Simulation Models of Cultural Evolution in R

Alex Mesoudi

2025-05-13

Contents

Introduction

This tutorial shows how to create very simple simulation or agent-based models of cultural evolution in R (R Core Team 2021). Currently these are:

- Model 1: Unbiased transmission
- Model 2: Unbiased and biased mutation
- Model 3: Biased transmission (direct/content bias)
- Model 4: Biased transmission (indirect bias)
- Model 5: Biased transmission (conformist bias)
- Model 6: Vertical and horizontal transmission
- Model 7: Migration
- Model 8: Blending inheritance
- Model 9: Demography and cultural gain/loss
- Model 10: Polarization
- Model 11: Cultural group selection
- Model 12: Historical dynamics
- Model 13: Social contagion
- Model 14: Social networks
- Model 15: Opinion formation
- Model 16: Bayesian iterated learning
- Model 17: Reinforcement learning
- Model 18: Evolution of social learning
- Model 19: Evolution of social learning strategies

How to use this tutorial

Each model is contained in a separate RMarkdown (Rmd) file. You can either (i) download each of these Rmd files from https://github.com/amesoudi/cultural_evolution_ABM_tutorial then open them in RStudio or another IDE, executing the code as you read the explanatory text, or (ii) read the online version of the tutorial at https://bookdown.org/amesoudi/ABMtutorial_bookdown/ which contains the compiled models with outputs. There is also a pdf version of the entire book on the github page. For maximum learning (and fun), I recommend (i), where you can execute and play around with the code yourself.

I assume you have basic knowledge of R as a programming language, e.g. the use of variables, dataframes, functions, subsetting and loops. If not, *Hands On Programming With R* by Garrett Grolemund is a good introduction.

I'm putting all model parameters in italicised *equation* text. This allows useful features such as superscripts (e.g. x^y) and subscripts (e.g. $x_{t=1}$). In RStudio, you can hover the cursor over the equation text to see this. All code variables are in regular *italics*, and all commands and functions in **bold**.

In RStudio use the green triangles above each code chunk to run that piece of code. Be sure to do this in the order they appear, as some chunks depend on previous chunks to work. You can also output the entire document including your executed code and formatting to html or pdf using the Knit button in the toolbar. Check the RMarkdown Cheat Sheet accessible via the Help menu for more details.

All the concepts covered here are introduced, discussed and mathematically modelled at an advanced level in Cavalli-Sforza & Feldman (1981) and Boyd & Richerson (1985). I explain them informally in Mesoudi (2011). A recent article (Mesoudi 2017) gives a current overview of cultural evolution research, with references to recent studies.

Each chapter has some exercises with suggestions for how to fully explore the models and extend them in interesting ways. Some chapters also have an 'Analytic Appendix', where I show how to derive the same results analytically.

How to cite this tutorial

You can cite the tutorial as:

- Mesoudi, Alex (2021) *Simulation models of cultural evolution in R.* doi:10.5281/zenodo.5155821. Available at https://github.com/amesoudi/cultural_evolution_ABM_tutorial and https://bookdown.org/amesoudi/ABMtutorial_bookdown/.

A longer and more detailed book-length resource, which builds on some of these models, can be found here:

- Alberto Acerbi, Alex Mesoudi, and Marco Smolla (2020) *Individual-based models of cultural evolution. A step-by-step guide using R.* doi:110.31219/osf.io/32v6a. Available at: https://acerbialberto.com/IBM-cultevo/

What is cultural evolution?

The theory of evolution is typically applied to genetic change. Darwin pointed out that the diversity and complexity of living things can be explained in terms of a deceptively simple process. Organisms vary in their characteristics. These characteristics are inherited from parent to offspring. Those characteristics that make an organism more likely to survive and reproduce will tend to increase in frequency. That's pretty much it. Since Darwin, biologists have filled in many of the details of this abstract idea. Geneticists have shown that 'characteristics' are determined by genes, and worked out where genetic variation comes from (e.g. mutation, recombination) and how genetic inheritance works (e.g. via Mendel's laws, and DNA). The details of selection have been explored, revealing the many reasons why some genes spread and others don't. Others realised that not all biological change results from selection, it can also result from random processes like population bottlenecks (genetic drift).

The theory of cultural evolution rests on the observation that culture constitutes a similar evolutionary process to that outlined above. By 'culture' we mean information that passes from one individual to another socially, rather than genetically. This could include things we colloquially call knowledge, beliefs, ideas, attitudes, customs, words, or values. These are all learned from others via various 'social learning' mechanisms such as imitation or spoken/written language. The key point is that social learning is an inheritance system. Cultural characteristics (or cultural traits) vary across individuals, they are passed from individual to individual, and in many cases some traits are more likely to spread than others. This is Darwin's insight, applied to culture. Cultural evolution researchers think that we can use similar evolutionary concepts, tools and methods to explain the diversity and complexity of culture, just as biologists have done for the diversity and complexity of living forms.

Importantly, we do not need to assume that cultural evolution is identical to genetic evolution. Many of the details will be different. To take an obvious example, we get DNA only from our two parents, but we can get ideas from many sources: teachers, strangers on the internet, long-dead authors' books, or even our parents. Cultural evolution researchers seek to build models and do empirical research to fill in these details.

Why model?

A formal model is a simplified version of reality, written in mathematical equations or computer code. Formal models are useful because reality is complex. We can observe changes in species or cultures over time, or particular patterns of biological or cultural diversity, but there are always a vast array of possible causes for any particular pattern or trend, and huge numbers of variables interacting in many different ways. A formal model is a highly simplified recreation of a small part of this complex reality, containing a few elements or processes that the modeller suspects are important. A model, unlike reality, can be manipulated and probed in order to better understand how each part works. No model is ever a complete recreation of reality. That would be pointless: we would have replaced a complex, incomprehensible reality with a complex, incomprehensible model. Instead, models are useful *because* of their simplicity.

Formal modelling is rare in the social sciences (with some exceptions, such as economics). Social scientists tend to be sceptical that very simple models can tell us anything useful about something as immensely complex as human culture. But the clear lesson from biology is that models are extremely useful in precisely this situation. Biologists face similarly immense complexity in the natural world. Despite this, models are useful. Population genetics models of the early 20th century helped to reconcile new findings in genetics with Darwin's theory of evolution. Ecological models helped understand interactions between species, such as predator-prey cycles. These models are hugely simplified: population genetics models typically make ridiculous assumptions like infinitely large populations and random mating. But they are useful because they precisely specify each part of a complex system, improving understanding of reality.

Another way to look at it is that all social scientists use models, but only some use *formal* models. Most models are verbal models, written in words. The problem is that words can be imprecise, and verbal models contain all kinds of hidden or unstated assumptions. The advantage of formal modelling is that we are forced to precisely specify every element and process that we propose, and make all of our assumptions explicit. Maths and code do not accept any ambiguity: they must be told absolutely everything. For more on the virtues of formal models for social scientists, see Paul Smaldino's 'Models are stupid, and we need more of them' (2017).

With these ideas in mind, let's turn to our first extremely simplified model of cultural evolution.

References

Boyd, R., & Richerson, P. J. (1985). Culture and the evolutionary process. University of Chicago Press.

Cavalli-Sforza, L. L., & Feldman, M. W. (1981). Cultural transmission and evolution: a quantitative approach. Princeton University Press.

Grolemund, G. (2014). Hands-on programming with R: Write your own functions and simulations. O'Reilly Media.

Mesoudi, A. (2011). Cultural evolution: How Darwinian theory can explain human culture and synthesize the social sciences. University of Chicago Press.

Mesoudi, A. (2017). Pursuing Darwin's curious parallel: Prospects for a science of cultural evolution. Proceedings of the National Academy of Sciences, 114(30), 7853-7860.

R Core Team (2021). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Smaldino, P. E. (2017). Models are stupid, and we need more of them. Computational Social Psychology, 311-331.

Model 1: Unbiased transmission

Introduction

Here we will simulate perhaps the simplest possible case of cultural evolution. We assume N individuals each of whom possesses one of two cultural traits, denoted A and B. Each generation, the N agents are replaced with N new agents. Each new agent picks a member of the previous generation at random and copies their cultural trait. This is known as unbiased oblique cultural transmission: unbiased because traits are copied entirely at random, and oblique because one generation learns from the previous non-overlapping generation (as opposed to horizontal cultural transmission where individuals copy members of the same generation, and vertical cultural transmission where offspring copy their parents - our model is way too simple to have actual parents and offspring though).

We are interested in tracking the proportion of individuals who possess trait A over successive generations. We will call this proportion p. We could also track the proportion who possess trait B, but this will always be $1 - p$ given that everyone who does not possess A must possess B, so there is really no need. For example, if 70% of the population have trait A, then $p = 0.7$. The remaining 30% must have trait B, which is a proportion $1 - p = 1 - 0.7 = 0.3$.

The output of the model will be a plot showing p over all generations (or timesteps) up to the last generation. Generations are denoted by t, where generation one is $t = 1$, generation two is $t = 2$, up to the last generation $t = t_{max}$. These could correspond to biological generations, but could equally be 'cultural generations' (or learning episodes) within the same fixed population, which would be much shorter.

Model 1

First we need to specify the fixed parameters of the model. These are N (the number of individuals) and t_{max} (the number of generations). Let's start with $N = 100$ and $t_{max} = 200$.

```
N <- 100
t_max <- 200
```

Run this code snippet in RStudio using the green 'play' triangle in the top right of the snippet. If you have the Environment pane visible on the right, you should be able to see N and t_{max} appear, with their assigned values. The Environment pane is useful for keeping track of active variables and their current values.

Now we need to create our agents. These will be stored in a dataframe, a commonly-used data format in R. The only information we need to keep about our agents is their cultural trait (A or B). Hence we need a dataframe that is N rows long, with a single column for the trait. We'll call this dataframe *agent*. Initially, we will give each agent either an A or B at random, using the **sample** command.

```
agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE))
```

Here, *trait* specifies the name of the sole variable/column in the *agent* dataframe. This is filled using **sample**. The first part of the **sample** command lists the elements to pick at random, in our case, the traits A and B. The second part gives the number of times to pick, in our case N times, once for each agent. The final part says to replace or reuse the elements after they've been picked. Otherwise, there would only be one copy of A and one copy of B, so we could only give two agents traits before running out.

We can see the first few lines of *agent* using the **head** command, to check it worked:

```
head(agent)
```

```
##   trait
## 1     A
## 2     B
## 3     B
## 4     A
## 5     B
## 6     B
```

As expected, there is a single column called *trait* containing As and Bs. Note that **head** only shows the first 6 rows for brevity; the other 94 are not shown. The numbers on the left are automatically generated in a dataframe as row labels, but in our case we can think of them as the agents' id numbers (agent 1, agent 2,..., agent 100).

A specific agent's trait can be retrieved using standard dataframe indexing. For example, agent 4's trait can be retrieved using:

```
agent$trait[4]
```

```
## [1] "A"
```

This should match the fourth row in the **head** output above.

We also need a dataframe to track the trait frequency p in each generation. This will have a single column with t_{max} rows, one for each generation. We'll call this dataframe *output*, because it is the output of the model. At this stage we don't know what p will be in each generation, so for now let's fill the *output* dataframe with lots of NAs, which is R's symbol for Not Available, or missing value. We can use the **rep** (repeat) command to repeat NA t_{max} times.

```
output <- data.frame(p = rep(NA, t_max))
```

Here, p is the name of the sole variable/column in the *output* dataframe, and it's filled entirely with NAs. We're using NA rather than, say, zero, because zero could be misinterpreted as $p = 0$, which would mean that all agents have trait B. This would be misleading, because at the moment we haven't yet calculated p, so it's non-existent, rather than zero.

We can, however, fill in the first value of p for our already-created first generation of agents, held in *agent*. The command below sums the number of As in *agent* and divides by N to get a proportion out of 1 rather than an absolute number. It then puts this proportion in the first slot of p in *output*, the one for the first generation, $t = 1$. We can again use **head** to check it worked.

```
output$p[1] <- sum(agent$trait == "A") / N
head(output)
```

```
##      p
## 1 0.46
## 2   NA
## 3   NA
## 4   NA
## 5   NA
## 6   NA
```

This first p value should be approximately 0.5: maybe not exactly, because we have a finite and relatively small population size. Analogously, flipping a coin 100 times will not always give exactly 50 heads and 50 tails. Sometimes we would get 51 heads, sometimes 52 heads, sometimes 48. Similarly, sometimes we will have 51 As, sometimes 48, etc.

Now we need to iterate our population over t_{max} generations. In each generation, we need to:

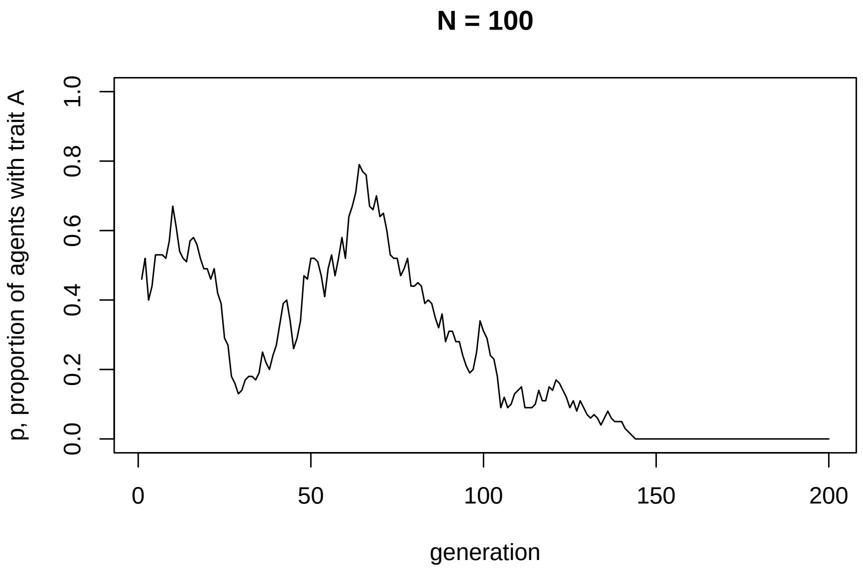
- copy the current agents to a separate dataframe called *previous_agent* to use as demonstrators for the new agents; this allows us to implement oblique transmission with its non-overlapping generations, rather than mixing up the generations and getting in a muddle
- create a new generation of agents, each of whose trait is picked at random from the *previous_agent* dataframe
- calculate p for this new generation and store it in the appropriate slot in *output*

To iterate, we'll use a for-loop, using t to track the generation. We've already done generation 1 so we'll start at generation 2. The random picking of models is done with **sample** again, but this time picking from the traits held in *previous_agent*. Note that I've added comments briefly explaining what each line does. This is perhaps superfluous in a tutorial like this, but it's always good practice. Code often gets cut-and-pasted into other places and loses its context. Explaining what each line does lets other people - and a future, forgetful you - know what's going on.

```r
for (t in 2:t_max) {

  # copy agent dataframe to previous_agent dataframe
  previous_agent <- agent

  # randomly copy from previous generation's agents
  agent <- data.frame(trait = sample(previous_agent$trait, N, replace = TRUE))

  # get p and put it into the output slot for this generation t
  output$p[t] <- sum(agent$trait == "A") / N

}
```

Now we should have 200 values of p stored in *output*, one for each generation. Let's plot them.

```r
plot(output$p,
     type = 'l',
     ylab = "p, proportion of agents with trait A",
     xlab = "generation",
     ylim = c(0,1),
     main = paste("N =", N))
```



Note the title of the graph, which gives the value of N for this simulation. It's easy to create plots and forget what the parameters were. It's therefore a good idea to include it somewhere on the graph. There's no need to include our other parameter, t_{max}, because that can be seen from the x-axis scale.

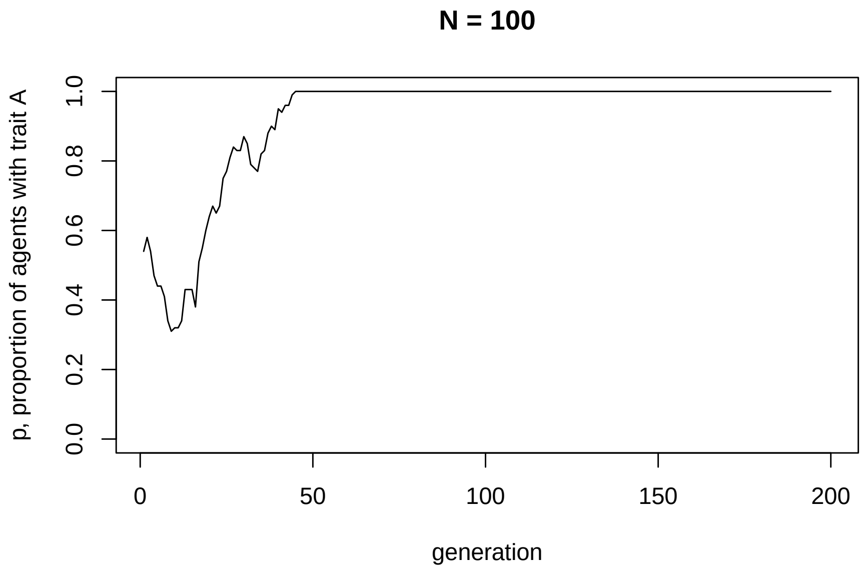
Unbiased transmission, or random copying, is by definition random, so different runs of this simulation will generate different plots. If you rerun all the code you'll get something different again. It probably starts off hovering around 0.5, the approximate starting value of p, and might go to 0 or 1 at some point. At $p = 0$ there are no As and every agent possesses B. At $p = 1$ there are no Bs and every agent possesses A. This is a typical feature of cultural drift, analogous to genetic drift: in small populations, with no selection or other directional processes operating, traits can be lost purely by chance.

Ideally we would like to repeat the simulation to explore this idea in more detail, perhaps changing some of the parameters. For example, if we increase N, are we more or less likely to lose one of the traits? With our code scattered about in chunks, it is hard to quickly repeat the simulation. Instead we can wrap it all up in a function, like so:

```r
UnbiasedTransmission <- function (N, t_max) {

  agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE))

  output <- data.frame(p = rep(NA, t_max))

  output$p[1] <- sum(agent$trait == "A") / N

  for (t in 2:t_max) {

    # copy agent to previous_agent dataframe
    previous_agent <- agent

    # randomly copy from previous generation
    agent <- data.frame(trait = sample(previous_agent$trait, N, replace = TRUE))

    # get p and put it into output slot for this generation t
    output$p[t] <- sum(agent$trait == "A") / N

  }

  plot(output$p,
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       main = paste("N =", N))

}
```

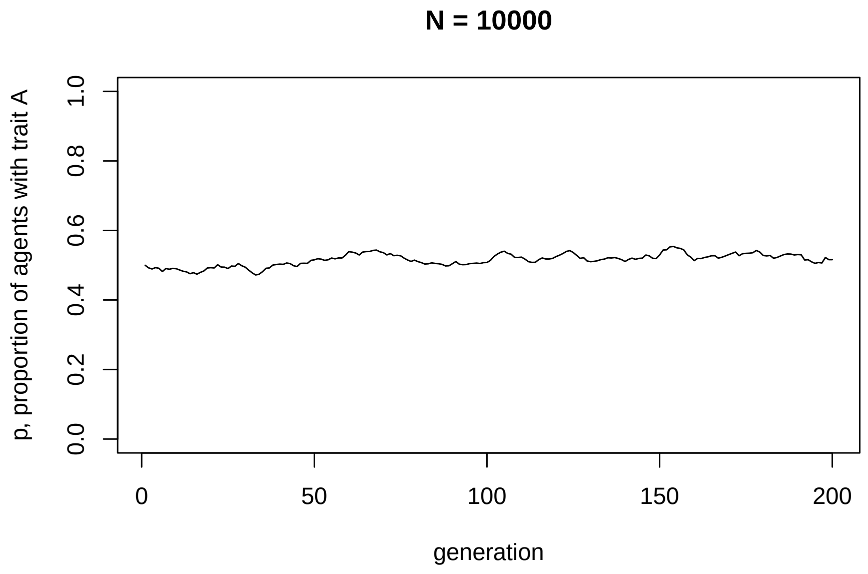
This is just all of the code snippets that we already ran above, but all within a function with parameters N and t_{max} as arguments to the function. Nothing will happen when you run the above code, because all you've done is define the function, not actually run it. The point is that we can now call the function in one go, easily changing the values of N and t_{max}. Let's try first with the same values of N and t_{max} as before, to check it works. You can re-run the code below several times, to see different dynamics. It should be different every time.

```r
UnbiasedTransmission(N = 100,
                     t_max = 200)
```



Now let's try changing the parameters. The following code re-runs the simulation with a much larger N.

```r
UnbiasedTransmission(N = 10000,
                     t_max = 200)
```



You should see much less fluctuation. Rarely in a population of $N = 10000$ will either trait go to fixation.

Wrapping a simulation in a function like this is good because we can easily re-run it with just a single command. However, it's a bit laborious to manually re-run it. Say we wanted to re-run the simulation 10 times with the same parameter values to see how many times A goes to fixation, and how many times B goes to fixation. Currently, we'd have to manually run the **UnbiasedTransmission** function 10 times and record somewhere else what happened in each run. It would be better to automatically re-run the simulation several times and plot each run as a separate line on the same plot. We could also add a line showing the mean value of p across all runs.

Let's use a new parameter r_{max} to specify the number of independent runs, and use another for-loop to cycle over the r_{max} runs. We also need to expand the number of columns in the *output* dataframe, because we need one column of p values for each of the runs. Let's rewrite the **UnbiasedTransmission** function to handle multiple runs.

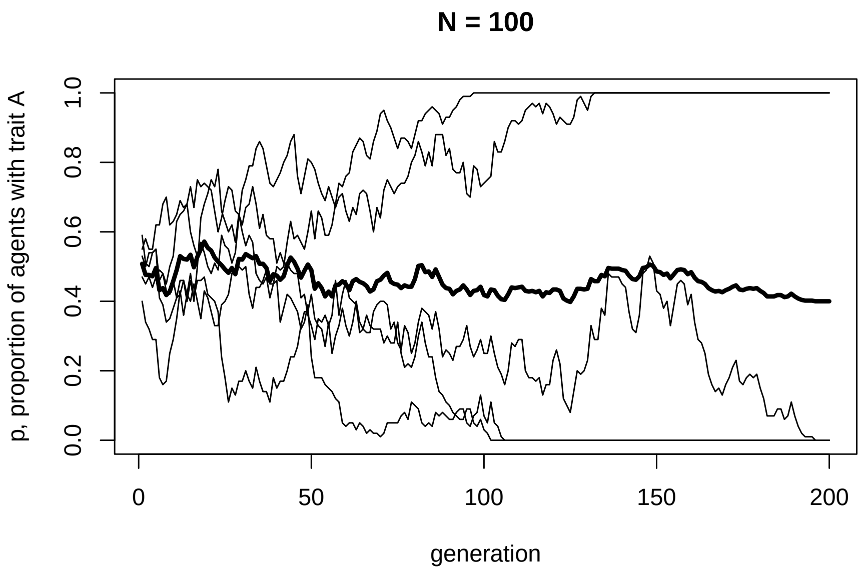
```
UnbiasedTransmission <- function (N, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")
  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # randomly copy from previous generation
      agent <- data.frame(trait = sample(previous_agent$trait, N, replace = TRUE))

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N =", N))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

There are a few changes here. First, we created *output* initially as a matrix, and then immediately converted it into a dataframe. This seems weird, but is just because it is easier to create a multi-row data structure using the **matrix** command than the **dataframe** command. The next command is purely cosmetic, and re-names the columns of *outcome* as run1, run2 etc.

Then we set up our *r* loop, which executes once for each run. The code is mostly the same as before, except that we now use the "[row,column]" notation to put *p* into the right place in *output*. The row is *t* as before, and now the column is *r*. The **plot** command is also changed to handle multiple runs: first we plot the mean as a thick line, then we add one plotted line for each run.

Finally, the **UnbiasedTransmission** function now ends with the *output* dataframe. This means that this dataframe will be exported from the function when it is run. This can be useful for storing data from simulations wrapped in functions, otherwise that data is lost after the function is executed. In the function call below, the raw data from the simulation is put into a dataframe called *data_model1*, as a record of what happened. Run it now to display the new plot.

```
data_model1 <- UnbiasedTransmission(N = 100,
                                    t_max = 200,
                                    r_max = 5)
```



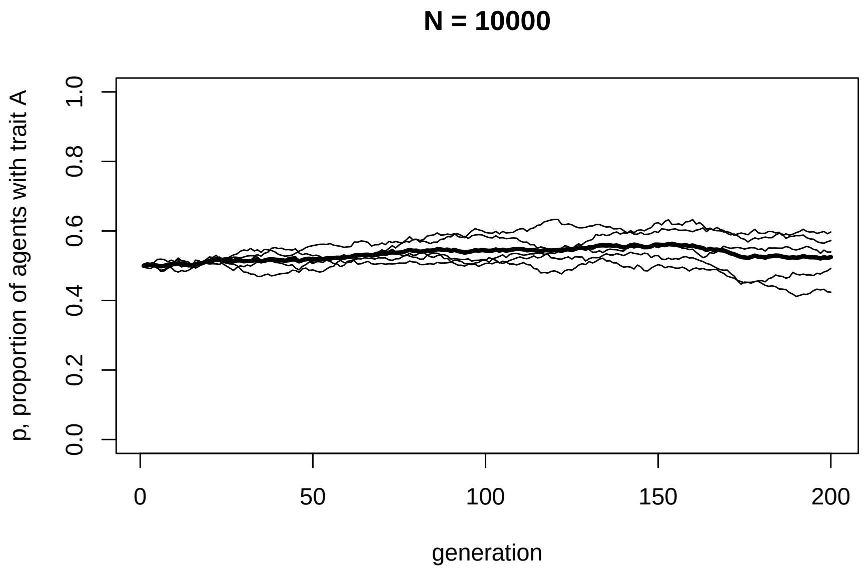
You should be able to see five independent runs of our simulation shown as regular thin lines, along with a thicker line showing the mean of these lines. Some runs have probably gone to 0 or 1, and the mean should be somewhere in between. The data is stored in *data_model1*, which we can inspect with **head**:

```
head(data_model1)
```

```
##   run1 run2 run3 run4 run5
## 1 0.40 0.59 0.53 0.55 0.47
## 2 0.34 0.51 0.50 0.58 0.45
## 3 0.32 0.50 0.54 0.55 0.47
## 4 0.29 0.54 0.54 0.55 0.44
## 5 0.29 0.55 0.55 0.62 0.47
## 6 0.18 0.46 0.41 0.62 0.49
```

Now let's run the updated **UnbiasedTransmission** model with $N = 10000$, to compare with $N = 100$.

```
data_model1 <- UnbiasedTransmission(N = 10000,
                                    t_max = 200,
                                    r_max = 5)
```



The mean line should be almost exactly at $p = 0.5$ now, with the five independent runs fairly close to it.

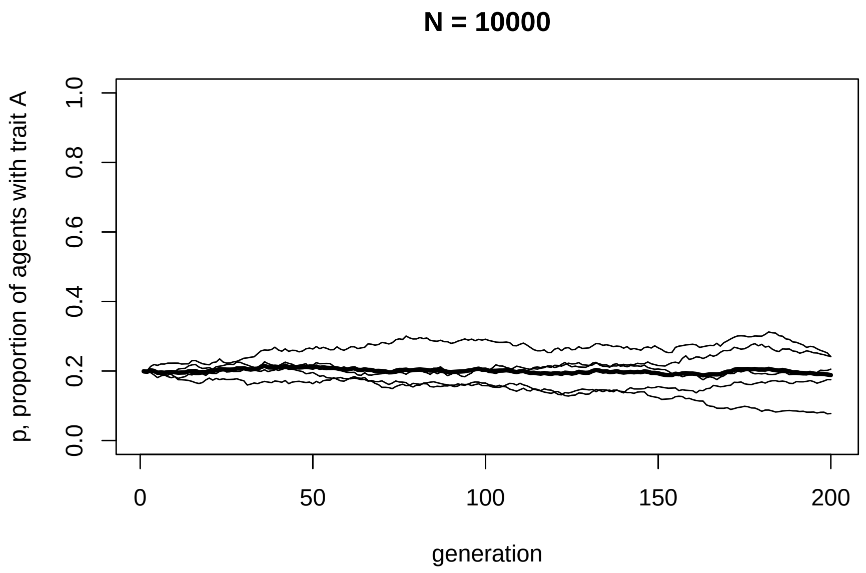
Let's add one final modification. So far the starting frequencies of A and B have been the same, roughly 0.5 each. But what if we were to start at different initial frequencies of A and B? Say, $p = 0.2$ or $p = 0.9$? Would unbiased transmission keep p at these initial values, or would it go to $p = 0.5$ as we have found so far?

To find out, we can add another parameter, p_0, which specifies the initial probability of drawing an A rather than a B in the first generation. Previously this was always $p_0 = 0.5$, but in the new function below we add it to the **sample** function to weight the initial allocation of traits in $t = 1$.

```
UnbiasedTransmission <- function (N, p_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA,t_max,r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                                       prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # randomly copy from previous generation
      agent <- data.frame(trait = sample(previous_agent$trait, N, replace = TRUE))

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N =", N))
  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

The only change here is the addition of p_0 as a definable parameter for the function, and the **prob** argument in the **sample** command. The **prob** argument gives the probability of picking each option, in our case A and B, in the first generation. The probability of A is now p_0, and the probability of B is now $1 - p_0$. Let's see what happens with a different value of p_0.

```
data_model1 <- UnbiasedTransmission(N = 10000,
                                    p_0 = 0.2,
                                    t_max = 200,
                                    r_max = 5)
```



With $p_0 = 0.2$, trait frequencies stay at $p = 0.2$. Unbiased transmission is truly non-directional: it maintains trait frequencies at whatever they were in the previous generation, barring random fluctuations caused by small population sizes.

A final useful skill is to be able to export the plot we created to an external image file. This file can then be inserted into other documents, e.g. as a figure in a manuscript. This is done in the code below by wrapping the **UnbiasedTransmission** function (or any function that outputs a plot) within two commands. First, the **png** command specifies the file format (you can also use **bmp**, **jpeg** or **tiff**), and takes arguments that set the file name, height and width, units of the height and width ("cm", "mm", "in" or "px"), and resolution (in ppi, pixels per inch). Then after the plot is created using **UnbiasedTransmission**, the **dev.off()** command tells R that we are done plotting and can export the file. It's wrapped in **invisible** just to hide the text output in this Rmd file.

Note that this code puts the plot file into the current working directory. To set the working directory to the same place as this Rmd file, use the menu command Session->Set Working Directory->To Source File Location before running the code below. Otherwise, use the command **setwd** to set the working directory to somewhere else, or add a full path to the **file** argument below, within the quote marks where the file name is given.

```r
png(file = "unbiased_transmission.png",
    height = 10,
    width = 12,
    units = "cm" ,
    res = 300)

data_model1 <- UnbiasedTransmission(N = 10000,
                                     p_0 = 0.2,
                                     t_max = 200,
                                     r_max = 5)

invisible(dev.off())
```

Summary

Even this extremely simple model provides some valuable insights. First, unbiased transmission does not in itself change trait frequencies. As long as populations are large, trait frequencies remain the same.

Second, the smaller the population size, the more likely traits are to be lost by chance. This is a basic insight from population genetics, known there as genetic drift, but it can also be applied to cultural evolution. More advanced models of cultural drift (sometimes called 'random copying') can be found in Cavalli-Sforza & Feldman (1981) and Bentley et al. (2004), with the latter showing that various real-world cultural traits exhibit dynamics consistent with this kind of process, including baby names, dog breeds and archaeological pottery types.

Furthermore, generating expectations about cultural change under simple assumptions like random cultural drift can be useful for detecting non-random patterns like selection. If we don't have a baseline, we won't know selection or other directional processes when we see them.

In Model 1 we have introduced several programming techniques that will be useful in later models. We've seen how to use dataframes to hold characteristics of agents, how to use loops to cycle through generations and simulation runs, how to use **sample** to pick randomly from sets of elements, how to wrap simulations in functions to easily re-run them with different parameter values, how to plot the results of simulations, how to store the output of the simulation in a dataframe that persists after the simulation function is run, and how to export a plot to an external image file.

Exercises

1. Try different values of p_0 with large N to confirm that unbiased transmission does not change the frequency of p, irrespective of the starting value p_0.
2. Re-run the **UnbiasedTransmission** model to find the approximate value of N at which one of the traits rarely, if ever, goes to fixation after a reasonably large number of timesteps ('fixation' means that p goes to either zero or one and stays there).
3. Modify the **UnbiasedTransmission** function to record and output the number of timesteps it takes for a run to go to fixation, i.e. p goes to zero or one. If a run does not go to fixation after a sufficiently large number of timesteps, record this as NA. Make a plot of time to fixation against population size N. Before creating the plot, what do you think the plot will look like?

Analytical Appendix

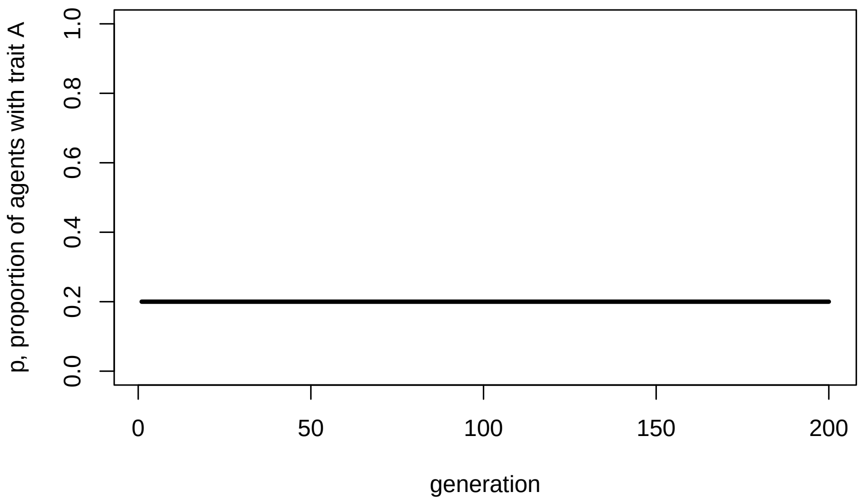
If p is the frequency of A in one generation, we are interested in calculating p', the frequency of A in the next generation under the assumption of unbiased transmission. Each new individual in the next generation picks a demonstrator at random from amongst the previous generation. The demonstrator will have A with probability p. The frequency of A in the next generation, then, is simply the frequency of A in the previous generation:

$$p' = p \tag{1.1}$$

Equation 1.1 simply says that under unbiased transmission there is no change in p over time. If, as we assumed above, the initial value of p in a particular population is p_0, then the equilibrium value of p, p^*, at which there is no change in p over time, is just p_0.

We can plot this recursion, to recreate the final simulation plot above:

```r
p_0 <- 0.2
t_max <- 200

p <- rep(NA, t_max)
p[1] <- p_0

for (i in 2:t_max) {
  p[i] <- p[i-1]
}

plot(p,
     type = 'l',
     ylab = "p, proportion of agents with trait A",
     xlab = "generation",
     ylim = c(0,1),
     lwd = 3)
```



Don't worry, it gets more complicated than this in later chapters. The key point here is that analytical or deterministic models like these assume infinite populations - note there is no N in the above recursion - and no stochasticity. Simulations with very large populations should give the same results as analytical models. Basically, the closer we can get in stochastic models to the assumption of infinite populations, the closer the match to infinite-population deterministic models. Deterministic models give the ideal case; stochastic models permit more realistic dynamics based on finite populations.

More generally, creating deterministic recursion-based models can be a good way of verifying simulation models, and vice versa. If the same dynamics occur in both agent-based and recursion-based models, then we can be more confident that those dynamics are genuine and not the result of a programming error or mathematical mistake.

References

Bentley, R. A., Hahn, M. W., & Shennan, S. J. (2004). Random drift and culture change. Proceedings of the Royal Society of London B, 271(1547), 1443-1450.

Cavalli-Sforza, L. L., & Feldman, M. W. (1981). Cultural transmission and evolution: a quantitative approach. Princeton University Press.

Model 2: Unbiased and biased mutation

Introduction

Evolution doesn't work without a source of variation that introduces new variation upon which selection, drift and other processes can act. In genetic evolution, mutation is almost always blind with respect to function. Beneficial genetic mutations are no more likely to arise when they are needed than when they are not needed - in fact most genetic mutations are neutral or detrimental to an organism. Cultural evolution is more interesting, in that novel variation may sometimes be directed to solve specific problems, or systematically biased due to features of our cognition. In the models below we'll simulate both unbiased and biased mutation.

Model 2a: Unbiased mutation

First we will simulate unbiased mutation in the same basic model as used in Model 1 (Unbiased transmission). We'll remove unbiased transmission to see the effect of unbiased mutation alone.

As in Model 1, we assume N individuals each of whom possesses one of two cultural traits, denoted A and B. In each generation from $t = 1$ to $t = t_{max}$, the N agents are replaced with N new agents. Instead of random copying, each agent now gives rise to a new agent with exactly the same cultural trait as them. (Another way of looking at this is in terms of timesteps, such as years: the same N agents live for t_{max} years, and keep their cultural trait from one year to the next.)

Each generation, there is a probability μ that each agent mutates from their current trait to the other trait. This probability applies to each agent independently; whether an agent mutates has no bearing on whether or how many other agents have mutated. On average, that means that μN agents mutate each generation. Like in Model 1, we are interested in tracking the proportion p of agents with trait A over time.

We'll wrap this in a function called **UnbiasedMutation**, using much of the same code as **UnbiasedTransmission**. Now though we have an extra parameter, μ, to specify.

```
UnbiasedMutation <- function (N, mu, p_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get N random numbers each between 0 and 1
      mutate <- runif(N)

      # if agent was A, with probability mu, flip to B
      agent$trait[previous_agent$trait == "A" & mutate < mu] <- "B"

      # if agent was B, with probability mu, flip to A
      agent$trait[previous_agent$trait == "B" & mutate < mu] <- "A"

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", mu = ", mu, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

The only changes from Model 1 are the addition of μ in the function definition and the plot title, and three new lines of code within the t **for** loop which replace the random copying command with unbiased mutation. Let's examine these three lines to see how they work.

The most obvious way of implementing unbiased mutation - which is NOT done above - would have been to set up another **for** loop. We would cycle through each agent one by one, each time calculating whether it should mutate or not based on $/mu$. This would certainly work, but R is notoriously slow at loops. It's always preferable in R, where possible, to use 'vectorised' code. That's what is done above in our three added lines.

First we pre-specify the probability of mutating for each agent. This is done with the **runif** (**r**andom draws from a **unif**orm distribution) command which generates N random numbers each between 0 and 1, and puts them into a variable called *mutate*. If the ith value in *mutate* is less than or equal to μ, then the ith agent in our *agent* dataframe will mutate. This is done on the subsequent two lines, first for agents that were previously A, and then for agents that were previously B.

We can think about this by imagining what happens at extreme values of μ. If $\mu = 1$, then that agent's *mutate* value will always be less than μ, because the maximum value of *mutate* is 1 (we can ignore the case when *mutate* happens to be exactly 1.000, as it will very rarely happen). The inequality is therefore always true, and the agent always mutates. That's what we want to happen, if $\mu = 1$. The following code illustrates this:

```
N <- 100
mu <- 1  # maximum mutation rate of 1

# get N random numbers each between 0 and 1
mutate <- runif(N)

# always mutate
mutate < mu
```

```
##   [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [16] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [31] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [46] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [61] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [76] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
##  [91] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

If $\mu = 0$, then that agent's *mutate* value will never be less than μ, because the minimum value of *mutate* is 0. The inequality is therefore never true, and the agent never mutates. Again, that's what we want, if $\mu = 0$. In code:

```
N <- 100
mu <- 0  # minimum mutation rate of 0

# get N random numbers each between 0 and 1
mutate <- runif(N)

# never mutate
mutate < mu
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [97] FALSE FALSE FALSE FALSE
```

At intermediate values, say $\mu = 0.1$, then the inequality is true for 10% of the *mutate* values, or in other words for 10% of the agents. Again, in code:

```
N <- 100
mu <- 0.1  # mutation rate of 10%

# get N random numbers each between 0 and 1
mutate <- runif(N)

# 10% of agents mutate
mutate < mu
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
##  [25]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
##  [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [61] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
##  [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [97] FALSE FALSE FALSE FALSE
```

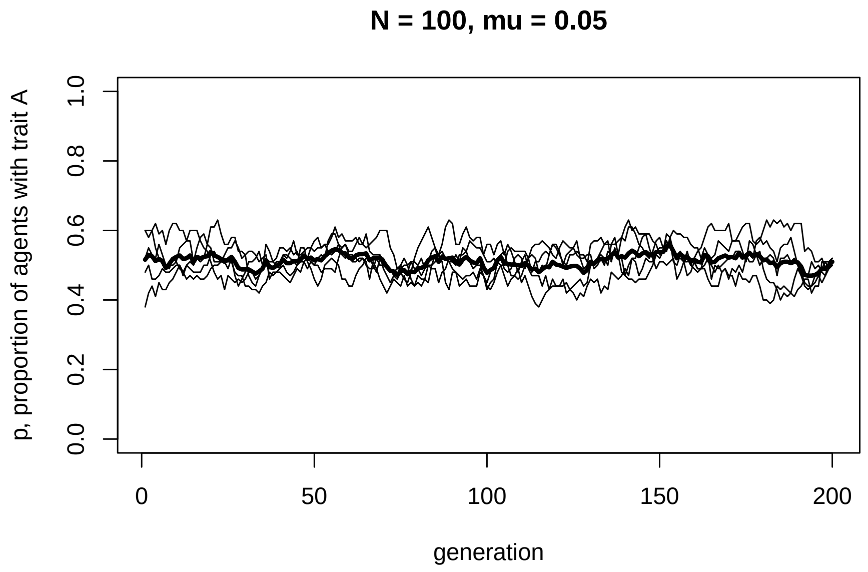
```
sum(mutate < mu) / N
```

```
## [1] 0.06
```

The value gives the proportion of mutating agents, and should be approximately (not necessarily exactly) 0.1.

Let's run this unbiased mutation model.

```
data_model2a <- UnbiasedMutation(N = 100, mu = 0.05, p_0 = 0.5, t_max = 200, r_max = 5)
```

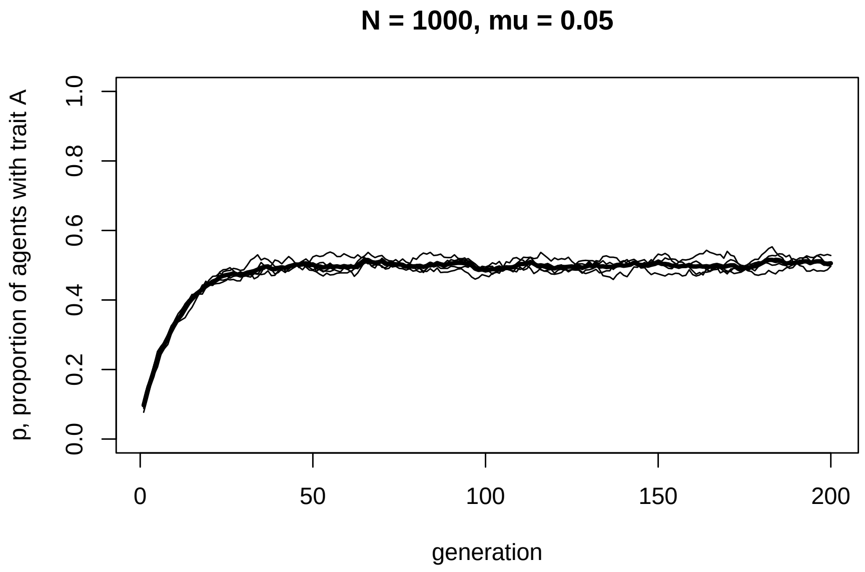


As one might expect, unbiased mutation produces random fluctuations over time, and does not alter the overall frequency of A which stays around $p = 0.5$. Because mutations from A to B are as equally likely as B to A, there is no overall directional trend.

But what if we were to start at different initial frequencies of A and B? Say, $p = 0.2$ or $p = 0.9$? Would unbiased mutation keep p at these initial values, like we saw unbiased transmission does in Model 1?

To find out, let's change p_0, which, as you may recall from Model 1, specifies the initial probability of drawing an A rather than a B in the first generation.

```
data_model2a <- UnbiasedMutation(N = 1000, mu = 0.05, p_0 = 0.1, t_max = 200, r_max = 
```



You should see p go from 0.1 up to 0.5. In fact, whatever the initial starting frequencies of A and B, unbiased mutation always leads to $p = 0.5$. Unlike the unbiased transmission simulated in Model 1, with unbiased mutation it is impossible for one trait to be lost and the other go to fixation. Unbiased mutation introduces and maintains cultural variation in the population.

Model 2b: Biased mutation

A more interesting case is biased mutation. Let's assume now that there is a probability μ_b that an agent with trait B mutates into A, but there is no possibility of trait A mutating into trait B. Perhaps trait A is a particularly catchy or memorable version of a story, or an intuitive explanation of a phenomenon, and B is difficult to remember or unintuitive.

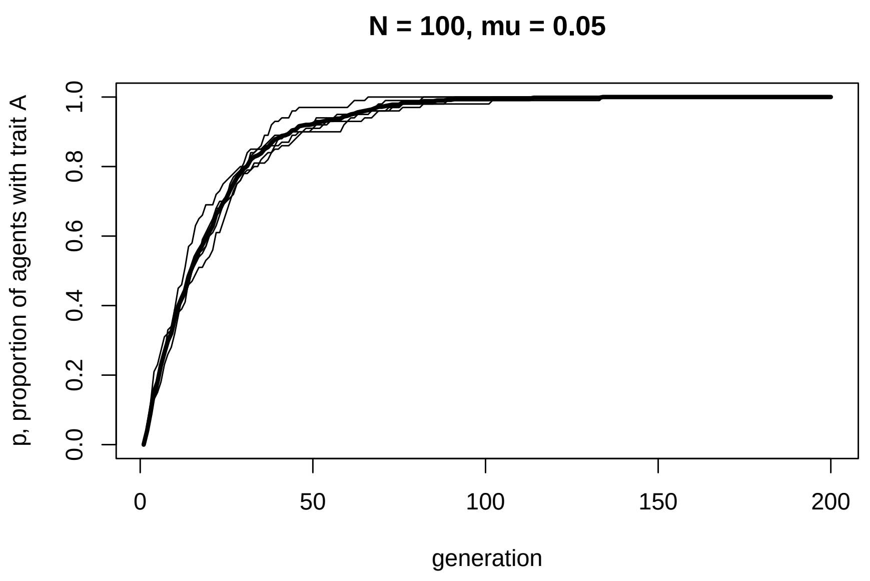
The function **BiasedMutation** captures this unidirectional mutation.

```r
BiasedMutation <- function (N, mu_b, p_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataframe
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")
  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get N random numbers each between 0 and 1
      mutate <- runif(N)

      # if agent was B, with prob mu_b, flip to A
      agent$trait[previous_agent$trait == "B" & mutate < mu_b] <- "A"

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", mu = ", mu_b, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

There are just two changes in this code compared to **UnbiasedMutation**. First, we've replaced μ with μ_b to keep the two parameters distinct and avoid confusion. Second, the line in **UnbiasedMutation** which caused agents with A to mutate to B has been deleted.

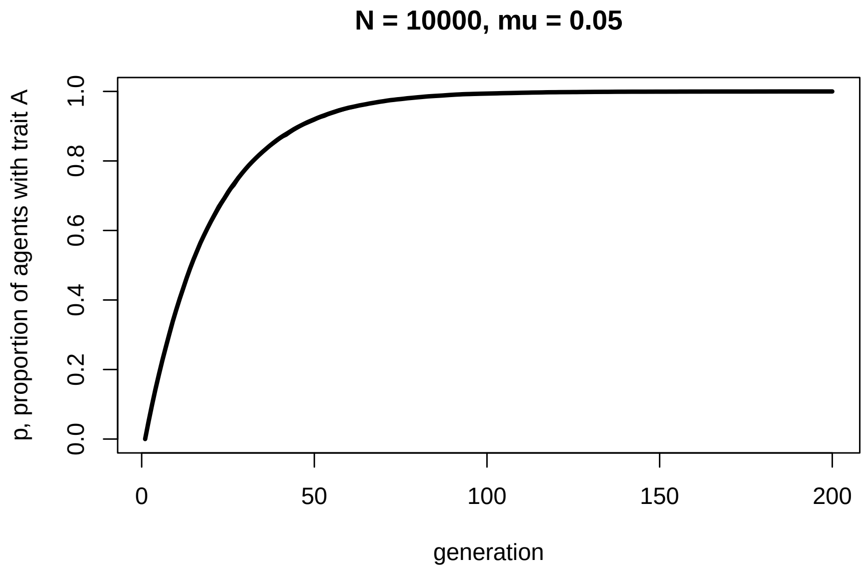
Let's see what effect this has by running **BiasedMutation**. We'll start with the population entirely composed of agents with B, i.e. $p_0 = 0$, to see how quickly and in what manner A spreads via biased mutation.

```r
data_model2b <- BiasedMutation(N = 100, mu_b = 0.05, p_0 = 0, t_max = 200, r_max = 5)
```



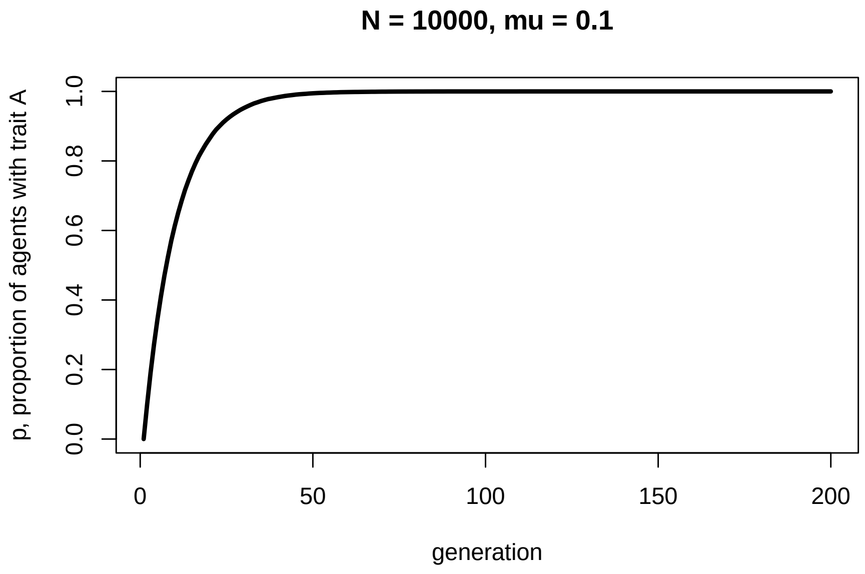
The plot should show a steep increase that slows and plateaus at $p = 1$ by around generation $t = 100$. There should be a bit of fluctuation in the different runs, but not much. Now let's try a larger sample size.

```r
data_model2b <- BiasedMutation(N = 10000, mu_b = 0.05, p_0 = 0, t_max = 200, r_max = 5)
```



With $N = 10000$ the line should be smooth with little fluctuation across the runs. But notice that it plateaus at about the same generation, around $t = 100$. Population size has little effect on the rate at which a novel trait spreads via biased mutation. μ_b, on the other hand, does affect this speed. Let's double the biased mutation rate to 0.1.

```
data_model2b <- BiasedMutation(N = 10000, mu_b = 0.1, p_0 = 0, t_max = 200, r_max = 5)
```



Now trait A reaches fixation around generation $t = 50$. Play around with N and μ_b to confirm that the latter determines the rate of diffusion of trait A, and that it takes the same form each time - roughly an ‘r’ shape with an initial steep increase followed by a plateauing at $p = 1$.

Summary

With this simple model we can draw the following insights. Unbiased mutation, which resembles genetic mutation in being non-directional, always leads to an equal mix of the two traits. It introduces and maintains cultural variation in the population. It is interesting to compare unbiased mutation to unbiased transmission from Model 1. While unbiased transmission did not change p over time, unbiased mutation always converges on $p^* = 0.5$, irrespective of the starting frequency. (NB $p^* = 0.5$ assuming there are two traits; more generally, $p^* = 1/v$, where v is the number of traits.)

Biased mutation, which is far more common - perhaps even typical - in cultural evolution, shows different dynamics. Novel traits favoured by biased mutation spread in a characteristic fashion - an r-shaped diffusion curve - with a speed characterised by the mutation rate μ_b. Population size has little effect, whether $N = 100$ or $N = 10000$. This is because mutation is an individual-level process, and does not depend on the traits of any other agent(s) in the population.

Whenever biased mutation is present ($\mu_b > 0$), the favoured trait goes to fixation, even if it is not initially present.

A form of unbiased cultural mutation can be observed when people try to copy artifact sizes or shapes, and the limits of our perceptual systems introduces random noise into the copied form. This has been studied in the context of the transmission of archaeological artifacts such as handaxes and arrowheads (Eerkens & Lipo 2005) and confirmed experimentally (Kempe et al. 2012).

Biased cultural mutation has been argued to result from universal features of human cognition, as different people or groups independently transform cultural traits towards certain 'attractive' forms. For example, studies have shown how portraits of subjects averting their gaze systematically mutated into portraits of subjects with direct eye gaze (Morin 2013), and how blood-letting as a medical practice independently emerged in different societies around the world (Miton et al. 2015).

In terms of programming techniques, the major novelty in Model 2 is the use of **runif** to generate a series of N random numbers from 0 to 1 and compare these to a fixed probability (in our case, μ or μ_b) to determine which agents should undergo whatever the fixed probability specifies (in our case, mutation). This could be done with a loop, but vectorising code in the way we did here is much faster in R than loops.

Exercises

1. Try different values of p_0 in **UnbiasedMutation** to confirm that any starting value converges on approximately $p = 0.5$.
2. Try different values of p_0 in **BiasedMutation** to confirm that any starting value converges on $p = 1$.
3. Try different values of N and μ_b in **BiasedMutation** to confirm that μ_b does, and N does not, affect the speed with which p reaches fixation.
4. Create an alternative **UnbiasedMutation** function which uses a for-loop to cycle through each agent and, for each one, mutate its cultural trait with probability μ. Use the function **system.time()** to compare the time that this looping function takes with the time the original vectorised **UnbiasedMutation** function takes, for the same parameter values. Is the vectorised version faster? If so, how many times faster?
5. Add a parameter μ_a to **BiasedMutation** which determines the probability that an agent with A mutates to B. Run the simulation to show that the equilibrium value of p, and the speed at which this equilibrium is reached, depends on the difference between mu_a and mu_b.

6. Add a third trait, C, to the **UnbiasedMutation** function. The first generation should have traits set using **sample** with c("A", "B", "C") rather than c("A", "B"). You will need to define a new parameter, q_0, which gives the probability of drawing a B, equivalent to how p_0 gives the probability of drawing an A. The probability of drawing a C is then $1 - p_0 - q_0$, given that all the probabilities have to add up to one. These probabilities can be entered into the *prob* argument of **sample**. Then modify the unbiased mutation lines such that, if an agent has trait A, with probability μ they have an equal chance of mutating into either B or C; agents with trait B similarly have a probability μ of mutating into either A or C; and agents with C mutate with probability μ into either A or B. Record and plot p, the frequency of A, as before. Does p still converge on 0.5, as it does with only two traits?

Analytical Appendix

If p is the frequency of A in one generation, we are interested in calculating p', the frequency of A in the next generation under the assumption of unbiased mutation. The next generation retains the cultural traits of the previous generation, except that μ of them switch to the other trait. There are therefore two sources of A in the next generation: members of the previous generation who had A and didn't mutate, therefore staying A, and members of the previous generation who had B and did mutate, therefore switching to A. The frequency of A in the next generation is therefore:

$$p' = p(1 - \mu) + (1 - p)\mu \tag{2.1}$$

The first term on the right-hand side of Equation 2.1 represents the first group, the $(1 - \mu)$ proportion of the p A-carriers who didn't mutate. The second term represents the second group, the μ proportion of the $1 - p$ B-carriers who did mutate.

To calculate the equilibrium value of p, p^*, we want to know when $p' = p$, or when the frequency of A in one generation is identical to the frequency of A in the next generation. This can be found by setting $p' = p$ in Equation 2.1, which gives:

$$p = p(1 - \mu) + (1 - p)\mu \tag{2.2}$$

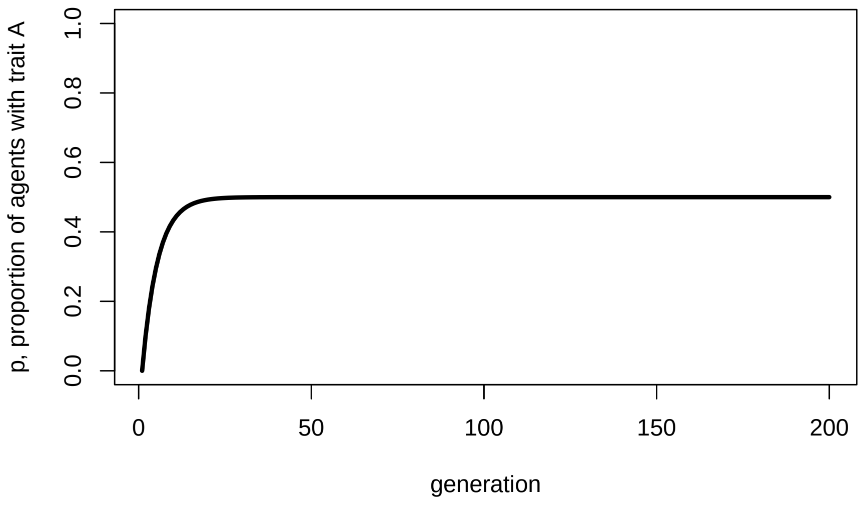
Rearranging Equation 2.2 gives:

$$\mu(1 - 2p) = 0 \tag{2.3}$$

The left-hand side of Equation 2.3 equals zero when either $\mu = 0$, which given our assumption that $\mu > 0$ cannot be the case, or when $1 - 2p = 0$, which after rearranging gives the single equilibrium $p^* = 0.5$. This matches our simulation results above. As we found in the simulations, this does not depend on μ or the starting frequency of p.

We can also plot the recursion in Equation 2.1 like so:

```
p_0 <- 0
t_max <- 200
mu <- 0.1

p <- rep(NA, t_max)
p[1] <- p_0

for (i in 2:t_max) {
  p[i] <- p[i-1]*(1 - mu) + (1-p[i-1])*mu
}

plot(p,
     type = 'l',
     ylab = "p, proportion of agents with trait A",
     xlab = "generation",
     ylim = c(0,1),
     lwd = 3)
```



This should resemble the figure generated by the simulations above, and confirm that $p^* = 0.5$.

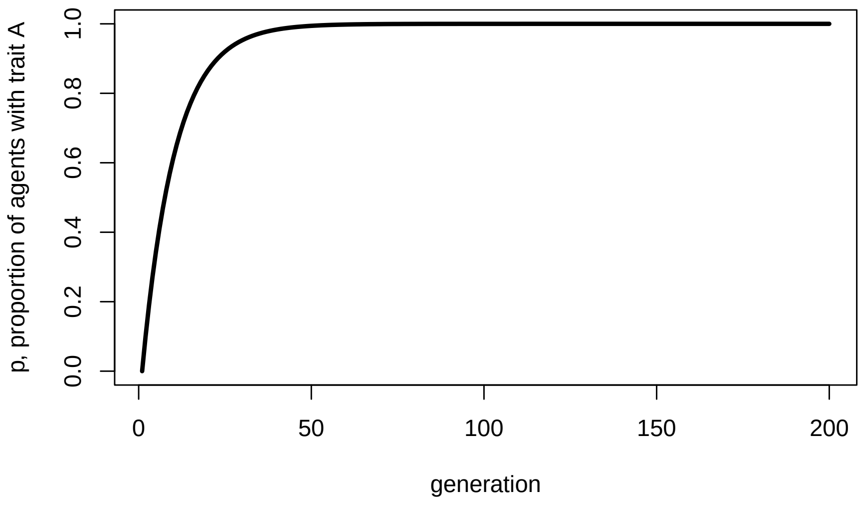
For biased mutation, assume that only Bs are switching to A, and with probability μ_b instead of μ. The first term on the right hand side becomes simply p, because As do not switch. The second term remains the same, but with μ_b. Thus,

$$p' = p + (1 - p)\mu_b \tag{2.4}$$

The equilibrium value p^* can be found by again setting $p' = p$ and solving for p. Assuming $\mu_b > 0$, this gives the single equilibrium $p^* = 1$, which again matches the simulation results.

We can plot the above recursion like so:

```r
p_0 <- 0
t_max <- 200
mu_b <- 0.1

p <- rep(NA, t_max)
p[1] <- p_0

for (i in 2:t_max) {
  p[i] <- p[i-1] + (1 - p[i-1])*mu_b
}

plot(p,
     type = 'l',
     ylab = "p, proportion of agents with trait A",
     xlab = "generation",
     ylim = c(0,1),
     lwd = 3)
```



Hopefully, this looks identical to the final simulation plot with the same value of μ_b.

Furthermore, we can specify an equation for the change in p from one generation to the next, or Δp. We do this by subtracting p from both sides of Equation 2.4, giving:

$$\Delta p = p' - p = (1 - p)\mu_b \tag{2.5}$$

Seeing this helps explain two things. First, the $1 - p$ part explains the r-shape of the curve. It says that the smaller is p, the larger Δp will be. This explains why p increases in frequency very quickly at first, when p is near zero, and the increase slows when p gets larger. We have already determined that the increase stops altogether (i.e. $\Delta p = 0$) when $p = p^* = 1$.

Second, it says that the rate of increase is proportional to μ_b. This explains our observation in the simulations that larger values of μ_b cause p to reach its maximum value faster.

References

Eerkens, J. W., & Lipo, C. P. (2005). Cultural transmission, copying errors, and the generation of variation in material culture and the archaeological record.

Journal of Anthropological Archaeology, 24(4), 316-334.

Kempe, M., Lycett, S., & Mesoudi, A. (2012). An experimental test of the accumulated copying error model of cultural mutation for Acheulean handaxe size. PLoS One, 7(11), e48333.

Miton, H., Claidière, N., & Mercier, H. (2015). Universal cognitive mechanisms explain the cultural success of bloodletting. Evolution and Human Behavior, 36(4), 303-312.

Morin, O. (2013). How portraits turned their eyes upon us: visual preferences and demographic change in cultural evolution. Evolution and Human Behavior, 34(3), 222-229.

Model 3: Biased transmission (direct / content bias)

Introduction

So far we have looked at unbiased transmission (Model 1) and unbiased/biased mutation (Model 2). Let's complete the set by looking at biased transmission. This occurs when one trait or one demonstrator is more likely to be copied than another trait or demonstrator. Trait-based copying is often called 'direct' or 'content' bias, while demonstrator-based copying is often called 'indirect' or 'context' bias. Both are sometimes also called 'cultural selection' because one thing (trait or demonstrator) is selected to be copied over another. In Model 3 we'll look at trait-based (direct, content) bias.

(As an aside, there is a confusing array of terminology in the field of cultural evolution, as illustrated by the preceding paragraph. That's why models are so useful. Words and verbal descriptions can be ambiguous. Often the writer doesn't realise that there are hidden assumptions or unrecognised ambiguities in their descriptions. They may not realise that what they mean by 'cultural selection' is entirely different to how someone else uses it. Models are great because they force us to specify exactly what we mean by a particular term or process. I can use the words in the paragraph above to describe biased transmission, but it's only really clear when I model it, making all my assumptions explicit.)

Model 3

As in Model 1 and Model 2, we assume there are two traits A and B. Let's assume that biased transmission favours trait A. Perhaps A is a more effective tool, more memorable story, or more easily pronounced word. We're not including any mutation in the model, so we need to include some As at the beginning of the simulation otherwise it would never appear. However, let's make it initially rare. Then we can see how selection favours this initially-rare trait.

To simulate biased transmission, following Model 1, we assume that each agent chooses another agent from the previous generation at random. But this time, if that chosen agent possesses trait A, then the focal agent copies trait A with probability s. This parameter s gives the strength of biased transmission, or the probability that an agent encountering another agent with a more favourable trait than their current trait abandons their current trait and adopts the new trait. If $s = 0$, there is no selection and agents never switch as a result of biased transmission. If $s = 1$, then agents always switch when encountering a favoured alternative.

Below is a function **BiasedTransmission** that implements all of these processes.

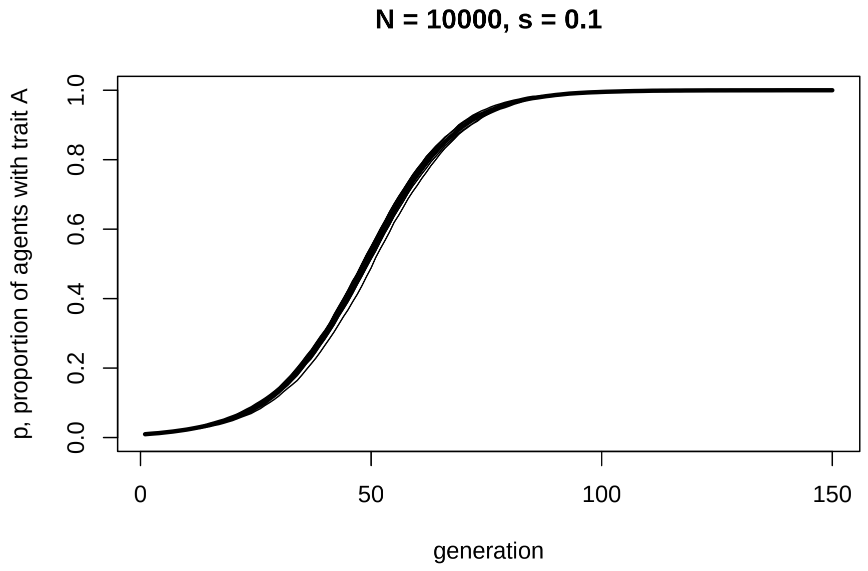
```
BiasedTransmission <- function (N, s, p_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # biased transmission

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # for each agent, pick a random agent from the previous generation
      # as demonstrator and store their trait
      demonstrator_trait <- sample(previous_agent$trait, N, replace = TRUE)

      # get N random numbers each between 0 and 1
      copy <- runif(N)
      # if demonstrator has A and with probability s, copy A from demonstrator
      agent$trait[demonstrator_trait == "A" & copy < s] <- "A"

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", s = ", s, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

Most of **BiasedTransmission** is recycled from Model 1 and Model 2. As before, we set up a dataframe to hold the *output* from multiple runs, and in generation $t = 1$ create a dataframe to hold the trait of each *agent*. The plot function is also similar, but now we add s to the plot title so we don't forget it.

The major change is that we now include biased transmission from the second generation onwards. Using vectorised code, we pick for each of N agents one of the previous generation's agents at random and store their trait in *demonstrator_trait*. Then we get random numbers between 0 and 1 for each agent and store these in *copy*. If the demonstrator has trait A (*demonstrator_trait* == "*A*"), and with probability s (*copy* $< s$), then the agent adopts trait A.

Let's run our **BiasedTransmission** model. Remember we are starting with a population with a small number of As, so $p_0 = 0.01$.

```
data_model3 <- BiasedTransmission(N = 10000, s = 0.1, p_0 = 0.01, t_max = 150, r_max =
```

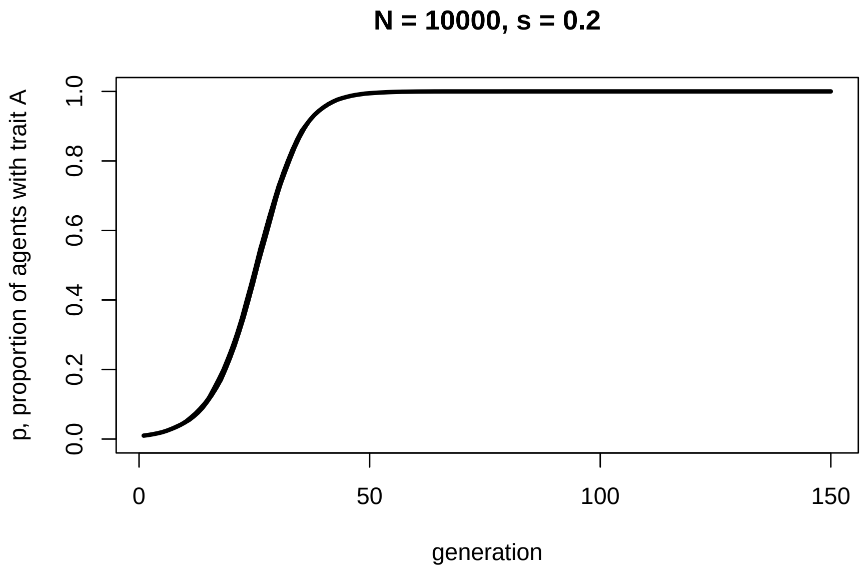


With a moderate selection strength of $s = 0.1$, we can see that A gradually replaces B and goes to fixation. It does this in a characteristic manner: the increase is slow at first, then picks up speed, then plateaus.

Note the difference to biased mutation. Where biased mutation was r-shaped, with a steep initial increase, biased transmission is s-shaped, with an initial slow uptake. This is because the strength of biased transmission, like selection in general, is proportional to the variation in the population. When A is rare initially, there is only a small chance of picking another agent with A. As A spreads, the chances of picking an A agent increases. As A becomes very common, there are few B agents left to switch.

Let's double the selection strength to $s = 0.2$, below.

```
data_model3 <- BiasedTransmission(N = 10000, s = 0.2, p_0 = 0.01, t_max = 150, r_max =
```

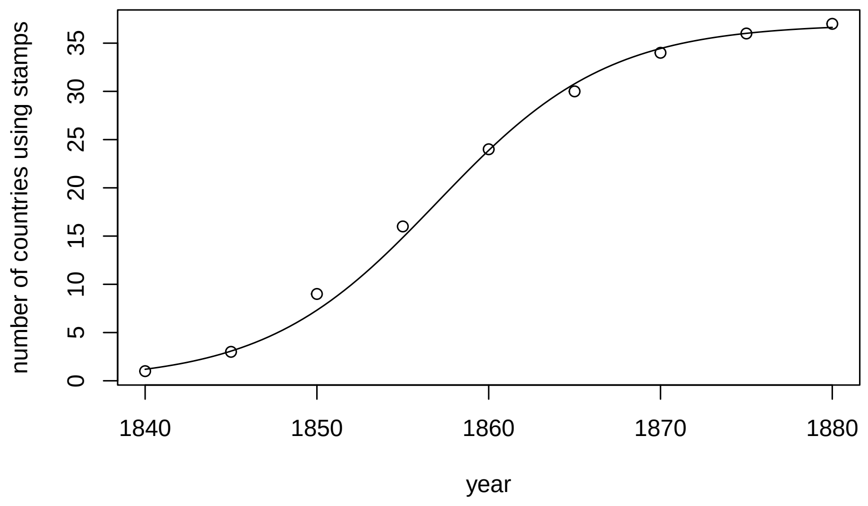


As we might expect, increasing the strength of selection increases the speed with which A goes to fixation. Note, though, that it retains the s-shape.

Summary

In Model 3 we saw how biased transmission causes a trait favoured by the selection bias to spread and go to fixation in a population, even when it is initially very rare. Biased transmission differs in its dynamics from biased mutation. Its action is proportional to the variation in the population at the time at which it acts. It is strongest when there is lots of variation (in our model, when there are equal numbers of A and B at $p = 0.5$), and weakest when there is little variation (when p is close to 0 or 1). This generates an s-shaped pattern of diffusion over time.

S-shaped diffusion curves like the ones we generated using Model 3 are ubiquitous in the real world. Rogers (2010) catalogued numerous examples of the s-shaped diffusion of novel technological and social innovations, from the spread of hybrid seed corn to the spread of new methods for teaching mathematics. Here is one example at the country level, concerning the spread of postage stamps in different European countries (data from Pemberton 1936):



Given that it is unlikely that 37 countries independently invented postage stamps over such a brief period, we can probably attribute the diffusion of postage stamps to a form of biased cultural transmission, as national postal services observed and copied the effective use of stamps in neighbouring countries. Henrich (2001) explicitly linked s-shaped diffusion curves to directly biased cultural transmission, rather than biased mutation, which as we saw in Model 2 generates r-shaped diffusion curves. Similarly, Newberry et al. (2017) provided evidence of s-shaped diffusion curves in the spread of novel grammatical forms, using them to distinguish biased transmission / cultural selection from unbiased transmission (see Model 1). However, we should also be cautious not to jump to conclusions. Many processes generate s-shaped diffusion curves, not just biased transmission, including sometimes purely individual-level biased mutation (Reader 2004; Hoppitt et al. 2010).

Exercises

1. Try different values of s to confirm that larger s increases the speed with which A goes to fixation.

2. Change s in **BiasedTransmission** to s_a, and add a new parameter s_b which specifies the probability of an agent copying trait B from a demonstrator who possesses that trait. Run the simulation to show that the

equilibrium value of p, and the speed at which this equilibrium is reached, depends on the difference between s_a and s_b. How do these dynamics differ from the mu_a and mu_b you implemented in Model 2 Q5?

Analytical Appendix

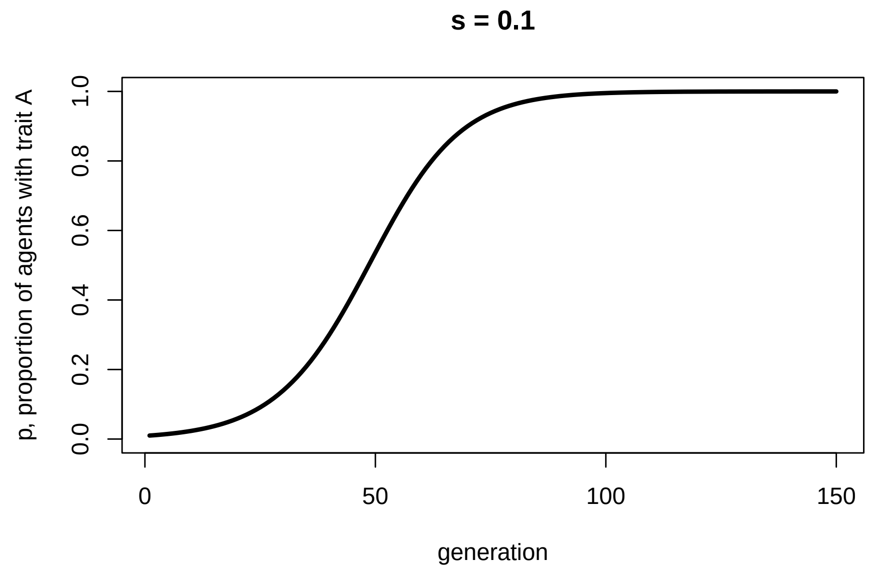
As before, we have p agents with trait A and $1 - p$ agents with trait B. The p agents with trait A keep their As, because A is favoured by biased transmission. The $1 - p$ agents with trait B pick another agent at random. If the random agent has B then nothing happens. However if the random agent has A, which they will with probability p, then with probability s they switch to that trait A. We can therefore write the recursion for p under biased transmission as:

$$p' = p + p(1 - p)s \tag{3.1}$$

The first term on the right-hand side is the unchanged A bearers, and the second term is the $1 - p$ B-bearers who find one of the p A-bearers and switch with probability s.

Here is some code to plot this biased transmission recursion:

```r
p <- rep(0, 150)
p[1] <- 0.01
s <- 0.1

for (i in 2:150) {
  p[i] <- p[i-1] + p[i-1]*(1-p[i-1])*s
}

plot(p,
     type = 'l',
     ylab = "p, proportion of agents with trait A",
     xlab = "generation",
     ylim = c(0,1),
     lwd = 3,
     main = paste("s = ", s, sep = ""))
```



The curve above should be identical to the simulation curve, given that the simulation had the same biased transmission strength s and a large enough N to minimise stochasticity.

From Equation 3.1 above, we can see how the strength of biased transmission depends on variation in the population, given that $p(1-p)$ is the formula for variance. This determines the shape of the curve, while s determines the speed with which the equilibrium p^* is reached.

But what is the equilibrium p^* here? In fact there are two. As before, the equilibrium can be found by setting the change in p to zero, or when:

$$p(1-p)s = 0 \tag{3.2}$$

There are three ways in which the left-hand side can equal zero: when $p = 0$, when $p = 1$ and when $s = 0$. The last case is uninteresting: it would mean that biased transmission is not occurring. The first two cases simply say that if either trait reaches fixation, then it will stay at fixation. This is to be expected, given that we have no mutation in our model. It contrasts with unbiased and biased mutation, where there is only one equilibrium value of p.

We can also say that $p = 0$ is an unstable equilibrium, meaning that any slight perturbation away from $p = 0$ moves p away from that value. This is essentially what we simulated above: a slight perturbation starting at $p = 0.01$ went all the way up to $p = 1$. In contrast, $p = 1$ is a stable equilibrium: any slight perturbation from $p = 1$ immediately goes back to $p = 1$.

References

Henrich, J. (2001). Cultural transmission and the diffusion of innovations: Adoption dynamics indicate that biased cultural transmission is the predominate force in behavioral change. American Anthropologist, 103(4), 992-1013.

Hoppitt, W., Kandler, A., Kendal, J. R., & Laland, K. N. (2010). The effect of task structure on diffusion dynamics: Implications for diffusion curve and network-based analyses. Learning & Behavior, 38(3), 243-251.

Newberry, M. G., Ahern, C. A., Clark, R., & Plotkin, J. B. (2017). Detecting evolutionary forces in language change. Nature, 551(7679), 223-226.

Pemberton, H. E. (1936). The curve of culture diffusion rate. American Sociological Review, 1(4), 547-556.

Reader, S. M. (2004). Distinguishing social and asocial learning using diffusion dynamics. Animal Learning & Behavior, 32(1), 90-104.

Rogers, E. M. (2010). Diffusion of innovations. Simon and Schuster.

Model 4: Biased transmission (indirect bias)

Introduction

In Model 3 we examined direct bias, where certain cultural traits are preferentially copied from a randomly chosen demonstrator. Here we will simulate *indirect bias* (Boyd & Richerson 1985). This is another form of biased transmission, or cultural selection, but where certain demonstrators are more likely to be copied than other demonstrators. For this reason, indirect bias is sometimes called 'demonstrator-based' or 'context' bias.

Indirect bias comes in several forms. Learners might preferentially copy demonstrators who have high success or payoffs (which may or may not derive from their cultural traits), demonstrators who are old (and perhaps have accrued valuable knowledge, or at least good enough to keep them alive to old age), demonstrators who are the same gender as the learner (if cultural traits are gender-specific), or demonstrators who possess high social status or prestige.

Model 4 presents a simple model of indirect bias, first showing how payoff-based indirect bias can look very similar to payoff-based direct bias (Model 4a), then exploring a more interesting case when payoff-based indirect bias allows a neutral trait to 'hitch-hike' along with a high-payoff functional trait when both are exhibited by high payoff demonstrators (Model 4b).

Model 4a: Payoff bias

In Model 4a we will simulate a case where new agents preferentially copy the cultural traits of agents from the previous generation who have higher relative payoffs. This is sometimes called 'payoff' or 'success' bias.

We will start with the skeleton of Model 3. As in Model 3, there are N agents, each of whom possess a single cultural trait, either A or B. The frequency of trait A is denoted p, and the initial frequency in the first generation is p_0. There are t_{max} timesteps and r_{max} independent runs.

In Model 3, agents were picked at random from the previous generation, and if that randomly-chosen agent possessed trait A then trait A was copied with probability s. For indirect bias, we need to change this. Demonstrator choice is no longer random: demonstrators are chosen non-randomly based on their payoffs. To implement this, we need to specify payoffs for each agent.

We will assume that an agent's payoff is determined solely by the agent's cultural trait. Agents with trait B have payoff of 1 (a 'baseline' payoff), while agents with trait A have payoff of $1 + s$. This means that trait A gives a payoff advantage to its bearers, relative to agents possessing trait B. The larger is s, the bigger this relative advantage.

Payoff-based indirect bias is then implemented by making the probability that an agent is chosen as a demonstrator proportional to that agents' relative payoff, i.e. its payoff relative to all other agents in the population. Once an agent is chosen, its trait is copied with no error and with probability 1.

Rather than going straight to a simulation function, let's explore this notion of relative payoff a bit further. First let's create a population of N agents, with traits determined by p_0, as usual.

```r
N <- 1000
p_0 <- 0.5
s <- 0.1

agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                    prob = c(p_0,1-p_0)))

head(agent)
```

```
##   trait
## 1     B
## 2     A
## 3     B
## 4     B
## 5     B
## 6     B
```

Given that $p_0 = 0.5$, you should see that the first few agents have a mix of trait A and trait B. Now let's add a payoff variable to the *agent* dataframe:

```r
agent$payoff[agent$trait == "A"] <- 1 + s
agent$payoff[agent$trait == "B"] <- 1

head(agent)
```

```
##   trait payoff
## 1     B    1.0
## 2     A    1.1
## 3     B    1.0
## 4     B    1.0
## 5     B    1.0
## 6     B    1.0
```

Agents with trait A are given fitness $1+s$, which in this case is 1.1, while agents with trait B are given fitness of 1.

The relative payoff of an agent can be calculated by dividing its payoff by the sum of all payoffs of all agents:

```
relative_payoffs <- agent$payoff / sum(agent$payoff)

head(relative_payoffs)
```

```
## [1] 0.0009507511 0.0010458262 0.0009507511 0.0009507511 0.0009507511
## [6] 0.0009507511
```

```
sum(relative_payoffs)
```

```
## [1] 1
```

These relative payoffs are obviously much smaller than the absolute payoffs because they have all been divided by N, which is a large number. They also all add up to 1. This is useful because in our simulation we can set the probability of picking an agent from whom to copy as equal to its relative fitness. Probabilities, like our relative payoffs, must sum to 1.

In previous models we implemented random copying by using the **sample** command to pick N previous-generation agents at random to copy. This worked because by default the **sample** command picks each item - in our case, each previous-generation agent - with equal probability. However, we can override this default by adding a **prob** argument. If we set **prob** to be our *relative_payoffs*, then each agent will be chosen in proportion to its relative payoff. The following code does this for one new generation, after putting *agent* into *previous_agent*.

```
previous_agent <- agent

agent$trait <- sample(previous_agent$trait, N, replace = TRUE, prob = relative_payoffs)

sum(previous_agent$trait == "A") / N
```

```
## [1] 0.518
```

```
sum(agent$trait == "A") / N
```

```
## [1] 0.527
```

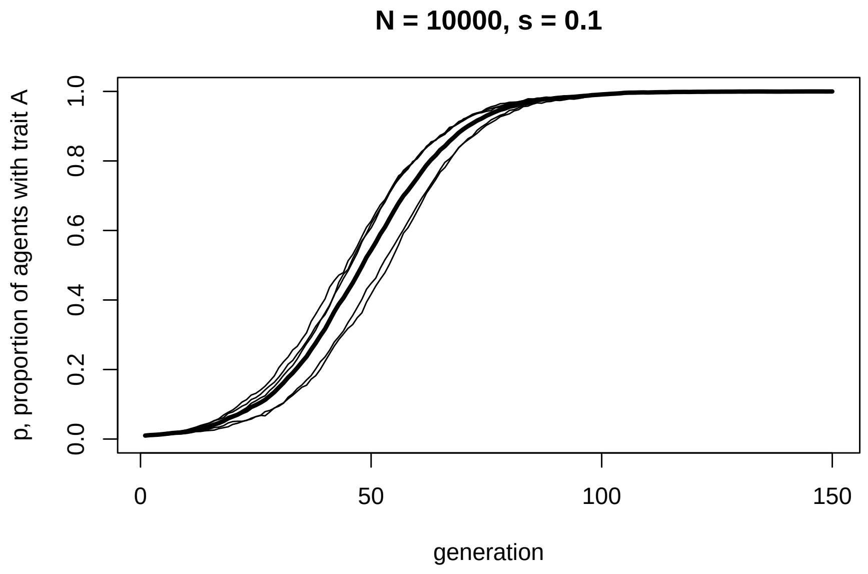
You should see that the frequency of trait *A* has increased from *previous_agent* to our new *agent* generation. This is what we would expect, given that there is a greater chance of selecting agents with the higher payoff trait *A*.

The following function incorporates the preceding code into our standard simulation model:

```
IndirectBias <- function (N, s, p_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add payoffs
    agent$payoff[agent$trait == "A"] <- 1 + s
    agent$payoff[agent$trait == "B"] <- 1

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get relative payoffs of previous agents
      relative_payoffs <- previous_agent$payoff / sum(previous_agent$payoff)

      # new traits copied from previous generation, biased by payoffs
      agent$trait <- sample(previous_agent$trait,
                            N, replace = TRUE,
                              prob = relative_payoffs)

      # add payoffs
      agent$payoff[agent$trait == "A"] <- 1 + s
      agent$payoff[agent$trait == "B"] <- 1

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", s = ", s, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

Now we can run **IndirectBias** with a small payoff advantage to agents with trait A:

```
data_model4a <- IndirectBias(N = 10000,
                             s = 0.1,
                             p_0 = 0.01,
                             t_max = 150,
                             r_max = 5)
```



This S-shaped curve is very similar to the one we generated in Model 3 for direct bias, with the same value of $s = 0.1$. This is not too surprising, given that in Model 4a an agent's payoff is entirely determined by their cultural trait. Under these assumptions, preferentially copying high payoff agents is functionally equivalent to preferentially copying high payoff traits. This is not always the case, however, as we will see in Model 4b.

Model 4b: Cultural hitch-hiking

A more interesting case of indirect bias occurs when individuals possess two cultural traits, one functional and the other neutral. Under certain circumstances, payoff based indirect bias can cause the neutral trait to 'hitch-hike' alongside the functional trait. Neutral traits can spread in the population simply because they are associated with high payoff traits in high payoff demonstrators, even though they have no effect on payoffs themselves.

The trait already incorporated into **IndirectBias** above is functional, in that one of its variants, A, has higher payoff than the alternative variant, B, when $s > 0$ (or vice versa when $s < 0$). A neutral trait has no effect on payoffs, and all variants are equally effective, much like the trait subject to unbiased transmission in Model 1.

In Model 4b we will add a neutral trait to **IndirectBias**. We keep the functional trait 1 that can be either A or B, with payoff advantage s to trait A. We add a second trait, trait 2, which can be either X or Y. We define q as the proportion of X in trait 2, with $1-q$ the proportion of Y. Whether an individual has trait X or Y has no effect on their payoff.

We will model a situation where the two traits may be initially linked. We are not going to be concerned here with why the two traits are initially linked. The link could, for example, have arisen by chance due to drift in historically small populations. We will leave this as an assumption of our model, which is fine as long as we are explicit about this. We define a parameter L that specifies the probability in the initial generation (at $t = 1$) that, if an individual has an A for trait 1, they also have an X for trait 2. With probability $1-L$, they pick X with probability q_0, analogously to how p_0 specifies the probability of picking an A for trait 1 in the first generation). In this model, q_0 will be fixed at 0.5, i.e. an equal chance of having X or Y.

The following code defines a new *agent* dataframe. What was previously *trait* now becomes *trait1*. A new *trait2* is initially set to *NA* for all agents. We then, with probability L, link *trait2* to *trait1*, otherwise set *trait2* at random.

```r
N <- 1000
p_0 <- 0.5
q_0 <- 0.5
L <- 1

agent <- data.frame(trait1 = sample(c("A","B"), N, replace = TRUE,
                                    prob = c(p_0,1-p_0)),
                    trait2 = rep(NA, N))

# with prob L, trait 2 is tied to trait 1, otherwise trait X with prob q_0
prob <- runif(N)
agent$trait2[agent$trait1 == "A" & prob < L] <- "X"
agent$trait2[agent$trait1 == "B" & prob < L] <- "Y"

agent$trait2[prob >= L] <- sample(c("X","Y"), sum(prob >= L), replace = TRUE,
                                  prob = c(q_0,1-q_0))

head(agent)
```

```
##   trait1 trait2
## 1      A      X
## 2      B      Y
## 3      A      X
## 4      A      X
## 5      A      X
## 6      A      X
```

When $L = 1$, there is maximum linkage between the two traits. All individuals with A also have X, and all individuals with B have Y. As L gets smaller, this linkage breaks down.

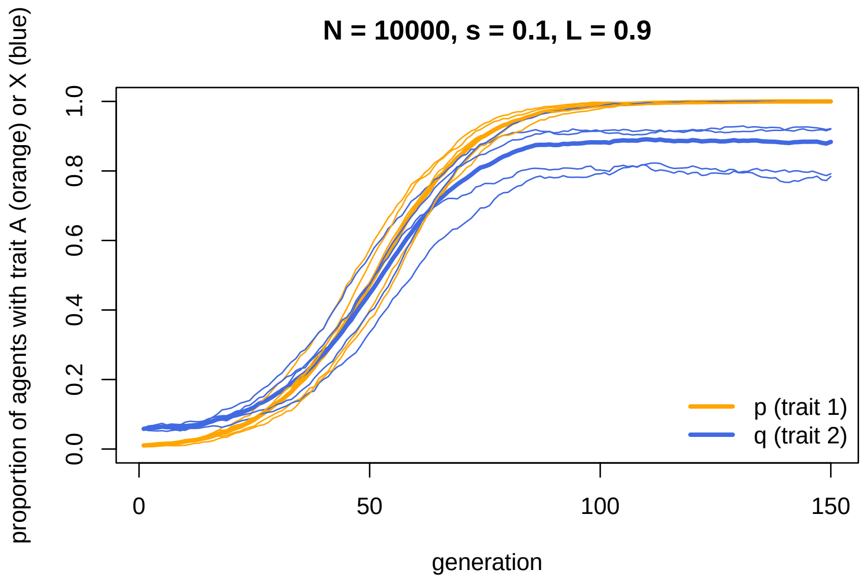
Now we can insert this code into the **IndirectBias** function from above, with a few other additions. Note that payoffs are calculated exactly as before, based solely on whether *trait1* is A or B. The *demonstrators* are picked as before, based on *relative_payoffs*, except now *trait2* is copied from the same demonstrator alongside *trait1*. Finally, we create an additional output dataframe for *trait2*, use this second output dataframe to record q, the frequency of X in trait 2, in each timestep, and use it to plot both p and q in different colours. Note the new **legend** command, which allows us to label the two lines in the plot, and the use of **list** to export two output dataframes, rather than one.

```
IndirectBias2 <- function (N, s, L, p_0, q_0, t_max, r_max) {

  # create matrices with t_max rows and r_max columns, fill with NAs, convert to dataf
  output_trait1 <- as.data.frame(matrix(NA, t_max, r_max))
  output_trait2 <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output_trait1) <- paste("run", 1:r_max, sep="")
  names(output_trait2) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait1 = sample(c("A","B"), N, replace = TRUE,
                                        prob = c(p_0,1-p_0)),
                        trait2 = rep(NA, N))

    # with prob L, trait 2 is tied to trait 1, otherwise trait X with prob q_0
    prob <- runif(N)
    agent$trait2[agent$trait1 == "A" & prob < L] <- "X"
    agent$trait2[agent$trait1 == "B" & prob < L] <- "Y"

    agent$trait2[prob >= L] <- sample(c("X","Y"), sum(prob >= L), replace = TRUE,
                                      prob = c(q_0,1-q_0))

    # add payoffs
    agent$payoff[agent$trait1 == "A"] <- 1 + s
    agent$payoff[agent$trait1 == "B"] <- 1

    # add first generation's p and q to first row of column r
    output_trait1[1,r] <- sum(agent$trait1 == "A") / N
    output_trait2[1,r] <- sum(agent$trait2 == "X") / N
    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get relative payoffs of previous agents
      relative_payoffs <- previous_agent$payoff / sum(previous_agent$payoff)

      # new traits copied from previous generation, biased by payoffs
      demonstrators <- sample(1:N,
                              N, replace = TRUE,
                              prob = relative_payoffs)

      agent$trait1 <- previous_agent$trait1[demonstrators]
      agent$trait2 <- previous_agent$trait2[demonstrators]

      # add payoffs
      agent$payoff[agent$trait1 == "A"] <- 1 + s
      agent$payoff[agent$trait1 == "B"] <- 1

      # put p and q into output slot for this generation t and run r
      output_trait1[t,r] <- sum(agent$trait1 == "A") / N
      output_trait2[t,r] <- sum(agent$trait2 == "X") / N

    }

  }

  # first plot a thick grey line for the mean p
  plot(rowMeans(output_trait1),
       type = 'l',
       ylab = "proportion of agents with trait A (orange) or X (blue)",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", s = ", s, ", L = ", L, sep = ""),
       col = "orange")

  # now thick orange line for mean q
  lines(rowMeans(output_trait2), col = "royalblue", lwd = 3)

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output_trait1[,r], type = 'l', col = "orange")
```

```
    lines(output_trait2[,r], type = 'l', col = "royalblue")

  }

  legend("bottomright",
         legend = c("p (trait 1)", "q (trait 2)"),
         lty = 1,
         lwd = 3,
         col = c("orange", "royalblue"),
         bty = "n")

  list(output_trait1, output_trait2)  # export data from function

  }
```

Now we can run **IndirectBias2** with a high linkage between trait 1 and trait 2, $L = 0.9$

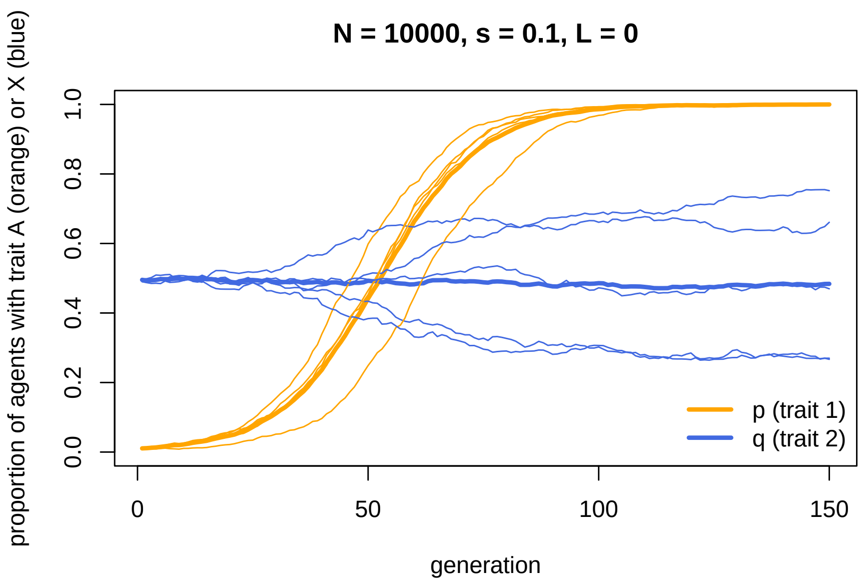
```
data_model4b <- IndirectBias2(N = 10000,
                              s = 0.1,
                              L = 0.9,
                              p_0 = 0.01,
                              q_0 = 0.5,
                              t_max = 150,
                              r_max = 5)
```



As you might expect, trait A (the orange line) goes to fixation exactly as for **IndirectBias** above. However, trait X (the blue line) shows a similar increase - not quite all the way to fixation, because $L < 1$, but very nearly. Hence the neutral trait X is hitch-hiking along with the functional trait A due to payoff based indirect bias. Trait X has no effect on payoffs, but because it happens to be associated with trait A at the start, it spreads through the population.

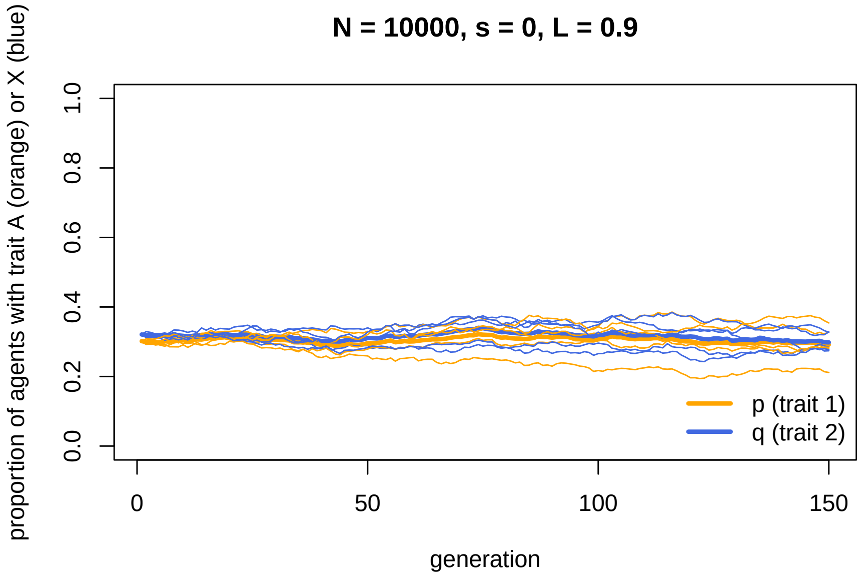
If we remove the linkage by setting $L = 0$, we see that this hitch-hiking no longer occurs. Instead trait X drifts randomly as we would expect for a neutral trait:

```
data_model4b <- IndirectBias2(N = 10000,
                              s = 0.1,
                              L = 0.0,
                              p_0 = 0.01,
                              q_0 = 0.5,
                              t_max = 150,
                              r_max = 5)
```



And when we remove the payoff biased indirect bias by setting $s = 0$, and reintroduce the linkage ($L = 0.9$), neither trait increases in frequency. Both are now effectively neutral.

```
data_model4b <- IndirectBias2(N = 10000,
                              s = 0.0,
                              L = 0.9,
                              p_0 = 0.3,
                              q_0 = 0.5,
                              t_max = 150,
                              r_max = 5)
```



Summary

Indirect bias occurs when individuals preferentially learn from demonstrators who have particular characteristics, such as being wealthy or socially successful (having high 'payoffs' in the language of modelling), being old, having a particular gender, or being prestigious. There is extensive evidence for these indirect transmission biases from the broad social and behavioural sciences (e.g. Rogers 2010; Labov 1972; Bandura, Ross & Ross 1963) and cultural evolution research specifically (e.g. Mesoudi 2011; Henrich & Gil-White 2001; Brand et al. 2020; Wood et al. 2012; McElreath et al. 2008; Henrich & Henrich 2010). Consequently, it is worth modelling indirect bias to understand its dynamics, and how it is similar or different to other forms of biased transmission.

Our first simple model of indirect bias, Model 4a, showed that when individuals preferentially learn from high payoff demonstrators, and demonstrator payoff is determined directly by the demonstrator's cultural trait, then the trait that gives demonstrators higher relative payoff spreads through the population. This is entirely expected and intuitive. The resulting S-shaped diffusion curve for Model 4a looks extremely similar to that generated in Model 3 for direct bias with the same selection parameter s. This is to be expected, given our assumptions that demonstrator payoff is determined solely by trait payoff. So in this case direct bias and indirect bias are functionally equivalent.

This should strike a note of caution. If we see the same cultural dynamics generated from different underlying models, then we cannot be certain which of these underlying processes generated similar real-world cultural dynamics. Nevertheless, it's better to know this, than to incorrectly attribute a dynamic to the wrong process.

Model 4b examined the more interesting case where there are two cultural traits, one functional and one neutral. If these traits are initially linked, via our parameter L, then the neutral trait can hitch-hike along with the functional trait. Everyday examples of cultural hitch-hiking might be where people copy fashion styles from wealthy and talented sportstars: a footballer's tattoos or hairstyle is unlikely to have anything to do with their sporting success, but these neutral traits may nevertheless be copied if people generally copy all traits associated with successful demonstrators. There is experimental evidence for cultural hitch-hiking (Mesoudi & O'Brien 2008), while Yeh et al. (2019) have created more elaborate models of cultural hitch-hiking with more than two traits and copyable links between those traits.

While we did not model the evolution of indirect bias itself, we might imagine that this global copying of successful demonstrators might be less costly to the learner when demonstrators exhibit multiple traits than direct bias, where learners would have to identify which of the multiple traits actually contribute to the demonstrator's success. The cost of indirect bias is potentially copying neutral traits, as we saw in Model 4b, or even maladaptive traits, alongside the functional traits. But in some circumstances this cost may be less than the aforementioned cost of direct bias. If it is, then this may be an explanation for the presence of neutral or maladaptive traits in human culture.

Boyd & Richerson (1985) modelled another interesting case of indirect bias, where the criterion of demonstrator success is copied alongside the actual cultural trait. For example, one might copy not only the specific tattoo of a successful footballer, but also copy their preference for tattoos. Under certain conditions this can lead to the runaway selection of both traits and preferences, with both getting more and more extreme over time. In our example this would lead to more and more extensive tattoos (which perhaps has happened in Micronesia, but for yam-growing rather than football: Boyd & Richerson 1985). This is roughly analogous to runaway sexual selection in genetic evolution that can lead to elaborate traits such as peacock's tails, which may occur because both elaborate tails and preferences for elaborate tails are genetically inherited together. Both cultural hitch-hiking and runaway selection are interesting dynamics of indirectly bias which are not observed for direct bias.

In terms of programming innovations, we saw in Model 4a how to assign payoffs to different agents on the basis of their cultural trait, and then use the *relative_payoffs* in the **sample** command to implement payoff-based selection of demonstrators from whom to learn. In Model 4b we introduced a second trait, keeping track of both the frequency of A in trait 1 (p) and the frequency of X in trait 2 (q). We also saw how to link these traits together using the L parameter. Finally, we saw how to draw two results lines on the same plot in different colours, include a legend for those lines, and export multiple dataframes from the simulation function as a list.

Exercises

1. Try different values of s in **IndirectBias** to confirm that larger s increases the speed with which A goes to fixation.

2. Try different values of L in **IndirectBias2** to confirm that as L gets smaller, the hitch-hiking effect gets weaker.

3. In **IndirectBias2** the hitch-hiking trait is neutral. But can a detrimental trait also hitch-hike? Modify the function to make the second trait functional, with a payoff disadvantage to X over Y (e.g. by introducing an equivalent selection parameter to s which affects X relative to Y, and making this new parameter negative). Set the initial linkage conditions such that the high payoff *trait1* variant A is associated with the low payoff *trait2* variant X. Are there any conditions under which the detrimental, low-payoff trait X hitch-hikes on the beneficial, high-payoff trait A?

Analytical Appendix

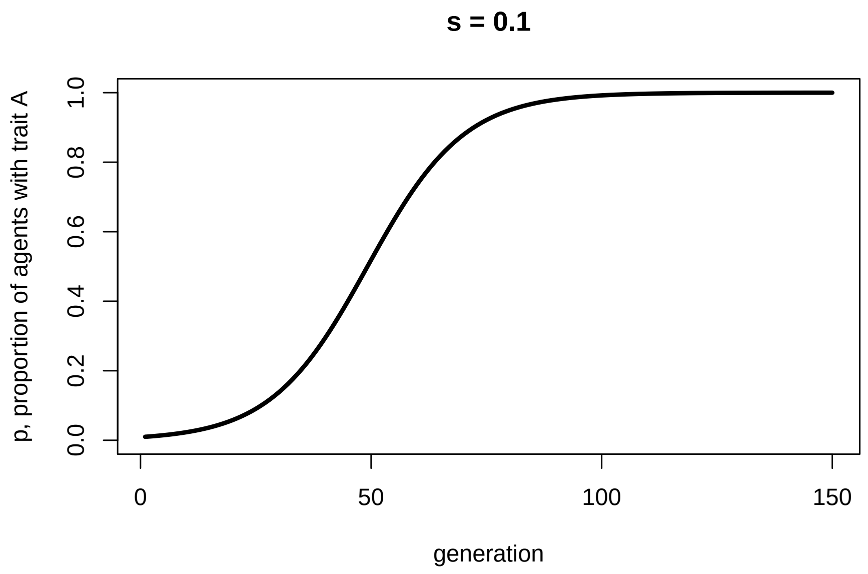
In Model 4a, the frequency of trait A in the next generation, p', is simply the relative payoff of all agents with trait A in the previous generation. This is given by:

$$p' = \frac{p(1+s)}{p(1+s)+(1-p)} = \frac{p(1+s)}{1+sp} \tag{4.1}$$

where $p(1+s)$ is the payoff of all agents with trait A, and $(1-p)$ is the payoff of all agents with trait B.

Plotting this recursion gives a curve that matches the one generated above by the simulation model:

```
p <- rep(0, 150)
p[1] <- 0.01
s <- 0.1

for (i in 2:150) {
  p[i] <- (p[i-1])*(1+s) / (1 + (p[i-1])*s)
}

plot(p,
     type = 'l',
     ylab = "p, proportion of agents with trait A",
     xlab = "generation",
     ylim = c(0,1),
     lwd = 3,
     main = paste("s = ", s, sep = ""))
```



To find the equilibria, we can set $p' = p$ in Equation 4.1, and rearrange to give:

$$p(1-p)s = 0 \tag{4.2}$$

This is identical to Equation 3.2, the equivalent for direct bias. Again, there are three equilibria, one when $p = 0$, one when $s = 0$, and one when $1 - p = 0$, or when $p = 1$. While Equation 4.1 is not the same as Equation 3.1, the resulting equilibria are identical.

References

Bandura, A., Ross, D., & Ross, S. A. (1963). A comparative test of the status envy, social power, and secondary reinforcement theories of identificatory learning. The Journal of Abnormal and Social Psychology, 67(6), 527.

Boyd, R., & Richerson, P. J. (1985). Culture and the evolutionary process. University of Chicago Press.

Brand, C. O., Heap, S., Morgan, T. J. H., & Mesoudi, A. (2020). The emergence and adaptive use of prestige in an online social learning task. Scientific Reports, 10(1), 1-11.

Henrich, J., & Gil-White, F. J. (2001). The evolution of prestige: Freely conferred deference as a mechanism for enhancing the benefits of cultural transmission. Evolution and Human Behavior, 22(3), 165-196.

Henrich, J., & Henrich, N. (2010). The evolution of cultural adaptations: Fijian food taboos protect against dangerous marine toxins. Proceedings of the Royal Society B: Biological Sciences, 277(1701), 3715-3724.

Labov, W. (1972). Sociolinguistic patterns. University of Pennsylvania press.

McElreath, R., Bell, A. V., Efferson, C., Lubell, M., Richerson, P. J., & Waring, T. (2008). Beyond existence and aiming outside the laboratory: estimating frequency-dependent and pay-off-biased social learning strategies. Philosophical Transactions of the Royal Society B: Biological Sciences, 363(1509), 3515-3528.

Mesoudi, A. (2011). An experimental comparison of human social learning strategies: payoff-biased social learning is adaptive but underused. Evolution and Human Behavior, 32(5), 334-342.

Mesoudi, A., & O'Brien, M. J. (2008). The cultural transmission of Great Basin projectile-point technology I: an experimental simulation. American Antiquity, 73(1), 3-28.

Rogers, E. M. (2010). Diffusion of innovations. Simon and Schuster.

Wood, L. A., Kendal, R. L., & Flynn, E. G. (2012). Context-dependent model-based biases in cultural transmission: Children's imitation is affected by model age over model knowledge state. Evolution and Human Behavior, 33(4), 387-394.

Yeh, D. J., Fogarty, L., & Kandler, A. (2019). Cultural linkage: the influence of package transmission on cultural dynamics. Proceedings of the Royal Society B, 286(1916), 20191951.

Model 5: Biased transmission (conformist bias)

Introduction

Model 3 looked at the case where one cultural trait is intrinsically more likely to be copied than another trait, and in Model 4 the case where one type of demonstrator is more likely to be copied than another. Here we will look at a third kind of biased transmission: conformity (or 'positive frequency dependent bias'). Here, individuals are disproportionately more likely to adopt the most common trait in the population, irrespective of its intrinsic characteristics or who bears it.

For example, imagine trait A has a frequency of 0.7 in the population, with the rest possessing trait B. An unbiased learner would adopt trait A with a probability exactly equal to 0.7. This is unbiased transmission, and is what happens in Model 1: by picking a member of the previous generation at random, the probability of adoption in Model 1 is equal to the frequency of that trait amongst the previous generation.

A conformist learner, on the other hand, would adopt trait A with a probability greater than 0.7. In other words, common traits get an 'adoption boost' relative to unbiased transmission. Uncommon traits get an equivalent 'adoption penalty'. The magnitude of this boost or penalty can be controlled by a parameter, which we will call D.

Let's keep things simple in our model. Rather than assuming that individuals sample across the entire population, which in any case might be implausible in large populations, let's assume they pick only three demonstrators at random. Why three? This is the minimum number of demonstrators that can yield a majority (i.e. 2 vs 1), which we need in order to implement conformity. When two demonstrators have one trait and the other demonstrator has a different trait, we want to boost the probability of adoption for the majority trait, and reduce it for the minority trait.

Following Boyd and Richerson (1985), we can specify the probability of adoption as in the following table:

Demonstrator 1	Demonstrator 2	Demonstrator 3	Probability of adopting trait A
A	A	A	1
A	A	B	$2/3 + D/3$
A	B	A	$2/3 + D/3$
B	A	A	$2/3 + D/3$
A	B	B	$1/3 - D/3$
B	A	B	$1/3 - D/3$
B	B	A	$1/3 - D/3$
B	B	B	0

The first row says that when all demonstrators have trait A, then trait A is definitely adopted. Similarly, the bottom row says that when all demonstrators have trait B, then trait A is never adopted, and by implication trait B is always adopted.

For the three combinations where there are two As and one B, the probability of adopting trait A is $2/3$, which it would be under unbiased transmission (because two out of three demonstrators have A), plus the conformist adoption boost specified by D. D is divided by three so that it varies from 0 to 1.

Similarly, for the three combinations where there are two Bs and one A, the probability of adopting A is $1/3$ minus the conformist adoption penalty specified by D.

Model 5

Let's implement these assumptions in the kind of agent-based model we've been building so far. As before, assume N agents each of whom possess one of two traits A or B. The frequency of A is denoted by p. The initial frequency of A in generation $t = 1$ is p_0. Rather than going straight to a function, let's go step by step.

First we'll specify our parameters, N and p_0 as before, plus the new conformity parameter D. We can also create an *agent* dataframe and fill it with As and Bs in the proportion specified by p_0, again exactly as before. To remind ourselves what *agent* looks like, we use the **head** command.

```
N <- 100
p_0 <- 0.5
D <- 1

# create first generation
agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                    prob = c(p_0,1-p_0)))

head(agent)
```

```
##   trait
## 1     A
## 2     A
## 3     A
## 4     B
## 5     B
## 6     A
```

Now we'll create a dataframe called *demonstrators* that picks, for each new agent in the next generation, three demonstrators at random from the current population of agents. It therefore needs three columns/variables, one for each of the demonstrators, and N rows, one for each new agent. We fill each column with randomly chosen traits from the *agent* dataframe. We can view this with **head**.

```
# create dataframe with a set of 3 randomly-picked demonstrators for each agent
demonstrators <- data.frame(dem1 = sample(agent$trait, N, replace = TRUE),
                            dem2 = sample(agent$trait, N, replace = TRUE),
                            dem3 = sample(agent$trait, N, replace = TRUE))

head(demonstrators)
```

```
##   dem1 dem2 dem3
## 1    A    A    A
## 2    A    A    A
## 3    B    A    B
## 4    A    A    B
## 5    B    A    A
## 6    A    A    B
```

Think of each row here as containing the traits of three randomly-chosen demonstrators chosen by each new next-generation agent. Now we want to calculate the probability of adoption of A for each of these three-trait demonstrator combinations.

First we need to get the number of As in each combination. Then we can replace the traits in *agent* based on the probabilities in the table above. When all demonstrators have A, we set to A. When no demonstrators have A, we set to B. When two out of three demonstrators have A, we set to A with probability $2/3 + D/3$ and B otherwise. When one out of three demonstrators have A, we set to A with probability $1/3 - D/3$ and B otherwise.

To check it works, we can add the new *agent* dataframe as a column to *demonstrators* and view the latter with **head**. This will let us see the three demonstrators and the resulting new trait side by side.

```r
# get the number of As in each 3-dem combo
numAs <- rowSums(demonstrators == "A")

agent$trait[numAs == 3] <- "A"  # for dem combos with all As, set to A
agent$trait[numAs == 0] <- "B"  # for dem combos with no As, set to B

prob <- runif(N)

# when A is a majority, 2/3
agent$trait[numAs == 2 & prob < (2/3 + D/3)] <- "A"
agent$trait[numAs == 2 & prob >= (2/3 + D/3)] <- "B"

# when A is a minority, 1/3
agent$trait[numAs == 1 & prob < (1/3 - D/3)] <- "A"
agent$trait[numAs == 1 & prob >= (1/3 - D/3)] <- "B"

# for testing only, add the new traits to the demonstrator dataframe and show it
demonstrators$newtrait <- agent$trait
head(demonstrators, 20)
```

```
##    dem1 dem2 dem3 newtrait
## 1     A    A    A        A
## 2     A    A    A        A
## 3     B    A    B        B
## 4     A    A    B        A
## 5     B    A    A        A
## 6     A    A    B        A
## 7     A    A    B        A
## 8     A    A    B        A
## 9     B    B    A        B
## 10    A    B    B        B
## 11    B    A    A        A
## 12    A    A    B         A
## 13    A    A    A         A
## 14    A    B    B         B
## 15    A    A    B         A
## 16    B    B    B         B
## 17    A    B    A         A
## 18    A    B    A         A
## 19    B    A    A         A
## 20    A    A    A         A
```

Because we set $D = 1$ above, we should see above that the new trait is always the majority trait amongst the three demonstrators. This is perfect conformity. We can weaken conformity by reducing D, in the code below.

```r
N <- 100
p_0 <- 0.5
D <- 0.1

# create first generation
agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                    prob = c(p_0,1-p_0)))

# create dataframe with a set of 3 randomly-picked demonstrators for each agent
demonstrators <- data.frame(dem1 = sample(agent$trait, N, replace = TRUE),
                            dem2 = sample(agent$trait, N, replace = TRUE),
                            dem3 = sample(agent$trait, N, replace = TRUE))

# get the number of As in each 3-dem combo
numAs <- rowSums(demonstrators == "A")

agent$trait[numAs == 3] <- "A"  # for dem combos with all As, set to A
agent$trait[numAs == 0] <- "B"  # for dem combos with no As, set to B

prob <- runif(N)

# when A is a majority, 2/3
agent$trait[numAs == 2 & prob < (2/3 + D/3)] <- "A"
agent$trait[numAs == 2 & prob >= (2/3 + D/3)] <- "B"

# when A is a minority, 1/3
agent$trait[numAs == 1 & prob < (1/3 - D/3)] <- "A"
agent$trait[numAs == 1 & prob >= (1/3 - D/3)] <- "B"

# for testing only, add the new traits to the demonstrator dataframe and show it
demonstrators$newtrait <- agent$trait
head(demonstrators, 20)
```

```
##    dem1 dem2 dem3 newtrait
## 1     B    A    B        A
## 2     A    B    A        A
## 3     A    B    A        A
## 4     B    A    B        A
## 5     A    A    B        A
## 6     A    B    B        A
## 7     B    A    B        B
## 8     B    B    B        B
## 9     B    B    A        A
## 10    A    A    B        A
## 11    A    B    B        B
## 12    A    A    B        B
## 13    A    B    B        B
## 14    A    B    A        A
## 15    A    A    B        A
## 16    B    B    A        A
## 17    B    A    A        A
## 18    A    B    A        A
## 19    B    A    A        B
## 20    A    A    B        B
```

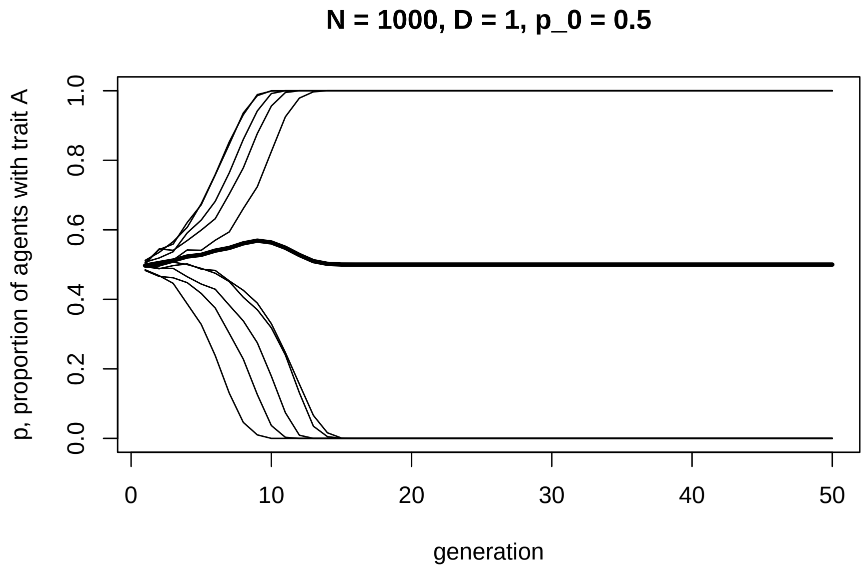
Now that conformity is weaker, sometimes the new trait is not the majority amongst the three demonstrators. With the small sample shown above, it's perhaps not possible to notice it. Hopefully when we put it all together now into a function and run it over multiple generations, we will notice an effect. The code below is a combination of Model 1 (unbiased transmission) and the code above for conformity.

```
ConformistTransmission <- function (N, p_0, D, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA,t_max,r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N
    for (t in 2:t_max) {

      # create dataframe with a set of 3 randomly-picked demonstrators for each agent
      demonstrators <- data.frame(dem1 = sample(agent$trait, N, replace = TRUE),
                                  dem2 = sample(agent$trait, N, replace = TRUE),
                                  dem3 = sample(agent$trait, N, replace = TRUE))

      # get the number of As in each 3-dem combo
      numAs <- rowSums(demonstrators == "A")

      agent$trait[numAs == 3] <- "A"  # for dem combos with all As, set to A
      agent$trait[numAs == 0] <- "B"  # for dem combos with no As, set to B

      prob <- runif(N)

      # when A is a majority, 2/3
      agent$trait[numAs == 2 & prob < (2/3 + D/3)] <- "A"
      agent$trait[numAs == 2 & prob >= (2/3 + D/3)] <- "B"

      # when A is a minority, 1/3
      agent$trait[numAs == 1 & prob < (1/3 - D/3)] <- "A"
      agent$trait[numAs == 1 & prob >= (1/3 - D/3)] <- "B"

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", D = ", D, ", p_0 = ", p_0, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }
```

```
  output  # export data from function
}
```

Note that we omit the testing code above (we've tested it and it works!), and there's no need to put *agent* into *previous_agent* because we have the *demonstrator* dataframe doing that job. Let's run the function.

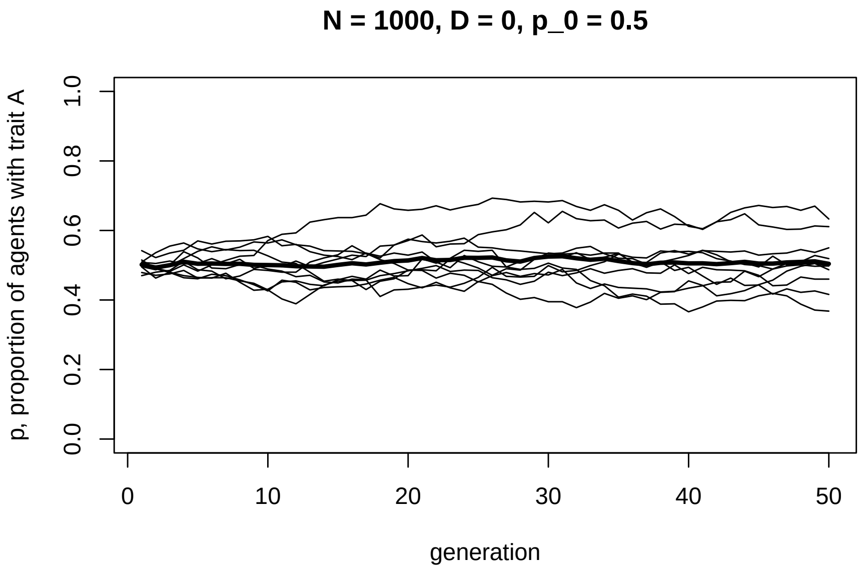
```
data_model5 <- ConformistTransmission(N = 1000, p_0 = 0.5, D = 1, t_max = 50, r_max = 
```



Here we should see some lines going to $p = 1$, and some lines going to $p = 0$. Conformity acts to favour the majority trait. This will depend on the initial frequency of A in the population. In different runs with $p_0 = 0.5$, sometimes there will be slightly more As, sometimes slightly more Bs (remember, in our model this is probabilistic, like flipping coins, so initial frequencies will rarely be precisely 0.5).

Let's compare conformity to unbiased transmission, by setting $D = 0$.

```
data_model5 <- ConformistTransmission(N = 1000, p_0 = 0.5, D = 0, t_max = 50, r_max = 
```

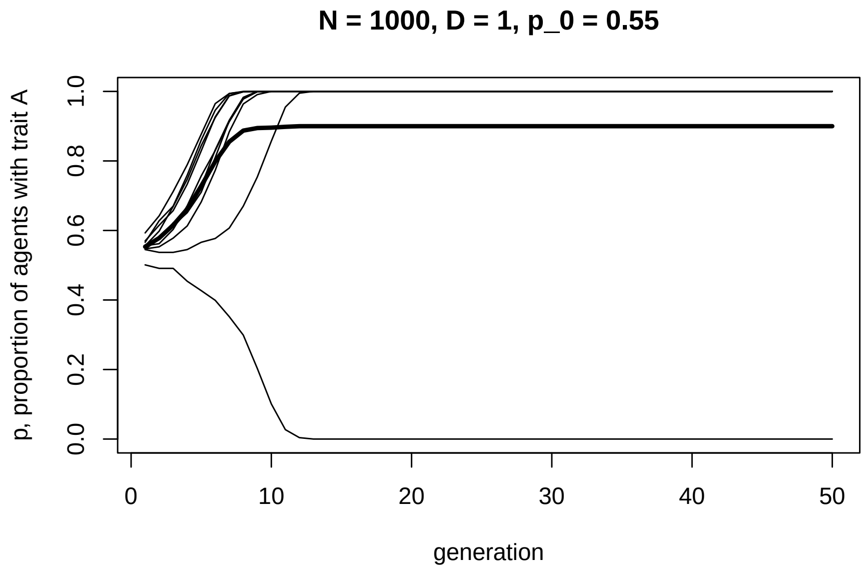


As in Model 1 with a sufficiently large N, we should see frequencies fluctuating around $p = 0.5$. This underlines the effect of conformity: it drives traits to fixation as they become more and more common.

As an aside, note that the last two graphs have roughly the same thick black mean frequency line, which hovers around $p = 0.5$. This highlights the dangers of looking at means alone. If we hadn't plotted the individual runs and relied solely on mean frequencies, we might think that $D = 0$ and $D = 1$ gave identical results. But in fact, they are very different. Always look at the underlying distribution that generates means.

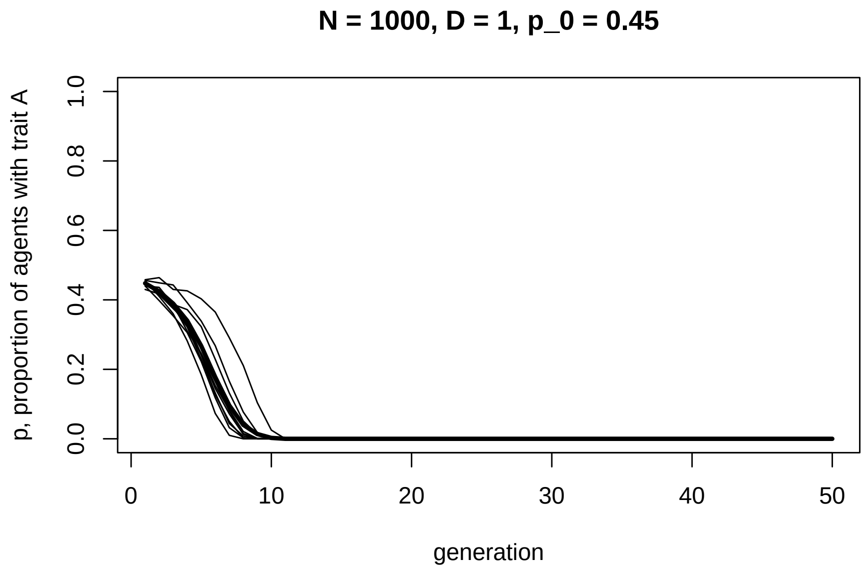
Now let's explore the effect of changing the initial frequencies by changing p_0, and adding conformity back in.

```
data_model5 <- ConformistTransmission(N = 1000, p_0 = 0.55, D = 1, t_max = 50, r_max = 10)
```



When A starts off in a slight majority ($p_0 = 0.55$), most if not all of the runs should result in A going to fixation. Now let's try the reverse.

```
data_model5 <- ConformistTransmission(N = 1000, p_0 = 0.45, D = 1, t_max = 50, r_max =
```



When A starts off in a minority ($p_0 = 0.45$), most if not all runs should result in A disappearing. These last two graphs show how initial conditions affect conformity. Whichever trait is more common is favoured by conformist transmission.

Summary

Model 5 explored conformist biased cultural transmission, or 'conformity' for short. This is where individuals are disproportionately more likely to adopt the most common trait among a set of demonstrators. We can contrast this with the direct or content biased transmission from Model 3, where one trait is intrinsically more likely to be copied. With conformity, the traits have no intrinsic attractiveness and are preferentially copied simply because they are common.

We saw how conformity increases the frequency of whichever trait is more common. Initial trait frequencies are important here: traits that are initially more common typically go to fixation. This in turn makes stochasticity important, which in small populations can affect initial frequencies.

Experimental studies have shown that people exhibit conformity as defined and modelled here (Efferson et al. 2008; Muthukrishna et al. 2016; Deffner et al. 2020), and models have extended Boyd & Richerson's (1985) initial treatment to consider more than two traits, more than three demonstrators, and the effects of spatial and temporal environmental variation (Henrich & Boyd 1998; Nakahashi et al. 2012; Mesoudi 2018). Conformity is thought to have important implications for real-world patterns of cultural evolution by affecting the spread of novel innovations through societies (Henrich 2001), and by acting to maintain between-group cultural variation in the face of migration, as we will explore further in a later model.

The major programming innovation in Model 5 was the use of an intermediate dataframe to hold the demonstrators. We then created the next generation using a table of probabilities, which specified for each combination of demonstrators the probability of adopting each trait.

Exercises

1. Try different values of D and p_0 to confirm that conformity acts to always favour the majority trait. Also try smaller values of N. How does the stochasticity at small values of N affect conformity?

2. The conformity parameter D can also be negative, which reduces the probability of adopting majority traits and increases the probability of adopting minority traits. This is *anti-conformity*, or *negative frequency-dependent cultural transmission*. Explore the effect on cultural dynamics of varying D between -1 and 0, for different values of p_0.

3. Create a new function **ConformityPlusBiasedTransmission**, using code from Model 3. First, agents should engage in directly biased transmission from the previous generation, according to parameter s. Then they should engage in conformity, with the demonstrators for conformity being the set of traits after biased transmission. Vary s (which favours trait A) and D (which favours the majority) starting at small values of p_0 to explore when selection can overpower conformity, and vice versa. See Henrich (2001) for a similar model of conformity plus directly biased transmission.

Analytic Appendix

An alternative way of doing all the above is with deterministic recursions, as Boyd & Richerson (1985) originally did.

Let's revise our table above to add the probabilities of each combination of three demonstrators coming together, assuming they are picked at random. These probabilities can be expressed in terms of p, the frequency of A, and $(1-p)$, the frequency of B.

Dem 1	Dem 2	Dem 3	Prob of adopting A	Prob of combination forming
A	A	A	1	p^3
A	A	B	$2/3 + D/3$	$p^2(1-p)$
A	B	A	$2/3 + D/3$	$p^2(1-p)$
B	A	A	$2/3 + D/3$	$p^2(1-p)$
A	B	B	$1/3 - D/3$	$p(1-p)^2$
B	A	B	$1/3 - D/3$	$p(1-p)^2$
B	B	A	$1/3 - D/3$	$p(1-p)^2$
B	B	B	0	$(1-p)^3$

To get the frequency of A in the next generation, p', we multiply, for each of the eight rows in the table, the probability of adopting A by the probability of that combination forming (i.e. the final two columns in the table), and add up all of these eight products. After rearranging, this gives the following recursion:

$$p' = p + Dp(1-p)(2p-1) \tag{5.1}$$

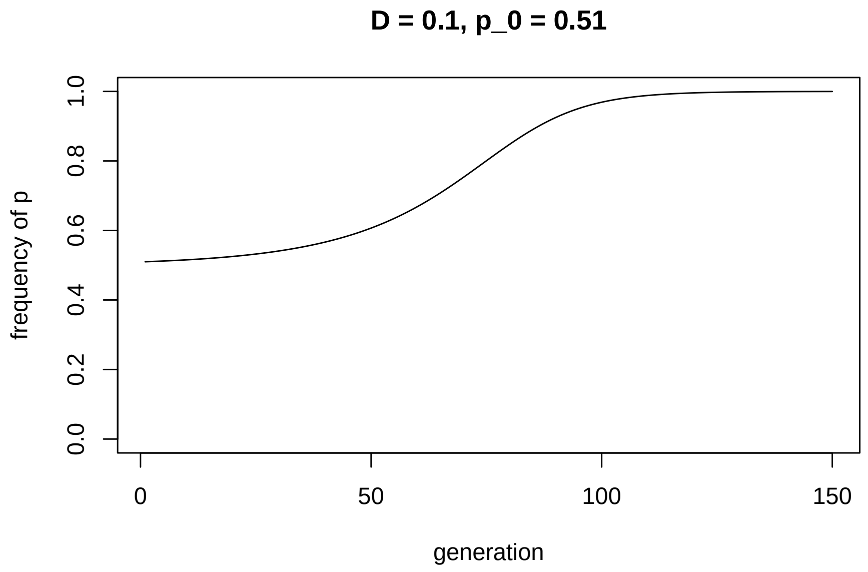
Now we can create a function for this recursion:

```
ConformityRecursion <- function(D, t_max, p_0) {

  p <- rep(0,t_max)
  p[1] <- p_0

  for (i in 2:t_max) {
    p[i] <- p[i-1] + D*p[i-1]*(1-p[i-1])*(2*p[i-1] - 1)
  }

  plot(p,
       type = "l",
       ylim = c(0,1),
       ylab = "frequency of p",
       xlab = "generation",
       main = paste("D = ", D, ", p_0 = ", p_0, sep = ""))

}
```

Here, we use a **for** loop to cycle through each generation, each time updating p according to the recursion equation above. Remember, there is no N here because the recursion is deterministic and assumes an infinite population size; hence there is no stochasticity due to finite population sizes. There is also no need to have multiple runs as each run is identical, hence no r_{max}.

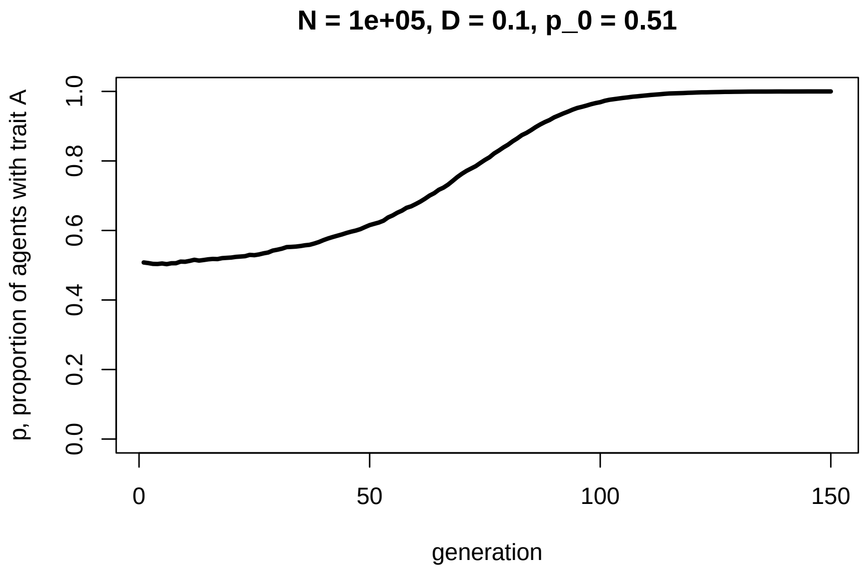
The following code runs the **ConformityRecursion** function with weak conformity ($D = 0.1$) and slightly more A in the initial generation ($p_0 = 0.51$).

```
ConformityRecursion(D = 0.1, t_max = 150, p_0 = 0.51)
```



As in the agent-based model, the initially most-frequent trait, here A, goes to fixation. Let's compare to the agent-based model with the same parameters, and a large enough N to make stochasticity unimportant.

```
data_model5 <- ConformistTransmission(N = 100000, p_0 = 0.51, D = 0.1, t_max = 150, r_
```



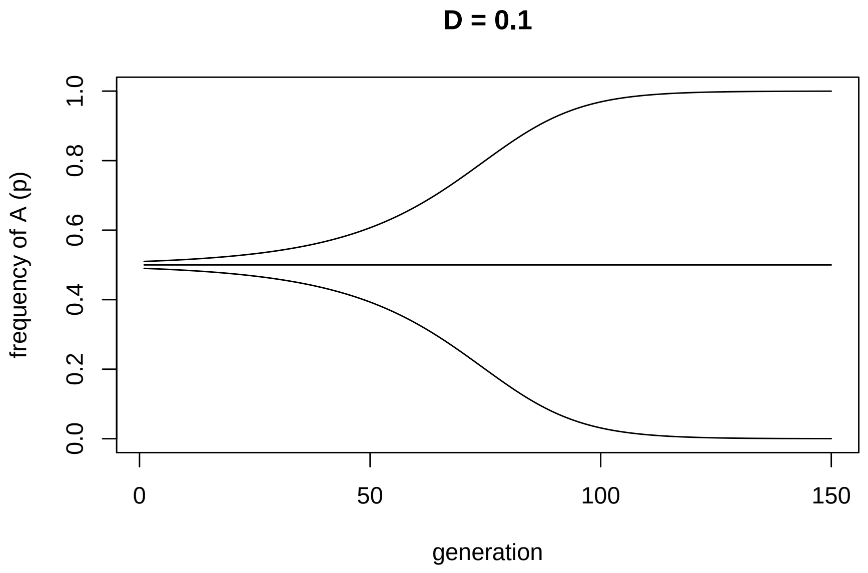
It should be a pretty good match. Try playing around with smaller N to show that stochastic agent-based models are most likely to match deterministic recursion models when N is large.

Let's modify the **ConformityRecursion** function to accept multiple values of p_0, so we can plot different starting frequencies on the same graph.

```
ConformityRecursion <- function(D, t_max, p_0) {

  numSims <- length(p_0)

  p <- as.data.frame(matrix(NA, nrow = t_max, ncol = numSims))
  p[1,] <- p_0

  for (i in 2:t_max) {
    p[i,] <- p[i-1,] + D*p[i-1,]*(1-p[i-1,])*(2*p[i-1,] - 1)
  }

  plot(p[,1],
       type = "l",
       ylim = c(0,1),
       ylab = "frequency of A (p)",
       xlab = "generation",
       main = paste("D =", D))

  if (numSims > 1) {
    for (i in 2:numSims) {
      lines(p[,i], type = 'l')
    }
  }

}
```

The following command plots three different values of p_0, one less than 0.5, one equal to 0.5, and one greater than 0.5. This should confirm that conformity favours whichever trait is initially most frequent.

```
ConformityRecursion(D = 0.1, t_max = 150, p_0 = c(0.49,0.5,0.51))
```



Again, this matches the simulations above where some runs are randomly initially above 0.5 and others below 0.5.

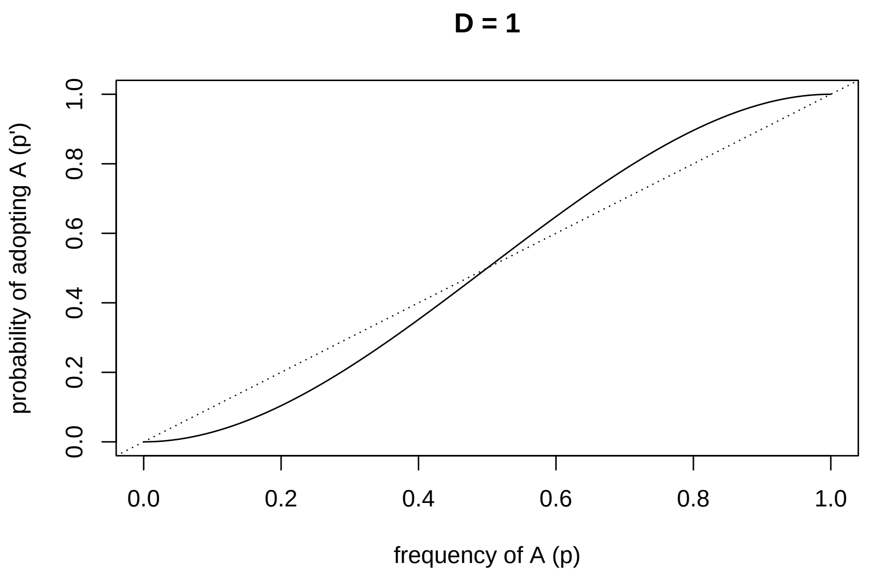
This result also matches the equilibria that emerge from Equation 5.1. If we set $p' = p$ we get

$$Dp(1-p)(2p-1) = 0 \tag{5.2}$$

Assuming $D > 0$, there are three ways in which the left hand side of Equation 5.2 can equal zero: $p = 0$, $1 - p = 0$ or $p = 1$, and $2p - 1 = 0$ or $p = 0.5$. This matches the three equilibria we see in the previous plot; which one we reach depends on starting conditions.

Finally, we can use the recursion equation to generate a plot that has become a signature for conformity in the cultural evolution literature. The following code plots, for all possible values of p, the probability of adopting p in the next generation.

```
p <- seq(0,1,length.out = 101)

D <- 1
p_next <- p + D*p*(1-p)*(2*p-1)

plot(p, p_next,
     type = 'l',
     ylab = "probability of adopting A (p')",
     xlab = "frequency of A (p)",
     main = paste("D =", D))

abline(a = 0, b = 1, lty = 3)
```



This encapsulates the process of conformity. The dotted line shows unbiased transmission: the probability of adopting A is exactly equal to the frequency of A in the population. The s-shaped solid curve shows conformist transmission. When A is common ($p > 0.5$), then the curve is higher than the dotted line: there is a disproportionately higher probability of adopting A. When A is uncommon ($p < 0.5$), then the curve is lower than the dotted line: there is a disproportionately lower probability of adopting A.

References

Boyd, R., & Richerson, P. J. (1985). Culture and the evolutionary process. University of Chicago Press.

Deffner, D., Kleinow, V., & McElreath, R. (2020). Dynamic social learning in temporally and spatially variable environments. Royal Society Open Science, 7(12), 200734.

Efferson, C., Lalive, R., Richerson, P. J., McElreath, R., & Lubell, M. (2008). Conformists and mavericks: the empirics of frequency-dependent cultural transmission. Evolution and Human Behavior, 29(1), 56-64.

Henrich, J. (2001). Cultural transmission and the diffusion of innovations: Adoption dynamics indicate that biased cultural transmission is the predominate force in behavioral change. American Anthropologist, 103(4), 992-1013.

Henrich, J., & Boyd, R. (1998). The evolution of conformist transmission and the emergence of between-group differences. Evolution and human behavior, 19(4), 215-241.

Mesoudi, A. (2018). Migration, acculturation, and the maintenance of between-group cultural variation. PloS one, 13(10), e0205573.

Muthukrishna, M., Morgan, T. J., & Henrich, J. (2016). The when and who of social learning and conformist transmission. Evolution and Human Behavior, 37(1), 10-20.

Nakahashi, W., Wakano, J. Y., & Henrich, J. (2012). Adaptive social learning strategies in temporally and spatially varying environments. Human Nature, 23(4), 386-418.

Model 6: Vertical and horizontal cultural transmission

Introduction

One obvious difference between cultural and genetic evolution is in their pathways of transmission. In genetic evolution - in humans at least - we get our genes exclusively from our two biological parents. In cultural evolution, we get cultural traits (ideas, attitudes, skills, languages etc.) from a wide array of sources, not just our biological parents but also other relatives, biologically unrelated teachers and peers, or complete strangers via books or the internet.

All of the models we have made so far assume *oblique cultural transmission*, such that agents learn from one or more members of the previous generation. In Model 6 we will consider *vertical cultural transmission*, in which agents learn from two parents, and *horizontal cultural transmission*, where agents learn from members of the same generation.

The models are still extremely simple, with no actual biological reproduction or proper family lineages. We are also maintaining the *non-overlapping generations* of prior models, which neatly separates each generation. Real life is more messy, with overlapping generations that blur distinctions between, say, oblique and horizontal transmission. But as we have seen, models are useful because of, not despite, their simplicity. In this case, we consider three interesting features of vertical and horizontal transmission. First, the case of assortative cultural mating, where one's parents are more similar in cultural traits than two randomly chosen members of the population. Second, the case where biases in vertical and horizontal transmission act in opposite directions. Third, we examine the claim that cultural evolution is faster than genetic evolution because it features horizontal transmission.

All of these pathways of transmission were modelled in depth by Cavalli-Sforza & Feldman (1981), and the simulation models here follow their general form.

Model 6a: Vertical cultural transmission

Following the same approach as we did for conformity in Model 4, we can make a table specifying the outcome of vertical cultural transmission for an offspring's cultural traits given its two parents' cultural traits. As before, we will use two discrete traits A and B, with the frequency of A being p and of B being $1-p$.

Mother's trait	Father's trait	Probability of child adopting A
A	A	1
A	B	$1/2 + s_v/2$
B	A	$1/2 + s_v/2$
B	B	0

This table assumes that if both parents have the same trait, then the child inherits that trait with 100% certainty. If parents have different traits, then the child has a 50% chance of inheriting either one, plus an additional chance $s_v/2$. This is similar to the conformity parameter D in giving an adoption boost to a trait, except that now trait A is always getting a boost, irrespective of which parent has A.

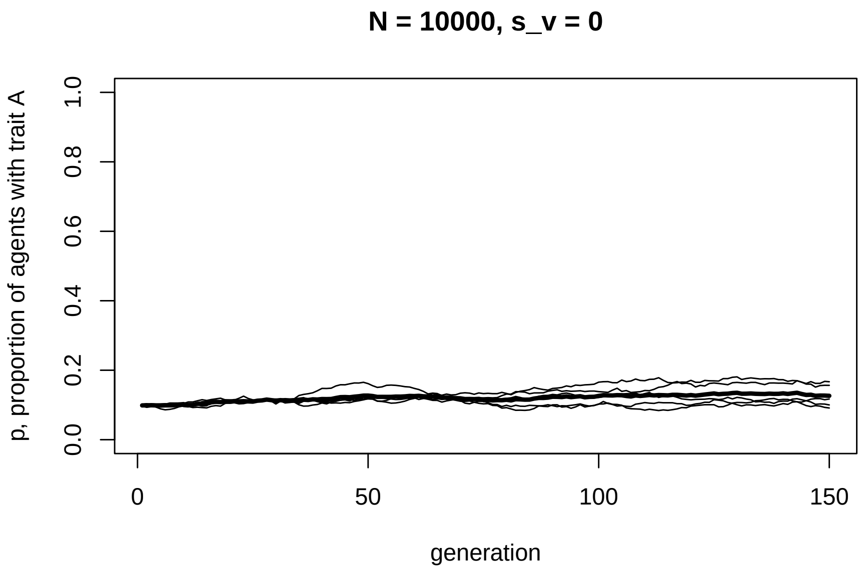
Consequently, s_v can be seen as a form of directly biased transmission or cultural selection, similar to that explored in Model 3: it gives the probability of preferentially adopting trait A above that expected under unbiased cultural transmission. It ranges from zero (unbiased transmission) to one (fully biased transmission, where A is always adopted if either parent has it). It can also be negative (up to -1), in which case trait B is favoured over A.

The following function takes the structure of **ConformistTransmission** from Model 5, replacing the three randomly-chosen demonstrators with two randomly-chosen parents, and setting the offspring traits as per the table above.

```r
VerticalTransmission <- function (N, p_0, s_v, t_max, r_max) {

  # create matrix with t_max rows and r_max columns, fill with NAs, convert to datafra
  output <- as.data.frame(matrix(NA,t_max,r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # create dataframe with a set of 2 randomly-picked parents for each agent
      parents <- data.frame(mother = sample(agent$trait, N, replace = TRUE),
                            father = sample(agent$trait, N, replace = TRUE))

      prob <- runif(N)

      # if both parents have A, child has A
      agent$trait[parents$mother == "A" & parents$father == "A"] <- "A"

      # if both parents have B, child has B
      agent$trait[parents$mother == "B" & parents$father == "B"] <- "B"

      # if mother has A and father has B, child has A with prob (1/2 + s_v/2), otherwise B
      agent$trait[parents$mother == "A" & parents$father == "B" &
                  prob < (1/2 + s_v/2)] <- "A"
      agent$trait[parents$mother == "A" & parents$father == "B" &
                  prob >= (1/2 + s_v/2)] <- "B"

      # if mother has B and father has A, child has A with prob (1/2 + s_v/2), otherwise B
      agent$trait[parents$mother == "B" & parents$father == "A" &
                  prob < (1/2 + s_v/2)] <- "A"
      agent$trait[parents$mother == "B" & parents$father == "A" &
                  prob >= (1/2 + s_v/2)] <- "B"

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", s_v = ", s_v, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

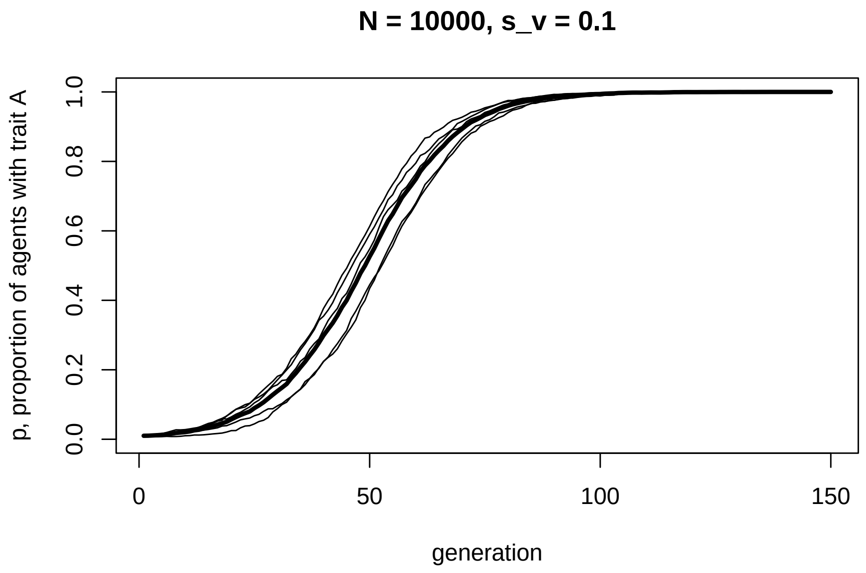
First we can check that when $s_v = 0$, we do indeed see unbiased transmission:

```
data_model6a <- VerticalTransmission(N = 10000,
                                     p_0 = 0.1,
                                     s_v = 0.0,
                                     t_max = 150,
                                     r_max = 5)
```



As in previous models, unbiased transmission results in no change in trait frequencies except random fluctuation due to chance. Now we can add selection:

```
data_model6a <- VerticalTransmission(N = 10000,
                                     p_0 = 0.01,
                                     s_v = 0.1,
                                     t_max = 150,
                                     r_max = 5)
```



This looks a lot like the s-shaped curve we found in Model 3 for $s = 0.1$. We have recapitulated unbiased transmission from Model 2 combined with biased transmission from Model 3, but assuming vertical cultural transmission from two parents, rather than randomly picking one demonstrator from the previous generation.

Model 6b: Assortative mating

We can use this vertical transmission model to explore what happens when we relax our assumption that parents form entirely at random. In reality, parents may be more culturally similar than average: two conservatives (or two liberals) may be more likely to get together than a conservative and a liberal; two vegans more likely than one vegan and one meat-eater. In evolutionary biology, this is known as *assortative mating*. For genetic evolution, it is assumed that the mates assort on genetically inherited characteristics. In cultural evolution, the mates assort on culturally inherited characteristics.

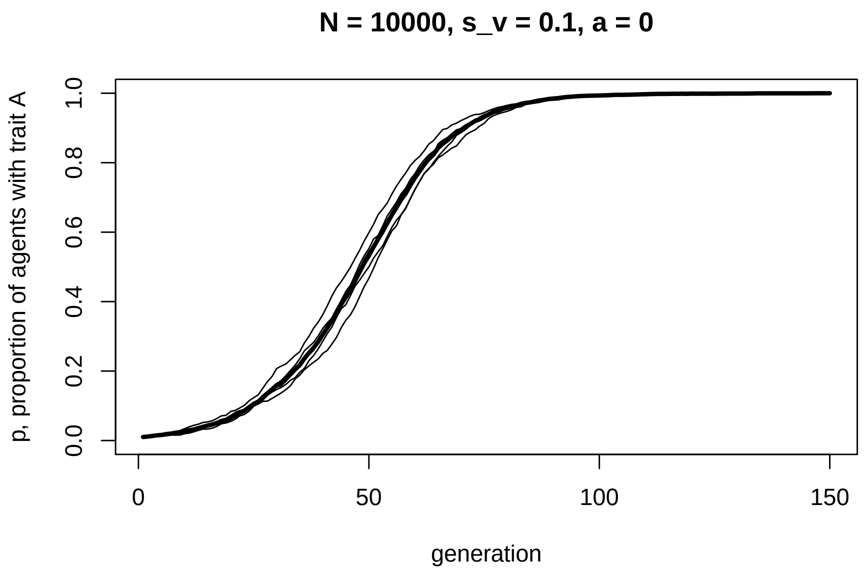
We now assume that a fraction a of matings are assortative, such that they must involve two parents with identical cultural traits, either both A or both B. A fraction $1 - a$ of matings are random as before, and can be any combination of traits. The following function implements this.

```
VerticalAssortative <- function (N, p_0, s_v, a, t_max, r_max) {

  # create matrix with t_max rows and r_max columns, fill with NAs, convert to datafra
  output <- as.data.frame(matrix(NA,t_max,r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                                   prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # 1. assortative mating:

      # create dataframe with a set of 2 parents for each agent
      # mother is picked randomly, father is blank for now
      parents <- data.frame(mother = sample(agent$trait, N, replace = TRUE),
                            father = rep(NA, N))

      # probabilities for a
      prob <- runif(N)

      # with prob a, make father identical
      parents$father[prob < a] <- parents$mother[prob < a]

      # with prob 1-a, pick random trait for father
      parents$father[prob >= a] <- sample(agent$trait,
                                          sum(prob >= a),
                                          replace = TRUE)

      # 2. vertical transmission:

      # new probabilities for s_v
      prob <- runif(N)
      # if both parents have A, child has A
      agent$trait[parents$mother == "A" & parents$father == "A"] <- "A"

      # if both parents have B, child has B
      agent$trait[parents$mother == "B" & parents$father == "B"] <- "B"

      # if mother has A and father has B, child has A with prob (1/2 + s_v/2), otherwise B
      agent$trait[parents$mother == "A" & parents$father == "B" &
                  prob < (1/2 + s_v/2)] <- "A"
      agent$trait[parents$mother == "A" & parents$father == "B" &
                  prob >= (1/2 + s_v/2)] <- "B"

      # if mother has B and father has A, child has A with prob (1/2 + s_v/2), otherwise B
      agent$trait[parents$mother == "B" & parents$father == "A" &
                  prob < (1/2 + s_v/2)] <- "A"
      agent$trait[parents$mother == "B" & parents$father == "A" &
                  prob >= (1/2 + s_v/2)] <- "B"

      # 3. store results:

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  # first plot a thick line for the mean p
  plot(rowMeans(output),
       type = 'l',
       ylab = "p, proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N, ", s_v = ", s_v, ", a = ", a, sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function
}
```

Most of **VerticalAssortative** is the same as **VerticalTransmission**, except for changes in the way parents are created. Rather than picking all mothers and fathers randomly, we first pick mothers randomly, then with probability a give the fathers the same cultural trait as the mother. With probability $1 - a$ we pick traits randomly for fathers, as in **VerticalTransmission**. The rest is the same, except that we add a to the figure title.

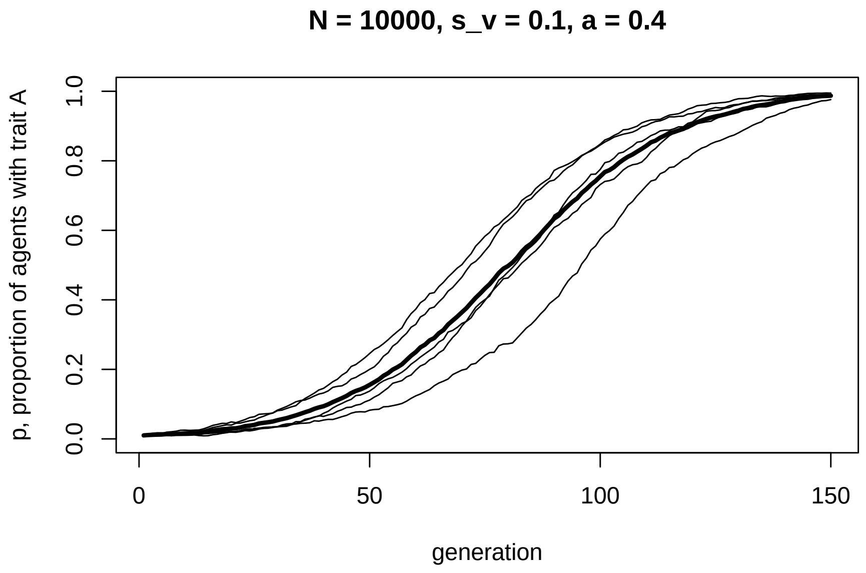
Let's first check that when $a = 0$, i.e. no assortative mating, the output is identical to that with **VerticalTransmission** above:

```
data_model6b <- VerticalAssortative(N = 10000,
                                    p_0 = 0.01,
                                    s_v = 0.1,
                                    a = 0,
                                    t_max = 150,
                                    r_max = 5)
```



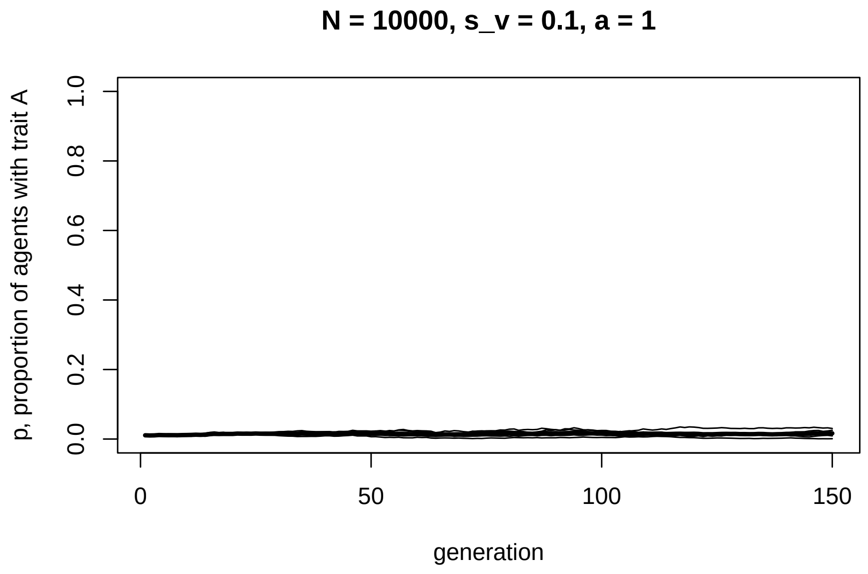
It's identical to before, with an s-shaped curve indicative of directly biased transmission. Now let's add some assortative mating, with $a = 0.4$:

```
data_model6b <- VerticalAssortative(N = 10000,
                                    p_0 = 0.01,
                                    s_v = 0.1,
                                    a = 0.4,
                                    t_max = 150,
                                    r_max = 5)
```



The curve is still s-shaped, but takes longer to reach $p = 1$. What happens when all mating is assortative, i.e. $a = 1$?

```
data_model6b <- VerticalAssortative(N = 10000,
                                    p_0 = 0.01,
                                    s_v = 0.1,
                                    a = 1,
                                    t_max = 150,
                                    r_max = 5)
```



Complete assortative mating results in no cultural change, beyond random fluctuations, even when selection is acting. In general, the more assortative mating there is, the weaker cultural selection will be.

In hindsight this is perhaps obvious. If assortative mating results in parents who have identical cultural traits, and as per the table above identical parents always give rise to identical children, then there will be no change resulting from these matings even when $s_v > 0$. However, under different assumptions about mating and transmission this may not always be the case. And hindsight does not always match foresight. It's always good to check and verify even simple predictions and intuitions.

Model 6c: Horizontal cultural transmission

Now we can add horizontal cultural transmission, which involves learning from members of the same generation. We will build this into the **VerticalAssortative** function above, such that vertical transmission (with assortative mating if $a > 0$) occurs first, and then horizontal cultural transmission occurs within the new generation of agents that form after vertical transmission.

There are many ways of implementing horizontal cultural transmission, just like there are many ways of implementing oblique transmission, as covered in other models (e.g. directly biased transmission, conformist biased transmission, blending inheritance). To allow a comparison with vertical transmission, here we will assume directly biased horizontal transmission. We will use a slightly different version of directly biased transmission from Model 3, modified to capture the key advantage of horizontal transmission over vertical transmission: the larger number of potential demonstrators.

Recall that Model 3 involved each agent randomly choosing one member of the previous generation and, if that demonstrator has trait A, then trait A is adopted with probability s. This reflects a situation where A is favoured by selection: perhaps A is a more effective tool, more memorable story, or more easily pronounced word. We are interested in when and how A spreads when initially rare in the population.

In Model 6c we assume that agents now choose n members of the *same* generation, i.e. the set of agents who have already undergone vertical transmission. If at least one of those n demonstrators has trait A, then the learner adopts trait A with probability s_h.

This is directly biased transmission, as in Model 3, because s_h allows agents with B to switch to A. There is no possibility of agents with A switching to B. The difference from Model 3 is that now there are n demonstrators rather than one, and we use the symbol s_h to distinguish this selection parameter from the one incorporated into vertical transmission, s_v, and the one originally used in Model 3, s.

The following function adds horizontal transmission to **VerticalAssortative**. We add n and s_h to the parameter list, and add some code to implement the horizontal transmission rule.

```r
VerticalHorizontal <- function (N, p_0, s_v, s_h, a, n, t_max, r_max, make_plot = TRUE) {

  # create matrix with t_max rows and r_max columns, fill with NAs, convert to dataframe
  output <- as.data.frame(matrix(NA,t_max,r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                        prob = c(p_0,1-p_0)))

    # add first generation's p to first row of column r
    output[1,r] <- sum(agent$trait == "A") / N

    for (t in 2:t_max) {

      # 1. assortative mating:
```

```
    # create dataframe with a set of 2 parents for each agent
    # mother is picked randomly, father is blank for now
    parents <- data.frame(mother = sample(agent$trait, N, replace = TRUE),
                          father = rep(NA, N))

    # probabilities for a
    prob <- runif(N)

    # with prob a, make father identical
    parents$father[prob < a] <- parents$mother[prob < a]

    # with prob 1-a, pick random trait for father
    parents$father[prob >= a] <- sample(agent$trait,
                                        sum(prob >= a),
                                        replace = TRUE)

    # 2. vertical transmission:

    # new probabilities for s_v
    prob <- runif(N)

    # if both parents have A, child has A
    agent$trait[parents$mother == "A" & parents$father == "A"] <- "A"

    # if both parents have B, child has B
    agent$trait[parents$mother == "B" & parents$father == "B"] <- "B"

    # if mother has A and father has B, child has A with prob (1/2 + s_v/2), otherwi
    agent$trait[parents$mother == "A" & parents$father == "B" &
                prob < (1/2 + s_v/2)] <- "A"
    agent$trait[parents$mother == "A" & parents$father == "B" &
                prob >= (1/2 + s_v/2)] <- "B"

    # if mother has B and father has A, child has A with prob (1/2 + s_v/2), otherwi
    agent$trait[parents$mother == "B" & parents$father == "A" &
                prob < (1/2 + s_v/2)] <- "A"
    agent$trait[parents$mother == "B" & parents$father == "A" &
                prob >= (1/2 + s_v/2)] <- "B"

    # 3. horizontal transmission:

    # create matrix for holding n demonstrators for N agents
    # fill with randomly selected agents from current gen
    demonstrators <- matrix(data = sample(agent$trait, N*n, replace = TRUE),
                            nrow = N, ncol = n)
```

```
      # record whether there is at least one A in each row
      oneA <- rowSums(demonstrators == "A") > 0

      # new probabilities for s_h
      prob <- runif(N)

      # adopt trait A if oneA is true and with prob s_h
      agent$trait[oneA & prob < s_h] <- "A"

      # 4. store results:

      # get p and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "A") / N

    }

  }

  if (make_plot == TRUE) {

    # first plot a thick line for the mean p
    plot(rowMeans(output),
        type = 'l',
        ylab = "p, proportion of agents with trait A",
        xlab = "generation",
        ylim = c(0,1),
        lwd = 3,
        main = paste("N = ", N, ", s_v = ", s_v, ", s_h = ", s_h,
                    ", a = ", a, ", n = ", n, sep = ""))

    for (r in 1:r_max) {

      # add lines for each run, up to r_max
      lines(output[,r], type = 'l')

    }

  }

  output  # export data from function
}
```

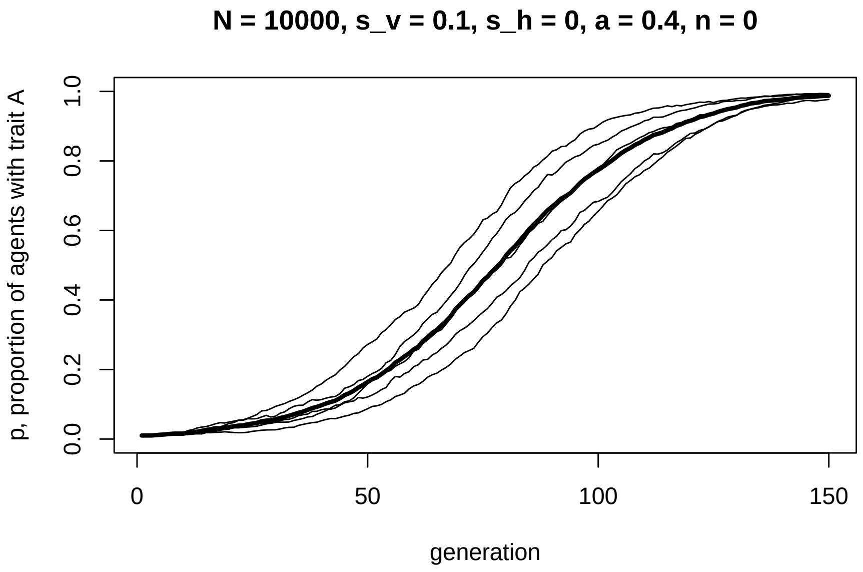
In the horizontal transmission section of **VerticalHorizontal** we first create a table of demonstrators using the **matrix** command. We use **matrix** rather than **data.frame** because in R the number of rows of a matrix can be generated on-the-fly, unlike dataframes. Here we create a matrix with n rows, representing the number of demonstrators, and N columns, representing the number of agents. We fill this with randomly-chosen members of the current generation using **sample**, as normal.

We then use the **rowSums** command to count the number of times in each row an A appears, and create a vector *oneA* which is *TRUE* whenever there is at least one A, and *FALSE* if there are no As. Then, if an agent has a *TRUE* in its corresponding *oneA* vector, and with probability s_h, it adopts A. Otherwise, it keeps the same trait that it received during vertical transmission.

There is one final modification. We add a variable *make_plot* to the function definition, and wrap all the plotting code within an **if** statement such that plots are only drawn when *make_plot == TRUE*. This is a useful way of turning off the plot generation, and will come in handy later. In the function call, the default value of *make_plot* is given as *TRUE*, so if we omit this in the function call, the plot is generated by default.

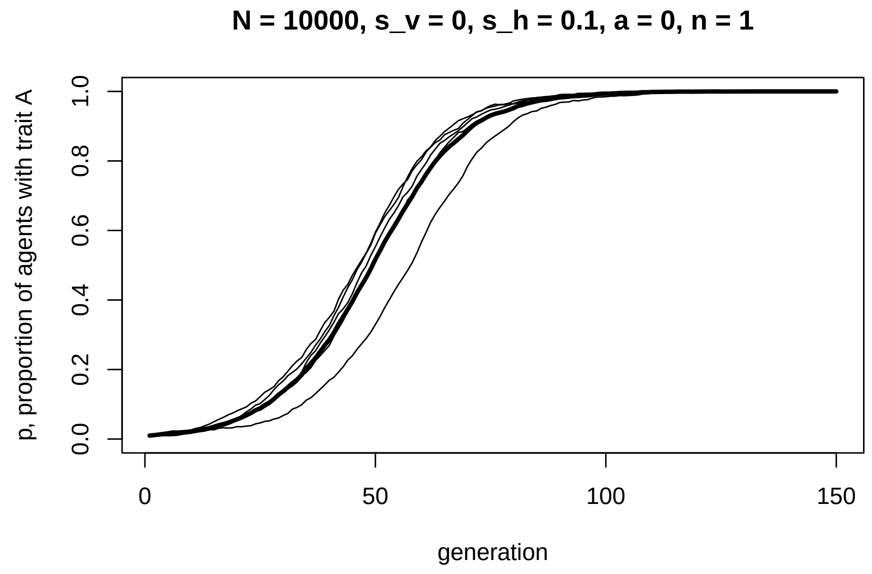
First let's check that vertical cultural transmission and assortative mating work as before, to make sure we didn't break anything.

```r
data_model6c <- VerticalHorizontal(N = 10000,
                                   p_0 = 0.01,
                                   s_v = 0.1,
                                   s_h = 0,
                                   a = 0.4,
                                   n = 0,
                                   t_max = 150,
                                   r_max = 5)
```



This should be roughly the same shape as the corresponding figure above with $s_v = 0.1$ and $a = 0.4$. Now let's turn off vertical transmission and try just horizontal transmission:

```
data_model6c <- VerticalHorizontal(N = 10000,
                                   p_0 = 0.01,
                                   s_v = 0,
                                   s_h = 0.1,
                                   a = 0,
                                   n = 1,
                                   t_max = 150,
                                   r_max = 5)
```

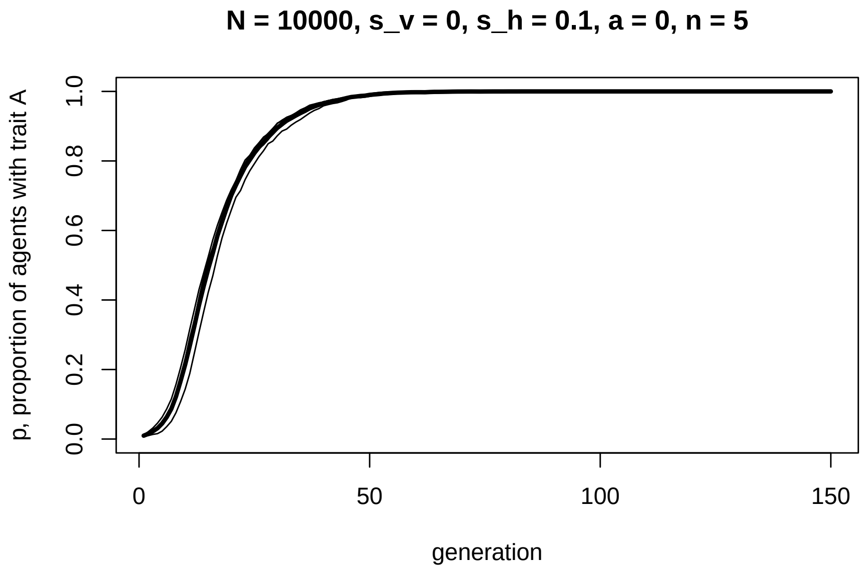


Horizontal transmission with $s_h = 0.1$ and $n = 1$ looks almost identical to the plot generated from **BiasedTransmission** in Model 3 with $s = 0.1$. This makes sense because **BiasedTransmission** used the same rule, but with n fixed at one.

The curve above also looks almost identical to the first vertical transmission curve generated above in Model 6a using **VerticalTransmission** with $s_v = 0.1$ and $a = 0$. Under these assumptions, vertical cultural transmission from two randomly chosen parents is equivalent to horizontal cultural transmission from one randomly chosen demonstrator.

Now let's increase n, the number of demonstrators in horizontal transmission:

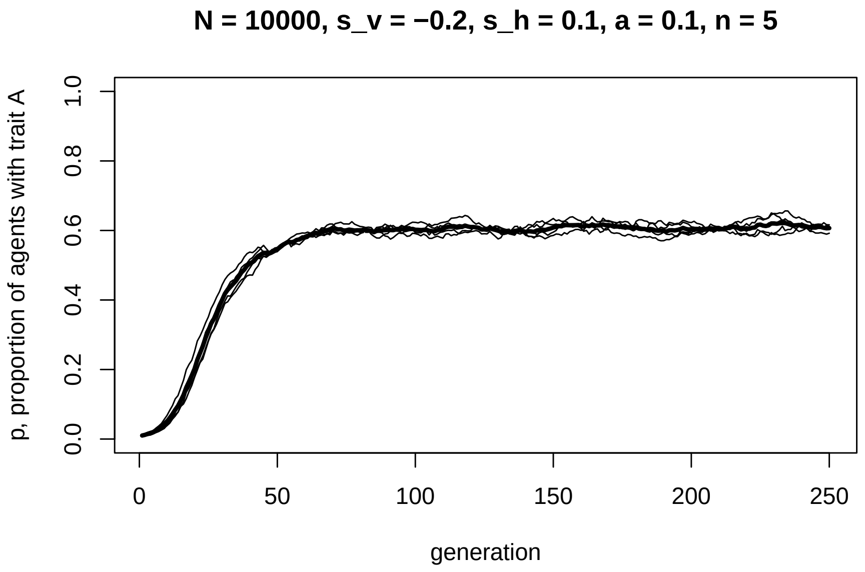
```r
data_model6c <- VerticalHorizontal(N = 10000,
                                   p_0 = 0.01,
                                   s_v = 0,
                                   s_h = 0.1,
                                   a = 0,
                                   n = 5,
                                   t_max = 150,
                                   r_max = 5)
```



Increasing n greatly increases the strength of selection due to horizontal cultural transmission. If you only need one out of five demonstrators to have A for selection to operate, then selection for A will be more frequent than if you need one out of one demonstrator to have A.

If we set $-1 < s_v < 0$ then biased vertical transmission favours B, rather than A. Combining this with $s_h > 0$ gives the case where vertical and horizontal transmission act in opposite directions. This might represent a 'clash of the generations' over traits such as smoking: parents exert pressure on children not to smoke, while peer pressure encourages smoking. The following illustrates such a case:

```
data_model6c <- VerticalHorizontal(N = 10000,
                                   p_0 = 0.01,
                                   s_v = -0.2,
                                   s_h = 0.1,
                                   a = 0.1,
                                   n = 5,
                                   t_max = 250,
                                   r_max = 5)
```



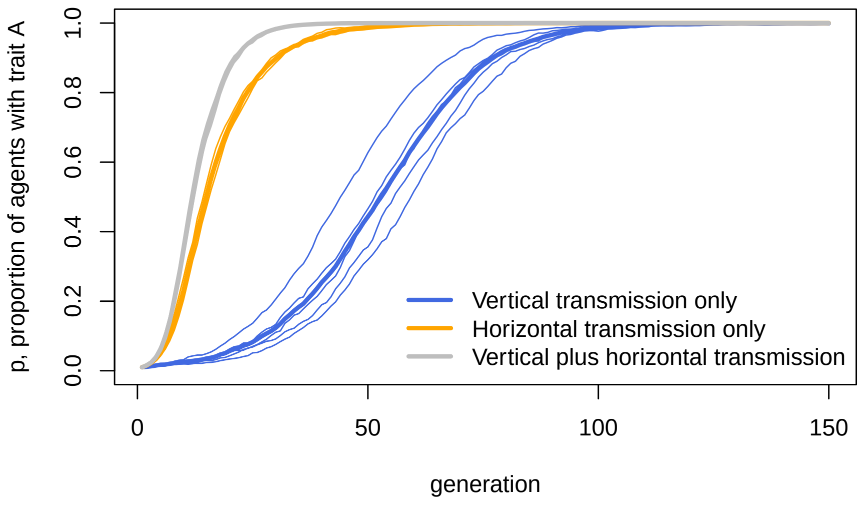
Whereas before there was always a single equilibrium at $p = 1$, here there is a stable mix of A and B agents co-existing at equilibrium at a point where the vertical and horizontal transmission biases balance out. In the plot above, this equilibrium value is approximately $p^* = 0.6$, but this varies with different combinations of s_v, a, s_h and n.

We started this discussion by comparing vertical-only genetic inheritance with cultural inheritance, which can be horizontal as well as (or instead of) vertical. Let's create a plot to compare three cases: vertical-only, horizontal-only and vertical-plus-horizontal. In each, vertical and horizontal transmission are now acting in the same direction, to favour trait A. We are interested in how quickly this favoured trait A spreads and goes to fixation.

The following code runs these three scenarios with *make_plot* = *FALSE* to avoid automatically plotting the results. Instead we store the output from each case, and plot all three on the same graph using different colours and a legend.

```
data_model6c_v <- VerticalHorizontal(N = 10000,
                                     p_0 = 0.01,
                                     s_v = 0.1,
                                     s_h = 0,
                                     a = 0.1,
                                     n = 0,
                                     t_max = 150,
                                     r_max = 5,
                                     make_plot = FALSE)
```

```r
data_model6c_h <- VerticalHorizontal(N = 10000,
                                     p_0 = 0.01,
                                     s_v = 0,
                                     s_h = 0.1,
                                     a = 0,
                                     n = 5,
                                     t_max = 150,
                                     r_max = 5,
                                     make_plot = FALSE)

data_model6c_vh <- VerticalHorizontal(N = 10000,
                                      p_0 = 0.01,
                                      s_v = 0.1,
                                      s_h = 0.1,
                                      a = 0.1,
                                      n = 5,
                                      t_max = 150,
                                      r_max = 5,
                                      make_plot = FALSE)

# plot vertical-only in blue
plot(rowMeans(data_model6c_v),
        type = 'l',
        ylab = "p, proportion of agents with trait A",
        xlab = "generation",
        ylim = c(0,1),
        lwd = 3,
        col = "royalblue")

for (r in 1:ncol(data_model6c_v)) {
  lines(data_model6c_v[,r], type = 'l', col = "royalblue")
}

# plot horizontal-only in orange
lines(rowMeans(data_model6c_h), type = 'l', lwd = 3, col = "orange")

for (r in 1:ncol(data_model6c_h)) {
  lines(data_model6c_h[,r], type = 'l', col = "orange")
}

# plot vertical-horizontal in grey
lines(rowMeans(data_model6c_vh), type = 'l', lwd = 3, col = "grey")

for (r in 1:ncol(data_model6c_vh)) {
  lines(data_model6c_vh[,r], type = 'l', col = "grey")
}

legend("bottomright",
       legend = c("Vertical transmission only",
                  "Horizontal transmission only",
                  "Vertical plus horizontal transmission"),
       lty = 1,
       lwd = 3,
       col = c("royalblue", "orange", "grey"),
       bty = "n")
```



As we might expect, vertical plus horizontal transmission is fastest, with both forms of biased transmission combining together to favour A. This is closely followed by horizontal transmission alone, which has the advantage of a large pool of demonstrators. The genetic-inheritance-like vertical transmission alone is slowest.

Summary

Our previous models all assumed oblique cultural transmission, where new agents learn from members of the previous generation. Here we modelled vertical cultural transmission, where agents learn from two parents (akin to genetic evolution), and horizontal cultural transmission, where agents learn from members of the same generation.

In Model 6a vertical cultural transmission was modelled in a simple way by picking two random members of the previous generation to be the parents for each new agent, then unbiased or biased transmission of traits occurs from those parents. This effectively recapitulated our previous unbiased and directly biased oblique transmission models but with slightly different assumptions. This consistency gives us confidence in the findings. It is always good to implement the same concept in different ways, to check that its effects are consistent or whether they depend on particular assumptions.

In Model 6b we extended vertical transmission by adding assortative mating, where the two parents are more culturally similar than expected under random mating. Assortative cultural mating acts in Model 6b to reduce the strength of cultural selection. This is because we assume that culturally identical parents always produce culturally identical offspring, and the selection bias only acts when parents have different traits. We can change these assumptions to explore other scenarios, and get different results (see Cavalli-Sforza & Feldman 1981). In general, assortative mating can potentially have positive effects, such as improving communication or cooperation between the parents thus making transmission more likely, or negative effects, such as generating cultural segregation or inequality in the population.

Finally, in Model 6c we implemented horizontal transmission, which we assume occurs after vertical transmission, is also directly biased, and occurs from n demonstrators in the same generation as the learner. Model 6c showed that the strength of directly biased horizontal transmission increases not only with the selection strength parameter s_h, but also with n. This reflects an often-cited advantage of horizontal over vertical transmission: one can potentially learn from many more sources than just your two parents.

When vertical and horizontal transmission act in different directions, i.e. to favour different traits, then we see a stable mixed equilibrium where A and B co-exist in the population. This might reflect a case where parents favour one cultural trait (e.g. not smoking), and peer-pressure favours a different trait (e.g. smoking). Whereas previously biased vertical and horizontal transmission both eliminate variation from the population, in this case cultural variation is maintained.

Model 6c showed that horizontal transmission can be faster at spreading favoured traits than vertical transmission, especially when there are many demonstrators from whom agents learn. Vertical plus horizontal transmission is faster still, assuming both act in the same direction. Empirically, vertical plus horizontal (or vertical plus oblique) transmission is thought to be typical of real life human cultural inheritance, with children initially learning from their parents then updating from peers and elders later in life (Aunger 2000;

Henrich & Broesch 2011).

The empirical record also shows that, over long timescales, cultural evolution is faster than genetic evolution (Perreault 2012). It is likely that horizontal cultural transmission is responsible for this speed, and has allowed human populations to adapt culturally to novel environments faster than they would have been able to via vertical-only genetic evolution alone. When a novel trait emerges in a population via mutation or migration, whether it is the bow-and-arrow or the smart phone, horizontal transmission allows it to spread far faster than if transmission were purely vertical. On the other hand, horizontal transmission might also allow harmful traits to rapidly spread before their negative effects become known, or before natural selection has had a chance to act on them.

The models here can be extended to look at uniparental vertical transmission, where either the mother or father is more culturally influential, rather than the biparental transmission we implemented above. Some traits are known to be transmitted uniparentally, or more strongly by one parent than the other. For example, one survey of Stanford students in the early 1980s found that religious denomination was more strongly maternally transmitted, and political orientation more strongly paternally transmitted (Cavalli-Sforza et al. 1982). Uniparental transmission will exhibit different long-term dynamics than biparental transmission (see Cavalli-Sforza & Feldman 1981).

Note that there is a subtle difference between vertical and horizontal transmission in this model. For vertical transmission, we assume (quite reasonably) that children come into the world lacking any cultural traits. Consequently, the vertical transmission table above gives the probabilities of adopting either A or B given the parents' values and the selection strength parameter s_v. For horizontal transmission, on the other hand, children already have a cultural trait, the one they obtained as a result of vertical transmission. We assume that if they already have A, there is no possibility of switching to B. If they already have B, then there is a chance of switching to A, depending on s_h and whether any of the n demonstrators have A. We can imagine here that individuals who have direct experience of A can be sure that it is better than B, so never switch. Individuals with B need one exposure to A in order to learn it, and do so with probability s_h. You might think that this unfairly weights the influence of horizontal transmission greater than that of vertical transmission. And that's fine! You are welcome to modify the model to better match how you think the transmission pathways should work, or better match empirical data. That's the beauty of models: because they are formally specified, it's easy to see where you might disagree with assumptions (which are often hidden or implicit in verbal models) and change them.

In terms of programming techniques, there is not much new here. We have recycled code from several previous models, especially the directly biased transmission from Model 3 and the 'mating table' approach of Model 5. Don't be afraid to re-use code, if it's been tried and tested elsewhere (with appropriate attribution, if it's not your own code). The one minor innovation was wrapping the plotting in a **for** statement and using a parameter *make_plot* to turn the automatic plotting on or off. This is useful if you want to combine plots from multiple runs, as we did in the final graph comparing the different pathways of inheritance.

Exercises

1. Try different values of s_v in **VerticalTransmission** to confirm that increasing s_v increases the speed with which A goes to fixation, and confirm that these dynamics are identical to how s acts in **BiasedTransmission** from Model 3.

2. Try running **VerticalTransmission** with $s_v = -0.1$ and $p_0 = 0.9$. What happens? What does a negative value of s_v mean?

3. Modify **VerticalTransmission** to allow the mother and the father to have different levels of influence. Replace the single s_v parameter with two selection parameters, one for the mother and one for the father. Explore the dynamics of uniparental biased transmission, compared to the biparental biased transmission implemented in the original model.

4. Try different values of a in **VerticalAssortative** to confirm that as a increases, selection slows. Write a function to record the number of timesteps it takes for a run to go to fixation ($p = 1$) for different values of a, and constant s_v. Plot a against this measure.

5. Try different values of s_h and n in **VerticalHorizontal** to confirm that as both parameters increase, the speed with which A goes to fixation increases. Again, create plots to show how fixation time varies with s_h and with n, with all other variables constant.

6. Try different negative values of s_v, along with different values of a, s_h and n, to confirm that there are different mixed equilibria depending on the balance of vertical and horizontal transmission bias.

7. Rewrite **VerticalHorizontal**, replacing the horizontal cultural transmission rule with conformity, using code from Model 5. Does this change the conclusions regarding the speed of vertical vs horizontal transmission?

Analytic Appendix

Following Cavalli-Sforza & Feldman (1981, Table 2.2.1), we can write out the full mating table for two parents as follows. For simplicity, we assume that their $b_3 = 1$, $b_0 = 0$, and $b_1 = b_2 = 1/2 + s/2$. In other words, there is no mutation and so two parents with the same trait always give rise to children with that trait, and when parental traits conflict then s_v represents the strength of selection for A under vertical transmission, irrespective of which parent the trait comes from.

Mother's trait	Father's trait	Probability of child adopting A	Probability of pair forming under random mating	Probability of pair forming under assortative mating
A	A	1	p^2	$p^2 + ap(1-p)$
A	B	$1/2 + s_v/2$	$p(1-p)$	$p(1-p)(1-a)$
B	A	$1/2 + s_v/2$	$p(1-p)$	$p(1-p)(1-a)$
B	B	0	$(1-p)^2$	$(1-p)^2 + ap(1-p)$

Considering random mating first, the frequency of p in the next generation, p', is given by multiplying the probability of a child adopting A (column 3) by the probability of a pair forming under random mating (column 4) for each parental trait combination, then summing these products. This gives

$$p' = p^2 + (1 + s_v)p(1-p)$$

which simplifies to

$$p' = p + p(1-p)s_v \tag{6.1}$$

This is identical to Equation 3.1 from Model 3 where there was a single randomly-selected demonstrator (i.e. one 'parent'), rather than two. As in Equation 3.1, the change in p from one generation to the next is proportional to the strength of selection, s_v, and the variance in p, $p(1-p)$. In the Analytic Appendix to Model 3 we plotted this recursion to show that it takes the form of an s-shaped curve.

Following Cavalli-Sforza & Feldman (1981, Table 2.5.1) we can alternatively use the probability of a pair forming under assortative mating (column 5). (Note that I've used a rather than Cavalli-Sforza & Feldman's m to avoid confusion with the migration parameter m used elsewhere.) As in the simulation model, a proportion a of matings are assortative, i.e. between two agents with the same cultural trait (either A and A or B and B), and $1-a$ matings are random as before. For example, for the first A x A row, there are $(1-a)$ matings that are random and so have probability p^2 as previously, and a matings that are assortative and will have probability p (because of the a matings that are assortative, the A x A pairings will occur with probability p and the B x B pairings will occur with probability $1-p$). This gives:

$$prob(A \text{ x } A) = (1-a)p^2+ap = p^2-ap^2+ap = p^2+ap(1-p) \tag{6.2}$$

and so on for the other rows.

As for random mating, we then multiply and sum the third and fifth columns to get p' under assortative mating:

$$p' = p^2 + ap(1-p) + (1+s_v)p(1-p)(1-a)$$

which simplifies to:

$$p' = p + p(1-p)s_v(1-a) \tag{6.3}$$

This is the same as Equation 6.1 but with the difference between p in successive generations also proportional to $(1-a)$. That is, the larger is a, the less change there is (at the extreme, $a = 1$, then $\Delta p = 0$). Assortative mating acts against selection.

Finally we can introduce a recursion for horizontal transmission. As in the simulation model, we assume that individuals observe n individuals and adopt A with probability s_h if at least one of those n individuals possesses trait A.

If p'' denotes the frequency of A after horizontal transmission, and p' the frequency before horizontal transmission but after vertical transmission as per Equation 6.1 or 6.3, then:

$$p'' = p' + (1-p')s_h(1-(1-p')^n) \tag{6.4}$$

Here, there are p' individuals who already have A and so can't change, and $(1-p')$ individuals who have B and change to A with probability equal to the strength of selection in horizontal transmission, s_h, multiplied by the probability that at least one of the n demonstrators has an A. This latter probability will be one minus the probability of none of the n individuals possessing A, i.e. $1-(1-p)^n$.

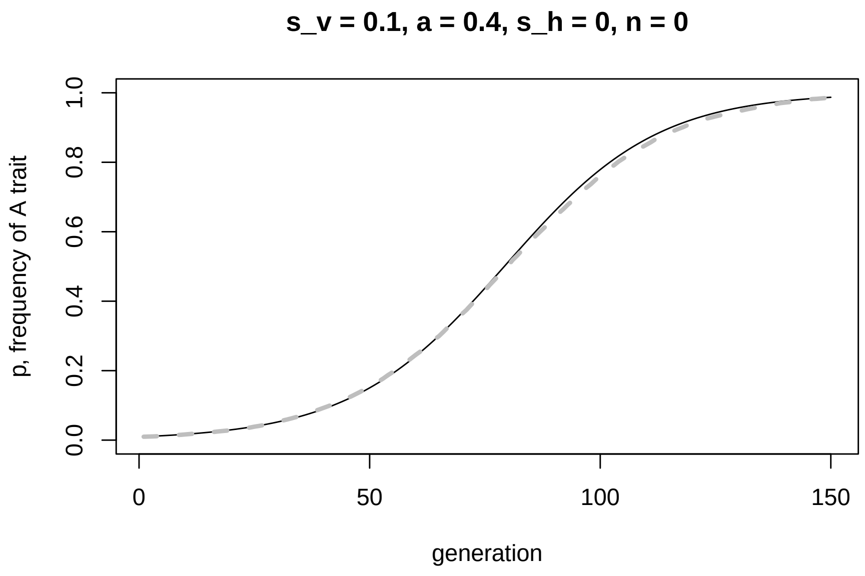
The potency of horizontal transmission lies in this last term. When $n = 1$, then $1-(1-p)^n$ reduces to p, and we retrieve the same directly biased transmission form as Equations 3.1 and 6.1. As n increases, then $1-(1-p)^n$ tends to 1. Because p must be less than or equal to 1, when $n > 1$ then horizontal transmission must be stronger than vertical transmission, for identical selection strength parameters, until an equilibrium is reached at $p = 1$ at which there is no change.

We can simulate the two recursions specified in Equations 6.3 and 6.4:

```
VerticalHorizontalRecursion <- function(s_v, a, s_h, n, t_max, p_0) {

  p <- rep(0,t_max)
  p[1] <- p_0

  for (i in 2:t_max) {

    p[i] <- p[i-1] + p[i-1]*(1-p[i-1])*s_v*(1-a)  # Eq 6.3

    p[i] <- p[i] + (1-p[i])*s_h*(1-(1-p[i])^n)  # Eq 6.4

  }

  plot(x = 1:t_max, y = p,
       type = "l",
       ylim = c(0,1),
       ylab = "p, frequency of A trait",
       xlab = "generation",
       main = paste("s_v = ", s_v, ", a = ", a, ", s_h = ", s_h, ", n = ", n, sep = ""

  p # output p
}
```

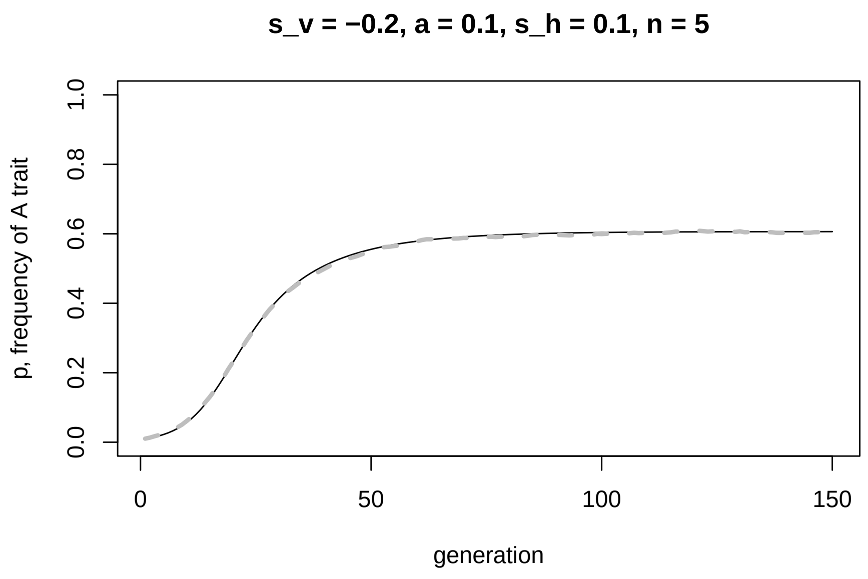
The following code plots the recursion line in black and the simulation data with a grey dashed line, for the same parameter values specifying vertical transmission only. We increase r_{max} to 20 to make the simulation mean as accurate as possible. They should match pretty well, confirming that both our simulation code and our maths is correct.

```
recursion_data <- VerticalHorizontalRecursion(p_0 = 0.01,
                                              s_v = 0.1,
                                              s_h = 0,
                                              a = 0.4,
                                              n = 0,
                                              t_max = 150)

simulation_data <- VerticalHorizontal(N = 10000,
                                      p_0 = 0.01,
                                      s_v = 0.1,
                                      s_h = 0,
                                      a = 0.4,
                                      n = 0,
                                      t_max = 150,
                                      r_max = 20,
                                      make_plot = FALSE)

lines(rowMeans(simulation_data), col = "grey", lwd = 3, lty = 2)
```



Here is the same for the case where vertical and horizontal transmission act in opposite directions:

```
recursion_data <- VerticalHorizontalRecursion(p_0 = 0.01,
                                              s_v = -0.2,
                                              s_h = 0.1,
                                              a = 0.1,
                                              n = 5,
                                              t_max = 150)

simulation_data <- VerticalHorizontal(N = 10000,
                                      p_0 = 0.01,
                                      s_v = -0.2,
                                      s_h = 0.1,
                                      a = 0.1,
                                      n = 5,
                                      t_max = 150,
                                      r_max = 20,
                                      make_plot = FALSE)

lines(rowMeans(simulation_data), col = "grey", lwd = 3, lty = 2)
```



Again, a good match.

The equilibrium value p^* can be found when $p = p''$. To make things easier, we can assume that $(1 - p')^n$ in Equation 6.4 is approximately zero when n is large, and so $(1-(1-p)^n)$ is approximately 1. Removing this term, substituting Equation 6.3 into 6.4, and setting $p = p''$ gives:

$$p = p + p(1-p)s_v(1-a) + s_h(1-(p + p(1-p)s_v(1-a)))$$

This can be rearranged to give:

$$(1-p)(p(s_v(1-a) - s_h s_v(1-a)) + s_h) = 0 \tag{6.5}$$

Hence there is one equilibrium when the first term $1 - p = 0$, such that $p^* = 1$. There is another where

$$p(s_v(1-a) - s_h s_v(1-a)) + s_h = 0$$

which rearranges to give

$$p^* = -\frac{s_h}{(1-a)(s_v - s_h s_v)} \tag{6.6}$$

With the values used in the previous graph, we can use Equation 6.6 to find the value of the internal equilibrium that we observed:

```
s_v <- -0.2
s_h <- 0.1
a <- 0.1

-s_h / ((1-a)*(s_v - s_h*s_v))
```

```
## [1] 0.617284
```

This should approximately match the final values of both the simulation and recursion data:

```
mean(as.numeric(simulation_data[nrow(simulation_data),]))
```

```
## [1] 0.606215
```

```
recursion_data[length(recursion_data)]
```

```
## [1] 0.606279
```

It's not an exact match due to the removal of the term with n. The match should get better as n increases.

Given that p^* must be less than one, an internal equilibrium can only exist when the right hand side of Equation 6.6 is less than one. After rearranging, this gives the inequality:

$$s_v(1-a) < -\frac{s_h}{(1-s_h)} \tag{6.7}$$

Because s_h must be positive by assumption, the right hand side of Equation 6.7 must be negative. Consequently the inequality in Equation 6.7 can only be true when s_v is negative, and moreover negative enough (after being reduced in strength by assortative cultural mating) to outweigh s_h acting in the opposite direction.

References

Aunger, R. (2000). The life history of culture learning in a face-to-face society. Ethos, 28(3), 445-481.

Cavalli-Sforza, L. L., & Feldman, M. W. (1981). Cultural transmission and evolution: a quantitative approach. Princeton University Press.

Cavalli-Sforza, L. L., Feldman, M. W., Chen, K. H., & Dornbusch, S. M. (1982). Theory and observation in cultural transmission. Science, 218(4567), 19-27.

Henrich, J., & Broesch, J. (2011). On the nature of cultural transmission networks: evidence from Fijian villages for adaptive learning biases. Philosophical Transactions of the Royal Society B, 366(1567), 1139-1148.

Perreault, C. (2012). The pace of cultural evolution. PLoS One, 7(9), e45150.

Model 7: Migration

Introduction

Along with selection, mutation and drift, another fundamental driver of genetic evolution is migration. The same is true of cultural evolution. Migration has been a permanent fixture of our species since we first dispersed out of Africa. The movement of people from group to group can shape between-group cultural diversity, and spread beneficial technologies or ideas from group to group.

Model 7

In Model 7 we will examine the effect of migration on between-group diversity. Previous models all featured a single group of agents. Now that we are interested in migration, we need to simulate multiple groups. Let's keep things simple and add one other group, making two in total: group 1 and group 2. As before, each group has N individuals in it.

As before, let's assume that there are two traits, A and B. We'll assume for now that they are neutral, i.e. not subject to selection / biased transmission. We assume that, initially, every member of group 1 has A, and every member of group 2 has B. Hence, in generation 1, $p = 1$, and $q = 0$, where p denotes the frequency of trait A in group 1 and q denotes the frequency of trait A in group 2. Such a case of maximum group difference might seem extreme, but is not too far from a situation where everyone in one society speaks one language and everyone in another society speaks another language, or everyone in one group practices one religion and everyone in another practices a different religion. In any case, remember that models are simplified, extreme cases designed to check the logic of verbal arguments, not exact recreations of reality.

The following code creates two agent dataframes, one for each group, and populates them with agents according to starting frequencies $p_0 = 1$ and $q_0 = 0$. Unlike before, we create another column recording which group this is, 1 or 2. We then combine the two dataframes into a single one using the **rbind** command, which combines dataframes by rows. Finally, we check it worked by calling the first five agents in group 1, i.e. agents one to five, who should all have trait A, and the first five agents in group 2, i.e. agents $N+1$ to $N+5$, who should all have trait B.

```r
N <- 100
p_0 <- 1
q_0 <- 0

# create first generation of group 1
agent1 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                     prob = c(p_0,1-p_0)),
                     group = 1)

# create first generation of group 2
agent2 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                     prob = c(q_0,1-q_0)),
                     group = 2)

# combine agent1 and agent2 into a single agent dataframe
agent <- rbind(agent1,agent2)

agent[1:5,]
```

```
##   trait group
## 1     A     1
## 2     A     1
## 3     A     1
## 4     A     1
## 5     A     1
```

```r
agent[(N+1):(N+5),]
```

```
##     trait group
## 101     B     2
## 102     B     2
## 103     B     2
## 104     B     2
## 105     B     2
```

We can also calculate the frequency of A in groups 1 and 2, i.e. p and q, and check that $p=1$ and $q=0$, with this code:

```r
p <- sum(agent$trait[agent$group == 1] == "A") / N
q <- sum(agent$trait[agent$group == 2] == "A") / N

paste("p =", p)
```

```
## [1] "p = 1"
```

```r
paste("q =", q)
```

```
## [1] "q = 0"
```

This code uses subsetting to get the number of As in group 1, and then in group 2, and divides both by N to get a proportion.

Now for migration. In each timestep, we assume that each agent has a probability m of migrating. There is a problem, though. Because this is stochastic, this might result in the groups changing size. Maybe in one generation five agents move from group 1 to group 2, while only two move from group 2 to group 1. Then, group 1 would be smaller than group 2, and neither would be N. This gets complicated to model, as the size of the dataframes holding agents would need to change during the simulations.

To avoid N from changing during the simulation, we can do the following, drawing inspiration from what's known as Wright's Island Model from population genetics. First, each agent migrates with probability m, ignoring group membership. On average, this gives $2Nm$ migrants across the entire population, given that there are two groups of N agents. We take the migrants out of their groups, leaving empty slots where they used to be. Then we put these migrants back into the empty slots at random, ignoring which group the slot is in. This keeps N constant: for every empty slot, there is a migrating agent. Some might go back into their original group, but this won't change anything so it doesn't really matter.

Finally, we are assuming that agents take their traits with them, for now. We will change this later.

The following code simulates one bout of migration, i.e. one generation in the model.

```r
m <- 0.1

# 2N probabilities, one for each agent, to compare against m
probs <- runif(1:(2*N))

# with prob m, add an agent's trait to list of migrants
```

```
migrants <- agent$trait[probs < m]

# put migrants randomly into empty slots
agent$trait[probs < m] <- sample(migrants, length(migrants))

p <- sum(agent$trait[agent$group == 1] == "A") / N
q <- sum(agent$trait[agent$group == 2] == "A") / N

paste("p =", p)
```

```
## [1] "p = 0.95"
```

```
paste("q =", q)
```

```
## [1] "q = 0.05"
```

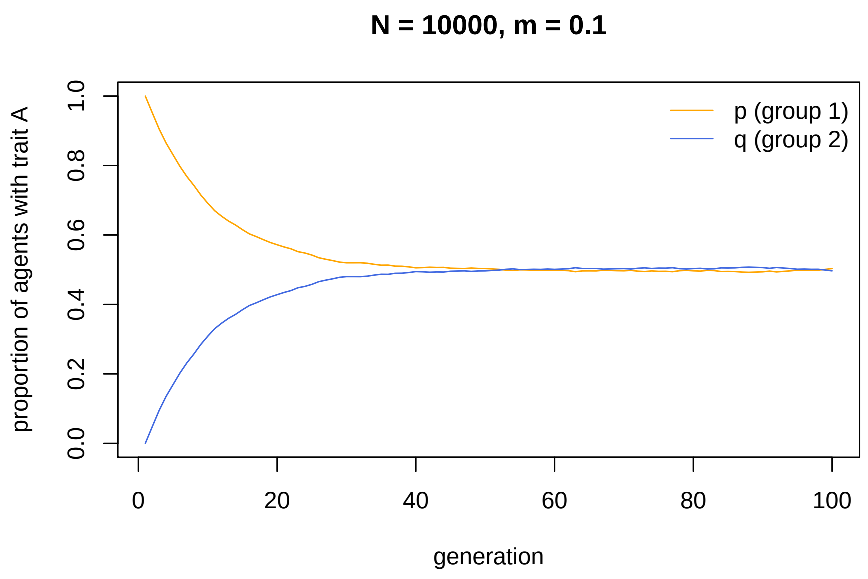
You should see here that p has decreased slightly to be less than 1, and q has increased slightly to be greater than 0. Migration seems to be bringing the frequencies closer together.

The following function simply repeats the above over t_{max} generations and plots p and q, all within a single function, similar to previous models. We omit multiple runs and r_{max}, for brevity.

```
Migration <- function (N, p_0, q_0, m, t_max) {

  # create output dataframe to hold t_max values of p and q
  output <- data.frame(p = rep(NA, t_max), q = rep(NA, t_max))

  # create first generation of group 1
  agent1 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                      prob = c(p_0,1-p_0)),
                      group = 1)

  # create first generation of group 2
  agent2 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                      prob = c(q_0,1-q_0)),
                      group = 2)

  # combine agent1 and agent2 into a single agent dataframe
  agent <- rbind(agent1,agent2)

  # store first generation frequencies
  output$p[1] <- sum(agent$trait[agent$group == 1] == "A") / N
  output$q[1] <- sum(agent$trait[agent$group == 2] == "A") / N

  for (t in 2:t_max) {

    # migration

    # 2N probabilities, one for each agent, to compare against m
    probs <- runif(1:(2*N))

    # with prob m, add an agent's trait to list of migrants
    migrants <- agent$trait[probs < m]

    # put migrants randomly into empty slots
    agent$trait[probs < m] <- sample(migrants, length(migrants))

    # store frequencies in output slot t
    output$p[t] <- sum(agent$trait[agent$group == 1] == "A") / N
    output$q[t] <- sum(agent$trait[agent$group == 2] == "A") / N

  }

  plot(x = 1:nrow(output), y = output$p,
       type = 'l',
       col = "orange",
       ylab = "proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       main = paste("N = ", N, ", m = ", m, sep = ""))

  lines(x = 1:nrow(output), y = output$q, col = "royalblue")

  legend("topright",
         legend = c("p (group 1)", "q (group 2)"),
         lty = 1,
         col = c("orange", "royalblue"),
         bty = "n")

  output  # export data from function
}
```

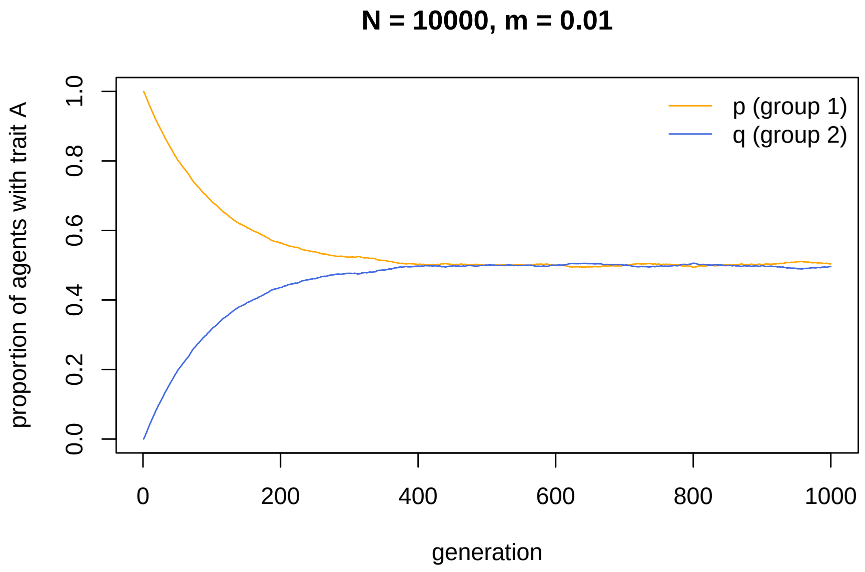
Now we can run the function with a reasonably strong migration rate of $m = 0.1$:

```
data_model7 <- Migration(N = 10000,
                         p_0 = 1,
                         q_0 = 0,
                         m = 0.1,
                         t_max = 100)
```



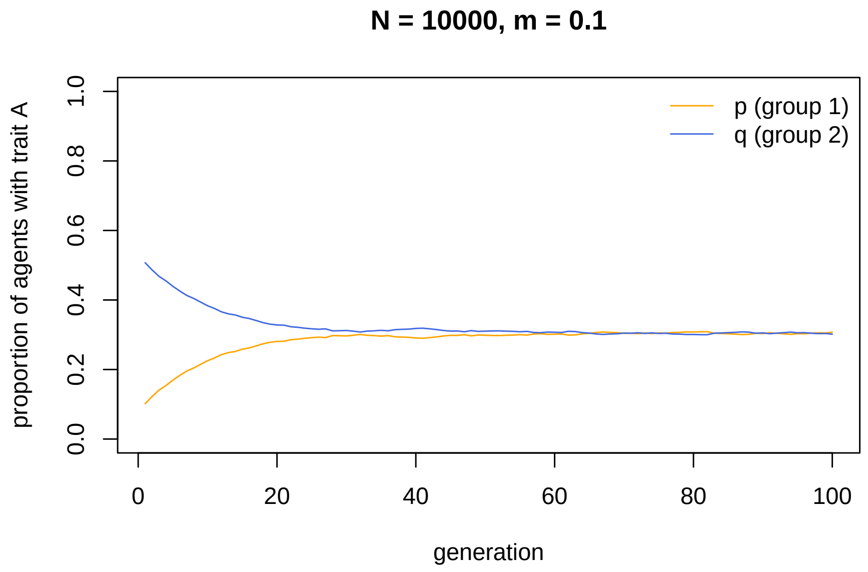
Migration causes both groups to converge on a 50:50 split of A and B, i.e. $p = q = 0.5$. Two groups that are initially entirely different in their cultural traits become identical, barring small random fluctuations. Even very small amounts of migration eventually yield between-group homogeneity:

```
data_model7 <- Migration(N = 10000,
                         p_0 = 1,
                         q_0 = 0,
                         m = 0.01,
                         t_max = 1000)
```



Changing the starting frequencies reveals that p and q do not always converge on 0.5. Rather, they converge on the average initial frequency of A across both groups (i.e. $(p_0 + q_0)/2$, which in the case below is 0.3):

```r
data_model7 <- Migration(N = 10000,
                         p_0 = 0.1,
                         q_0 = 0.5,
                         m = 0.1,
                         t_max = 100)
```



This consequence of migration, to break down between-group differences and make each group identical, is well known from population genetics. In order to maintain between-group variation in the face of even small amounts of migration, we therefore need some additional process.

In genetic evolution one such process is natural selection, if selection favours different alleles in different groups. This might happen, for example, if the groups inhabit different environments in which different traits are optimal. In Model 3 and Model 5 we saw how directly biased transmission and conformist transmission can act as forms of cultural selection. Perhaps, then, these processes can maintain between-group variation in cultural evolution.

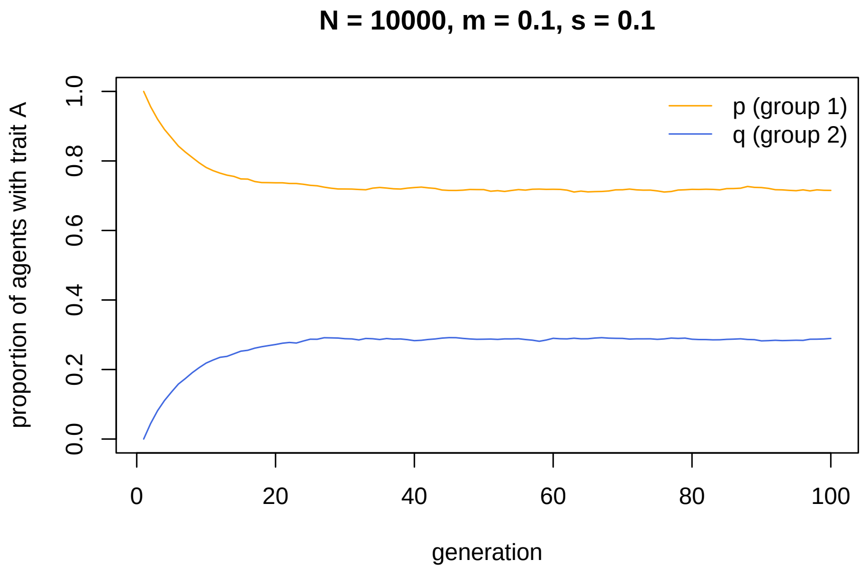
The following function adds directly biased transmission using code from the **BiasedTransmission** function of Model 3. We assume that trait A is favoured in group 1, and trait B is favoured in group 2. The parameter s determines the strength of this biased transmission / cultural selection, which we assume is equal in strength (but opposite in direction) in each group. For simplicity again we'll omit the multiple runs, hence no r_{max}.

```
MigrationPlusBiasedTransmission <- function (N, p_0, q_0, m, s, t_max) {

  # create output dataframe to hold t_max values of p and q
  output <- data.frame(p = rep(NA, t_max), q = rep(NA, t_max))

  # create first generation of group 1
  agent1 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                       prob = c(p_0,1-p_0)),
                       group = 1)

  # create first generation of group 2
  agent2 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                       prob = c(q_0,1-q_0)),
                       group = 2)

  # combine agent1 and agent2 into a single agent dataframe
  agent <- rbind(agent1,agent2)

  # store first generation frequencies
  output$p[1] <- sum(agent$trait[agent$group == 1] == "A") / N
  output$q[1] <- sum(agent$trait[agent$group == 2] == "A") / N

  for (t in 2:t_max) {

    # migration:

    # 2N probabilities, one for each agent, to compare against m
    probs <- runif(1:(2*N))

    # with prob m, add an agent's trait to list of migrants
    migrants <- agent$trait[probs < m]

    # put migrants randomly into empty slots
    agent$trait[probs < m] <- sample(migrants, length(migrants))

    # biased transmission:

    # get 2N random numbers, one per agent, each between 0 and 1
    copy <- runif(2*N)

    # group 1 favours A:
    # for each group 1 agent, pick a random agent as demonstrator and store their trait
    demonstrator_trait <- sample(agent$trait[agent$group == 1], N, replace = TRUE)
    # if demonstrator has A and with probability s, copy A from demonstrator
    agent$trait[agent$group == 1 & demonstrator_trait == "A" & copy < s] <- "A"

    # group 2 favours B:
    # for each group 1 agent, pick a random agent as demonstrator and store their trait
    demonstrator_trait <- sample(agent$trait[agent$group == 2], N, replace = TRUE)
    # if demonstrator has B and with probability s, copy B from demonstrator
    agent$trait[agent$group == 2 & demonstrator_trait == "B" & copy < s] <- "B"
```

```
    # store frequencies in output slot t
    output$p[t] <- sum(agent$trait[agent$group == 1] == "A") / N
    output$q[t] <- sum(agent$trait[agent$group == 2] == "A") / N

  }

  plot(x = 1:nrow(output), y = output$p,
       type = 'l',
       col = "orange",
       ylab = "proportion of agents with trait A",
       xlab = "generation",
       ylim = c(0,1),
       main = paste("N = ", N, ", m = ", m, ", s = ", s, sep = ""))

  lines(x = 1:nrow(output), y = output$q, col = "royalblue")

  legend("topright",
         legend = c("p (group 1)", "q (group 2)"),
         lty = 1,
         col = c("orange", "royalblue"),
         bty = "n")

  output  # export data from function
}
```

And we run the function with $s = 0.1$:

```
data_model7 <- MigrationPlusBiasedTransmission(N = 10000,
                                               p_0 = 1,
                                               q_0 = 0,
                                               m = 0.1,
                                               s = 0.1,
                                               t_max = 100)
```



Here we can see how adding cultural selection in the form of directly biased transmission maintains some degree of between-group cultural variation. Group 1 has around 70% A and 30% B, while group 2 has around 30% A and 70% B. You can play around with different values of s and m to see how the frequencies change in response. When s is large relative to m, then the groups maintain more distinctive cultural profiles.

Finally, we can see how conformist cultural transmission can also act to maintain between-group cultural variation in the face of migration. The following function integrates the original **Migration** function above with the **Conformist-Transmission** function from Model 5. Note that conformity operates *within* each group, on the assumption that individuals are interacting only with other members of their own group, and never with members of the other group.

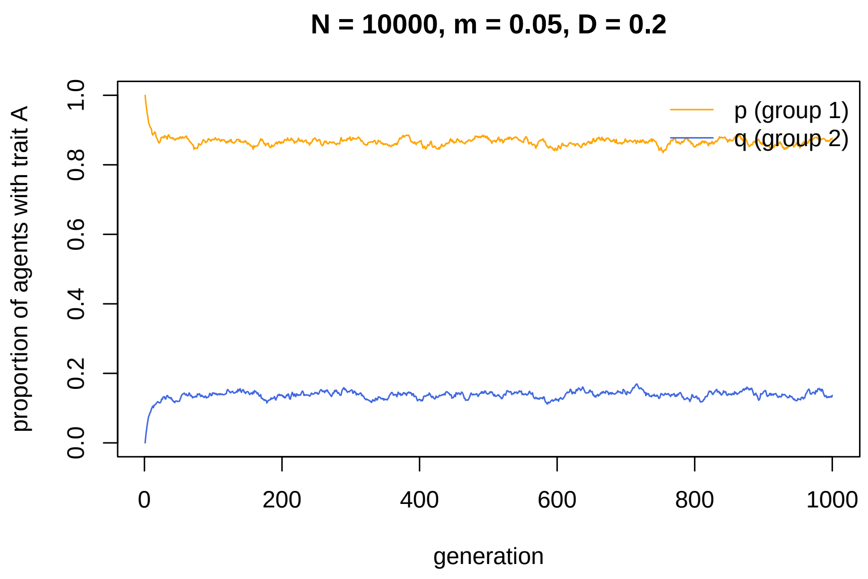
```
MigrationPlusConformity <- function (N, p_0, q_0, m, D, t_max) {

  # create output dataframe to hold t_max values of p and q
  output <- data.frame(p = rep(NA, t_max), q = rep(NA, t_max))

  # create first generation of group 1
  agent1 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                       prob = c(p_0,1-p_0)),
                       group = 1)

  # create first generation of group 2
  agent2 <- data.frame(trait = sample(c("A","B"), N, replace = TRUE,
                       prob = c(q_0,1-q_0)),
                       group = 2)

  # combine agent1 and agent2 into a single agent dataframe
  agent <- rbind(agent1,agent2)

  # store first generation frequencies
  output$p[1] <- sum(agent$trait[agent$group == 1] == "A") / N
  output$q[1] <- sum(agent$trait[agent$group == 2] == "A") / N

  for (t in 2:t_max) {

    # migration:

    # 2N probabilities, one for each agent, to compare against m
    probs <- runif(1:(2*N))

    # with prob m, add an agent's trait to list of migrants
    migrants <- agent$trait[probs < m]

    # put migrants randomly into empty slots
    agent$trait[probs < m] <- sample(migrants, length(migrants))

    # conformity in group 1:

    # create dataframe with a set of 3 randomly-picked group 1 demonstrators for each
    demonstrators <- data.frame(dem1 = sample(agent$trait[agent$group == 1], N, replac
                                dem2 = sample(agent$trait[agent$group == 1], N, replac
                                dem3 = sample(agent$trait[agent$group == 1], N, replac

    # get the number of As in each 3-dem combo
    numAs <- rowSums(demonstrators == "A")

    agent$trait[agent$group == 1 & numAs == 3] <- "A"  # for dem combos with all As, s
    agent$trait[agent$group == 1 & numAs == 0] <- "B"  # for dem combos with no As, se

    prob <- runif(N)

    # when A is a majority, 2/3
    agent$trait[agent$group == 1 & numAs == 2 & prob < (2/3 + D/3)] <- "A"
    agent$trait[agent$group == 1 & numAs == 2 & prob >= (2/3 + D/3)] <- "B"

    # when A is a minority, 1/3
    agent$trait[agent$group == 1 & numAs == 1 & prob < (1/3 - D/3)] <- "A"
    agent$trait[agent$group == 1 & numAs == 1 & prob >= (1/3 - D/3)] <- "B"
```

```
    # conformity in group 2:

    # create dataframe with a set of 3 randomly-picked group 2 demonstrators for each agent
    demonstrators <- data.frame(dem1 = sample(agent$trait[agent$group == 2], N, replace = TRUE),
                                dem2 = sample(agent$trait[agent$group == 2], N, replace = TRUE),
                                dem3 = sample(agent$trait[agent$group == 2], N, replace = TRUE))

    # get the number of As in each 3-dem combo
    numAs <- rowSums(demonstrators == "A")

    agent$trait[agent$group == 2 & numAs == 3] <- "A"  # for dem combos with all As, set to A
    agent$trait[agent$group == 2 & numAs == 0] <- "B"  # for dem combos with no As, set to B

    prob <- runif(N)

    # when A is a majority, 2/3
    agent$trait[agent$group == 2 & numAs == 2 & prob < (2/3 + D/3)] <- "A"
    agent$trait[agent$group == 2 & numAs == 2 & prob >= (2/3 + D/3)] <- "B"

    # when A is a minority, 1/3
    agent$trait[agent$group == 2 & numAs == 1 & prob < (1/3 - D/3)] <- "A"
    agent$trait[agent$group == 2 & numAs == 1 & prob >= (1/3 - D/3)] <- "B"

    # store frequencies in output slot t
    output$p[t] <- sum(agent$trait[agent$group == 1] == "A") / N
    output$q[t] <- sum(agent$trait[agent$group == 2] == "A") / N

  }

  plot(x = 1:nrow(output), y = output$p,
       type = 'l',
       col = "orange",
       ylab = "proportion of agents with trait A",
       xlab = "generation", ylim = c(0,1),
       main = paste("N = ", N, ", m = ", m, ", D = ", D, sep = ""))

  lines(x = 1:nrow(output), y = output$q, col = "royalblue")

  legend("topright",
         legend = c("p (group 1)", "q (group 2)"),
         lty = 1,
         col = c("orange", "royalblue"),
         bty = "n")

  output  # export data from function
```

```
}
```

With a moderate amount of conformity, $D = 0.2$, i.e. a 20% chance of adopting the majority trait:

```
data_model7 <- MigrationPlusConformity(N = 10000,
                                       p_0 = 1,
                                       q_0 = 0,
                                       m = 0.05,
                                       D = 0.2,
                                       t_max = 1000)
```



Here again, conformist transmission - a form of cultural selection - maintains between-group cultural variation even in the face of migration. The difference to directly biased transmission is that we do not need to assume different environments or selection pressures in the two groups. Conformity works simply on the basis of the different frequencies of the traits in the different groups.

Summary

Model 7 looked at how migration across group boundaries can break down between-group cultural variation to create a homogenous, undifferentiated mass culture where every group is culturally identical. While globalisation via the mass media or global markets has perhaps been making real-life human societies more similar to one another in recent years, we can still discern culturally distinct societies across the world, marked by traits such as dress, language, religion, psychological characteristics, cuisine and so on. Prior to the invention of mass transit and mass communication, societies would have been even more distinct. Yet migration has been a constant fixture of our species since we dispersed across the world. How then can we reconcile the extensive between-group cultural variation of our species with this frequent migration?

We modelled two potential answers to this question, both forms of cultural selection. Where different traits are favoured in different groups, then directly biased transmission (see Model 3) can maintain between-group cultural variation even in the face of migration. Similarly, conformity (see Model 5) can maintain between-group cultural variation by causing migrants to adopt the majority trait in their new society. The latter works even for neutral, arbitrary traits, which may describe well many real-life group markers. Directly and conformist biased transmission can be seen as mechanisms of *acculturation*, which describes the cultural change that may occur as a result of migration. Given that psychological processes such as conformity are unique to cultural evolution, this may also be an explanation for why there is a lot more between-group cultural variation in our species than there is between-group genetic variation (for further details and data on migration, conformity and between-group cultural variation, see Henrich & Boyd 1998; Bell et al. 2009; Mesoudi 2018; Deffner et al. 2020).

The major programming innovation in Model 7 was the introduction of two groups of agents. This allows us to simulate migration between groups, as well as processes that are likely to occur predominantly within groups such as conformity. There are many ways of extending Model 7, including adding more groups, adding non-random migration, using measures like F_{ST} to quantify the amount of between-group cultural variation, making the cultural traits cooperative / non-cooperative rather than neutral, and modelling other processes that might maintain variation such as punishment (see Mesoudi 2018).

Exercises

1. Try different values of m, p_0 and q_0 in the **Migration** function to confirm that migration causes groups to converge on the initial mean frequency of trait A across all groups, i.e. $(p_0 + q_0)/2$.

2. Try different values of m and s in **MigrationPlusBiasedTransmission**, and m and D in **MigrationPlusConformity**, to explore how strong s and D need to be relative to m to prevent population homogenisation.

3. Add a third group of N agents. As before, m is the probability that each agent moves to a random group. Where before there were two groups to randomly select, now there are three. Show that migration still causes convergence on the initial mean frequency of trait A across all three groups.

4. Change the function to allow a user-defined number of groups. The function should take the number of groups as a parameter, and create this number of groups on-the-fly. Either allow the user to specify the starting frequencies of A across all groups (e.g. by having a parameter that can take sets of values such as *c(0.3,0.5,0.8,0.9)*), or make the starting values random. Show that migration still causes convergence on the initial mean frequency of trait A across all groups.

5. Make migration non-random, such that agents are more likely to migrate to groups that have a higher frequency of trait A. How do the dynamics of non-random migration compare to those of the random migration simulated in the original **Migration** function?

Analytic Appendix

Let's derive the recursions and equilibrium frequency for the basic migration model, to understand analytically why the frequency of A in each group converges on the initial average frequency of A across the two groups.

Recall that the frequency of A in generation t in the first group is p_t and in the second group is q_t. The frequency of A in the previous generation is denoted p_{t-1} and q_{t-1} respectively, and in the first generation it is p_0 and q_0 respectively. Let's consider group 1 first. We want to write an expression for p_t, in terms of p_{t-1}, the migration rate m, and any other parameter that is necessary.

In generation t, $1-m$ individuals in group 1 do not migrate. The frequency of A amongst these $1-m$ individuals will therefore be the same as in the previous generation in group 1, i.e. p_{t-1}. This gives an overall frequency for these group 1 non-migrants of $p_{t-1}(1-m)$.

The other m individuals in group 1 are migrants. These migrants are drawn from the entire population, which in this case is groups 1 and 2 combined. The frequency of A in the entire population is the average frequency of A across the two groups, which we will denote $\bar{x}$. The frequency of A amongst the m migrants in generation t in group 1 will therefore be $\bar{x}m$. What is the value of $\bar{x}m$? Because there is no mutation or drift in this model, new As never appear, and existing As never disappear, when considering the entire population. Hence the total frequency of A in the entire population will always remain the same, from the first generation to the last. This will be $\bar{x} = (p_0+q_0)/2$, i.e. the average frequency of A across the two groups in the first generation. This means that $\bar{x}$ is a constant, i.e. it never changes from one generation to the next.

Putting these together, the frequency of A in generation t in group 1, p_t, will be

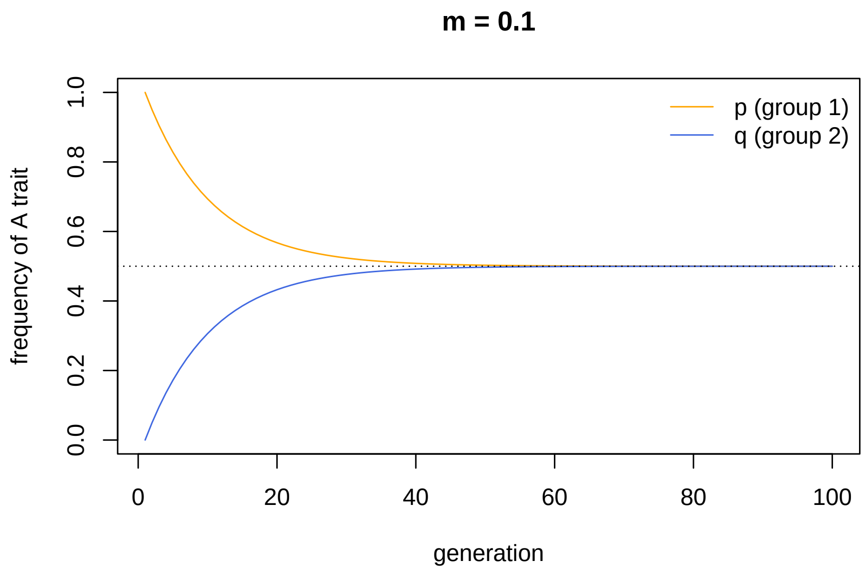
$$p_t = p_{t-1}(1 - m) + \bar{x}m \tag{7.1}$$

Equivalently, the frequency of A in generation t in group 2, q_t, will be

$$q_t = q_{t-1}(1 - m) + \bar{x}m \tag{7.2}$$

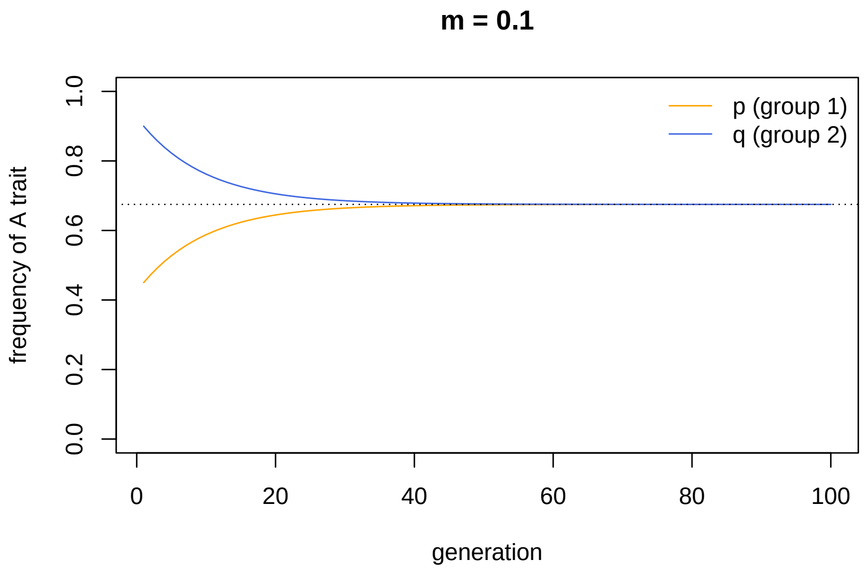
Now we have two recursions, and we can plot them.

```
MigrationRecursion <- function(m, t_max, p_0, q_0) {

  p <- rep(0,t_max)
  p[1] <- p_0

  q <- rep(0,t_max)
  q[1] <- q_0

  for (i in 2:t_max) {
    x_bar <- (p[i-1] + q[i-1]) / 2

    p[i] <- p[i-1]*(1-m) + x_bar*m

    q[i] <- q[i-1]*(1-m) + x_bar*m
  }

  plot(x = 1:t_max, y = p,
       type = "l",
       ylim = c(0,1),
       ylab = "frequency of A trait",
       xlab = "generation",
       col = "orange",
       main = paste("m = ", m, sep = ""))

  lines(x = 1:t_max, y = q, col = "royalblue")

  abline(h = (p_0 + q_0) / 2, lty = 3)

  legend("topright",
         legend = c("p (group 1)", "q (group 2)"),
         lty = 1,
         col = c("orange", "royalblue"),
         bty = "n")

}

MigrationRecursion(m = 0.1, t_max = 100, p_0 = 1, q_0 = 0)
```



This looks almost identical to the first plot we created above from the simulations. I added a dotted line denoting the initial mean frequency of A in the entire population, $\bar{x}$, to verify that the two groups converge on this mean. We can change the starting values to provide a further check:

```
MigrationRecursion(m = 0.1, t_max = 100, p_0 = 0.45, q_0 = 0.9)
```



To understand better why this convergence occurs, we need to solve these recursions. Hartl & Clark (1997, p.168) provide a trick for solving such equations, where the recursion can be expressed in the form $p_t - X = (p_{t-1} - X)Y$, where X and Y are constants. Rearranging this expression gives $p_t = p_{t-1}Y - XY + X = p_{t-1}Y - X(Y+1)$. This fits equation 7.1 if we say that $Y = 1 - m$ and $X = \bar{x}$. Hence we can rewrite equation 7.1 as:

$$p_t - \bar{x} = (p_{t-1} - \bar{x})(1-m) \tag{7.3}$$

Because the relation between p_{t-1} and p_{t-2} is the same as that between p_t and p_{t-1}, all the way back to p_0, the solution to equation 7.3 is

$$p_t - \bar{x} = (p_0 - \bar{x})(1-m)^t \tag{7.4}$$

After many generations, when t becomes very large, then the last term on the right which is raised to the power of t will be approximately zero. Even if m is very small, and $(1-m)$ is very close to 1, any number less than 1 raised to a large power (i.e. multiplied by itself many times) will be approximately zero. Consequently the whole right hand side of equation 7.4 becomes zero, and p_t remains the same generation after generation. So we can set the right-hand side of equation 7.4 to zero and rearrange to find this equilibrium value, p^*:

$$p^* = \bar{x} \tag{7.5}$$

And as we noted at the beginning, $\bar{x} = (p_0 + q_0)/2$, i.e. the average of the two starting frequencies of A in the two groups. This is what we found in the agent-based simulations as well as the recursion simulations in this section.

References

Bell, A. V., Richerson, P. J., & McElreath, R. (2009). Culture rather than genes provides greater scope for the evolution of large-scale human prosociality. Proceedings of the National Academy of Sciences, 106(42), 17671-17674.

Deffner, D., Kleinow, V., & McElreath, R. (2020). Dynamic social learning in temporally and spatially variable environments. Royal Society Open Science, 7(12), 200734.

Hartl, D. L., Clark, A. G. (1997). Principles of population genetics. Sunderland, MA: Sinauer associates.

Henrich, J., & Boyd, R. (1998). The evolution of conformist transmission and the emergence of between-group differences. Evolution and Human Behavior, 19(4), 215-241.

Mesoudi, A. (2018). Migration, acculturation, and the maintenance of between-group cultural variation. PLOS ONE, 13(10), e0205573.

Model 8: Blending inheritance

Introduction

When Darwin wrote *The Origin of Species*, it was typically thought that biological inheritance involved the blending of parental traits. For example, the offspring of a tall and a short individual would have intermediate height, the average (or blend) of its two parents. This presented a problem for Darwin's theory, one which Darwin himself acknowledged. If traits blend together, then they are unlikely to persist for long enough for selection to act on them. All traits will quickly converge on a blended average, destroying the variation that is necessary for evolution.

This puzzle was resolved when Mendel's famous pea plant experiments were rediscovered around the turn of the 20th century. These showed that genetic inheritance is actually particulate, not blending. This means that inheritance involves the passing of discrete particles of information - what became known as 'genes' - in an all-or-nothing fashion from parent to offspring. Some simple phenotypic traits, like Mendel's pea plant varieties, show this directly. A white plant crossed with a purple plant gives rise to either a white or purple offspring, not a blend. Complex phenotypic traits like human height, which appear to 'blend' in offspring, are actually determined by many discrete genes.

A common objection to cultural evolution follows similar lines. Cultural traits, it is argued, are often continuous, and appear to blend in learners. Someone who grew up in Britain before moving to the States might have a blended 'transatlantic' accent. Someone with a liberal father and conservative mother might end up with moderate political views. If cultural inheritance is blending, then the same objection as that levied at Darwin above should also apply: variation will be rapidly lost, and evolution can't operate. This is a common specific criticism of memetics, which assumes that there are discrete particles of inheritance analogous to genes.

One problem with this objection is that it is not at all clear that cultural inheritance *is* blending. We don't know enough about how brains store and receive information to say with certainty that there is not a discrete, particulate inheritance system underlying what appears to be blending at the 'phenotypic' level, just like biological traits such as height appear to blend but are actually determined by an underlying particulate inheritance system. Another problem is that the objection cannot be true: there is huge cultural variation in the world that patently does exist, and upon which selection can and does act. So even if cultural inheritance is blending, perhaps some other feature of cultural evolution means that blending does not have the problematic consequences described above. Either way, it is worth exploring the case of blending inheritance with formal models to go beyond verbal arguments. Indeed, it was not until the formal population genetic models of R.A. Fisher and others in the early 20th century that Mendelian genetic inheritance and Darwinian evolution were definitively shown to be consistent with one another.

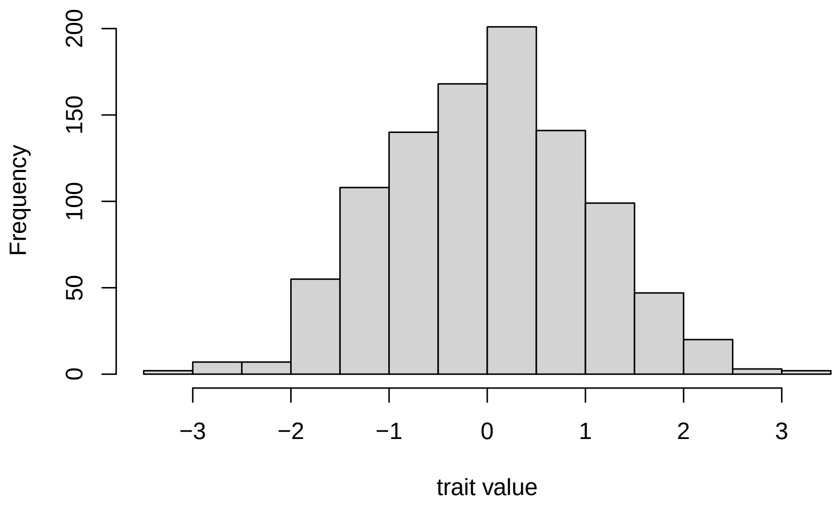
Model 8a: Blending inheritance

In Model 8a we will simulate blending cultural inheritance, inspired by a formal mathematical model presented by Boyd & Richerson (1985). We assume a population of N individuals. Each of these individuals possesses a value of a continuously varying cultural trait. While previous models have assumed discrete traits that can take on one of two forms (A or B), in this case we are modelling a trait that can take any value on a continuous scale. Many cultural traits are continuous, from handaxe length to political orientation.

In the first generation / timestep, we assume that trait values are drawn randomly from a standard normal distribution. This is a symmetrical, bell-shaped distribution with mean 0 and standard deviation 1. It does not really matter what distribution or parameter values we use here. However, lots of naturally-occurring cultural traits are normally distributed. We might think of political orientation, where some people are on the extreme left or extreme right, and most are somewhere in the middle.

The **rnorm** command generates random numbers from this standard normal distribution, like this:

```r
N <- 1000
values <- rnorm(N)
hist(values, main = "", xlab = "trait value")
```



Hence this will be our initial distribution of trait values in the first generation. Then, in each new generation the N individuals are replaced with N new individuals. Each of these new agents picks n agents from the previous timestep at random and adopts the mean trait value of these n demonstrators. We assume that $n > 1$ because we cannot really take a blended mean of a single trait value, and we assume that $n \leq N$ because learners can only learn from a previous generation agent once.

In each timestep we track the mean trait value to see whether blending inheritance generates directional cultural change. We also track the variance of the trait across the entire population, in order to see whether and when blending inheritance destroys variation as per the objection above.

Below I have taken the skeleton of Model 1's **UnbiasedTransmission** function, which has multiple independent runs and keeps track of and plots mean trait frequency, and adapted it to create a **BlendingInheritance** function. Examine the code before reading the explanation of the changes afterwards.

```
BlendingInheritance <- function (N, n, t_max, r_max) {

  # create a matrix for trait means with t_max rows and r_max columns,
  # fill with NAs, convert to dataframe
  trait_mean <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(trait_mean) <- paste("run", 1:r_max, sep="")
```

```
# same for holding trait variance
trait_var <- as.data.frame(matrix(NA, t_max, r_max))
names(trait_var) <- paste("run", 1:r_max, sep="")

for (r in 1:r_max) {

  # create first generation, N random numbers from a standard normal distribution
  agent <- rnorm(N)

  # add first generation's mean to first row of column r
  trait_mean[1,r] <- mean(agent)

  # add first generation's variance to first row of column r
  trait_var[1,r] <- var(agent)

  for (t in 2:t_max) {

    # create matrix with N rows and n columns,
    # fill with traits from random members of agent
    m <- matrix(sample(agent, N*n, replace = TRUE), N, n)

    # create new generation by taking rowMeans, i.e. mean of n demonstrators,
    # to implement blending inheritance
    agent <- rowMeans(m)

    # get mean trait value and put it into output slot for this generation t and run
    trait_mean[t,r] <- mean(agent)

    # get trait variance and put it into output slot for this generation t and run r
    trait_var[t,r] <- var(agent)

  }

}

# create two plots, one for means and one for variances
par(mfrow=c(1,2)) # 1 row, 2 columns

# plot a thick line for the mean mean of all runs
plot(rowMeans(trait_mean),
     type = 'l',
     ylab = "trait mean",
     xlab = "generation",
     ylim = c(min(trait_mean,-1), max(trait_mean,1)),
     lwd = 3,
       main = paste("N = ", N, ", n = ", n, sep = ""))

  # add lines for each run, up to r_max
  for (r in 1:r_max) {
    lines(trait_mean[,r], type = 'l')
  }

  # plot a thick line for the mean variance across all runs
  plot(rowMeans(trait_var),
       type = 'l',
       ylab = "trait variance",
       xlab = "generation",
       ylim = c(0, max(trait_var)),
       lwd = 3,
       main = paste("N = ", N, ", n = ", n, sep = ""))

  # add lines for each run, up to r_max
  for (r in 1:r_max) {
    lines(trait_var[,r], type = 'l')
  }

  # export data from function
  list(final_agent = agent, mean = trait_mean, variance = trait_var)
}
```

First, rather than a single *output* dataframe, we create two, one called *trait_mean* to hold the mean trait value over all timesteps for all runs, and one called *trait_variance* to hold the variance of the trait over all timesteps and all runs. Then, in each run, we create an initial population which has N agents each with a trait value drawn randomly from a standard normal distribution using the **rnorm** command, as shown above. We then record the initial mean and variance in the appropriate slots of *trait_mean* and *trait_variance* respectively.

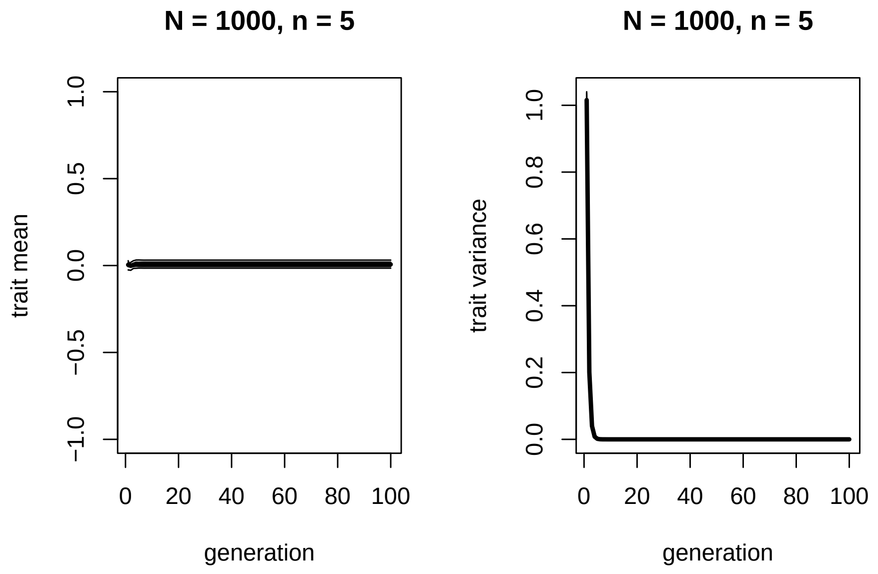
Within the timestep loop, we then create a matrix with N rows and n columns, and fill it with trait values from agents randomly selected using the **sample** command. Each row here represents a new agent, and each column of that row contains the trait values of n randomly chosen demonstrators, just like for horizontal transmission in Model 6c. We then use the **rowMeans** command to get the means of each row of this matrix. This is the blending inheritance rule. This mean (or blend) of the n demonstrators is put into the *agent* dataframe, over-writing the previous generation's traits, and the new mean and variance are added to the output dataframes.

The subsequent plotting code now makes two plots side-by-side, one for the means and one for the variances. This is done with the **par(mfrow([rows],[cols]))** command, where [rows] is the number of rows and [cols] the number of columns. Here I've set one row and two columns. The actual plots use the same code as in previous models, with means across all runs plotted with thick lines and separate lines for each run. We do this twice, once for means, and once for variances.

Finally, rather than outputting a single dataframe, we now output three dataframes after the function is run: the *final_agent* dataframe, which allows us to explore the distribution of trait values at the very end of the simulation, and the *trait_mean* and *trait_variance* dataframes which contain the means and variances. We do this by exporting a list, which contains three dataframes.

Let's run this function with a reasonably large N and small n:

```
data_model8 <- BlendingInheritance(N = 1000,
                                   n = 5,
                                   t_max = 100,
                                   r_max = 5)
```



The left-hand plot shows that the mean across all runs remains at roughly 0, which is the mean of the initial standard normal distribution. Blending inheritance does not change mean trait value over time. It is non-directional, just like unbiased mutation or unbiased transmission. This is to be expected, as all we are doing is taking the mean of random draws from the previous generation's trait distribution, which will be the same as the mean in the previous generation.

The right-hand plot shows that the variance quickly drops to zero. Again as we expected, blending inheritance acts to reduce variation to zero. We can confirm this by displaying some of the values of the final agent dataframe, to confirm that every agent has exactly the same trait:

```r
head(data_model8$final_agent)
```

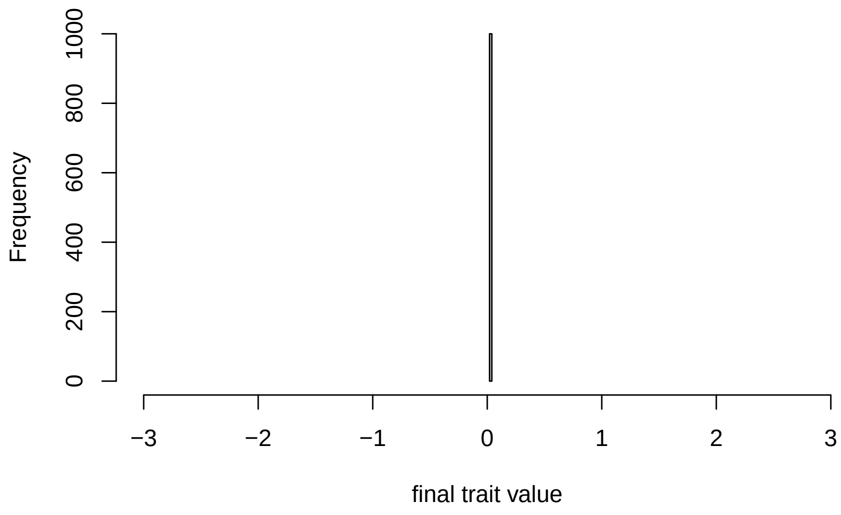
```
## [1] 0.03113359 0.03113359 0.03113359 0.03113359 0.03113359 0.03113359
```

```r
tail(data_model8$final_agent)
```

```
## [1] 0.03113359 0.03113359 0.03113359 0.03113359 0.03113359 0.03113359
```

We can also plot the final trait distribution on the same scale as the graph above showing the initial standard normal distribution, confirming that all of the initial variation has been lost.

```r
hist(data_model8$final_agent, xlim = c(-3,3), main = "", xlab = "final trait value")
```



Model 8b: Blending inheritance with mutation

All we have done so far is confirm the intuition that blending inheritance destroys variation, just as Darwin's critics originally argued. If we assume that cultural traits in real life blend (which, as noted above, is far from certain), then it is hard to reconcile this conclusion with the observation of extensive cultural variation in real life.

However, we must remember that other aspects of cultural evolution may also differ from genetic evolution. Boyd & Richerson (1985) pointed out that blending inheritance is potentially problematic in biological evolution because rates of genetic mutation are low, and so cannot replenish the variation that blending inheritance destroys. Yet rates of cultural mutation may be much higher. In Model 8b we will follow their example and add unbiased cultural mutation, or copying error, to the blending inheritance model.

Previously we assumed that new agents could copy the traits of n agents from the previous timestep with no error, and then take the mean of these 100% accurate trait values. Now we assume that each of the n cultural traits are copied with some error, similar to unbiased cultural mutation in Model 2. So before the blended mean is taken, random error is added to each copied trait value.

We do this by again using the **rnorm** command to draw randomly from a normal distribution. This time, the normal distribution has a mean of the copied trait values, which are stored in the m matrix, and the random error or deviation is introduced to each one using a new parameter e. This is the variance of this normal distribution. The larger is e, the more error or mutation there is in the estimates of each of the n traits. Note that each of the n copied traits are subject to independent error. Some might be copied quite faithfully, others less faithfully. Blending inheritance then proceeds as before, but now it is the mean of the modified trait values.

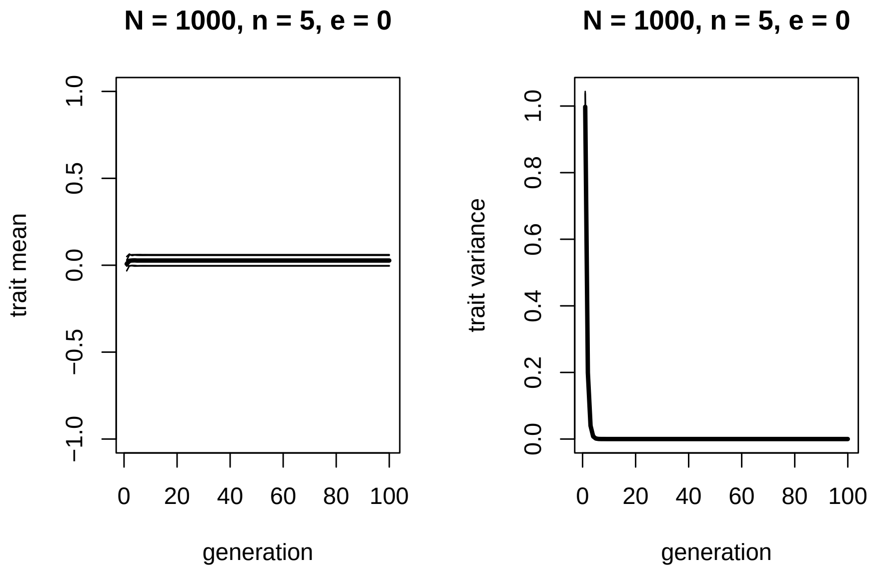
The following code modifies the **BlendingInheritance** function above by adding a parameter e, and adding a line after the trait copying in which the m matrix is modified according to e. Note that because e is the variance of the error distribution (following Boyd & Richerson 1985), and the **rnorm** function takes a standard deviation, we need to give the function the square root of e. We also add e to the plot title.

```
BlendingInheritance <- function (N, n, e, t_max, r_max) {

  # create a matrix for trait means with t_max rows and r_max columns,
  # fill with NAs, convert to dataframe
  trait_mean <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(trait_mean) <- paste("run", 1:r_max, sep="")

  # same for holding trait variance
  trait_var <- as.data.frame(matrix(NA, t_max, r_max))
  names(trait_var) <- paste("run", 1:r_max, sep="")
```

```
for (r in 1:r_max) {

  # create first generation, N random numbers from a standard normal distribution
  agent <- rnorm(N)

  # add first generation's mean to first row of column r
  trait_mean[1,r] <- mean(agent)

  # add first generation's variance to first row of column r
  trait_var[1,r] <- var(agent)

  for (t in 2:t_max) {

    # create matrix with N rows and n columns,
    # fill with traits from random members of agent
    m <- matrix(sample(agent, N*n, replace = TRUE), N, n)

    # add random error to each demonstrator value, with variance e
    m <- matrix(rnorm(N*n, mean = m, sd = sqrt(e)), N, n)

    # create new generation by taking rowMeans, i.e. mean of n demonstrators,
    # to implement blending inheritance
    agent <- rowMeans(m)

    # get mean trait value and put it into output slot for this generation t and run r
    trait_mean[t,r] <- mean(agent)

    # get trait variance and put it into output slot for this generation t and run r
    trait_var[t,r] <- var(agent)

  }

}

# create two plots, one for means one for variances
par(mfrow=c(1,2)) # 1 row, 2 columns

# plot a thick line for the mean mean of all runs
plot(rowMeans(trait_mean),
     type = 'l',
     ylab = "trait mean",
     xlab = "generation",
     ylim = c(min(trait_mean,-1), max(trait_mean,1)),
     lwd = 3,
     main = paste("N = ", N, ", n = ", n, ", e = ", e, sep = ""))
  # add lines for each run, up to r_max
  for (r in 1:r_max) {
    lines(trait_mean[,r], type = 'l')
  }

  # plot a thick line for the mean variance across all runs
  plot(rowMeans(trait_var),
       type = 'l',
       ylab = "trait variance",
       xlab = "generation",
       ylim = c(0, max(trait_var)),
       lwd = 3,
       main = paste("N = ", N, ", n = ", n, ", e = ", e, sep = ""))

  # add lines for each run, up to r_max
  for (r in 1:r_max) {
    lines(trait_var[,r], type = 'l')
  }

  # export data from function
  list(final_agent = agent, mean = trait_mean, variance = trait_var)
}
```

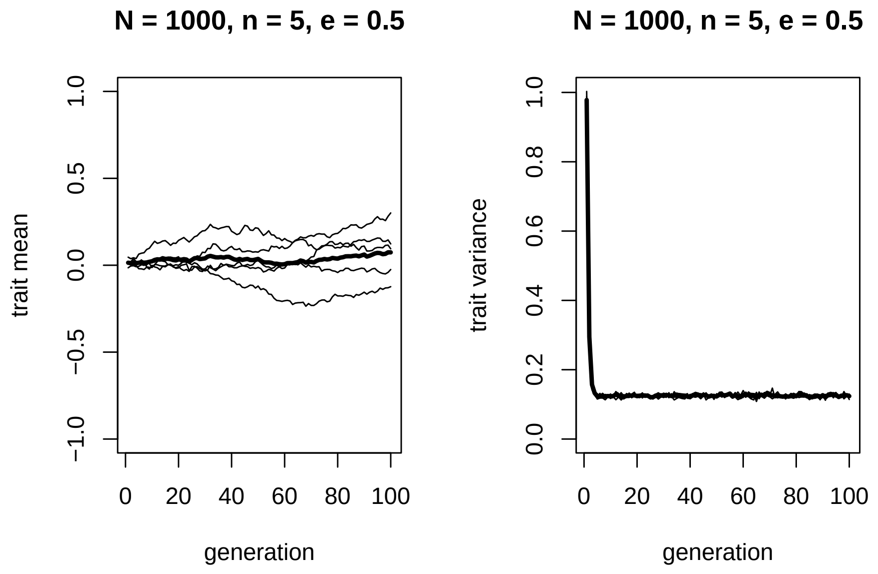
First we can run this code with $e = 0$, to confirm that this matches the original function which had no error.

```
data_model8 <- BlendingInheritance(N = 1000,
                                   n = 5,
                                   e = 0,
                                   t_max = 100,
                                   r_max = 5)
```



As before, the mean does not change, and variation decreases to zero. Now let's increase e:

```
data_model8 <- BlendingInheritance(N = 1000,
                                   n = 5,
                                   e = 0.5,
                                   t_max = 100,
                                   r_max = 5)
```



There are two changes here. First, in the left hand plot we can see that while the mean of the mean trait value still does not change and remains around zero, there is more variation around this line in the different runs. Consistent with this, the right hand plot shows that while the variance still drops, it does not drop to zero. For $n = 5$ and $e = 0.5$, the variance converges on around 0.125. We can see this more exactly by displaying the mean variances for each run over the last 50 timesteps (to avoid the initial value from skewing the estimate):

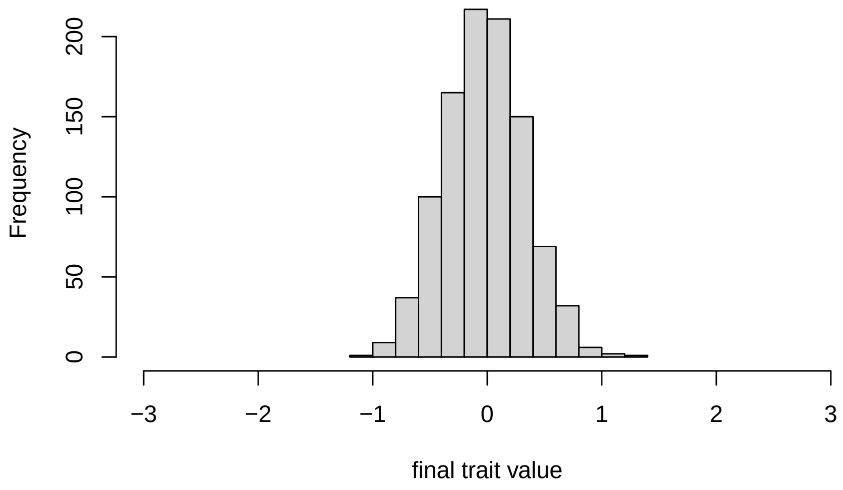
```
colMeans(data_model8$variance[50:100,])
```

```
##      run1      run2      run3      run4      run5 
## 0.1258589 0.1242682 0.1254471 0.1256430 0.1252636
```

Try playing around with n and e to show that the equilibrium amount of variance increases as e increases and as n decreases. In the analytical appendix we will see how to calculate this equilibrium value from n and e.

Finally, we can plot the trait distribution in the final generation.

```
hist(data_model8$final_agent, xlim = c(-3,3), main = "", xlab = "final trait value")
```



While the first plot above showed that the initial distribution ranged from around -3 to 3, this final distribution ranges from around -1 to 1. So while there is less variation than in the first generation, there is still variation, and it still resembles a normal distribution.

Summary

A common objection to cultural evolution is that cultural inheritance blends different traits together, destroying the variation that is necessary for evolution to operate. While the claim that cultural inheritance is blending can be contested, and obviously it is not the case that there is no cultural variation in the real world, we can still use formal models to explore these claims.

Model 8a confirmed that blending inheritance destroys variation, as expected. However, Model 8b showed that this can be countered by cultural mutation. While genetic mutation is very infrequent, cultural mutation is probably much more common. People typically misremember facts, distort stories, and modify tools as they attempt to copy others. So even if cultural inheritance does take the form of blending, it is quite plausible that cultural mutation is potent enough to re-introduce variation that blending destroys. While we did not model this here, a similar argument can be made for cultural selection. While natural selection tends to be weak, cultural selection may be much stronger. So even if blending inheritance reduces variation, strong cultural selection can still act before the variation is depleted.

There are also a few programming innovations introduced in Model 8. We modelled a continuously varying cultural trait, rather than a discrete trait that could take one of two values as in previous models. This means we had to specify the distribution of the continuous trait, and track the mean and variance of that distribution, rather than a trait frequency. We used the command **rnorm** to draw random trait values from a normal distribution in order to create the initial generation's trait values. We also used **rnorm** to simulate random error or mutation in the transmission of the continuous trait. The copied trait value is set as the mean of the normal distribution, with the error around this value set by the standard deviation (or variance) of the normal distribution. Finally, we saw how to create two plots side-by-side using the **par(mfrow=c([rows],[cols]))** command.

Exercises

1. Try different values of N and n in **BlendingInheritance** to see how blending inheritance reduces variation to zero whenever $n > 1$.

2. Try different values of n and e to show that the former increases the blending effect, and the latter reduces it.

3. Forgetting about blending for now, combine Model 3 and Model 8 to create a function that models selection (i.e. directly biased transmission, from Model 3) on a continuous cultural trait (as implemented in Model 8). Instead of s being the probability of switching to trait A upon encountering a demonstrator with that trait as in Model 3, instead assume that there is a particular value of the continuous trait that is particularly attractive, memorable or intuitive. Allow the user to set this value when calling the function. For example, it might be zero (the mean of the starting normal distribution), or a different value (say +3, or -2). Then, each agent picks a random member of the previous generation, and if the demonstrator's value is closer to the favoured value, and with probability s (as in Model 3), then the copying agent adopts the demonstrator's value. Otherwise they do not copy and retain their previous value. Run the simulation to see what happens to the mean and variance over time.

4. Modify the function you just created to allow selection for two different trait values, rather than just one. Now, with probability s, each agent picks a random demonstrator from the previous generation, and if the demonstrator's value is closer to either of the two favoured values, then the

agent adopts that closest value. Run the simulation to see what happens to the mean and variance over time.

5. Add blending inheritance to the function you just created, using the code from Model 8. Use a parameter to switch blending on (e.g. *blend = TRUE*) or off (e.g. *blend = FALSE*). With $e = 0$, explore under what conditions selection (via s) prevents blending from destroying variation.

Analytic Appendix

The simulation model presented above recreates an analytical model presented in Chapter 3 of Boyd & Richerson (1985), and we replicate their conclusions. They also include assortative transmission, which has a similar effect to mutation: if the n demonstrators have similar cultural traits, then blending inheritance is less effective at reducing variation. Box 3.22 in Boyd & Richerson (1985) provides a proof that the mean trait value after blending inheritance, $\bar{X}'$, equals the mean trait value before blending inheritance, $\bar{X}$. Their equation 3.28 does the same for variance under the assumption of transmission error, where variance after blending inheritance, V', is given by:

$$V' = (1/n)(V + \bar{E}) \tag{8.1}$$

where $\bar{E}$ is the mean value of e across all demonstrators, and e is defined as in the simulation model. Because e is a constant across all demonstrators, and the errors are independent, $\bar{E}$ will equal e. To get the equilibrium variance, $\hat{V}$, we can set $V' = V$ in the above equation and rearrange to find:

$$\hat{V} = \frac{e}{n-1} \tag{8.2}$$

For the $e = 0.5$ and $n = 5$ simulated above, this gives $\hat{V} = 0.5/4 = 0.125$, as was found in the simulation. This equation confirms that the variation remaining after blending inheritance increases with e and decreases with n.

References

Boyd, R., & Richerson, P. J. (1985). Culture and the evolutionary process. University of Chicago Press.

Model 9: Demography and cultural gain / loss

Introduction

Demography refers to the study of populations: their size, structure, and movements in and out. We have already examined demography in previous models. This was most explicit in Model 7 where we looked at migration between two separate sub-populations. However, in all of the models so far, we have seen how small populations are more likely to lose traits purely by chance, while large populations have more stable dynamics that more closely match analytical models with infinite population sizes.

Here we will pursue this further by examining an influential model created by Henrich (2004) that links demography - specifically, population size, N - to cultural losses and cultural gains. Henrich (2004) did this in the context of an archaeological example. Around 12,000 years ago, Tasmania became cut off from the mainland of Australia due to rising sea levels. Upon first contact with European explorers, the Aboriginal Tasmanians had strikingly simple technology compared to the Aborigines of the mainland: they lacked bone tools, fishhooks, traps and nets, spearthrowers and boomerangs, all of which were used on the mainland. Henrich argued that this technological loss or stagnation was not because the Tasmanians were less smart or skilful than their mainland counterparts. Rather, it was because when they were cut off from the mainland their population size dropped dramatically. Tools such as fishhooks are hard to make, and subject to error. When there are few skilled fishhook-makers, fishhook technology is easily lost and unlikely to be reinvented. In large populations, however, there will be lots of skilled fishhook makers. Even if some forget or fail to pass on their fishhook-making skill, there will be others to carry on the tradition. This is an example of cumulative cultural evolution, where individuals acquire skills and knowledge from others that they could not invent on their own. While Henrich framed his model in the context of prehistoric Tasmania, we can generalise this principle to any case where hard-to-learn traits are dependent on population size.

Model 9

Henrich (2004) used a simple model to explore this scenario of cultural loss due to small population size. We assume N individuals who each possess a value of a continuously varying, culturally transmitted skill (e.g. fishhook making). This skill is denoted z_i for individual i, where i ranges from 1 to N. Note that, technically, N is the 'effective' population size. This is the number of individuals who would be able to possess the trait in question. It may be less than the actual ('census') population size if, for example, infants or some other class of individual do not or cannot make fishhooks.

In each new generation, every individual copies the z value of the most highly skilled individual from the previous generation, denoted z_h. This is a form of biased transmission or cultural selection, similar to the payoff-based indirectly-biased transmission explored in Model 4. Here, demonstrators are preferentially selected based on their superior culturally-transmitted skill or knowledge.

If individuals perfectly copied the highest trait value from the previous generation, then the model would not be very interesting. It would also not be very realistic. Instead, we assume that this indirectly biased transmission is imperfect. It is imperfect in two ways. First, there is skill loss due to mistakes in the copying process. This always reduces the copier's z value relative to the target z_h. Second, there are experiments, guesses and inferences. These sometimes result in worse skill, but sometimes better skill than the target z_h.

To formalise these two kinds of variation, Henrich (2004) assumed that naive learners' skill z_i is drawn from a gumbel distribution. This is similar to the blending inheritance of Model 8 where we drew copied values from a normal distribution. A gumbel distribution is generated when extreme values are repeatedly picked. This is what we are doing when we pick the highest-skilled demonstrator to learn from in each generation, so it is more appropriate than a normal distribution.

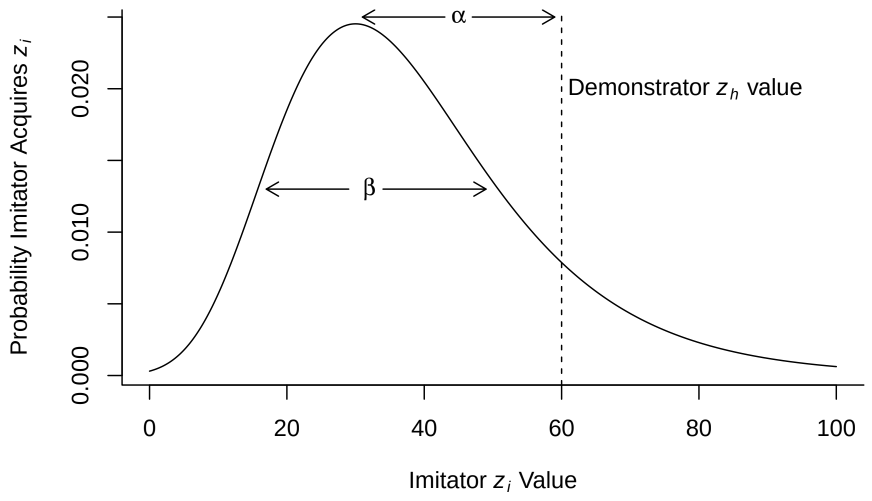
Unfortunately R does not provide built-in functions to work with gumbel distributions. However, we can write our own, using the formula for the probability density of the distribution. The following functions, **dgumbel** and **rgumbel**, generate the probability density and random draws for a gumbel distribution. I won't go into details on the formulae, but you can look them up if you want.

```r
# probability density of the gumbel, for values x
# a is the mode, beta is the dispersion
# defaults to standard gumbel with a=0 and beta=1
dgumbel <- function(x, a = 0, beta = 1) {
  b <- (x - a)/beta
  (1/beta)*exp(-(b+exp(-b)))
}
```

```
# n random draws from a gumbel distribution with mode a and dispersion beta
# defaults to standard gumbel with a=0 and beta=1
rgumbel <- function(n, a = 0, beta = 1) {
  a + beta * (-log(-log(runif(n))))
}
```

Gumbel distributions take two parameters, a mode a and a dispersion β. Here is an example gumbel distribution, with additional labels showing Henrich's model parameters (recreating Fig 1 in Henrich 2004).

```
x <- seq(0,100,0.1)  # x-axis values
a <- 30  # mode
beta <- 15  # dispersion

# plot gumbel distribution for the parameters defined above
plot(x, dgumbel(x, a, beta),
     type = 'l',
     ylab = expression("Probability Imitator Acquires "*italic("z"[i])),
     xlab = expression("Imitator "*italic("z"[i])*" Value"),
     bty='l')

# add a vertical dotted line for the demonstrator with z_h
alpha <- 30
abline(v = a + alpha, lty=2)
text(a + alpha + 18, 0.02,
     labels = expression("Demonstrator "*italic("z"[h])*" value"))

# add beta label and arrows
text(a+2, 0.013,
     labels = expression(beta))
arrows(a+4, 0.013, a+19, 0.013, length=0.1)
arrows(a-1, 0.013, a-13, 0.013, length=0.1)

# add alpha label and arrows
text(a+alpha/2, 0.025,
     labels = expression(alpha))
arrows(a+2+alpha/2, 0.025, a+14+alpha/2, 0.025, length=0.1)
arrows(a-2+alpha/2, 0.025, a-14+alpha/2, 0.025, length=0.1)
```



Notice how the gumbel distribution looks like an asymmetric normal distribution which is elongated at the upper end. This reflects the fact that we are sampling extreme high values. A demonstrator z_h value is shown on the plot with a dotted vertical line. This is the highest-skill value that the imitator is trying to copy. Their actual copied z_i value is drawn randomly from this gumbel distribution.

The α parameter determines the first kind of deviation described above, the transmission error. This is the difference between the demonstrator z_h value and the mode of the gumbel distribution (which is a - try not to confuse the mode a with the transmission error parameter α. It's not ideal notation, but I'll stick with it because that's what Henrich used. z_h is therefore $a + \alpha$). The larger is α, the further the mode will be below the z_h value, and the more likely the copied z_i value will be less than z_h.

The β parameter is the dispersion of the gumbel distribution, controlling the width. This represents the second kind of deviation described above, the inferential guesses or experiments. The larger is β, the more likely the copied z_i value will be different to the copied z_h value. Occasionally, the copied z_i value will be higher than the demonstrator z_h value. This occurs with probability equal to the area under the curve to the right of the dotted vertical line, in the extreme right-hand tail of the distribution. The larger is β, the larger this area will be. So while α always makes copied values worse than z_h, β increases the chances that copied values will occasionally exceed z_h.

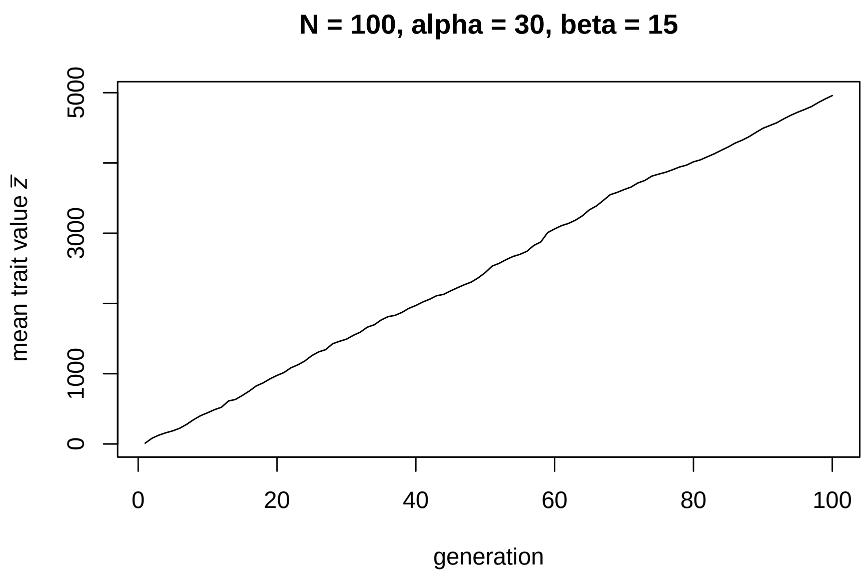
Let's simulate this model to confirm this, and find out when the mean skill in the population $\bar{z}$ increases, representing cultural gain, and when it decreases, representing cultural loss.

The following function follows the structure of previous models. After repeating the **rgumbel** function definition from above (just to make sure that this function is defined, and make **DemographyModel** self-contained), first we set up an output dataframe. In this case the output dataframe needs to store the mean trait value $\bar{z}$ in each generation from 1 to t_{max}. Then we set up the agent dataframe: each of N agents' initial trait value is randomly drawn from a gumbel distribution with mode 0 and dispersion β. After storing the first generation mean trait value, we start cycling through generations. Each generation, we find the maximum skill value for the current generation z_h, then draw N new agents from a gumbel distribution with mode $z_h - \alpha$ and dispersion β as per the figure above and using **rgumbel**. Each generation we store the new mean trait value, before plotting them all at the end and outputting the data from the function. Note the use of **expression()** in the plot axis labels; this allows us to use italics, subscripts and bars in the plot text.

```
DemographyModel <- function(N, alpha, beta, t_max) {

  rgumbel <- function(n, a = 0, beta = 1) {
    a + beta * (-log(-log(runif(n))))
  }

  # create output dataframe to hold mean trait value for each generation
  output <- data.frame(z_bar = rep(NA, t_max))

  # create 1st generation, N random draws from a gumbel with mode a=0 and dispersion beta
  agent <- data.frame(trait = rgumbel(N, beta = beta))

  # store first generation trait mean
  output$z_bar[1] <- sum(agent$trait) / N

  for (t in 2:t_max) {

    # get highest skill from current generation
    z_h <- max(agent$trait)

    # draw new values
    agent$trait <- rgumbel(N, a = z_h - alpha, beta = beta)

    # store new mean trait value
    output$z_bar[t] <- sum(agent$trait) / N

  }

  plot(x = 1:nrow(output), y = output$z_bar,
       type = 'l',
       ylab = expression("mean trait value "*italic(bar("z"))),
       xlab = "generation",
       main = paste("N = ", N, ", alpha = ", alpha, ", beta = ", beta, sep = ""))

  output
}
```

Let's run the demography model with the values of α and β used in the first figure above, and $N = 100$:

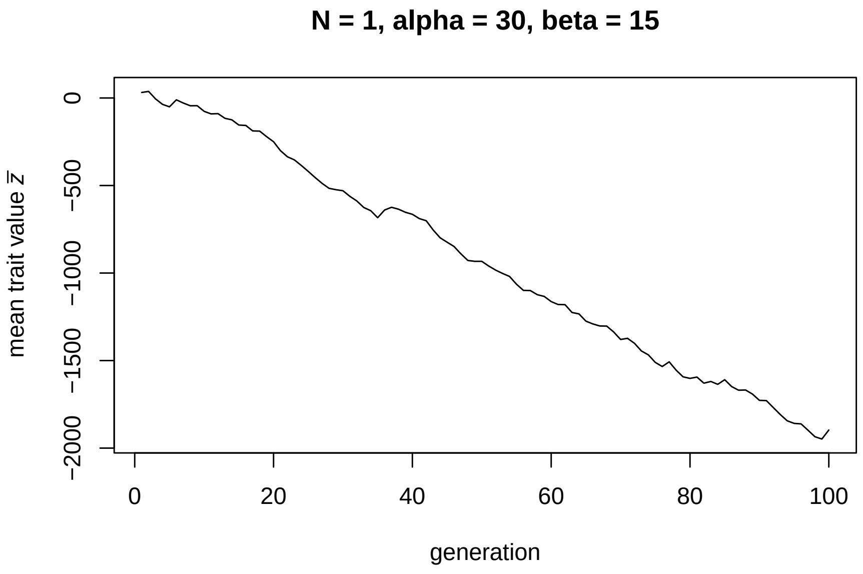
```
data_model9 <- DemographyModel(N = 100,
                               alpha = 30,
                               beta = 15,
                               t_max = 100)
```



Here we can see that, for these parameter values, there is cultural gain: the mean trait value $\bar{z}$ increases linearly over time.

Now let's try the smallest population size possible, $N = 1$:

```
data_model9 <- DemographyModel(N = 1,
                               alpha = 30,
                               beta = 15,
                               t_max = 100)
```



Now there is cultural loss: the mean trait value $\bar{z}$ decreases linearly over time.

Note that in both of these cases, trait values either increase indefinitely, or decrease indefinitely. Obviously this is unrealistic: at the least, there will be a minimum value ('zero skill') below which skills cannot fall. These upper and lower bounds are outside the scope of this model (although see Mesoudi 2011 for how to implement an upper bound, derived from the increasing costs of acquiring increasingly-complex traits). But the key aim of this model is not to realistically simulate actual cumulative cultural change, it is to identify qualitatively the parameter values - particularly population size - which lead to cultural gains and cultural losses.

Now that we know that there are usually only two outcomes of the model, linear cultural gain and linear cultural loss, we can switch our focus to a different outcome measure, $\Delta\bar{z}$. This is the change in mean trait value from one generation to the next. Because the change in $\bar{z}$ is linear, $\Delta\bar{z}$ should be the same on average for all generations.

The following function runs the basic demography model for t_{max} generations across a range of values of N, from 1 to N_{max}. For each N, we record $\Delta\bar{z}$ in the output dataframe. This allows us to explore what values of N lead to cultural gain, i.e. $\Delta\bar{z} > 0$ and what values lead to cultural loss, i.e. $\Delta\bar{z} < 0$. Note that this function can take a long time to run when N_{max} is large. Consequently, there are a few tweaks to make it run as fast as possible. First, we ditch the dataframes and use simple vectors instead, for both *output* and *agent*. Vectors are faster than dataframes, and the latter are really not necessary with only a single measure and trait respectively. Second, rather than storing each $\Delta\bar{z}$ for each generation, we keep a running total and divide by t_{max} at the end of the loop. Finally, we remove the plot for now, and instead create plots from the output later on.

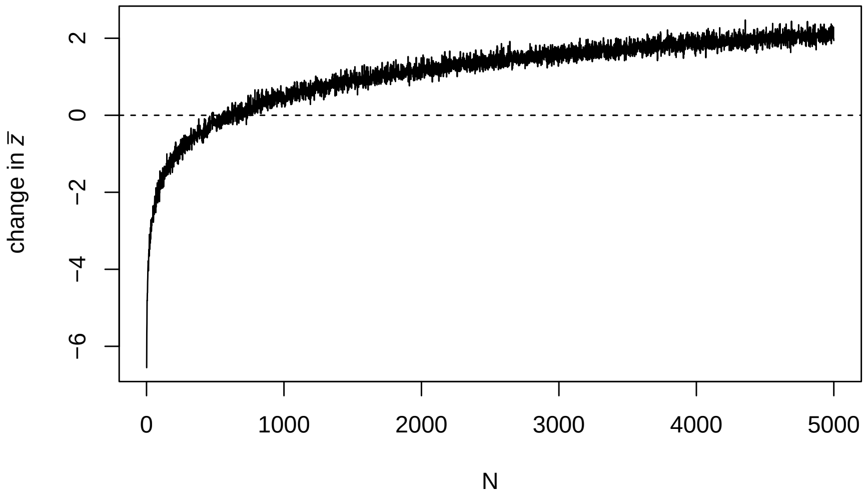
```
DemographyModel2 <- function(N_max, alpha, beta, t_max) {

  rgumbel <- function(n, a = 0, beta = 1) {
    a + beta * (-log(-log(runif(n))))
  }

  # create output dataframe to hold change in mean trait value for each N
  output <- rep(0, N_max)

  for (N in 1:N_max) {

    # create 1st generation, n random draws from a gumbel with mode a=0 and dispersion
    agent <- rgumbel(N, beta = beta)

    for (t in 1:t_max) {

      # current mean z
      z_bar <- sum(agent) / N

      # get highest skill from current generation
      z_h <- max(agent)

      # draw new values
      agent <- rgumbel(N, a = z_h - alpha, beta = beta)

      # add new delta_z_bar to total delta_z_bar for this n
      output[N] <- output[N] + sum(agent) / N - z_bar

    }

    # divide total delta_z_bar by t_max to get mean
    output[N] <- output[N] / t_max

  }

output

}
```

Let's run this for $t_{max} = 50$ and $N_{max} = 5000$, and $\alpha = 7$ and $\beta = 1$. It might take a little while.

```
data_model9_1 <- DemographyModel2(N_max = 5000,
                                  alpha = 7,
                                  beta = 1,
                                  t_max = 100)
```

Now we can plot the results:

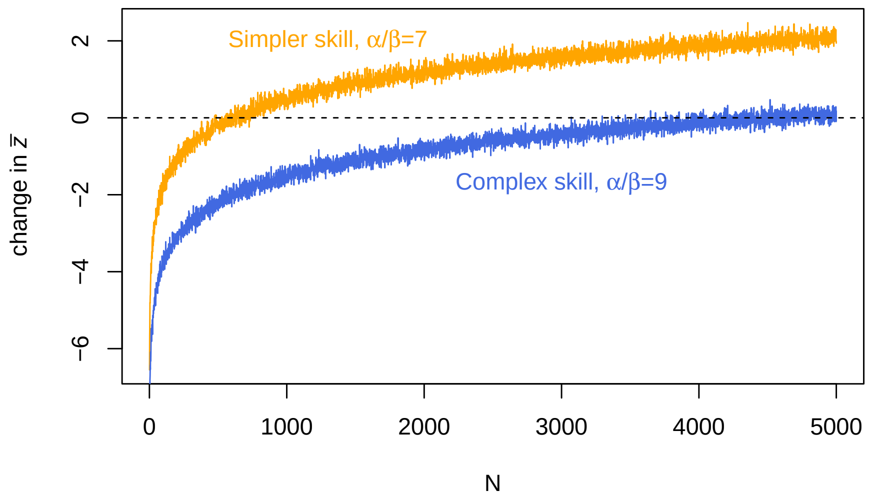
```
plot(1:length(data_model9_1), data_model9_1,
     type = 'l',
     ylab = expression("change in "*italic(bar("z"))),
     xlab = "N")

abline(h = 0, lty = 2)
```



This plot shows that as N increases, $\Delta\bar{z}$ flips from negative to positive. At low N there is cultural loss, at high N there is cultural gain. Note that there is some noise increasing the thickness of the line. This comes from the stochasticity of the simulation. The more generations we run, the less noise there is, but also the longer the simulation will take to run.

We can also run another pair of α and β and compare the two lines on the same plot:

```
data_model9_2 <- DemographyModel2(N_max = 5000,
                                  alpha = 9,
                                  beta = 1,
                                  t_max = 100)

plot(1:length(data_model9_1), data_model9_1,
     type = 'l',
     ylab = expression("change in "*italic(bar("z"))),
     xlab = "N",
     col = "orange")

lines(data_model9_2, col = "royalblue")

abline(h = 0, lty = 2)

text(1300, 2,
     labels = expression("Simpler skill, "*alpha*"/"*beta*"=7"),
     col = "orange")
text(3000, -1.7,
     labels = expression("Complex skill, "*alpha*"/"*beta*"=9"),
     col = "royalblue")
```



Here we have recreated Henrich's (2004) Figure 3, comparing a 'simple' skill and a 'complex' skill. A simple skill is easier to learn than a complex skill, such that it has a smaller α relative to β (remember, α is copying error that always reduces skill, while β is inferences and guesses that sometimes leads to higher skill). The plot above shows that simpler skills require a smaller population size to be maintained ($\Delta\bar{z} = 0$) or improve ($\Delta\bar{z} > 0$) than more complex skills. This reflects the postulated Tasmanian case that we started with: when population sizes dropped when Tasmania became cut off from the mainland, complex skills like fishhooks were lost and never reinvented, while simpler skills remained. In the figure above, this would be like going from $N = 5000$ to $N = 1000$.

Summary

Cultural evolution occurs within populations of certain sizes and structures, and this population size and structure affects evolutionary dynamics. In previous models we have seen how population size can lead to the accidental loss of traits, which is a basic principle of genetic and cultural drift. Model 9 extended this basic finding to show how population size can determine whether there is cultural gain or cultural loss of a continuous cultural trait such as the skill required to make a hard-to-learn tool. Assuming that copying is imperfect, small populations are less able to maintain and accumulate cultural skill than large populations, even when every learner can identify and is attempting to learn from the highest skilled individual from the previous generation. Fewer learners means less chance that one of those learners will match or exceed that highest skilled individual from the previous generation. This effect of population size interacts with the 'learnability' and 'improvability' of the trait: more easily learned (lower α) and more easily improved (higher β) traits require smaller populations to maintain and improve.

This model by Henrich (2004) has inspired a large body of subsequent models (e.g. Powell, Shennan & Thomas, 2009; Mesoudi 2011) and empirical research (e.g. Derex et al. 2013; Kline & Boyd 2010). These have extended the predictions of the model, and tested those predictions both in the laboratory and in real world datasets (see Derex & Mesoudi 2020). To be sure, population size is not the only determinant of cultural gains and losses. Even Henrich's original model demonstrates this, given that learnability and improvability are also crucial. And more recent work has focused on population structure (e.g. social networks) rather than simply the number of individuals that are present. But there is no doubt that demography cannot be ignored when studying cultural evolution. Simulation models are often useful for studying demography, as complex population structures are difficult to implement in analytical models.

A few new programming techniques were used in Model 9. First, we introduced another continuous distribution, the gumbel distribution. Unlike the normal distribution used in Model 8, there are no built-in R functions for using the gumbel. We therefore had to write our own, and incorporate the **rgumbel** function into the simulation function. It is common practice to write user-defined functions for small routines and then call them within other functions. Indeed, it is often good practice, especially when the same set of lines of code are repeated in multiple places. Second, we used commands like **expression**, **text** and **arrows** to annotate plots. It is preferable to create plots entirely in R and then export the finished version (see Model 1 for how to do this), rather than export basic plots and annotate them in separate visual editing software or Powerpoint. Creating plots entirely in R is more reproducible because others can re-run your code and re-create your figures exactly. It also avoids loss of image resolution. Finally, we saw a few tricks to speed up code that takes a long time to run. Dataframes can be replaced with one-dimensional vectors when they are made up of just one variable. Means can be calculated by keeping a running total within a loop then dividing by the total number of data points once the loop has finished. Simplifying code as much as possible can often lead to significantly faster simulation runs, although try not to sacrifice code understandability in doing so.

Exercises

1. Try different values of α, β and N in **DemographyModel** to determine that, for given values of α and β, there is a threshold value of N above which there is cultural gain, and below which there is cultural loss.

2. Change **DemographyModel** so that there are two different population sizes, N_1 and N_2, both user-defined in the function call. For the first half of the generations, i.e. from $t = 1$ to $t = t_{max}/2$, the population size is N_1. For the second half, i.e. from $t = (t_{max}/2) + 1$ to $t = t_{max}$, the population size is N_2. For a given pair of α and β values, pick an N_1 that leads to cultural gain, and an N_2 that leads to cultural loss. Run the simulation to see what happens. Is this a reasonable simulation of the purported 'Tasmanian' case described above and in Henrich (2004), where isolation due to rising sea levels caused a reduction in population size?

3. In Mesoudi (2011) I suggested that a simple way of making the Henrich (2004) model more realistic is to tie the copying error parameter α to the mean skill level, $\bar{z}$, in each generation. As skills become ever more complex, they should become harder to learn. If they are harder to learn, there will be more error in their transmission. Thus as $\bar{z}$ increases over the generations, so should α. Modify the **DemographyModel** function to implement this additional assumption. As before, there should be a user-defined starting value of α. However, in each new timestep from $t = 2$ onwards, α should be multiplied by the mean skill level in the previous

timestep (the latter is already being stored in the output dataframe). Run the simulation to see how this changes the dynamics of the model.

Analytic Appendix

Henrich's (2004) model was analytical rather than simulation-based. We can use the analytical model to check our simulation results above, as well as additional things like calculate the exact N at which losses switch to gains for a given α and β.

As in the model above, we assume N individuals indexed by i, with the ith individual possessing trait value z_i. Each individual of each new generation draws a value from the gumbel distribution shown above, with mode $z_h - \alpha$ and dispersion β. As in the simulation model, we want to calculate $\Delta\bar{z}$, the change in the mean trait value from one generation to the next.

The next generation mean trait value, $\bar{z}'$, is given by:

$$\bar{z}' = z_h + \Delta z_h \qquad (9.1)$$

Equation 9.1 says that the new mean trait value is the highest trait value in the previous generation, z_h, plus any change in z_h as a result of imperfect copying. Henrich shows that the former can be approximated by:

$$z_h = a + \beta(\epsilon + \ln(N)) \qquad (9.2)$$

where ϵ is Euler's constant, which is approximately 0.577.

The other term Δz_h is given by $z'_h - z_h$. The new maximum value, z'_h, is given by the mean of the gumbel distribution, $a + \beta\epsilon$. The old maximum, z_h, is given by $a + \alpha$, as per the gumbel distribution figure above. Consequently:

$$\Delta z_h = a + \beta\epsilon - a - \alpha = -\alpha + \beta\epsilon \qquad (9.3)$$

Substituting Equation 9.2 and 9.3 into Equation 9.1 gives:

$$\bar{z}' = a + \beta(\epsilon + \ln(N)) - \alpha + \beta\epsilon$$

As $\bar{z}' = \bar{z} + \Delta\bar{z}$ and $a = \bar{z} - \beta\epsilon$,

$$\Delta\bar{z} = \bar{z} - \beta\epsilon + \beta(\epsilon + \ln(N)) - \alpha + \beta\epsilon - \bar{z}$$

Simplifying, this gives:

$$\Delta\bar{z} = -\alpha + \beta(\epsilon + \ln(N)) \tag{9.4}$$

which is Henrich's Equation 2. This says that, as we found in the simulation model, whether we see cultural gain or cultural loss depends on α, β and N, but it also provides an exact relation between them.

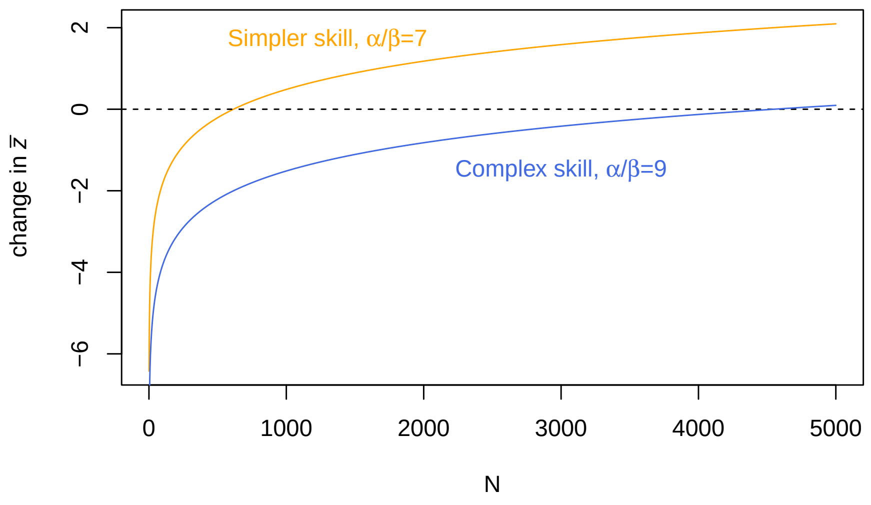
By setting $\Delta\bar{z} = 0$ and rearranging, we find that the equilibrium value of N at which there is neither gain nor loss, N^*, is:

$$N^* = e^{(\frac{\alpha}{\beta} - \epsilon)} \tag{9.5}$$

and this is Henrich's Equation 3. For the values $\alpha = 30$ and $\beta = 15$ used in the simulation model above, equation 9.5 gives $N^* = 4.15$. When N is 4 or less we would expect cultural loss, and when N is 5 or more we would expect cultural gain. Try using the simulation model to confirm this.

Finally, we can use Equation 9.4 to recreate the figure above with the 'simpler' and 'complex' skills:

```r
N_max <- 5000
delta_z <- rep(NA,N_max)

alpha <- 7
beta <- 1

for (N in 1:N_max) {

  delta_z[N] <- -alpha + beta * (-digamma(1) + log(N))

}

plot(1:length(delta_z), delta_z,
     type = 'l',
     ylab = expression("change in "*italic(bar("z"))),
     xlab = "N",
     col = "orange")

abline(h = 0, lty = 2)

alpha <- 9
beta <- 1

for (N in 1:N_max) {
  delta_z[N] <- -alpha + beta * (-digamma(1) + log(N))

}

lines(delta_z, col = "royalblue")

text(1300, 1.7,
     labels = expression("Simpler skill, "*alpha*"/"*beta*"=7"),
     col = "orange")
text(3000, -1.5,
     labels = expression("Complex skill, "*alpha*"/"*beta*"=9"),
     col = "royalblue")
```



Note that Euler's constant is provided in R as **-digamma(1)**. Here we have reproduced the simulation figure above but in a fraction of the time, and with less noise.

References

Derex, M., Beugin, M. P., Godelle, B., & Raymond, M. (2013). Experimental evidence for the influence of group size on cultural complexity. Nature, 503(7476), 389-391.

Derex, M., & Mesoudi, A. (2020). Cumulative cultural evolution within evolving population structures. Trends in Cognitive Sciences, 24(8), 654-667.

Henrich, J. (2004). Demography and cultural evolution: How adaptive cultural processes can produce maladaptive losses - The Tasmanian case. American Antiquity, 69(2), 197-214.

Kline, M. A., & Boyd, R. (2010). Population size predicts technological complexity in Oceania. Proceedings of the Royal Society B: Biological Sciences, 277(1693), 2559-2564.

Mesoudi, A. (2011). Variable cultural acquisition costs constrain cumulative cultural evolution. PloS ONE, 6(3), e18239.

Powell, A., Shennan, S., & Thomas, M. G. (2009). Late Pleistocene demography and the appearance of modern human behavior. Science, 324(5932), 1298-1301.

Model 10: Polarization

Introduction

A central question in cultural evolution research is how cultural diversity is generated and maintained. One counter-intuitive answer to this question was offered by Axelrod (1997) using what has become an influential agent-based model. Axelrod showed that regional differences in cultural traits - or ‘polarization’ when neighbouring regions are maximally culturally different to one another - can be generated and maintained even though individual agents are trying to become as similar as possible to other agents. In Axelrod’s terminology, global cultural divergence at the population level can emerge despite local cultural convergence at the individual level. This conclusion has implications for a variety of social phenomena, from political polarization in social media to racial segregation in cities.

Axelrod demonstrated this using a spatially explicit agent-based model. ‘Spatially explicit’ means that rather than a homogenous mass of agents existing all with an equal probability of interacting with any other agent, instead each agent exists in a specific location in space, with a set of neighbouring agents with whom they may be more likely to interact than agents further away in space. Agent-based models are particularly suited for this kind of spatially explicit simulation compared to analytical models. Our previous models have already incorporated non-random interaction between agents and minimal population structure, such as the assortative cultural mating of Model 6 (vertical/horizontal transmission) and the movement across groups of Model 7 (migration). In Model 10 we will go further and recreate Axelrod’s classic model, learning how to create and analyse spatially explicit agent-based models where each agent inhabits a specific position in a spatial grid.

Another distinct feature of Axelrod’s model is his assumption that agents possess multiple cultural traits, and these can serve as markers of identity and influence interaction. Specifically, he assumed that each agent possesses a number of cultural ‘features’. Each feature can take on one of several trait values. In his basic model, there are five cultural features, each of which can take one of ten possible trait values. When modelling cultural diversity, it makes sense to model multiple cultural traits. Each member of a real society possesses multiple culturally-transmitted traits - language(s) spoken, dialects of those languages, dress customs, whether cars are driven on the left or the right, using knives and forks or chopsticks, bowing vs handshakes, etc. These traits in combination - rather than any single trait - define a society's culture. While we modelled two cultural traits in Model 4 (indirect bias), in Model 10 we will model more than two.

Axelrod's model, like all good agent-based models, is simple. Each agent is placed in a fixed position on a square grid. There are N_{side} agents along each side of the grid, giving ${N_{side}}^2$ agents in total. Each agent has $g = 5$ cultural features. Each feature takes one of ten values, denoted with the integers 0-9. For example, an agent might have traits 58290. Here, the first feature is 5, the second 8, and so on.

In the first timestep each agent's trait values are picked at random. Then in each timestep, the following three rules are applied:

1. Pick an agent at random (the focal agent)

2. Pick one of the focal agent's neighbours at random. A neighbour is an agent to the immediate north, south, east or west of the focal agent's grid position. Focal agents at the corner or edge of the grid may have no neighbour in one of those positions, and so have only two or three neighbours respectively.

3. With probability equal to the proportion of shared cultural traits between the focal agent and its chosen neighbour, pick one feature at random that differs between focal agent and its neighbour, and set the focal agent's value of that feature to the neighbour's value.

These rules plausibly assume that agents are more likely to interact with, and be influenced by, other agents to the extent that they are culturally similar, as indexed by the proportion of cultural traits that they share. When two agents are completely dissimilar, then the proportion of shared features will be zero, and no interaction / influence will occur. The more traits they share, the more likely they are to interact and potentially become even more similar.

The outcome that we are interested in is the cultural diversity or homogeneity that emerges after a certain number of timesteps, or iterations of the above three rules. The only cultural change that can occur according to the three rules above is that agents become more similar to one another. There is no mutation, and no rules that make agents more dissimilar. We would expect, therefore, that all agents will gradually become more and more similar, perhaps identical, and eventually every agent will be culturally identical.

Model 10

Let's build the model step by step before putting it together in a function. First we need to create agents and their traits. Rather than a dataframe, we will use a matrix. A matrix is perfect because it has a fixed number of rows and columns, and these rows and columns will serve as the coordinates for our spatial grid. In our case the matrix will be square, with N_{side} rows and N_{side} columns. Each agent inhabits a fixed position in the matrix. Indexing starts at the top left and has the notation [row,column]. The agent in the top left position is therefore at position [1,1]; its neighbour to the east is in position [1,2]; and its neighbour to the south is at [2,1].

First we create a 10x10 matrix (i.e. $N_{side} = 10$) filled for now with NAs:

```
N_side <- 10

# make agent matrix of size N_size x N_size
agent <- matrix(NA,
                nrow = N_side,
                ncol = N_side)

agent
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [2,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [3,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [4,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [5,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [6,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [7,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [8,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
##  [9,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
## [10,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
```

The rows are labelled along the left hand side, and the columns along the top.

Now we need g features each of which initially takes one integer from 0 to 9. The following code sets $g = 5$, and then uses **sample.int** to pick g integers from 0 to 9, with replacement (so integers can repeat). Note that the **sample.int** command returns random integers from one up to the first argument. Because we want to include 0 as a trait, we pick random integers from 1 to 10, then subtract 1 to get the range 0 to 9.

```r
g <- 5

sample.int(10, g, replace = TRUE) - 1
```

```
## [1] 6 7 1 6 0
```

Because we want each one to fit into a single cell of the matrix, we need to condense them into a single value. We could turn them into a *g*-digit number. However, this would lose the leading zeroes (e.g. 0 0 4 8 5 would become 485, not 00485). Instead we will convert to a character (chr) variable, using the **paste** command with collapse = " " to remove the spaces:

```r
paste(sample.int(10, g, replace = TRUE) - 1,
      collapse = "")
```

```
## [1] "73863"
```

Now we can fill the *agent* matrix with random traits and display them (the **options** command simply widens the print area on the pdf output to see the matrix properly).

```r
  # fill agent with g numbers each a random integer 0-9, stored as chr
  for (n_width in 1:N_side) {

    for (n_height in 1:N_side) {

      agent[n_width,n_height] <- paste(sample.int(10, g, replace = TRUE) - 1,
                                       collapse = "")

    }

  }

options(width = 300)
agent
```

```
##        [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]    [,9]    [,10]
##   [1,] "09824" "83434" "76836" "89679" "43362" "58190" "11079" "06196" "00208" "9814
##   [2,] "26365" "70335" "70985" "03637" "85784" "21754" "12268" "38154" "45285" "5641
##   [3,] "79948" "93830" "10586" "75392" "14961" "13280" "36694" "44395" "93217" "6928
##   [4,] "94043" "16966" "64829" "01889" "63986" "25328" "33222" "00992" "61906" "0551
##   [5,] "33986" "60163" "44886" "46142" "50222" "95191" "37398" "69424" "60800" "9154
##   [6,] "69508" "97012" "33451" "35061" "03410" "50696" "46497" "28094" "65684" "3813
##  [7,] "96206" "66892" "29139" "01158" "07108" "63105" "52054" "67169" "40486" "36619"
##  [8,] "37030" "50965" "63789" "10144" "29123" "20818" "12823" "23038" "14803" "06941"
##  [9,] "74654" "15617" "27543" "69472" "96570" "06624" "06747" "95798" "76656" "45741"
## [10,] "80230" "49977" "17613" "30822" "10396" "39349" "74794" "46297" "79317" "87032"
```

Here we have reproduced Table 1 from Axelrod (1997): a 10 x 10 grid containing *g*-length cultural traits for 100 agents.

Now to convert the three rules above into code. First we pick a random agent to be the focal agent. We do this by picking a random row value and a random column value, and setting *focal* to the agent in that position.

```
# pick a focal agent at random
focal_row <- sample(1:N_side, 1)
focal_col <- sample(1:N_side, 1)
focal <- agent[focal_row, focal_col]

focal
```

```
## [1] "79317"
```

Now we pick one of the focal's neighbours at random, as per rule 2. This is complicated by the fact that some focal agents will have four neighbours (those in the middle of the grid), some will have three (those at the edges) and some will have only two (those in the corners). We don't want to pick a non-existent neighbour to compare with the focal. The following code creates a vector of *all_neighbours*, adds ('appends') to this vector only those neighbours that exist given the focal's position, and picks one of the existing neighbours at random to be the *neighbour*.

```
# pick one of its neighbours at random
# ignoring non-existent agents outside boundaries

all_neighbours <- NULL

if (focal_row-1 >= 1)
  all_neighbours <- append(all_neighbours,
                           agent[focal_row-1, focal_col])
if (focal_row+1 <= N_side)
  all_neighbours <- append(all_neighbours,
                           agent[focal_row+1, focal_col])
if (focal_col-1 >= 1)
  all_neighbours <- append(all_neighbours,
                           agent[focal_row, focal_col-1])
if (focal_col+1 <= N_side)
```

```
  all_neighbours <- append(all_neighbours,
                           agent[focal_row, focal_col+1])

neighbour <- sample(all_neighbours, 1)

neighbour
```

```
## [1] "76656"
```

We can now simulate rule 3, the cultural change. First we convert the *focal* and *neighbour*'s traits from a single character into numbers using the **unlist** and **strsplit** commands. This is essentially the reverse of the **paste** command above. We can then get the cultural *similarity* between *focal* and *neighbour* by adding the number of features that are identical and dividing by the total number of features. Then, if this *similarity* is greater than zero and less than one, i.e. the two agents are not completely dissimilar nor identical, then with probability equal to *similarity* we pick a random feature that differs between the *focal* and *neighbour* and store it as *feature*. These are identified using the **which** command (i.e. `which(focal != neighbour)`). If there is a single dissimilar feature (i.e. `sum(focal != neighbour) == 1`) then this single dissimilar feature is stored in *feature*. If there is more than one, then we pick one at random using *sample*. Note that we do this because, if there is a single dissimilar feature, **sample** will not work properly. Always make sure **sample** is picking from more than one numeric element, never a single numeric element (try running `sample(2, 10, replace = TRUE)` to see this). Finally, we set the *focal* agent's chosen dissimilar trait value to that of its *neighbour*, and insert the modified *focal* traits back into the agent matrix.

```
# separate out traits and make them numeric, for comparing
focal <- as.numeric(unlist(strsplit(focal, split = NULL)))
neighbour <- as.numeric(unlist(strsplit(neighbour, split = NULL)))

# get similarity
similarity <- sum(focal == neighbour) / length(focal)

if (similarity > 0 & similarity < 1) {

  if (runif(1) < similarity) {

    if (sum(focal != neighbour) == 1) {
      feature <- which(focal != neighbour)
    } else {
      feature <- sample(which(focal != neighbour), 1)
    }
    focal[feature] <- neighbour[feature]

    agent[focal_row, focal_col] <- paste(focal, collapse = "")

  }

}

paste(focal, collapse = "")
```

```
## [1] "79317"
```

```
paste(neighbour, collapse = "")
```

```
## [1] "76656"
```

```
similarity
```

```
## [1] 0.2
```

```
agent[focal_row, focal_col]
```

```
## [1] "79317"
```

The output shows the original focal agent, its neighbour, their similarity, and the modified focal agent. The latter may well be identical to the original focal traits, given that our randomised agents are currently highly dissimilar to one another (probably *similarity* = *0*, in which case there is definitely no change). Try repeating the above code but with the *focal* and *neighbour* set to be maximally similar without being identical, e.g. “12345” and “12340” respectively. In this case *similarity* = *0.8* and the final feature in *focal* is likely to flip from 5 to 0.

We now have all the code to write a function. **Polarization** below combines all the previous code, wrapping the three rules in a loop to repeat them t_{max} times. The output of the simulation is the final *agent* matrix at $t = t_{max}$ and the number of timesteps at which this was produced (this will be used later when plotting the results).

```
Polarization <- function(N_side, g, t_max) {

  # make agent matrix of size N_size x N_size
  agent <- matrix(NA,
                  nrow = N_side,
                  ncol = N_side)

  # fill agent with g numbers each a random integer 0-9, stored as chr
  for (n_width in 1:N_side) {

    for (n_height in 1:N_side) {

      agent[n_width,n_height] <- paste(sample.int(10, g, replace = TRUE) - 1,
                                       collapse = "")

    }

  }

  for (t in 1:t_max) {

    # pick a focal agent at random
    focal_row <- sample(1:N_side, 1)
    focal_col <- sample(1:N_side, 1)
    focal <- agent[focal_row, focal_col]

    # pick one of its neighbours at random
    # ignoring non-existent agents outside boundaries

    all_neighbours <- NULL

    if (focal_row-1 >= 1)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row-1, focal_col])
    if (focal_row+1 <= N_side)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row+1, focal_col])
    if (focal_col-1 >= 1)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row, focal_col-1])
    if (focal_col+1 <= N_side)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row, focal_col+1])

    neighbour <- sample(all_neighbours, 1)

    # compare focal and neighbour agents:
    # if there's at least one dissimilar trait,
    # and with prob equal to the similarity between focal and neighbout,
    # set a random dissimilar focal trait to that of the neighbour's

    # separate out traits and make them numeric, for comparing
    focal <- as.numeric(unlist(strsplit(focal, split = NULL)))
    neighbour <- as.numeric(unlist(strsplit(neighbour, split = NULL)))

    # get similarity
    similarity <- sum(focal == neighbour) / length(focal)

    if (similarity > 0 & similarity < 1) {

      if (runif(1) < similarity) {

          if (sum(focal != neighbour) == 1) {
            feature <- which(focal != neighbour)
          } else {
            feature <- sample(which(focal != neighbour), 1)
          }

        focal[feature] <- neighbour[feature]

        agent[focal_row, focal_col] <- paste(focal, collapse = "")

      }

    }

  }

  # output agent matrix and t_max
  list(agent = agent, t = t)

}
```

Here is one run of the *Polarization* function, with $N_{side} = 10$, $g = 5$ and $t_{max} = 100000$. We display the final *agent* matrix that has been stored in *data_model10*.

```
data_model10 <- Polarization(N_side = 10,
                             g = 5,
                             t_max = 100000)

data_model10$agent
```

```
##       [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]    [,9]    [,10]
##  [1,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [2,] "22434" "22434" "22434" "22434" "22434" "22434" "80303" "22434" "22434" "2243
##  [3,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [4,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [5,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [6,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [7,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [8,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
##  [9,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
## [10,] "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "22434" "2243
```

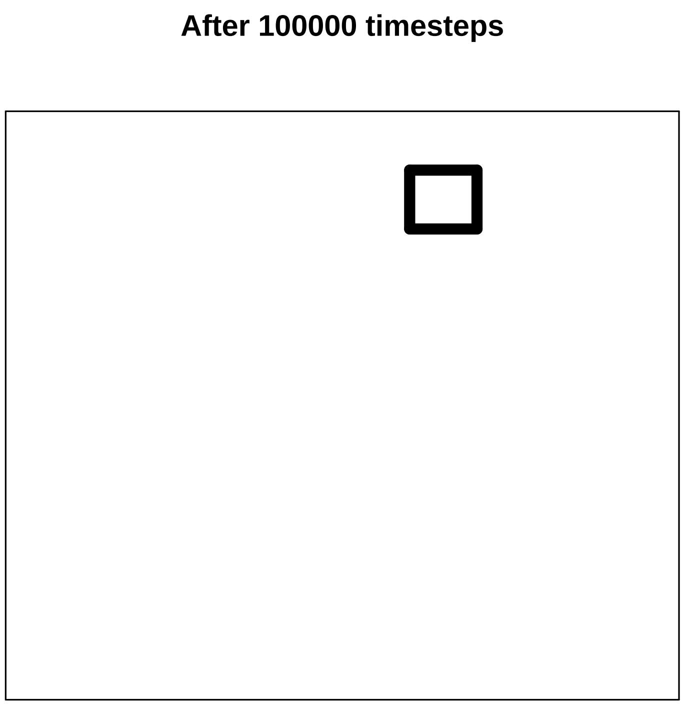
After 100,000 timesteps, there should be noticeable similarity in the agent traits above. Probably not complete similarity, however. While most agents should have the same traits, there should be some areas of the matrix where some agents have a different trait.

Axelrod plotted the output of the model using lines of different thickness to denote the cultural similarity between neighbouring agents. The function **PolarizationPlot** below does this, using the *data_model10* output from **Polarization**. First we retrieve the *agent* matrix and t_{max} from *data_model10*, as well as N_{side} from *agent*. We then make an empty N_{side} x N_{side} plot, add a title recording the number of timesteps, and add lines around the sides using **segments**. Then we cycle through each agent along the rows (the *Row* loop) and columns (the *Col* loop) to draw vertical lines. We pick each agent and its neighbour to the east, calculate their *similarity* in the same way as we did above, set the *lwd* (line width) in proportion to the *similarity*, and draw a line using **segments**. Once all the vertical lines are drawn we cycle through again, this time comparing each agent to its neighbour to the south and drawing horizontal lines at the boundaries between them.

```
PolarizationPlot <- function(data_model10) {

  agent <- data_model10$agent
  t_max <- data_model10$t

  # retrieve N_side from matrix
  N_side <- dim(agent)[1]

  # make an empty N_side x N_side plot
  plot(NULL,
       ylim = c(0, N_side),
       xlim = c(0, N_side),
       ylab = "",
       xlab = "",
       axes = FALSE,
       main = paste("After ", format(t_max, scientific = F), " timesteps",
                    sep = ""))

  # add a frame around the edges
  segments(x0 = 0, y0 = 0,
           x1 = 0, y1 = N_side)
  segments(x0 = 0, y0 = 0,
           x1 = N_side, y1 = 0)
  segments(x0 = 0, y0 = N_side,
           x1 = N_side, y1 = N_side)
  segments(x0 = N_side, y0 = 0,
           x1 = N_side, y1 = N_side)

  # vertical lines

  for (Row in 1:N_side) {

    for (Col in 1:(N_side-1)) {

      # agent2 is to the right of agent1
      agent1 <- agent[Row,Col]
      agent2 <- agent[Row,Col+1]

      # make numeric
      agent1 <- as.numeric(unlist(strsplit(agent1, split = NULL)))
      agent2 <- as.numeric(unlist(strsplit(agent2, split = NULL)))

      # get similarity
      similarity <- sum(agent1 == agent2) / length(agent1)

      # set line thickness
      if (similarity < 1)    lwd <- 0.5
      if (similarity <= 0.8) lwd <- 1
      if (similarity <= 0.6) lwd <- 3
      if (similarity <= 0.4) lwd <- 5
      if (similarity <= 0.2) lwd <- 7

      if (similarity < 1) {

        segments(x0 = Col, x1 = Col,
                 y0 = N_side-Row, y1 = N_side-Row+1,
                 lwd = lwd)

      }

    }
  }

  # horizontal lines

  for (Row in 1:(N_side-1)) {

    for (Col in 1:N_side) {

      # agent2 is directly below agent1
      agent1 <- agent[Row,Col]
      agent2 <- agent[Row+1,Col]

      # make numeric
      agent1 <- as.numeric(unlist(strsplit(agent1, split = NULL)))
      agent2 <- as.numeric(unlist(strsplit(agent2, split = NULL)))

      # get similarity
      similarity <- sum(agent1 == agent2) / length(agent1)

      # set line thickness
      if (similarity < 1)    lwd <- 0.5
      if (similarity <= 0.8) lwd <- 1
      if (similarity <= 0.6) lwd <- 3
      if (similarity <= 0.4) lwd <- 5
      if (similarity <= 0.2) lwd <- 7

      if (similarity < 1) {

        segments(y0 = N_side-Row, y1 = N_side-Row,
                 x0 = Col-1, x1 = Col,
                 lwd = lwd)

      }

    }

  }

}
```

Below is the plot for the run created above. It should match the *agent* matrix, with lines separating agents that are dissimilar. The thicker the lines, the more dissimilar the agents are.

```
PolarizationPlot(data_model10)
```



The plot should show a small number of regions that are dissimilar to the neighbouring regions. Despite individual agents' preferences to interact with and copy similar others, different regions of the grid have polarized into groups with different cultural traits.

How many distinct regions are there in the plot above? We could count them visually, but better would be to use the code below. We convert the *agent* matrix to a vector, and count the number of unique traits using the **unique** command.

```
length(unique(as.vector(data_model10$agent)))
```

```
## [1] 2
```

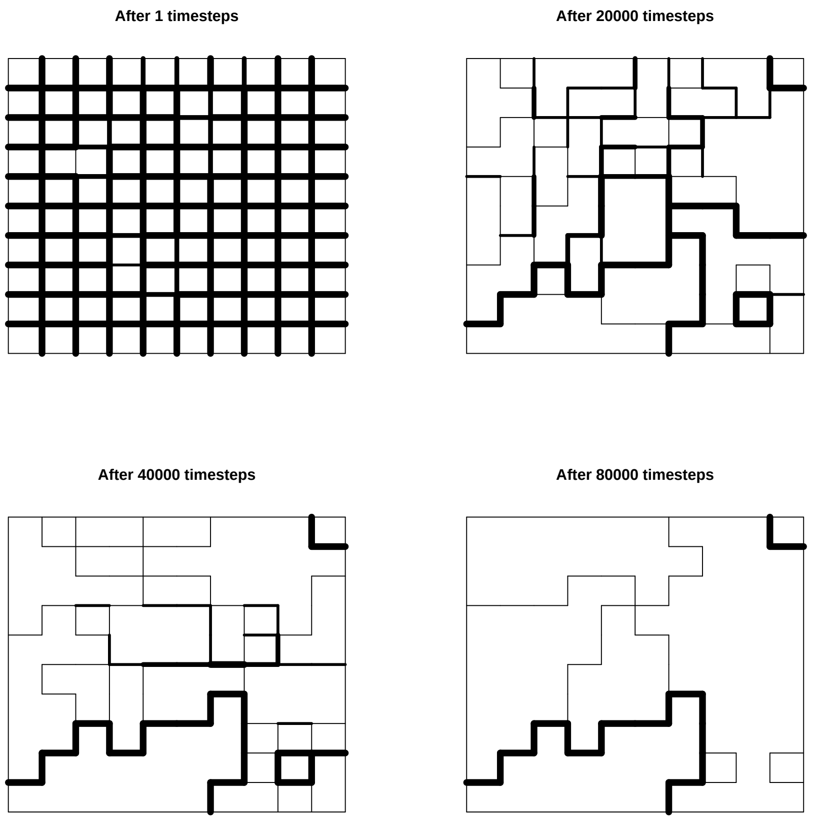
Note that this is the number of unique traits in the entire population. It may not be the same as the number of regions shown in the plot, if there are two non-contiguous regions that have the same trait. Axelrod doesn't specify whether he recorded the number of unique traits or the number of non-contiguous regions. These are unlikely to differ too much though, as the chances of two non-contiguous regions having identical traits is slim.

How did these regions form over time? The function **PolarizationMultiplot** below creates a multi-panel 2 x 2 plot which plots the *agent* matrix at four different points of the same run, as in Axelrod's Figure 1. The parameter t_{plot} replaces t_{max}. t_{plot} is a list of four timesteps at which the output is plotted; the final value in t_{plot} will effectively therefore be t_{max}. Most of the code in **PolarizationMultiplot** is the same as **Polarization**, except that we set up a 2 x 2 plot using `par(mfrow = c(2,2))`, store each of the four *agent* matrices in *agent_list* whenever the timestep t matches one of the t_{plot} values (using `if (t %in% t_plot)`), and plot each *agent* matrix using **PolarizationPlot** at the end.

```
PolarizationMultiplot <- function(N_side, g, t_plot) {

  # make agent matrix of size N_size x N_size
  agent <- matrix(NA,
                  nrow = N_side,
                  ncol = N_side)

  # fill agent with g-digit codes, stored as chr
  for (n_width in 1:N_side) {

    for (n_height in 1:N_side) {

      agent[n_width,n_height] <- paste(sample.int(10, g, replace = TRUE) - 1,
                                       collapse = "")

    }

  }

  # set up 2x2 plot panels
  par(mfrow = c(2,2))

  # set up agent list to store data
  agent_list <- list(NULL)

  for (t in 1:max(t_plot)) {

    # pick a focal agent at random
    focal_row <- sample(1:N_side, 1)
    focal_col <- sample(1:N_side, 1)
    focal <- agent[focal_row, focal_col]

    # pick one of its neighbours at random
    # ignoring non-existent agents outside boundaries

    all_neighbours <- NULL

    if (focal_row-1 >= 1)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row-1, focal_col])
    if (focal_row+1 <= N_side)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row+1, focal_col])
    if (focal_col-1 >= 1)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row, focal_col-1])
    if (focal_col+1 <= N_side)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row, focal_col+1])

    neighbour <- sample(all_neighbours, 1)

    # compare focal and neighbour agents:
    # if there's at least one dissimilar trait,
    # and with prob equal to the similarity between focal and neighbout,
    # set a random dissimilar focal trait to that of the neighbour's

    # separate out traits and make them numeric, for comparing
    focal <- as.numeric(unlist(strsplit(focal, split = NULL)))
    neighbour <- as.numeric(unlist(strsplit(neighbour, split = NULL)))

    # get similarity
    similarity <- sum(focal == neighbour) / length(focal)

    if (similarity > 0 & similarity < 1) {

      if (runif(1) < similarity) {

        if (sum(focal != neighbour) == 1) {
          feature <- which(focal != neighbour)
        } else {
          feature <- sample(which(focal != neighbour), 1)
        }
        focal[feature] <- neighbour[feature]

        agent[focal_row, focal_col] <- paste(focal, collapse = "")

      }

    }

    # add agent to agent_list if t is in t_plot
    if (t %in% t_plot) {

      agent_list[[match(t, t_plot)]] <- agent

    }

  }

  # create plots using PolarizationPlot
  for (t in 1:length(t_plot)) {

    PolarizationPlot(list(agent = agent_list[[t]],
                          t = t_plot[t]))

  }

  # output agent matrix list and t from function
  list(agent = agent_list, t = t_plot)

}
```

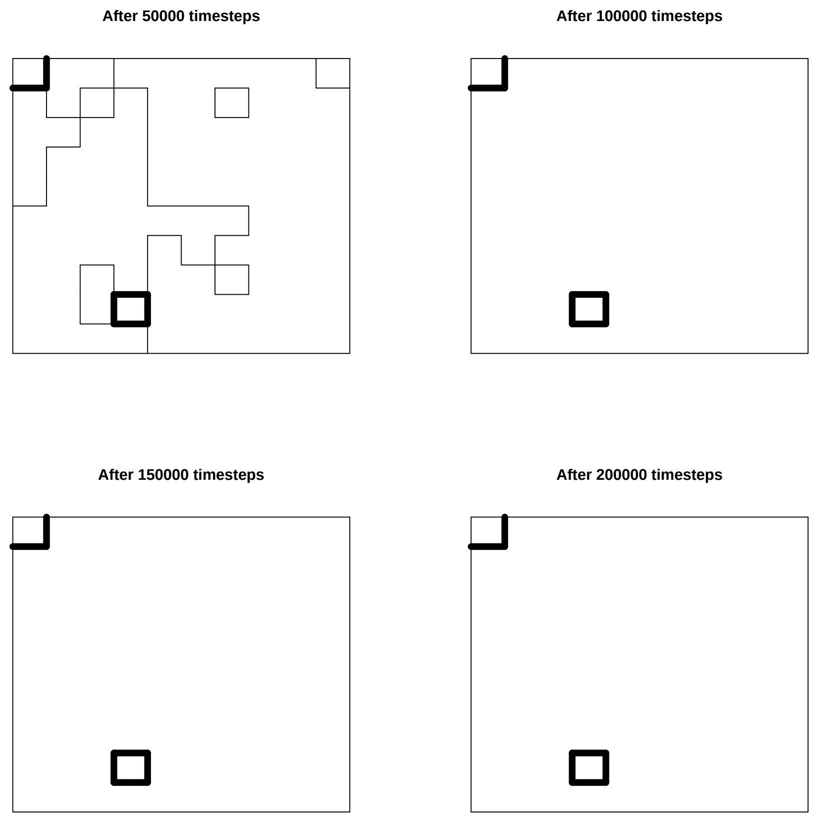
Now we can recreate Axelrod's Figure 1, with plots for $t = 1, 20000, 40000, 80000$. You should be able to see the thick lines that indicate dissimilarity gradually disappear, leaving a small number of distinct regions by $t = 80000$.

```
data_model10 <- PolarizationMultiplot(N_side = 10,
                                      g = 5,
                                      t_plot = c(1,20000,40000,80000))
```



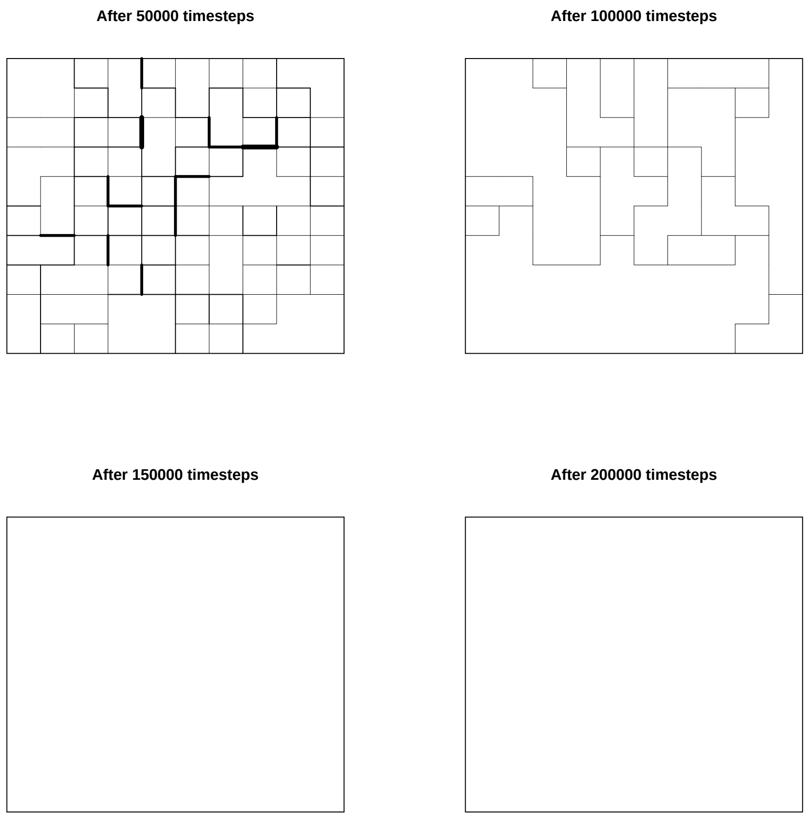
It is likely that by $t = 80000$ the population has reached stability. This means that all neighbours of every agent are either culturally identical (indicated in the plot by no lines) or maximally culturally dissimilar (indicated by the thickest lines). In both of these cases, no further change can occur. Let's run the simulation for more timesteps. It should be guaranteed to reach stability by $t = 200000$.

```
data_model10 <- PolarizationMultiplot(N_side = 10,
                                      g = 5,
                                      t_plot = c(50000,100000,150000,200000))
```



Now let's vary g, the number of cultural features, to see how this affects the number of stable regions. With $g = 15$:

```
data_model10 <- PolarizationMultiplot(N_side = 10,
                                      g = 15,
                                      t_plot = c(50000,100000,150000,200000))
```



Interestingly, with a larger g we are almost certain to converge on a single stable region that encompasses the entire grid. What's more, at $t = 100000$ there are no highly dissimilar boundaries, only weakly similar ones. This was also observed by Axelrod. Counter-intuitively, having more cultural features increases the likelihood of global homogeneity rather than polarization. This is because with more features, there is a higher chance that two agents will have the same trait value on at least one of those features, and therefore be able to interact and switch traits.

It would be useful to know exactly when a simulation run reaches stability. If a run reaches stability before t_{max}, there is little point continuing the run. We can save time by stopping the simulation at the point at which stability is reached and recording the final *agent* configuration. We also don't have to try to guess beforehand when stability might be reached, and risk ending the simulation at t_{max} before that point.

The following function **stability** takes an *agent* matrix as its input and tests whether stability has been reached. The function cycles through each agent (much like in **PolarizationPlot**, first along rows then along columns), and calculates the similarity of each pair of neighbouring agents. If any of these similarities are greater than zero and less than one, then a variable *stable* is set to FALSE and the loops are broken using **break**. Because *stable* starts out initially TRUE, if the function gets to the end of the cycles without finding a non-stable pair of neighbours, then *stable* remains TRUE. *stable* is then outputted from the function at the end.

```
stability <- function(agent) {

  # cycle thru each agent, compare with neighbours,
  # if similarity >0 and <1 then set stable=false and break the loop(s)

  # retrieve N_side from agent matrix
  N_side <- dim(agent)[1]

  # start with the assumption of stability
  stable <- TRUE

  # cycle thru agents horizontally
  for (Row in 1:N_side) {

    for (Col in 1:(N_side-1)) {

      # agent2 is to the right of agent1
      agent1 <- agent[Row,Col]
      agent2 <- agent[Row,Col+1]

      # make numeric
      agent1 <- as.numeric(unlist(strsplit(agent1, split = NULL)))
      agent2 <- as.numeric(unlist(strsplit(agent2, split = NULL)))

      # get similarity
      similarity <- sum(agent1 == agent2) / length(agent1)

      if (similarity > 0 & similarity < 1) {

        stable <- FALSE
        break

      }
    }

    if (stable == FALSE) break
  }

  # if still possibly stable, cycle vertically
  if (stable == TRUE) {

    for (Row in 1:(N_side-1)) {

      for (Col in 1:N_side) {

        # agent2 is directly below agent1
        agent1 <- agent[Row,Col]
        agent2 <- agent[Row+1,Col]

        # make numeric
        agent1 <- as.numeric(unlist(strsplit(agent1, split = NULL)))
        agent2 <- as.numeric(unlist(strsplit(agent2, split = NULL)))

        # get similarity
        similarity <- sum(agent1 == agent2) / length(agent1)

        if (similarity > 0 & similarity < 1) {

          stable <- FALSE
          break

        }
      }

      if (stable == FALSE) break

    }

  }

  # return stable
  stable

}
```

Now we can rewrite the **Polarization** function to add this stability test. We add a new parameter *stable_stop* to the function definition. When *stable_stop* is FALSE (the default), **Polarization** functions as before and continues until t_{max}. When *stable_stop* is TRUE, then the stability test is activated. At the end of every timestep, the **stability** function is called. If the returned *stable* variable is TRUE, then the t loop is broken. As before, t is part of the output of the function. While previously this was always t_{max}, now it is the t at which the simulation reached stability, if *stable_stop* is TRUE and stability was reached before t_{max}.

```r
Polarization <- function(N_side, g, t_max, stable_stop = FALSE) {

  # make agent matrix of size N_size x N_size
  agent <- matrix(NA,
                  nrow = N_side,
                  ncol = N_side)

  # fill agent with g numbers each a random integer 0-9, stored as chr
  for (n_width in 1:N_side) {

    for (n_height in 1:N_side) {

      agent[n_width,n_height] <- paste(sample.int(10, g, replace = TRUE) - 1,
                                       collapse = "")

    }

  }

  for (t in 1:t_max) {

    # pick a focal agent at random
    focal_row <- sample(1:N_side, 1)
    focal_col <- sample(1:N_side, 1)
    focal <- agent[focal_row, focal_col]

    # pick one of its neighbours at random
    # ignoring non-existent agents outside boundaries

    all_neighbours <- NULL

    if (focal_row-1 >= 1)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row-1, focal_col])
    if (focal_row+1 <= N_side)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row+1, focal_col])
    if (focal_col-1 >= 1)
      all_neighbours <- append(all_neighbours,
                               agent[focal_row, focal_col-1])
    if (focal_col+1 <= N_side)
      all_neighbours <- append(all_neighbours,
                              agent[focal_row, focal_col+1])

    neighbour <- sample(all_neighbours, 1)

    # compare focal and neighbour agents:
    # if there's at least one dissimilar trait,
    # and with prob equal to the similarity between focal and neighbout,
    # set a random dissimilar focal trait to that of the neighbour's

    # separate out traits and make them numeric, for comparing
    focal <- as.numeric(unlist(strsplit(focal, split = NULL)))
    neighbour <- as.numeric(unlist(strsplit(neighbour, split = NULL)))

    # get similarity
    similarity <- sum(focal == neighbour) / length(focal)

    if (similarity > 0 & similarity < 1) {

      if (runif(1) < similarity) {

        if (sum(focal != neighbour) == 1) {
          feature <- which(focal != neighbour)
        } else {
          feature <- sample(which(focal != neighbour), 1)
        }

        focal[feature] <- neighbour[feature]

        agent[focal_row, focal_col] <- paste(focal, collapse = "")

      }

    }

    # if stable_stop is TRUE, break the t-loop if stability is reached
    if (stable_stop) {
      stable <- stability(agent)
      if (stable) break
    }

  }

  # output agent matrix and t
  list(agent = agent, t = t)
```

```
}
```

Running **Polarization** with *stable_stop* = *TRUE* and a very large $t_{max} = 1000000$ reveals that stability is reached long before this point, most likely less than 100000 timesteps.

```
data_model <- Polarization(N_side = 10,
                           g = 5,
                           t_max = 1000000,
                           stable_stop = TRUE)

data_model$agent
```

```
##       [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]    [,8]    [,9]    [,10]
##  [1,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [2,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [3,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [4,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [5,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [6,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [7,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [8,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
##  [9,] "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "15291" "1529
## [10,] "15291" "15291" "15291" "54084" "15291" "15291" "15291" "44007" "44007" "1529
```

```
data_model$t
```

```
## [1] 168614
```

Now we have a more nimble simulation, we can systematically vary parameters and measure the effect on the average number of stable regions across multiple runs. The code below runs **Polarization** for $g = 5$, $g = 10$ and $g = 15$, each for $r_{max} = 10$ independent runs. The output, both the number of stable regions at stability and the number of timesteps at which this was reached, is recorded in a dataframe called *g_analysis*. As in previous models, we add the values of each run to a running total then divide by r_{max} at the end. Finally, because it can take a while to run, we add progress messages using the **cat** function. This tells us which g value and which run the simulation is currently at. While this is largely cosmetic, it can be useful to know that a simulation is still running and hasn't hung up, and how long it might take to finish so you know whether you have time to go get a cup of tea.

```r
g_values <- c(5, 10, 15)

r_max <- 10

g_analysis <- data.frame(cultural_features = g_values,
                         stable_regions = 0,
                         timesteps = 0)

for (g in 1:length(g_values)) {

  cat("Running g =", g_values[g], fill=T)

  for (r in 1:r_max) {

    cat("...Run", r, "of", r_max, fill=T)

    data_model10 <- Polarization(N_side = 10,
                                 g = g_values[g],
                                 t_max = 1000000,
                                 stable_stop = TRUE)

    g_analysis$stable_regions[g] <- g_analysis$stable_regions[g] +
      length(unique(as.vector(data_model10$agent)))

    g_analysis$timesteps[g] <- g_analysis$timesteps[g] +
      data_model10$t

  }

}
```

```
## Running g = 5
## ...Run 1 of 10
## ...Run 2 of 10
## ...Run 3 of 10
## ...Run 4 of 10
## ...Run 5 of 10
## ...Run 6 of 10
## ...Run 7 of 10
## ...Run 8 of 10
## ...Run 9 of 10
## ...Run 10 of 10
## Running g = 10
## ...Run 1 of 10
## ...Run 2 of 10
## ...Run 3 of 10
## ...Run 4 of 10
## ...Run 5 of 10
## ...Run 6 of 10
## ...Run 7 of 10
## ...Run 8 of 10
## ...Run 9 of 10
## ...Run 10 of 10
## Running g = 15
## ...Run 1 of 10
## ...Run 2 of 10
## ...Run 3 of 10
## ...Run 4 of 10
## ...Run 5 of 10
## ...Run 6 of 10
## ...Run 7 of 10
## ...Run 8 of 10
## ...Run 9 of 10
## ...Run 10 of 10
```

```
g_analysis$stable_regions <- g_analysis$stable_regions / r_max
g_analysis$timesteps <- g_analysis$timesteps / r_max

g_analysis
```

```
##   cultural_features stable_regions timesteps
## 1                 5            4.8   61202.4
## 2                10            1.0  102878.6
## 3                15            1.0  123569.5
```

The *stable_regions* column in the *g_values* dataframe should show that when there are five cultural features ($g = 5$), there is more than one stable region, i.e. polarization. When there are ten or fifteen features, there is a single stable region, i.e. homogeneity. This matches the middle column of Axelrod's Table 2, for 10 traits per feature (Axelrod found a mean of 3.2 stable regions for $g = 5$, and 1.0 for $g = 10$ and $g = 15$). The *timesteps* column in *g_values* reassures us that stability was reached, because they are all much less than the t_{max} value that was defined.

Summary

The notable feature of Model 10, a replication of Axelrod (1997), is the apparent contradiction between the local interactions of agents and the global dynamics of the model. At the local level, individual agents preferentially interact with neighbours who are more culturally similar to themselves. Their only possible action is to become even more similar by adopting their neighbour's traits with a probability proportional to their similarity. Yet at the global level, we often see the emergence of stark cultural divisions between regional clusters of agents who are maximally different to other clusters.

Global polarization can therefore emerge despite local preferences for cultural convergence. This perhaps gives some insight into real-world polarization, from racial and socio-economic segregation in cities, to the balkanization of regions within states, to political polarization on social media. These socially undesirable patterns may not necessarily be the result of people's tendencies to denigrate outgroups or make themselves as dissimilar as possible to others. They may instead be a by-product of the benign desire to preferentially assort with culturally similar others, and become even more similar to those similar others. If the latter is true (and such a claim needs to be empirically tested), this may change how we try to tackle such real world social problems.

Interestingly and counter-intuitively, polarization is more likely to be observed when there are fewer cultural features (a smaller g in the model). One way of avoiding polarization may therefore be to emphasise the many ways in which people differ. There are not just conservatives and liberals, there are conservatives and liberals who are also football fans and enjoy hiking and like romantic comedies and listen to country music. The more features people may assort on, the more likely interactions are to occur. Axelrod (1997) also found that polarization is reduced when (i) agents interact with more agents, i.e. have more neighbours than just the four we simulated in Model 10, and (ii) when territories are larger, i.e. there are more agents overall. Increasing the range of interactions and number of interactants may therefore be another way to reduce polarization.

On the other hand, the alternative outcome of the model is complete cultural homogeneity. This is also often undesirable. Cultural evolution requires variation on which selection for better alternatives can act. No variation means no adaptation. The loss of minority knowledge, customs and traditions leaves societies less culturally rich and less resilient to environmental change. The cause of homogeneity in Model 10 is similar to the assortation from Model 6, but on multiple traits not just one. The outcome is also similar to conformity (Model 5), which similarly causes the cultural majority to subsume the minority. To preserve minority traditions, some way is needed to counter the benign desire to assort with and copy similar others.

Of course, there are many processes missing from this simple model. Cultural mutation would constantly introduce new variation, acting against both homogeneity and polarization (although see Klemm et al. 2005). Inter-regional competition may shift or dissolve the boundaries between stable regions. The cultural features in Model 10 are simply markers of identity; if instead they affected payoffs, then individuals might happily adopt high-payoff traits from otherwise dissimilar others. Agents are fixed in position and stuck with the same neighbours forever, rather than being able to form new links to other agents (see Centola et al. 2007). Real-world cultural evolution is complex, but the value of simple models like Axelrod's is to better understand the many simple processes that contribute to a messy, complex reality. What's more, the dynamics of even simple models like this are often counter-intuitive, as we have seen with the emergence of global polarization despite local convergence, and the effect of g.

One programming innovation of Model 10 was using a matrix to simulate agents within a spatially explicit grid. Each agent's traits were placed in a fixed position in a square matrix. The [row,column] matrix notation was used to retrieve an agent's position as well as their neighbours with whom they interact. Matrices can be rectangular as well as square, and also one-dimensional (essentially a line of agents). We also modelled multiple cultural traits per agent, specifically g cultural features each of which could take one of ten values. This goes beyond the one or two traits and trait values of previous models. We encoded these as character variables to preserve the leading zeroes, which would disappear if we used numeric variables. We converted the character to numeric to compare agents and their neighbours. You should use whatever variable type is most appropriate for the task. Some other minor coding tips: first, use **break** to break out of loops once you know there will be no further change, or once some condition has been fulfilled, to avoid wasting time; second, always make sure that **sample** is picking from more than one element, never a single element, otherwise things can go wrong; and third, add messages using **cat** to keep track of the progress of the simulation.

Exercises

1. Create some code to track the number of unique regions in every timestep, averaged across r_{max} independent runs. Plot this average value over time. Does the number of unique regions decline steadily over time or show a different pattern?

2. In the Model 10 simulations, the number of traits per feature is fixed at 10 (the integers 0-9). Axelrod varied this, comparing 10 with 5 and 15. Modify **Polarization** to make the number of traits per feature a user-defined variable. Run a similar analysis to the *g_analysis* above, but varying both g and the number of traits per feature. Use this to replicate

all of Axelrod's Table 2, showing the mean number of stable regions for every combination of five, ten and fifteen features and traits per feature.

3. Analyse the effect of varying N_{side}, the size of the grid within which agents are placed, and hence the number of agents in the population. Run a similar analysis to the *g_analysis* above, but varying N_{side}, and keeping $g = 5$ and fifteen traits per feature. Plot the average number of stable regions against N_{side} to replicate Axelrod's Figure 2.

4. In **Polarization** we assumed that each agent had at most four neighbours, to the north, south, east and west (agents at edges and corners had fewer). This is known as a von Neumann neighbourhood, after the mathematician and computer scientist John von Neumann. An alternative is to also include the four agents to the north-west, north-east, south-west and south-east. This gives a maximum of eight neighbours (again, agents at edges and corners will have fewer). This is known as a Moore neighbourhood, after the mathematician and computer scientist Edward F Moore. Modify **Polarization** such that agents have a Moore neighbourhood rather than a von Neumann neighbourhood. Does this increase or decrease the number of stable regions?

5. Modify **Polarization** (with either von Neumann or Moore neighbourhoods) such that neighbourhoods wrap around to the other side of the grid. In other words, an agent at the extreme west edge of the grid will have neighbours to the north, east and south as previously, but also a neighbour to the west, which will be the agent in the same row at the extreme east of the grid. Agents in the corners wrap around in both the north-south and east-west axes. Every agent therefore has exactly four neighbours assuming von Neumann neighbourhoods, or exactly eight assuming Moore neighbourhoods. Do overlapping edges increase or decrease the number of stable regions?

6. Add mutation to **Polarization**. In each timestep, pick an agent at random, pick one of their features at random, and with a certain probability determined by a user-defined parameter, switch this feature to a different, random trait value. Complete stability will never be reached with mutation, but explore the effect of mutation on the number of approximately stable regions, and the speed with which this approximate stability is reached, compared to the case of no mutation.

References

Axelrod, R. (1997). The dissemination of culture: A model with local convergence and global polarization. Journal of Conflict Resolution, 41(2), 203-226.

Centola, D., Gonzalez-Avella, J. C., Eguiluz, V. M., & San Miguel, M. (2007). Homophily, cultural drift, and the co-evolution of cultural groups. Journal of Conflict Resolution, 51(6), 905-929.

Klemm, K., Eguiluz, V. M., Toral, R., & San Miguel, M. (2005). Globalization, polarization and cultural drift. Journal of Economic Dynamics and Control, 29(1-2), 321-334.

Model 11: Cultural group selection

Introduction

The evolution of cooperation is a major topic across the biological and social sciences. Cooperation is defined as helping another individual, often at some cost to the helper (costly cooperation is sometimes called 'altruism' or 'altruistic cooperation'). Cooperation is rife in nature, from parents looking after their offspring to sterile worker bees feeding their queen. It also underpins all human societies, from sharing food to paying taxes. Yet altruistic cooperation is a puzzle. All else being equal, a defector (or 'free-rider') who never cooperates yet receives help from others will do better than an altruistic cooperator who bears the cost of helping. Consequently, the emergence and maintenance of cooperation requires explanation (West et al. 2007).

This basic free-rider problem applies to the genetic evolution of cooperative behaviour in nature, where costs and benefits are incurred and accrued in the currency of lifetime biological fitness (West et al. 2007). It also applies to human societies where costs and benefits take the form of monetary or other kinds of more immediate payoffs (Apicella & Silk 2019). Examples include taxation systems where free-riding is not paying tax yet benefiting from publicly-funded roads, schools and hospitals, or communal living, where defectors who never wash dishes free-ride on the washing up effort of others. Such systems are characterised by a conflict between the individual and group level: within groups, free-riding individuals do better than cooperating individuals, yet groups of cooperators collectively do better than groups of free-riders. Non-dish-washing free-riders expend less effort than diligent dish-washers yet benefit from using clean dishes washed by others. Yet if everyone free-rides, the dishes quickly pile up.

Solutions to the free-rider problem include kin selection, where cooperators direct help to genetic relatives; reciprocity, where cooperators direct help to others who are likely to return the favour; and punishment, where cooperators punish defectors for not cooperating (West et al. 2007; Apicella & Silk 2019). Yet each of these has limitations with respect to human cooperation: kin selection cannot explain cooperation towards non-kin, which humans do frequently; reciprocity cannot explain cooperation between strangers and breaks down in large groups, which are again common features of human societies; and punishment is itself costly, leading to a second-order free-rider problem where non-punishing cooperators outcompete punishing cooperators.

Consequently, some cultural evolution researchers have suggested that human cooperation can arise via *cultural group selection* (Richerson et al. 2016). This occurs when more-cooperative groups outcompete less-cooperative groups in between-group competition. Genetic group selection does not seem to work: between-group genetic variation is easily broken down by migration, and individual-level natural selection is too strong. Cultural evolution, however, provides better conditions: rapid cultural selection (Model 3) can generate differences between groups, and processes such as conformity (Model 5) or polarisation (Model 10) can maintain between-group cultural variation despite frequent migration (see Model 7). Between-group competition can occur via direct conquest, e.g. warfare, where more-cooperative societies full of self-sacrificial fighters out-compete less-cooperative societies full of deserting back-stabbers. Or it can occur indirectly, such as where more-cooperative societies, e.g. ones with better social welfare systems, attract more migrants than less-cooperative societies. Success in between-group competition does not necessarily require the death of the defeated group's members. Instead, defeated group members can disband and join the winning group.

The evolution of cooperation is a highly contentious topic. Verbal arguments often go round in circles because different scholars have different understandings of terms like cooperation, punishment and group selection. This makes formal models crucial for clarifying assumptions and arguments.

Here we will recreate an influential agent-based model of cultural group selection from Boyd, Gintis, Bowles & Richerson (2003). This model features costly (i.e. altruistic) punishment within groups, mixing / migration between groups, payoff-biased social learning and between-group selection. While it is not the only model of cultural group selection, it provides a good example to help understand and evaluate the hypothesis.

Model 11

Let's introduce the model step by step, with code, before putting it all inside a function. There are N groups each containing a fixed number of n individuals. Individuals come in three types, characterised by three different behaviours: cooperators (C), defectors (D) and punishers (P).

As the model is quite complex, we will use matrices instead of dataframes as they are faster. The code below creates a matrix with N columns, one per group, and n rows, one per agent, to hold the type (C, D or P). As before, we use [row,column] notation to access a specific agent. For example, `agent[3,2]` returns the behaviour of the third agent in the second group. For now we assume $N = 4$ groups with $n = 4$ agents per group. One group has Cs and Ds, one Ps and Ds, one a mix of all three types, and one all Ds.

```r
N <- 4  # 4 groups
n <- 4  # 4 agents per group

# create agent matrix, each column is a group
# can be C (cooperator), D (defector) or P (punisher)
agent <- matrix(nrow = n, ncol = N)

agent[,1] <- c("C", "C", "D", "D")
agent[,2] <- c("D", "D", "P", "P")
agent[,3] <- c("P", "P", "C", "D")
agent[,4] <- c("D", "D", "D", "D")

agent
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "C"  "D"  "P"  "D"
## [2,] "C"  "D"  "P"  "D"
## [3,] "D"  "P"  "C"  "D"
## [4,] "D"  "P"  "D"  "D"
```

Now we create a payoff matrix. This has the same structure as *agent*. Each position in *payoff* holds the payoff for the individual in the equivalent position in *agent*. For example, `payoff[3,2]` holds the payoff of the third agent in group 2. All individuals start with a baseline payoff of 1, from which various costs are subtracted. (NB While this baseline payoff of 1 is not explicitly specified in Boyd et al. 2003, it is needed to avoid negative payoffs. Negative payoffs would mess up subsequent calculations of relative payoffs.)

```r
# create payoff matrix, with baseline payoff 1
payoff <- matrix(1, nrow = n, ncol = N)

payoff
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1
## [3,]    1    1    1    1
## [4,]    1    1    1    1
```

Now we simulate five stages that occur in each timestep.

Stage 1 is cooperation. Cooperators and punishers cooperate with probability $1 - e$ and defect with probability e. The parameter e represents errors, where Cs or Ps defect by mistake when they meant to cooperate. For now we will set $e = 0$ to keep the output the same when you run it, but you can increase it to explore its effect here. We will increase it in the simulations below to $e = 0.01$.

Cooperation reduces the cooperating agent's payoff by c. This makes cooperation costly, i.e. altruistic. Defectors always defect and pay no cost. We set $c = 0.2$, representing a fairly substantial cost.

The following vectorised code implements this. To identify contributors we create a matrix of probabilities, *contribute*, to compare against $1 - e$ for each P and C agent. This gives a matrix *contributors* that is TRUE if the agent contributes and FALSE if they don't. Contributing agents' payoffs are then reduced by c. Because $e = 0$, the matrices will show that all the Ps and Cs pay the cost, and the Ds do not. At this stage, defection pays.

```r
e <- 0   # cooperation error rate
c <- 0.2  # cost of cooperation

# probs for contribution (1-e)
contribute <- matrix(runif(n*N), nrow = n, ncol = N)

# contributors are Ps or Cs with prob 1-e
contributors <- (agent == "P" | agent == "C") & contribute > e

# reduce payoffs of contributing Ps and Cs by c
payoff[contributors] <- payoff[contributors] - c

contributors
```

```
##       [,1]  [,2]  [,3]  [,4]
## [1,]  TRUE FALSE  TRUE FALSE
## [2,]  TRUE FALSE  TRUE FALSE
## [3,] FALSE  TRUE  TRUE FALSE
## [4,] FALSE  TRUE FALSE FALSE
```

```r
payoff
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.8  1.0  0.8    1
## [2,]  0.8  1.0  0.8    1
## [3,]  1.0  0.8  0.8    1
## [4,]  1.0  0.8  1.0    1
```

Stage 2 is punishment. Punishers (Ps) punish every agent in their group who defected in the first stage. Punishment reduces each punished agent's payoff by p/n at a cost of k/n to the punisher. The parameter k makes punishment costly, i.e. altruistic.

Note that punishment scales with the number of defectors: the more punishers there are in a group, the more punishment defectors receive. The more defectors there are, the more costly punishment is to the punisher. We set $p = 0.8$ and $k = 0.2$ such that punishment is more costly for the punished than the punisher. For example, the cost of burning down someone's house is less than the cost of having your house burned down.

In the following code, we first identify groups that have at least one P using *Pgroups*. We only need to apply our punishment routine to these groups. This saves a bit of time. Then we get a matrix of *defections*, those agents who are D or who are P or C with probability e. Then we cycle through each agent i of each punishment group j using two **for** loops. For each punishing agent, we reduce the punisher's payoff by k/n per defector in their group, and each defector's payoff by p/n. Note that to get all group members except the focal individual, we use the negative sign. For example, `payoff[-i,j]` returns all payoffs of agents in group j who are not i.

```r
p <- 0.8  # cost to punished
k <- 0.2  # cost to punisher

# columns/groups with at least one P
Pgroups <- unique(which(agent=="P", arr.ind=TRUE)[,"col"])

# defections (Ds and Ps/Cs with probability e)
defections <- agent == "D" | ((agent == "P" | agent == "C") & contribute <= e)

# cycle thru Pgroups (j) and agents (i)
for (j in Pgroups) {

  for (i in 1:n) {

    # if the agent is P
    if (agent[i,j] == "P") {

      # reduce punisher's payoff by k/n per defection
      payoff[i,j] <- payoff[i,j] - sum(defections[-i,j]) * k / n

      # reduce each defector's payoff by p/n
      payoff[-i,j][defections[-i,j]] <- payoff[-i,j][defections[-i,j]] - p / n

    }
  }

}

payoff
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  0.8  0.6 0.75    1
## [2,]  0.8  0.6 0.75    1
## [3,]  1.0  0.7 0.80    1
## [4,]  1.0  0.7 0.60    1
```

In the resulting *payoff* matrix above, the first and last groups are unchanged because they contain no punishers and so there is no punishment. The Ds of the second group have each been punished by two Ps, reducing their payoffs by $2p/n = 0.4$. The Ps in group 2 pay the cost of punishing two individuals, which is $2k/n = 0.1$. This is subtracted from the 0.8 they already had after paying the costs of cooperation. In the third group, the two Ps each punish the single D via similar calculations. The C is unchanged, as it neither punishes nor is punished.

Note the effect of punishment. Ds now do worst in the second and third groups because they receive punishment. However, in the third group, the C does better than the Ps because punishment is costly. This illustrates the second-order free rider problem that characterises punishment: it's better to let others punish than to do the punishment yourself. But if no one punishes, defectors do best.

Stage 3 is payoff-biased social learning combined with inter-group mixing. With probability $1 - m$, agents interact with a random member of their own group. With probability m, agents interact with a random member of another group. Hence m controls the rate of between-group mixing. We will set $m = 0.01$, giving a 1/100 chance of interacting with members of other groups. Technically this is not quite 'migration', because the original agent remains in its group. But it has the same effect of transmitting behavioural types between groups.

Once a demonstrator is chosen, the focal individual then copies the behaviour (C, D or P) of the demonstrator with probability

$$\frac{W_{dem}}{(W_{dem} + W_{focal})}$$

where W_{dem} is the payoff of the demonstrator and W_{focal} is the payoff of the focal agent, after the cooperation and punishment costs have been subtracted. High-payoff behaviours are therefore more likely to be copied.

The following code implements this by cycling though each agent. We store *agent* as *previous_agent* to make sure that all social leaning occurs from the same population of agents. Otherwise, as we cycle through the agents, the later agents might copy agents who have already copied. Then we cycle through groups (j) and agents (i). Each agent picks a random demonstrator from the same group with probability $1 - m$ and a different group with probability m. *dem* takes two integers that denote the [row,col] coordinates of the demonstrator. This is done with **sample** on a set of numbers excluding self (i) for within-group copying (assuming you cannot copy yourself), and excluding one's own group (j) for other-group copying. Relative fitness W is calculated as per the equation above, and with this probability, the agent adopts the demonstrator's behaviour.

```
m <- 0.01  # rate of between-group mixing

# store agent in previous agent to avoid overlap
previous_agent <- agent

# cycle thru groups (j) and agents (i)
for (j in 1:N) {

  for (i in 1:n) {

    # with prob 1-m, choose demonstrator from same group (excluding self)
    if (runif(1) > m) {

      dem <- c(sample((1:n)[(1:n)!=i], 1), j)

    # with prob m, choose demonstrator from different group
    } else {

      dem <- c(sample(1:n, 1),
               sample((1:N)[(1:N)!=j], 1))

    }

    # get W, relative payoff of demonstrator
    W <- payoff[dem[1],dem[2]] / (payoff[dem[1],dem[2]] + payoff[i,j])

    # copy dem's behaviour with prob W
    # use previous_agent to avoid copying an agent who has already copied
    if (runif(1) < W) {

      agent[i,j] <- previous_agent[dem[1],dem[2]]

    }

  }
}

agent
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "D"  "D"  "C"  "D"
## [2,] "C"  "P"  "P"  "D"
## [3,] "D"  "P"  "D"  "D"
## [4,] "D"  "P"  "P"  "D"
```

Each output will be different, but you should see that some agents in each group have switched to a different behaviour. It will be hard to see here, but over multiple generations higher-payoff behaviours will spread at the expense of lower-payoff behaviours.

Stage 4 is group selection. Groups are paired at random and with probability ε enter into a contest. We set $\varepsilon = 0.5$, an unrealistically high rate of inter-group conflict for demonstration purposes. (Boyd et al. set $\varepsilon = 0.015$ to reflect the estimated rate of inter-group conflict in small-scale societies.)

Groups with more cooperators (i.e. cooperating Cs and Ps) are more likely to win contests. This reflects the assumption that cooperation contributes to group success. More cooperation means greater effort in battle, greater willingness to pay taxes that fund armies, lower likelihood of selling secrets to the enemy, etc., all of which improves a group's chances of winning a contest relative to less-cooperative groups full of back-stabbing, deserting, tax-evading free-riders.

Formally, the probability that group 1 defeats group 2 in a pair is

$$\frac{1}{2} + \frac{(d_2 - d_1)}{2}$$

where d_1 is the proportion of defectors in group 1 of a pair and d_2 is the proportion of defectors in group 2 of the pair. When both groups have the same proportion of defectors, then $d_2 - d_1 = 0$ and there is a 50% chance of either group winning. When group 1 has fewer defectors than group 2, then $d_2 - d_1 > 0$ and group 1 has a greater chance (>50%) of winning. When group 2 has fewer defectors than group 1, then $d_2 - d_1 < 0$ and group 1 has a smaller chance (<50%) of winning.

Groups that lose a contest are replaced in the next timestep with a replica of the winning group, i.e. one that contains the same set of behaviours as the winning group.

The following code implements this group selection. First we pair up each group at random in a dataframe *contests*, with a proportion ε kept and the rest discarded. (We use a dataframe here because sometimes we have only one contest left; a single-row matrix becomes a vector, which becomes a problem later on.) After re-calculating *defections* given the new post-social-learning behaviours, we cycle through each contest and calculate the probability d of the first group winning based on the equation above. With this probability, group 1 wins and group 2 takes on group 1's behaviours. Otherwise group 2 wins and group 1 takes on group 2's behaviours.

```r
epsilon <- 0.5  # frequency of conflict

# dataframe of randomly selected pairs of groups
contests <- as.data.frame(matrix(sample(N), nrow = N/2, ncol = 2))

# keep contests with prob epsilon
contests <- contests[runif(N/2) < epsilon,]

# recalculate defections (Ds and Ps/Cs with probability e)
defections <- agent == "D" | ((agent == "P" | agent == "C") & contribute <= e)

# if there are any contests left
if (nrow(contests) > 0) {

  # cycle thru pairs
  for (i in 1:nrow(contests)) {

    # prob group 1 beats group 2 in pair i
    d1 <- sum(defections[,contests[i,1]]) / n
    d2 <- sum(defections[,contests[i,2]]) / n
    d <- 0.5 + (d2 - d1)/2

    # group 1 wins
    if (runif(1) < d) {

      agent[,contests[i,2]] <- agent[,contests[i,1]]

    # group 2 wins
    } else {

      agent[,contests[i,1]] <- agent[,contests[i,2]]

    }

  }

}

agent
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "D"  "D"  "C"  "D"
## [2,] "C"  "P"  "P"  "D"
## [3,] "D"  "P"  "D"  "D"
## [4,] "D"  "P"  "P"  "D"
```

With $\varepsilon = 0.5$ there is a good chance that one of the groups has been replaced with another (re-run the code above if not). The winning group has effectively replicated themselves, at the expense of the losing group. This is cultural group selection.

Stage 5 is mutation. There is a probability μ that each agent will mutate into one of the other two types (i.e. C into D or P; P into C or D; and D into C or P). This keeps a small, constant supply of new variation coming into the population so that we are not entirely reliant on the starting combination of behaviours.

The following code does this, with an unrealistically high $\mu = 0.5$ for demonstration purposes. Much like in Model 2, we create $N * n$ probabilities for each agent, and if the probability for an agent is less than μ then we mutate into one of the other two types at random. In the resulting *agent* matrix, roughly half of the agents should have switched behaviours.

```
mu <- 0.5  # mutation rate

# probs for mutation
mutate <- runif(N*n)

# store agent in previous agent to avoid overlap
previous_agent <- agent

# mutating D agents
agent[mutate < mu & previous_agent == "D"] <-
  sample(c("P","C"),
         sum(mutate < mu & previous_agent == "D"),
         replace = TRUE)

# mutating C agents
agent[mutate < mu & previous_agent == "C"] <-
  sample(c("P","D"),
         sum(mutate < mu & previous_agent == "C"),
         replace = TRUE)

# mutating P agents
agent[mutate < mu & previous_agent == "P"] <-
```

```
  sample(c("D","C"),
         sum(mutate < mu & previous_agent == "P"),
         replace = TRUE)

agent
```

```
##      [,1] [,2] [,3] [,4]
## [1,] "C"  "P"  "C"  "P"
## [2,] "P"  "P"  "C"  "D"
## [3,] "D"  "P"  "C"  "C"
## [4,] "P"  "P"  "P"  "C"
```

These five stages comprise the events that happen in each timestep of the model, from $t = 1$ to $t = t_{max}$. As is often the case, it's not clear what will happen in the long run. Defectors have higher payoffs within groups, unless they are punished in which case punishers can do better, unless there are cooperators who get the benefit of punishment without its cost. Which type will payoff-biased social learning favour? Group selection favours groups of cooperators over defectors, but will there be enough Cs or Ps in a group to allow this to happen? To find out, we need to simulate over many generations.

The code below puts all five stages together into a single function called **Cooperation**. As in previous models, we add a t-loop iterating all of the five stages, create a dataframe called *output* which records in each timestep the frequency of cooperating agents (Cs and Ps combined) and just the Ps, and end by plotting the results. The plot contains a solid line for overall cooperation (all Cs and Ps), and a dotted line for just the Ps, over all timesteps. The output of the model comprises the final generation *agent* matrix, the *output* dataframe, and the mean cooperation in the last 50% of generations as reported by Boyd et al. (2003).

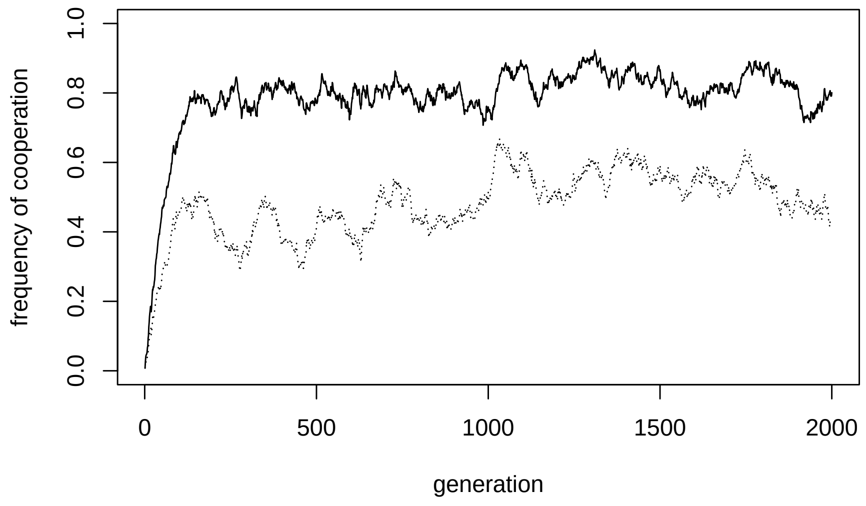
Following Boyd et al., the initial generation consists of one group containing all Ps, and the rest containing all Ds. This reflects a situation where drift or individual learning has created a single punishing group amongst a larger population of defectors. Also following Boyd et al., we assume $N = 128$ groups, so the odds are stacked in the D's favour.

Given that there are lots of parameters, we specify default parameter values from Boyd et al. (2003) so there is no need to remember them.

```
Cooperation <- function(N = 128,
                        n = 4,
                        t_max = 2000,
                        e = 0.02,
                        c = 0.2,
                        m = 0.01,
                        p = 0.8,
                        k = 0.2,
                        mu = 0.01,
                        epsilon = 0.015,
                        show_plot = TRUE) {

  # create agent matrix, each column is a group
  # can be C (cooperator), D (defector) or P (punisher)
  agent <- matrix(nrow = n, ncol = N)

  # initial conditions: group 1 all punishers, others are all defectors
  agent[,1] <- "P"
  agent[,-1] <- "D"

  # create output for freq of cooperation in each timestep
  # and freq of punishment
  output <- data.frame(PandC = rep(NA, t_max),
                       P = rep(NA, t_max))

  # store for t = 1
  output$PandC[1] <- sum(agent == "C" | agent == "P") / (N*n)
  output$P[1] <- sum(agent == "P") / (N*n)

  for (t in 2:t_max) {

    # create/initialise payoff matrix, with baseline payoff 1
    payoff <- matrix(1, nrow = n, ncol = N)

    # 1. Cooperation

    # probs for contribution (1-e)
    contribute <- matrix(runif(n*N), nrow = n, ncol = N)

    # contributors are Ps or Cs with prob 1-e
    contributors <- (agent == "P" | agent == "C") & contribute > e

    # reduce payoffs of contributing Ps and Cs by c
    payoff[contributors] <- payoff[contributors] - c

    # 2. Punishment

    # columns/groups with at least one P
    Pgroups <- unique(which(agent=="P", arr.ind=TRUE)[,"col"])

    # defections (Ds and Ps/Cs with probability e)
    defections <- agent == "D" | ((agent == "P" | agent == "C") & contribute <= e)

    # cycle thru Pgroups (j) and agents (i)
    for (j in Pgroups) {

      for (i in 1:n) {

        # if the agent is P
        if (agent[i,j] == "P") {

          # reduce punisher's payoff by k/n per defection
          payoff[i,j] <- payoff[i,j] - sum(defections[-i,j]) * k / n

          # reduce each defector's payoff by p/n
          payoff[-i,j][defections[-i,j]] <- payoff[-i,j][defections[-i,j]] - p / n

        }

      }

    }

    # 3. Social learning

    # store agent in previous agent to avoid overlap
    previous_agent <- agent

    # cycle thru groups (j) and agents (i)
    for (j in 1:N) {

      for (i in 1:n) {

        # with prob 1-m, choose demonstrator from same group (excluding self)
        if (runif(1) > m) {

          dem <- c(sample((1:n)[(1:n)!=i], 1), j)

        # with prob m, choose demonstrator from different group
        } else {

          dem <- c(sample(1:n, 1),
                   sample((1:N)[(1:N)!=j], 1))

        }
        # get W, relative payoff
        W <- payoff[dem[1],dem[2]] / (payoff[dem[1],dem[2]] + payoff[i,j])

        # copy dem's behaviour with prob W
        # use previous_agent to avoid copying an agent who has already copied
        if (runif(1) < W) {

          agent[i,j] <- previous_agent[dem[1],dem[2]]

        }

      }

    }

    # 4. Group selection

    # dataframe of randomly selected pairs of groups
    contests <- as.data.frame(matrix(sample(N), nrow = N/2, ncol = 2))

    # keep contests with prob epsilon
    contests <- contests[runif(N/2) < epsilon,]

    # recalculate defections (Ds and Ps/Cs with probability e)
    defections <- agent == "D" | ((agent == "P" | agent == "C") & contribute <= e)

    # if there are any contests left
    if (nrow(contests) > 0) {

      # cycle thru pairs
      for (i in 1:nrow(contests)) {

        # prob group 1 beats group 2 in pair i
        d1 <- sum(defections[,contests[i,1]]) / n
        d2 <- sum(defections[,contests[i,2]]) / n
        d <- 0.5 + (d2 - d1)/2

        # group 1 wins
        if (runif(1) < d) {

          agent[,contests[i,2]] <- agent[,contests[i,1]]

        # group 2 wins
        } else {
          agent[,contests[i,1]] <- agent[,contests[i,2]]

        }

      }

    }

    # 5. Mutation

    # probs for mutation
    mutate <- runif(N*n)

    # store agent in previous agent to avoid overlap
    previous_agent <- agent

    # mutating D agents
    agent[mutate < mu & previous_agent == "D"] <-
      sample(c("P","C"),
             sum(mutate < mu & previous_agent == "D"),
             replace = TRUE)

    # mutating C agents
    agent[mutate < mu & previous_agent == "C"] <-
      sample(c("P","D"),
             sum(mutate < mu & previous_agent == "C"),
             replace = TRUE)

    # mutating P agents
    agent[mutate < mu & previous_agent == "P"] <-
      sample(c("D","C"),
             sum(mutate < mu & previous_agent == "P"),
             replace = TRUE)

    # 6. Record freq of cooperation

    output$PandC[t] <- sum(agent == "C" | agent == "P") / (N*n)
    output$P[t] <- sum(agent == "P") / (N*n)

  }

  if (show_plot == TRUE) {

    plot(x = 1:nrow(output),
         y = output$PandC,
         type = 'l',
         ylab = "frequency of cooperation",
         xlab = "generation",
         ylim = c(0,1))

    # dotted line for freq of Ps
    lines(x = 1:nrow(output),
          y = output$P,
          type = 'l',
          lty = 3)

  }

  # output final agent, full output, and mean cooperation of last 50% of timesteps
  list(agent = agent,
       output = output,
       mean_coop = mean(output$PandC[(t_max/2):t_max]))

}
```

One run of the model with default values gives the following plot:

```
data_model11 <- Cooperation()
```



In small groups of $n = 4$, cooperation increases to around 0.8. For the above output, the mean cooperation in the final 50% of timesteps is 0.83. A slight majority are Ps, with the rest Cs.

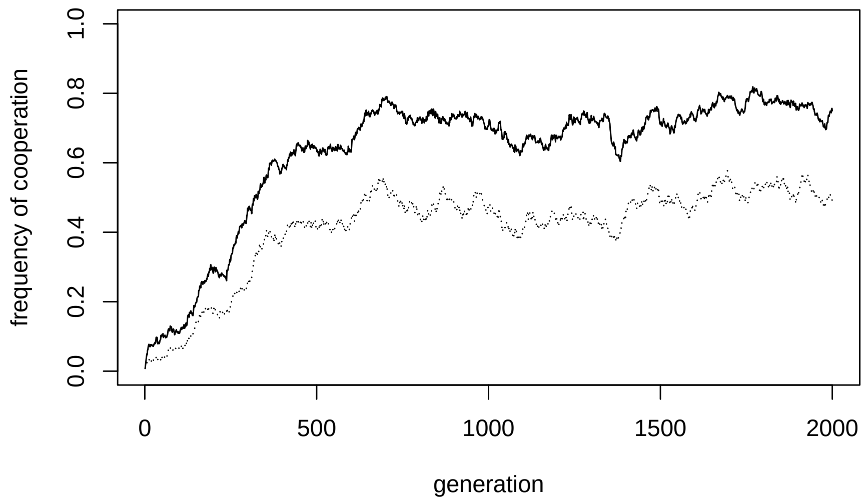
A look at a sample of the final generation population shows that groups are quite homogenous, comprising either all Ps, all Cs or all Ds. There are some dissimilar agents, most likely due to mutation or between-group social learning. This within-group homogeneity and between-group heterogeneity is a hallmark of cultural group selection.

```r
data_model11$agent[,1:25]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
## [1,] "C"  "P"  "P"  "P"  "C"  "P"  "C"  "D"  "C"  "D"   "P"   "D"   "P"   "C"  
## [2,] "C"  "P"  "P"  "P"  "C"  "P"  "C"  "D"  "C"  "D"   "P"   "D"   "C"   "C"  
## [3,] "C"  "P"  "P"  "P"  "C"  "P"  "C"  "D"  "C"  "D"   "P"   "D"   "D"   "C"  
## [4,] "C"  "P"  "P"  "P"  "C"  "P"  "C"  "D"  "C"  "D"   "P"   "D"   "C"   "C"  
##      [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25]
## [1,] "C"   "P"   "P"   "C"   "D"   "C"   "C"   "P"   "C"   "P"   "C"  
## [2,] "C"   "P"   "P"   "C"   "D"   "C"   "C"   "P"   "C"   "P"   "P"  
## [3,] "C"   "P"   "P"   "C"   "D"   "C"   "C"   "P"   "C"   "P"   "C"  
## [4,] "C"   "P"   "P"   "C"   "D"   "C"   "C"   "P"   "C"   "P"   "C"
```

Increasing the group size to $n = 32$ reduces the frequency of cooperation slightly, to around 0.7.

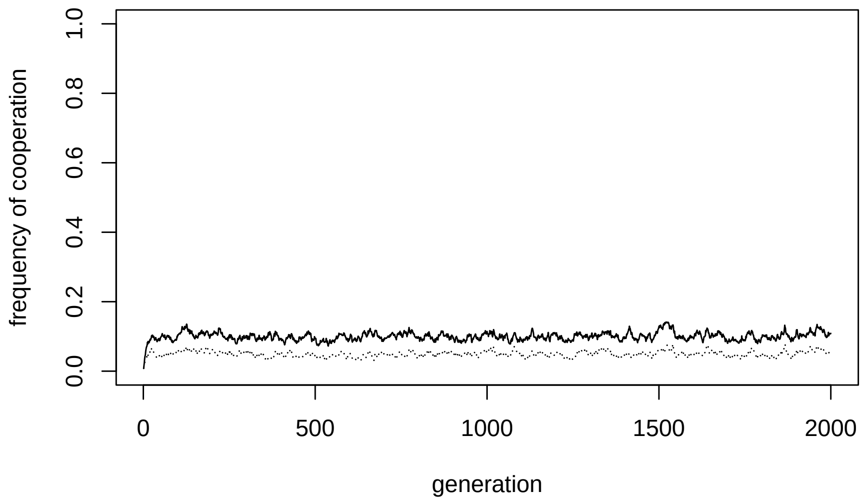
```r
data_model11 <- Cooperation(n = 32)
```



In the plot above, the mean cooperation in the last 50% of timesteps is 0.72. It is a well-established finding that cooperation is harder to maintain in larger groups, but here cooperation only slightly declines.

By setting $p = k = 0$ we can remove punishment from the model:

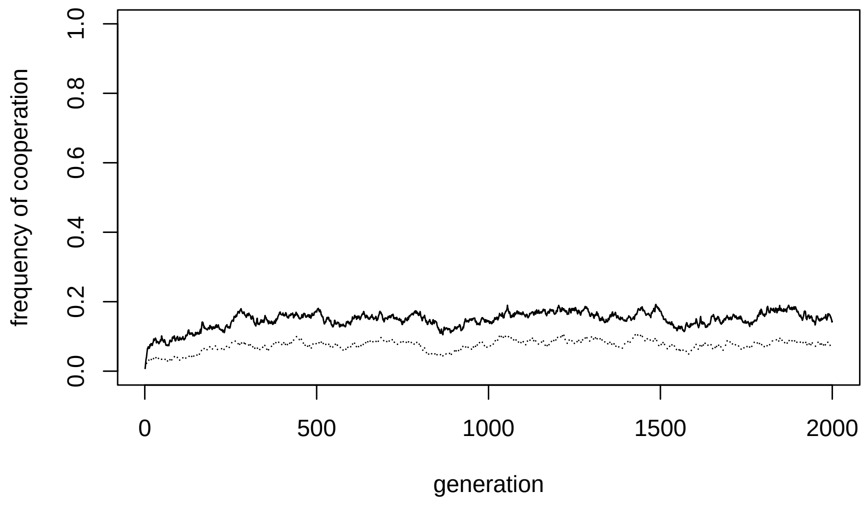
```r
data_model11 <- Cooperation(n = 32, p = 0, k = 0)
```



Without punishment, cooperation almost disappears (in the plot above, the frequency of cooperation is 0.1). This shows that, even with group selection, some mechanism is needed to maintain cooperation within groups. These cooperative groups are then favoured by group selection. But if there is no such mechanism, group selection has nothing to select.

Conversely, we can remove group selection by setting the probability of intergroup competition $\varepsilon = 0$, and restoring punishment.

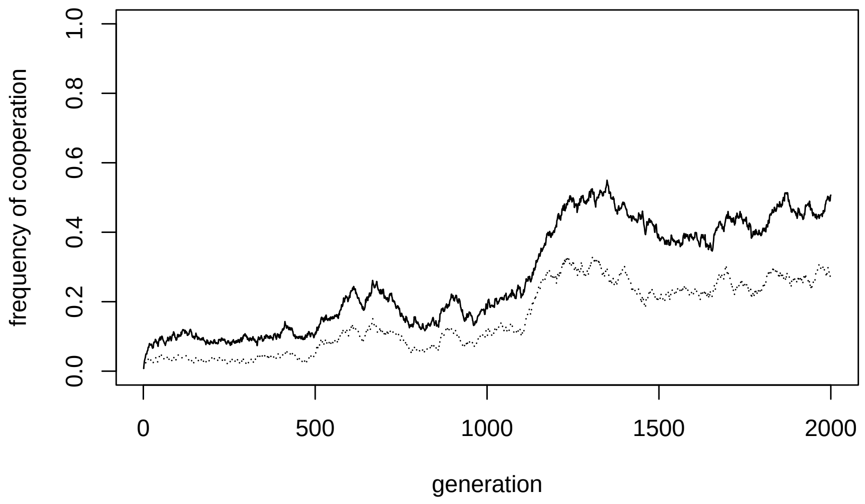
```r
data_model11 <- Cooperation(n = 32, epsilon = 0)
```



Again, cooperation has all but disappeared (to 0.16 in the above plot). Without group selection, cooperation has no benefit, and is selected against.

Now let's increase the inter-group mixing rate from $m = 0.01$ to $m = 0.05$:

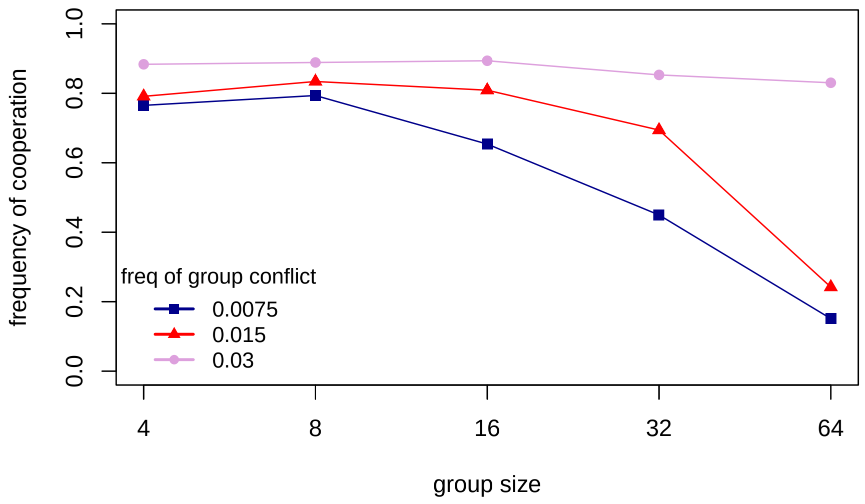
```
data_model11 <- Cooperation(n = 32, m = 0.05)
```



Inter-group mixing reduces cooperation (in the plot above to 0.41). Mixing, like migration, breaks down between-group variation and prevents group selection from acting. Because mixing operates alongside payoff-biased social learning, and defectors typically have higher fitness than cooperators, this results in the spread of defectors.

Finally, let's recreate one of the figures from Boyd et al. (2003). Their Figure 1b shows the frequency of cooperation across a range of group sizes (n), for three different rates of intergroup conflict (ε). The following code recreates this figure. Warning: it can take a while to run so many simulations. (NB Boyd et al. took an average of 10 independent runs for each parameter combination and went up to $n = 256$; the code below only has one run per parameter combination and goes up to $n = 64$. However, the results are qualitatively the same.)

```
n <- c(4,8,16,32,64)
epsilon <- c(0.0075,0.015,0.03)

output <- data.frame(matrix(NA, ncol = length(epsilon), nrow = length(n)))
rownames(output) <- n
colnames(output) <- epsilon

for (j in 1:length(epsilon)) {

  for (i in 1:length(n)) {

    output[i,j] <- Cooperation(n = n[i],
                              epsilon = epsilon[j],
                              show_plot = FALSE)$mean_coop

  }

}

plot(x = 1:length(n), y = output[,1],
     type = 'o',
     ylab = "frequency of cooperation",
     xlab = "group size",
     col = "darkblue",
     pch = 15,
     ylim = c(0,1),
     xlim = c(1,length(n)),
     xaxt = "n")

axis(1, at=1:length(n), labels=n)

lines(x = 1:length(n), output[,2],
      type = 'o',
      col = "red",
      pch = 17)

lines(x = 1:length(n), output[,3],
      type = 'o',
      col = "plum",
      pch = 16)

legend("bottomleft",
        legend = epsilon,
        title = "freq of group conflict",
        lty = 1,
        lwd = 2,
        pch = c(15,17,16),
        col = c("darkblue", "red","plum"),
        bty = "n",
        cex = 0.9)
```



The above figure shows that (i) cooperation declines with group size; (ii) higher rates of inter-group conflict maintain higher frequencies of cooperation; and (iii) at the highest rates of inter-group conflict, cooperation is maintained even in large groups.

Summary

Cooperation underpins the social lives of countless species, including (and perhaps especially) human societies. Given the advantage that free-riders have over cooperators, cooperation requires special explanation. Many such explanations have been offered. Model 11, recreating a model by Boyd et al. (2003), provides one such explanation, combining altruistic punishment, payoff-biased social learning and cultural group selection.

We found, as did Boyd et al. (2003), that cooperation can be maintained in relatively large groups when these processes are acting together. Within groups, payoff-biased social learning is the selection mechanism. Because defectors have higher payoffs than cooperators, defection is selected and spreads within groups. Adding punishers turns the tables, with defectors' payoffs reduced by punishment. Yet without group selection, altruistic punishers will be out-competed by non-punishing cooperators, who benefit from others' punishment but do not pay the costs of punishing. But then we are back where we started, and defectors will out-compete the non-punishing cooperators. With group selection, however, groups of cooperating punishers spread at the expense of groups of defectors due to the benefit of cooperation in inter-group competition. Hence, cooperation spreads. Neither punishment nor group selection alone maintain cooperation; both are needed.

The model presented here is one version of a broad class of models of cultural group selection (Smith 2020). This one involves punishment. Others assume that rapid cultural adaptation or conformity maintain between-group cultural variation, and inter-group competition favours more-cooperative groups. Some models assume preferential migration plus acculturation (see Model 7) rather than group extinction/replacement as we simulated here. There is no single 'cultural group selection' hypothesis, there are several. It is important to recognise that within-group mechanisms like punishment are not alternatives to cultural group selection, but rather are complementary. In Model 11, punishment favours cooperation within groups, while group selection solves the second-order free-rider problem that comes with punishment.

Cultural group selection is a controversial explanation for human cooperation (see Richerson et al. 2016 and associated commentaries; and also Smith 2020). This may be a legacy of the rejection of naive genetic group selection in biology in the 1960s. However, cultural group selection is different: it involves cultural rather than genetic variation, it applies only to humans, and it incorporates special assumptions about biased social learning, inter-group interactions etc. Nevertheless, models like Model 11 are just that: models. Evidence is needed to demonstrate that cultural group selection has been an important driver of cooperation in real human societies. Some such evidence exists, such as findings of substantial between-group cultural variation in cooperation-related behaviours (Richerson et al. 2016). Other evidence opposes cultural group selection, such as findings that people do not reliably socially learn cooperative behaviour (Lamba 2014). The value of models like Model 11 lies in clarifying theoretical assumptions and predictions, allowing those assumptions and predictions to be tested empirically.

In terms of programming, Model 11 is probably the most complex in this series so far. There are several stages within each timestep, and more parameters than any previous model. As a result, run times can be slow, particularly for large groups. Where possible we used matrices rather than dataframes and vectorised the code to improve speed. Unfortunately there is no straightforward way of vectorising the punishment and social learning stages (if you can think of one let me know!). We must accept that sometimes complex models take a while to run. Grab a cup of tea and watch TV, smug in the knowledge that your code is working for you in the background. More seriously, the important thing with complex models is to make and test each stage in turn under manageable assumptions (e.g. $N = 4$ rather than $N = 128$) before putting them all together, as we did above. This makes it much easier to catch bugs and to make sure that your code does what you think it does.

Exercises

1. Use the **Cooperation** function to explore the effect of the remaining parameters on the frequency of cooperation: (a) the error rate e; (b) the cost of cooperation c; (c) the mutation rate μ; and (d) the number of groups N.

2. Recreate Boyd et al.'s (2003) other figures, adapting the code above used to recreate their Figure 1b. Do you get the same results?

3. Change the starting conditions, to (a) one group of all Cs and the rest all Ds; (b) all Ds (with Ps and Cs only appearing via mutation); and (c) random behaviours. Does cooperation still emerge with these different starting conditions?

4. Add a new parameter b to the **Cooperation** function. Each cooperator (Cs and Ps with probability $1 - e$) generates a payoff benefit b which is shared equally amongst all group members (including defectors). This simulates the standard 'Public Goods Game' from economics. Now cooperation generates within-group benefits, as well as between-group benefits via group selection. Explore the effect of different values of b on overall cooperation levels.

References

Apicella, C. L., & Silk, J. B. (2019). The evolution of human cooperation. Current Biology, 29(11), R447-R450.

Boyd, R., Gintis, H., Bowles, S., & Richerson, P. J. (2003). The evolution of altruistic punishment. Proceedings of the National Academy of Sciences, 100(6), 3531-3535.

Lamba, S. (2014). Social learning in cooperative dilemmas. Proceedings of the Royal Society B, 281(1787), 20140417.

Richerson, P., Baldini, R., Bell, A. V., Demps, K., Frost, K., Hillis, V., … & Zefferman, M. (2016). Cultural group selection plays an essential role in explaining human cooperation: A sketch of the evidence. Behavioral and Brain Sciences, 39, E30.

Smith, D. (2020). Cultural group selection and human cooperation: a conceptual and empirical review. Evolutionary Human Sciences, 2.

West, S. A., Griffin, A. S., & Gardner, A. (2007). Evolutionary explanations for cooperation. Current Biology, 17(16), R661-R672.

Model 12: Historical dynamics

Introduction

As discussed in the context of Model 11 (Cultural Group Selection), cooperation is crucial to the functioning of large human societies. While Model 11 was all about how cooperation emerges within groups, in Model 12 we will zoom out to examine how cooperation, specifically the mechanism of cultural group selection, can determine broader historical dynamics at the level of the group and above.

In particular, we are interested in how large multi-group societies, or *empires*, rise and fall throughout human history. The rise and fall of empires is a standard historical pattern. At any point in history many empires exist. These empires compete with one another, some growing and becoming dominant for a time (think the Roman Empire), and all eventually collapsing (again, like the Roman Empire). Unlike in our previous models, there are no stable equilibria in human history. The 'best' empire or empires (whatever 'best' might mean) do not reach a stable size and stay there. Rather, there are continual oscillations, with new empires constantly emerging and growing, and existing empires constantly declining and disappearing. These dynamics are also often described as *chaotic*, in that it is virtually impossible to predict at any one time point exactly which empires will subsequently grow or collapse. Nevertheless, the overall dynamics of multiple co-existing, competing states going through cycles of emergence, dominance and collapse are worthy of explanation.

Turchin (2003) presented a cultural evolutionary theory of how empires rise and fall through human history. Turchin's theory drew on the notion of cultural group selection (see Model 11), but combined this with historical detail concerning the various pressures acting on real empires from history. In Model 12 we will recapitulate the spatially explicit agent-based simulation of this theory from Chapter 4 of Turchin's (2003) book.

One key element of Turchin's (2003) cultural evolutionary theory of historical dynamics is the concept of *asabiya*. This term was coined by 14th century

Islamic scholar Ibn Khaldun to describe the degree of within-group cooperation, or collective solidarity, possessed by a group. Members of groups high in asabiya generate public goods together, fight for one another, defend each other, monitor and punish free-riders, and generally engage in the kind of costly cooperative acts described in Model 11. Members of groups low in asabiya free-ride on each other, desert during conflicts, shirk costly punishment of free-riders, and engage in other non-cooperative acts. As in Model 11, groups higher in asabiya do better in intergroup conflict than groups lower in asabiya. This is cultural group selection.

A second key element of Turchin's (2003) theory is the importance of frontier regions. These are regions where different empires or groups of different ethnicities meet. Turchin argues that asabiya is highest at these frontiers because of the presence and threat of competing groups. As modern day examples, religious beliefs tend to be stronger in regions like Northern Ireland or the Middle East where different competing religions exist in close proximity, compared to more religiously homogenous regions.

Turchin's theory works as follows. In frontier regions, small groups high in asabiya engage in frequent intergroup conflict. Eventually one of these groups expands and takes over rival groups, becoming a multi-group empire. As this empire expands, its internal non-frontier region gets larger, causing the empire's overall asabiya to drop. At the frontiers, new groups emerge that are high in asabiya. One of these grows big enough to invade the previous empire, which is weakened by its low asabiya. The new empire thus replaces the old empire. The new empire grows larger, its asabiya drops, and it in turn is invaded by a new empire that has emerged at its frontier. This cycle continues, generating the oscillatory dynamics characteristic of real human history.

Model 12

Model 12 simulates this process. The agents in Model 12 are groups. Individuals within groups are not explicitly modelled. Each group exists in one fixed position within an N_{side} x N_{side} square grid, like in Model 10 (Polarisation). Groups can either be independent entities existing outside of any empire, or they can belong to an empire. Each empire is denoted by a number, e.g. Empire 1, Empire 2 and so on. We use a matrix called E to store the empire id of each group: 0 indicates no empire, a positive integer indicates an empire. As in Model 10 (Polarisation), the columns and rows of the matrix give the x and y spatial coordinates of the group. For example, `E[3,6]` gives the group in the 3rd row down and 6th column across, while `E[3,7]` gives its neighbour to the east. We start with all non-empire agents, and $N_{side} = 10$:

```
# N_side x N_side grid
N_side <- 10
```

```
# matrix for empire id, initially all 0 (no empire)
E <- matrix(0, nrow = N_side, ncol = N_side)
```

We initialise the simulation with a single 4 x 4 empire in a random internal position. This is labelled Empire 1, and can be seen in the E matrix.

```
# create a starting 4x4-cell empire 1
row_E1 <- sample(3:(N_side-5), 1)
col_E1 <- sample(3:(N_side-5), 1)
E[row_E1:(row_E1+3),col_E1:(col_E1+3)] <- 1

E
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    0    0    0    0    0    0    0     0
##  [2,]    0    0    0    0    0    0    0    0    0     0
##  [3,]    0    0    0    0    0    0    0    0    0     0
##  [4,]    0    0    0    0    1    1    1    1    0     0
##  [5,]    0    0    0    0    1    1    1    1    0     0
##  [6,]    0    0    0    0    1    1    1    1    0     0
##  [7,]    0    0    0    0    1    1    1    1    0     0
##  [8,]    0    0    0    0    0    0    0    0    0     0
##  [9,]    0    0    0    0    0    0    0    0    0     0
## [10,]    0    0    0    0    0    0    0    0    0     0
```

Each group also has a value of asabiya ranging from 0 (no cooperation) to 1 (maximum cooperation). We create another matrix, S, containing these values. All groups initially have $S = 0.1$.

```
# matrix for asabiya, S
S <- matrix(0.1, nrow = N_side, ncol = N_side)

S
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [2,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [3,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [4,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [5,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [6,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [7,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [8,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
##  [9,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
## [10,]  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1   0.1
```

Next we create an *output* dataframe. Our measure of interest is the area of each empire at each generation. When this is plotted, we will be able to see whether one or more empires reaches equilibrium size, whether all empires disappear, or whether we see the more realistic rise and fall of different empires over time.

The code below creates a dataframe with three variables, *generation*, *empire* and *area*. We add the area of Empire 1 at generation 1, which is 4 x 4 = 16 groups. Note that unlike previous output dataframes, we can't rely on each row representing one generation. Sometimes there will be more than one empire present in one generation. Because we don't know how many empires there will be, we start with generation 1 and later add rows depending on how many empires are present in each generation.

```
# record area of empire 1 in output dataframe
output <- data.frame(generation = 1,
                     empire = 1,
                     area = sum(E == 1))

output
```

```
##   generation empire area
## 1          1      1   16
```

The final step before starting the generation loop is to create a variable *max_empire* to keep track of the maximum empire id. This will be used later when new empires are created. Right now it's set to 1.

```
# store highest empire number, to subsequently create new empires
max_empire <- 1
```

Now we write code for events that repeat in each generation. First, the asabiya of each group is updated according to whether it is on a frontier or not. Groups that have at least one neighbour to the north, south, east or west which belongs to a different empire increase in asabiya. Groups with no dissimilar neighbours decrease in asabiya.

To implement this, we could cycle through each group using a **for**-loop, and for each one cycle through each of its four neighbours using another **for**-loop, tracking whether each is similar or dissimilar. As mentioned previously, however, running so many loops takes a long time in R. A faster and more efficient approach is to use a few lines of vectorised code instead.

In the code below, we create a matrix *dissimilar_neighbour* to record whether each group has at least one dissimilar neighbour, initialised with zeroes. We then compare E with a duplicate of E shifted up, left, down and right. If these north, west, south and east neighbours are dissimilar to (i.e. not equal to, or !=) E, then 1 is added to *dissimilar_neighbour*.

```
# 1. Update asabiya (S)

# matrix of whether group has at least one dissimilar-empire neighbour
dissimilar_neighbour <- matrix(0, nrow = N_side, ncol = N_side)

# south: compare E with 1st row removed and duplicate 21st row (shifted up) vs regular E
dissimilar_neighbour <- dissimilar_neighbour + (rbind(E[-1,], E[N_side,]) != E)

# east: compare E with 1st col removed and duplicate 21st col (shifted left) vs regular E
dissimilar_neighbour <- dissimilar_neighbour + (cbind(E[,-1], E[,N_side]) != E)

# north: compare E with duplicate 1st row (shifted down) vs regular E
dissimilar_neighbour <- dissimilar_neighbour + (rbind(E[1,], E[-N_side,]) != E)

# west: compare E with duplicate 1st column (shifted right) vs regular E
dissimilar_neighbour <- dissimilar_neighbour + (cbind(E[,1], E[,-N_side]) != E)

dissimilar_neighbour
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    0    0    0    0    0    0    0     0
##  [2,]    0    0    0    0    0    0    0    0    0     0
##  [3,]    0    0    0    0    1    1    1    1    0     0
##  [4,]    0    0    0    1    2    1    1    2    1     0
##  [5,]    0    0    0    1    1    0    0    1    1     0
##  [6,]    0    0    0    1    1    0    0    1    1     0
##  [7,]    0    0    0    1    2    1    1    2    1     0
##  [8,]    0    0    0    0    1    1    1    1    0     0
##  [9,]    0    0    0    0    0    0    0    0    0     0
## [10,]    0    0    0    0    0    0    0    0    0     0
```

As shown above, there are 24 groups who have exactly one dissimilar neighbour (16 nonempire groups that border one Empire 1 group, and 8 Empire 1 groups that border one nonempire group), and four groups with two dissimilar neighbours (the four corner Empire 1 groups that each border two nonempire groups).

In fact it's not important in Turchin's model how many dissimilar neighbours a group has. We therefore convert *dissimilar_neighbour* to a TRUE/FALSE matrix, TRUE indicating at least one dissimilar neighbour, FALSE indicating no dissimilar neighbours.

```
# remove the multiple borders, reduce to TRUE/FALSE
dissimilar_neighbour <- dissimilar_neighbour > 0
```

```
dissimilar_neighbour
```

```
##        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
##  [1,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [2,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [3,] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
##  [4,] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
##  [5,] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
##  [6,] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
##  [7,] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
##  [8,] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
##  [9,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [10,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

The asabiya of groups with at least one dissimilar neighbour (`dissimilar_neighbour == TRUE`) increases according to the following equation:

$$S_t = S_{t-1} + r_0 S_{t-1}(1 - S_{t-1})$$

This describes logistic (S-shaped) growth in asabiya, where S_t is the group's asabiya in generation t, S_{t-1} is that group's asabiya in the previous generation and r_0 determines the rate of increase. A logistic function features a slow initial rate of increase, followed by a rapid increase, then a slow rate of increase again. Turchin argues that a logistic function is appropriate because of the way cooperation increases within groups. The slow initial increase describes how it is initially hard for cooperators to establish themselves amongst a majority of free-riders; the rapid increase occurs because cooperation gets easier as there are more cooperators present and cooperators can more easily assort; the final slowdown occurs as fewer free-riders are left to switch to cooperation.

The asabiya of groups with no dissimilar neighbours (`dissimilar_neighbour == FALSE`) decreases according to the following equation:

$$S_t = S_{t-1} - \delta S_{t-1}$$

This describes exponential decline with δ determining the rate of decline. Exponential decline represents the intrinsic evolutionary advantage of free-riding relative to cooperation: once some free-riders are present, cooperation quickly declines as cooperators refuse to be exploited by those free-riders.

Note that these functions (logistic for S increasing at frontiers and exponential for S decreasing at non-frontiers) are simply Turchin's choices, and are driven partly by mathematical convenience and partly by the reasoning described above. Others might disagree with these assumptions, and that's fine. They - and you -

are free to change them to see how robust the results are to different assumptions, or to better fit the evidence.

The following code implements these two equations, with the default values $r_0 = 0.2$ and $\delta = 0.1$ used by Turchin. We can see the increased asabiya of frontier groups and decreased asabiya of non-frontier groups in the S matrix.

```
r_0 <- 0.2
delta <- 0.1

# increase S of groups with dissimilar neighbours according to r_0
S[dissimilar_neighbour] <- S[dissimilar_neighbour] +
  r_0 * S[dissimilar_neighbour] * (1 - S[dissimilar_neighbour])

# decrease S of groups without dissimilar neighbours according to delta
S[!dissimilar_neighbour] <- S[!dissimilar_neighbour] -
  delta * S[!dissimilar_neighbour]

S
```

```
##       [,1] [,2] [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
##  [1,] 0.09 0.09 0.09 0.090 0.090 0.090 0.090 0.090 0.090  0.09
##  [2,] 0.09 0.09 0.09 0.090 0.090 0.090 0.090 0.090 0.090  0.09
##  [3,] 0.09 0.09 0.09 0.090 0.118 0.118 0.118 0.118 0.090  0.09
##  [4,] 0.09 0.09 0.09 0.118 0.118 0.118 0.118 0.118 0.118  0.09
##  [5,] 0.09 0.09 0.09 0.118 0.118 0.090 0.090 0.118 0.118  0.09
##  [6,] 0.09 0.09 0.09 0.118 0.118 0.090 0.090 0.118 0.118  0.09
##  [7,] 0.09 0.09 0.09 0.118 0.118 0.118 0.118 0.118 0.118  0.09
##  [8,] 0.09 0.09 0.09 0.090 0.118 0.118 0.118 0.118 0.090  0.09
##  [9,] 0.09 0.09 0.09 0.090 0.090 0.090 0.090 0.090 0.090  0.09
## [10,] 0.09 0.09 0.09 0.090 0.090 0.090 0.090 0.090 0.090  0.09
```

The second within-generation event is intergroup conflict. Each non-edge group is chosen once per generation in a random order to potentially attack their neighbouring groups. The chosen attacker cycles through each of its four north-south-east-west neighbours (its von Neumann neighbourhood) again in a random order. For each neighbour which is of a different empire to the attacker, we compare the power P of the attacker and the defender. The power P of group x which belongs to empire y is given by:

$$P_x = A_y \bar{S}_y \exp(-d_{x,y}/h)$$

where A_y is the size in number of groups of empire y, $\bar{S}_y$ is the mean asabiya of all groups belonging to empire y, $d_{x,y}$ is the distance from group x to the centre of empire y, and h is a constant. The centre of an empire is calculated by taking the mean row number and mean column number of all groups belonging to empire y. The distance $d_{x,y}$ is then the Euclidean distance between the group and the centre.

This equation says that the power of a group increases with the size and average asabiya of its parent empire, and declines with its distance from the centre of the empire. The latter is an exponential decline determined by the constant h. Again, these are assumptions made by Turchin, but seem plausible: groups belonging to larger, more cohesive empires will have more resources and motivation to fight, while groups further from the empire centre will suffer supply-chain and communication difficulties which reduce their ability to fight.

For nonempire groups, who are essentially a mini-empire of one group, $A_y = 1$, $\bar{S}_y = \bar{S}_x$, and $d_{x,y} = 0$.

Once the power of both attacker, P_{att}, and defender, P_{def}, are calculated, their difference is compared against a threshold δ_P:

$$P_{att} - P_{def} > \delta_P$$

If this inequality is satisfied, then the attacker successfully defeats the defender. If not, then nothing happens. If an attack is successful, the defender joins the empire of the attacker, and the defender's asabiya S is set to the mean of its previous S and the attacker's S. If the attacker is a nonempire group ($E = 0$), then a new empire forms. This is given the label *max_empire* + 1, and *max_empire* is then incremented by 1.

The following code implements all of the above description. It's rather long, but the comments should link back to each of the steps above.

```r
h <- 2  # rate of decline in power with distance from empire centre
delta_P <- 0.1  # threshold for difference in power between attacker and defender

# 2. Intergroup conflict

# each non-edge cell is chosen in random order to attack their NSEW neighbouring cells

# cells to enter conflicts, excluding edge cells (row/col 1 and 21) which do not attac
attacker <- as.vector(matrix(1:(N_side^2),N_side,N_side)[-c(1,N_side),-c(1,N_side)])

# randomise the order of attackers
attacker <- sample(attacker)

for (i in attacker) {

  # random order of NSEW neighbours to attack
  defender <- sample(c(i+1,i-1,i+N_side,i-N_side))
```

```r
# cycle thru neighbours
for (j in defender) {

  # if defender j is a valid opponent
  # and attacker and defender are of different empires, or both are non-empires
  if (j %in% attacker & (E[j] != E[i] | (E[i] == 0 & E[j] == 0))) {

    # non-empire cells have area A=1, their own S, and d=0
    if (E[i] == 0) {

      A_att <- 1
      S_att <- S[i]
      d_att <- 0

    }

    if (E[j] == 0) {

      A_def <- 1
      S_def <- S[j]
      d_def <- 0

    }

    # empire cells have area A of their empire, mean S of their empire,
    #and d distance from centre of their empire
    if (E[i] > 0) {

      A_att <- sum(E == E[i])
      S_att <- mean(S[E == E[i]])

      col_centre <- mean(which(E == E[i], arr.ind = T)[,"col"])
      row_centre <- mean(which(E == E[i], arr.ind = T)[,"row"])
      Row <- (i-1) %% N_side + 1
      Col <- ceiling(i/N_side)
      d_att <- sqrt((Row - row_centre)^2 + (Col - col_centre)^2)

    }

    if (E[j] > 0) {

      A_def <- sum(E == E[j])
      S_def <- mean(S[E == E[j]])

      col_centre <- mean(which(E == E[j], arr.ind = T)[,"col"])
```

```
        row_centre <- mean(which(E == E[j], arr.ind = T)[,"row"])
        Row <- (j-1) %% N_side + 1
        Col <- ceiling(j/N_side)
        d_def <- sqrt((Row - row_centre)^2 + (Col - col_centre)^2)

      }

      # power of attacker and defender
      P_att <- A_att * S_att * exp(-d_att / h)
      P_def <- A_def * S_def * exp(-d_def / h)

      # if P_att - P_def is greater than delta_P, then attack is successful
      if (P_att - P_def > delta_P) {

        # if attacker is already part of an empire
        if (E[i] > 0) {

          # defender adopts empire of attacker
          E[j] <- E[i]

        # if attacker is a nonempire cell
        } else {

          # attacker and defender become a new empire
          E[i] <- max_empire + 1
          E[j] <- max_empire + 1

          # update max_empire
          max_empire <- max_empire + 1

        }

        # all defenders adopt the average of their prior S and the attacker's S
        S[j] <- (S[j]+S[i])/2

      }

    }

  }

}

E
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    0    0    0    0    0    0    0     0
##  [2,]    0    0    1    1    1    1    1    1    0     0
##  [3,]    0    0    1    1    1    1    1    1    1     0
##  [4,]    0    0    1    1    1    1    1    1    1     0
##  [5,]    0    1    1    1    1    1    1    1    1     0
##  [6,]    0    0    1    1    1    1    1    1    1     0
##  [7,]    0    0    0    1    1    1    1    1    1     0
##  [8,]    0    0    0    1    1    1    1    1    1     0
##  [9,]    0    0    0    0    1    1    1    1    1     0
## [10,]    0    0    0    0    0    0    0    0    0     0
```

Inspection of E shows that Empire 1 has expanded by taking over several nonempire groups. This makes sense given the power equation. Empire groups have greater power due to the larger size $A = 16$ of Empire 1, compared to the $A = 1$ of nonempire cells. Asabiya S is roughly the same for empire and nonempire groups on the frontier (if anything it's lower for empire groups, as their $\bar{S}$ will include nonfrontier groups), and Empire 1 is too small for distance effects to matter much.

The third within-generation event is empire collapse. If the mean asabiya, S, of an empire is less than a threshold S_{crit}, that empire is dissolved and all of its groups become nonempire groups ($E = 0$). The following code does this, remembering that we don't know before each generation which empires exist.

```r
S_crit <- 0.003  # mean asabiya below which empires collapse

# 3. Imperial collapse

# if mean S of an empire is less than S_crit, empire collapses

# list of empires (excluding zero/nonempires)
empires <- unique(as.vector(E[E>0]))

# if there are any empires left
if (length(empires) > 0) {

  # for each empire
  for (i in 1:length(empires)) {

    if (mean(S[E == empires[i]]) < S_crit) {

      # dissolve empire
      E[E == empires[i]] <- 0

    }
  }

}

E
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    0    0    0    0    0    0    0     0
##  [2,]    0    0    1    1    1    1    1    1    0     0
##  [3,]    0    0    1    1    1    1    1    1    1     0
##  [4,]    0    0    1    1    1    1    1    1    1     0
##  [5,]    0    1    1    1    1    1    1    1    1     0
##  [6,]    0    0    1    1    1    1    1    1    1     0
##  [7,]    0    0    0    1    1    1    1    1    1     0
##  [8,]    0    0    0    1    1    1    1    1    1     0
##  [9,]    0    0    0    0    1    1    1    1    1     0
## [10,]    0    0    0    0    0    0    0    0    0     0
```

Nothing will have changed here, as the mean asabiya of Empire 1 of 0.109 is much higher than our default $S_{crit} = 0.003$.

The fourth within-generation event is to deal with groups around the grid edges. In this model edge cells are 'reflecting', which means that they take the E and S values of the nearest non-edge group. The code below does this, first for each corner cell and then for each of the four sides.

```
# 4. Reset boundary cells

# boundary cells take empire number and S of nearest non-edge cell

# top left corner
E[1,1] <- E[2,2]
S[1,1] <- S[2,2]

# top right corner
E[1,N_side] <- E[2,N_side-1]
S[1,N_side] <- S[2,N_side-1]

# bottom left corner
E[N_side,1] <- E[N_side-1,2]
S[N_side,1] <- S[N_side-1,2]

# bottom right corner
E[N_side,N_side] <- E[N_side-1,N_side-1]
S[N_side,N_side] <- S[N_side-1,N_side-1]
```

```
# top row
E[1,2:(N_side-1)] <- E[2,2:(N_side-1)]
S[1,2:(N_side-1)] <- S[2,2:(N_side-1)]

# bottom row
E[N_side,2:(N_side-1)] <- E[(N_side-1),2:(N_side-1)]
S[N_side,2:(N_side-1)] <- S[(N_side-1),2:(N_side-1)]

# left column
E[2:(N_side-1),1] <- E[2:(N_side-1),2]
S[2:(N_side-1),1] <- S[2:(N_side-1),2]

# right column
E[2:(N_side-1),N_side] <- E[2:(N_side-1),(N_side-1)]
S[2:(N_side-1),N_side] <- S[2:(N_side-1),(N_side-1)]

E
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    1    1    1    1    1    1    0     0
##  [2,]    0    0    1    1    1    1    1    1    0     0
##  [3,]    0    0    1    1    1    1    1    1    1     1
##  [4,]    0    0    1    1    1    1    1    1    1     1
##  [5,]    1    1    1    1    1    1    1    1    1     1
##  [6,]    0    0    1    1    1    1    1    1    1     1
##  [7,]    0    0    0    1    1    1    1    1    1     1
##  [8,]    0    0    0    1    1    1    1    1    1     1
##  [9,]    0    0    0    0    1    1    1    1    1     1
## [10,]    0    0    0    0    1    1    1    1    1     1
```

Finally we record the area of each empire in the *output* dataframe. Because we don't know beforehand which empires are present in generation t, we first store this in a vector called *empires*. Then we calculate each of these empires' areas and store it in a holding dataframe *output_new*, before adding *output_new* to the end of *output* using **rbind**. If there are no empires present, we put *NA* in the *empire* and *area* columns.

```
t <- 2

# 5. Update output

# list of empires (excluding zero/nonempires)
empires <- unique(as.vector(E[E>0]))
```

```
# if there are any empires left
if (length(empires) > 0) {

  # get area of each empire
  A <- rep(NA,length(empires))
  for (i in 1:length(empires)) {
    A[i] <- sum(E == empires[i])
  }

  output_new <- data.frame(generation = rep(t, length(empires)),
                           empire = empires,
                           area = A)

} else {

  # if no empires left, set to NA
  output_new <- data.frame(generation = t,
                           empire = NA,
                           area = NA)

}

# add output_new to end of output
output <- rbind(output, output_new)

output
```

```
##   generation empire area
## 1          1      1   16
## 2          2      1   72
```

As expected, *output* records the increased size of Empire 1 in the second generation.

Now to put everything together into a single function. **EmpireDynamics** combines all the code above, declaring the variables and their default values as arguments, putting all the within-generation events inside a t-loop, and ending by exporting the *output* dataframe along with the final empire matrix E. Note that we switch to $N_{side} = 21$ which was used by Turchin.

```
EmpireDynamics <- function(r_0 = 0.2,
                           delta = 0.1,
                           h = 2,
                           delta_P = 0.1,
                           S_crit = 0.003,
                           N_side = 21,
                           t_max = 200) {

  # matrix for empire id, initially all 0 (no empire)
  E <- matrix(0, nrow = N_side, ncol = N_side)

  # create a starting 4x4-cell empire 1
  row_E1 <- sample(2:(N_side-4), 1)
  col_E1 <- sample(2:(N_side-4), 1)
  E[row_E1:(row_E1+3),col_E1:(col_E1+3)] <- 1

  # matrix for asabiya, S
  S <- matrix(0.1, nrow = N_side, ncol = N_side)

  # record area of empire 1 in output dataframe
  output <- data.frame(generation = 1,
                       empire = 1,
                       area = sum(E == 1))

  # store highest empire number, to subsequently create new empires
  max_empire <- 1

  for (t in 2:t_max) {

    # 1. Update asabiya (S)

    # matrix of whether group has at least one dissimilar-empire neighbour
    dissimilar_neighbour <- matrix(0, nrow = N_side, ncol = N_side)

    # south: compare E with 1st row removed and duplicate 21st row (shifted up) vs regular E
    dissimilar_neighbour <- dissimilar_neighbour + (rbind(E[-1,], E[N_side,]) != E)

    # east: compare E with 1st col removed and duplicate 21st col (shifted left) vs regular E
    dissimilar_neighbour <- dissimilar_neighbour + (cbind(E[,-1], E[,N_side]) != E)

    # north: compare E with duplicate 1st row (shifted down) vs regular E
    dissimilar_neighbour <- dissimilar_neighbour + (rbind(E[1,], E[-N_side,]) != E)

    # west: compare E with duplicate 1st column (shifted right) vs regular E
    dissimilar_neighbour <- dissimilar_neighbour + (cbind(E[,1], E[,-N_side]) != E)

    # remove the multiple borders, reduce to TRUE/FALSE
    dissimilar_neighbour <- dissimilar_neighbour > 0

    # increase S of groups with dissimilar neighbours according to r_0
```

```
S[dissimilar_neighbour] <- S[dissimilar_neighbour] +
  r_0 * S[dissimilar_neighbour] * (1 - S[dissimilar_neighbour])

# decrease S of groups without dissimilar neighbours according to delta
S[!dissimilar_neighbour] <- S[!dissimilar_neighbour] -
  delta * S[!dissimilar_neighbour]


# 2. Intergroup conflict

# each non-edge cell is chosen in random order to attack their NSEW neighbouring c

# cells to enter conflicts, excluding edge cells (row/col 1 and 21) which do not a
attacker <- as.vector(matrix(1:(N_side^2),N_side,N_side)[-c(1,N_side),-c(1,N_side)]

# randomise the order of attackers
attacker <- sample(attacker)

for (i in attacker) {

  # random order of NSEW neighbours to attack
  defender <- sample(c(i+1,i-1,i+N_side,i-N_side))

  # cycle thru neighbours
  for (j in defender) {

    # if defender j is a valid opponent
    # and attacker and defender are of different empires, or both are non-empires
    if (j %in% attacker & (E[j] != E[i] | (E[i] == 0 & E[j] == 0))) {

      # non-empire cells have area A=1, their own S, and d=0
      if (E[i] == 0) {

        A_att <- 1
        S_att <- S[i]
        d_att <- 0

      }

      if (E[j] == 0) {

        A_def <- 1
        S_def <- S[j]
        d_def <- 0
```

```r
          }

          # empire cells have area A of their empire, mean S of their empire,
          #and d distance from centre of their empire
          if (E[i] > 0) {

            A_att <- sum(E == E[i])
            S_att <- mean(S[E == E[i]])

            col_centre <- mean(which(E == E[i], arr.ind = T)[,"col"])
            row_centre <- mean(which(E == E[i], arr.ind = T)[,"row"])
            Row <- (i-1) %% N_side + 1
            Col <- ceiling(i/N_side)
            d_att <- sqrt((Row - row_centre)^2 + (Col - col_centre)^2)

          }

          if (E[j] > 0) {

            A_def <- sum(E == E[j])
            S_def <- mean(S[E == E[j]])

            col_centre <- mean(which(E == E[j], arr.ind = T)[,"col"])
            row_centre <- mean(which(E == E[j], arr.ind = T)[,"row"])
            Row <- (j-1) %% N_side + 1
            Col <- ceiling(j/N_side)
            d_def <- sqrt((Row - row_centre)^2 + (Col - col_centre)^2)

          }

          # power of attacker and defender
          P_att <- A_att * S_att * exp(-d_att / h)
          P_def <- A_def * S_def * exp(-d_def / h)

          # if P_att - P_def is greater than delta_P, then attack is successful
          if (P_att - P_def > delta_P) {

            # if attacker is already part of an empire
            if (E[i] > 0) {

              # defender adopts empire of attacker
              E[j] <- E[i]

            # if attacker is a nonempire cell
            } else {
```

```
          # attacker and defender become a new empire
          E[i] <- max_empire + 1
          E[j] <- max_empire + 1

          # update max_empire
          max_empire <- max_empire + 1

        }

        # all defenders adopt the average of their prior S and the attacker's S
        S[j] <- (S[j]+S[i])/2

      }

    }

  }

}

# 3. Imperial collapse

# if mean S of an empire is less than S_crit, empire collapses

# list of empires (excluding zero/nonempires)
empires <- unique(as.vector(E[E>0]))

# if there are any empires left
if (length(empires) > 0) {

  # for each empire
  for (i in 1:length(empires)) {

    if (mean(S[E == empires[i]]) < S_crit) {

      # dissolve empire
      E[E == empires[i]] <- 0

    }

  }

}
```

```
  # 4. Reset boundary cells

  # boundary cells take empire number and S of nearest non-edge cell

  # top left corner
  E[1,1] <- E[2,2]
  S[1,1] <- S[2,2]

  # top right corner
  E[1,N_side] <- E[2,N_side-1]
  S[1,N_side] <- S[2,N_side-1]

  # bottom left corner
  E[N_side,1] <- E[N_side-1,2]
  S[N_side,1] <- S[N_side-1,2]

  # bottom right corner
  E[N_side,N_side] <- E[N_side-1,N_side-1]
  S[N_side,N_side] <- S[N_side-1,N_side-1]

  # top row
  E[1,2:(N_side-1)] <- E[2,2:(N_side-1)]
  S[1,2:(N_side-1)] <- S[2,2:(N_side-1)]

  # bottom row
  E[N_side,2:(N_side-1)] <- E[(N_side-1),2:(N_side-1)]
  S[N_side,2:(N_side-1)] <- S[(N_side-1),2:(N_side-1)]

  # left column
  E[2:(N_side-1),1] <- E[2:(N_side-1),2]
  S[2:(N_side-1),1] <- S[2:(N_side-1),2]

  # right column
  E[2:(N_side-1),N_side] <- E[2:(N_side-1),(N_side-1)]
  S[2:(N_side-1),N_side] <- S[2:(N_side-1),(N_side-1)]

  # 5. Update output

  # list of empires (excluding zero/nonempires)
  empires <- unique(as.vector(E[E>0]))

  # if there are any empires left
  if (length(empires) > 0) {

    # get area of each empire
```

```
      A <- rep(NA,length(empires))
      for (i in 1:length(empires)) {
        A[i] <- sum(E == empires[i])
      }

      output_new <- data.frame(generation = rep(t, length(empires)),
                               empire = empires,
                               area = A)

    } else {

      # if no empires left, set to NA
      output_new <- data.frame(generation = t,
                               empire = NA,
                               area = NA)

    }

    # add output_new to end of output
    output <- rbind(output, output_new)

  }

  # export output and the final generation empire matrix
  list(output = output, E = E)

}
```

Here is one run of the simulation using default parameter values, showing the first few rows of *output* and the final generation empires.

```
data_model12 <- EmpireDynamics()

head(data_model12$output)
```

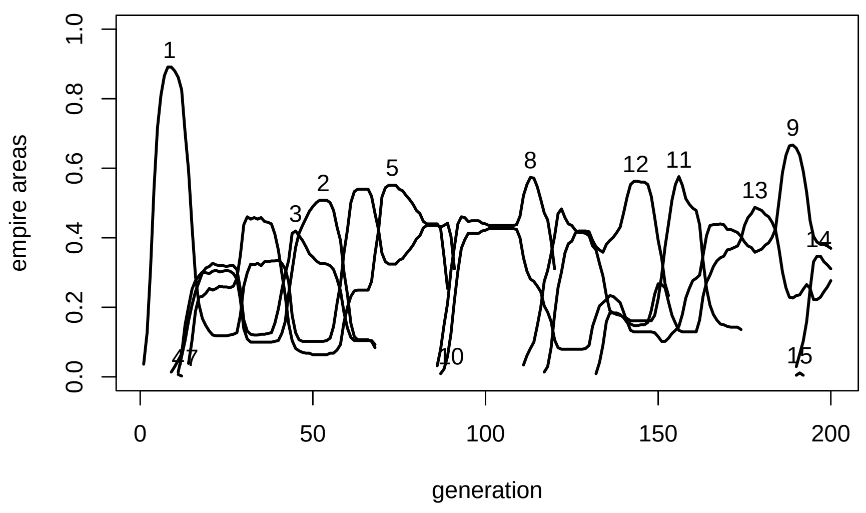
```
##   generation empire area
## 1          1      1   16
## 2          2      1   56
## 3          3      1  137
## 4          4      1  239
## 5          5      1  316
## 6          6      1  357
```

```
data_model12$E
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
##  [1,]   14   14   14   14   14   14   14   14   14    14    14    14    14
##  [2,]   14   14   14   14   14   14   14   14   14    14    14    14    14
##  [3,]   14   14   14   14   14   14   14   14   14    14    14    14    14
##  [4,]   14   14   14   14   14   14   14   14   14    14    14    14    14
##  [5,]   14   14   14   14   14   14   14   14   14    14    14    14     9
##  [6,]   14   14   14   14   14   14   14   14   14    14    14    14     9
##  [7,]   14   14   14   14   14   14   14   14   14    14    14     9     9
##  [8,]   14   14   14   14   14   14   14   14   14    14    14     9     9
##  [9,]   14   14   14   14   14   14   14   14   14    14    14     9     9
## [10,]   14   14   14   14   14   14   14   14   14    14     9     9     9
## [11,]   14   14   14   14   14   14   14   14   14    14     9     9     9
## [12,]   14   14   14   14   14   14   13   13   13    13     9     9     9
## [13,]   14   14   13   13   13   13   13   13   13    13    13     9     9
## [14,]   13   13   13   13   13   13   13   13   13    13    13    13     9
## [15,]   13   13   13   13   13   13   13   13   13    13    13    13     9
## [16,]   13   13   13   13   13   13   13   13   13    13    13    13    13
## [17,]   13   13   13   13   13   13   13   13   13    13    13    13    13
## [18,]   13   13   13   13   13   13   13   13   13    13    13    13    13
## [19,]   13   13   13   13   13   13   13   13   13    13    13    13    13
## [20,]   13   13   13   13   13   13   13   13   13    13    13    13    13
## [21,]   13   13   13   13   13   13   13   13   13    13    13    13    13
##       [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21]
##  [1,]     9     9     9     9     9     9     9     9
##  [2,]     9     9     9     9     9     9     9     9
##  [3,]     9     9     9     9     9     9     9     9
##  [4,]     9     9     9     9     9     9     9     9
##  [5,]     9     9     9     9     9     9     9     9
##  [6,]     9     9     9     9     9     9     9     9
##  [7,]     9     9     9     9     9     9     9     9
##  [8,]     9     9     9     9     9     9     9     9
##  [9,]     9     9     9     9     9     9     9     9
## [10,]     9     9     9     9     9     9     9     9
## [11,]     9     9     9     9     9     9     9     9
## [12,]     9     9     9     9     9     9     9     9
## [13,]     9     9     9     9     9     9     9     9
## [14,]     9     9     9     9     9     9     9     9
## [15,]     9     9     9     9     9     9     9     9
## [16,]     9     9     9     9     9     9     9     9
## [17,]    13     9     9     9     9     9     9     9
## [18,]    13     9     9     9     9     9     0     0
## [19,]    13     9     9     0     0     0     0     0
## [20,]    13    13     0     0     0     0     0     0
## [21,]    13    13     0     0     0     0     0     0
```

To get a better idea what's happening here, we can write a function to plot empire areas over time, recreating Turchin's (2003) Figure 4.4a. **AreaPlot** below takes the output dataframe from **EmpireDynamics** and, for each empire, plots a line for its area over all generations. We also add numbered labels for each empire at its maximum area using the **text** command. Note that we also need to know N_{side} to calculate the proportion of the grid each empire inhabits. Because this isn't contained in *output*, we get it from the final generation matrix E at the start.

```
AreaPlot <- function(data_model12) {

  output <- data_model12$output
  N_side <- nrow(data_model12$E)

  plot(x = 1:max(output$generation),
       type = "n",
       ylim = c(0,1),
       ylab = "empire areas",
       xlab = "generation")

  # remove NAs, otherwise things go wrong
  output <- na.omit(output)

  empires <- unique(output$empire)

  for (i in empires) {

    output_i <- output[output$empire == i,]

    # draw lines
    lines(x = output_i$generation,
          y = output_i$area/N_side^2,
          lwd = 2)

    # add numbered labels

    # generation at which empire i is at maximum area
    x_coord <- mean(output_i$generation[output_i$area == max(output_i$area)])

    # maximum area of empire i
    y_coord <- max(output_i$area/N_side^2)[1]

    text(x_coord, y_coord + 0.05, i)
  }

}

AreaPlot(data_model12)
```



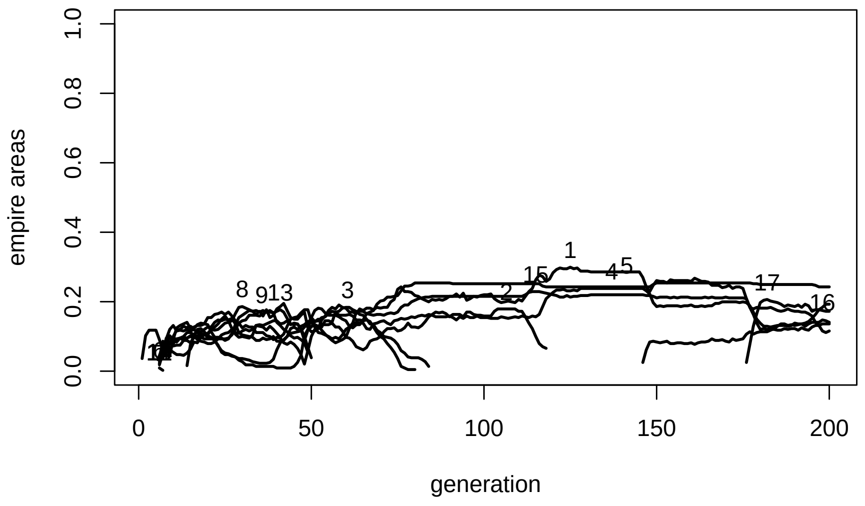
The plot above shows the oscillatory dynamics that we were hoping for, and which resemble real world historical dynamics. No empire dominates the entire time period. Empire 1 soon collapses making way for new empires. Some of these collapse very quickly, others persist for several generations. Even by the final generation here, empires continue to rise and fall, and would continue to do so if we extended the simulation further.

Now let's systematically remove different elements of the model to see which are necessary for generating these dynamics.

First we manipulate h, the constant which determines how quickly power declines with distance from the empire centre. As h increases, this distance penalty gets smaller.

Here we reduce h from the default of $h = 2$ to $h = 1$:

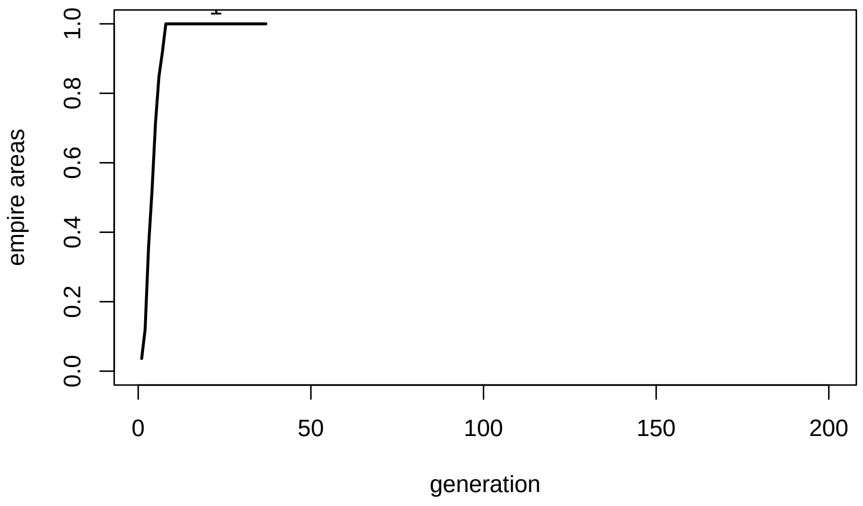
```
data_model12 <- EmpireDynamics(h = 1)
AreaPlot(data_model12)
```



Reducing h reduces and homogenises the area of each empire. The largest empire barely reaches a third of the total grid, and all empires have roughly the same area. Reducing h increases the distance penalty in the power equation above. Empires are therefore less able to expand to large sizes, resulting in several smaller empires of roughly equal power continually exchanging frontier groups. While still featuring oscillatory cycles, the lack of large empires and the homogenous empire sizes are not very realistic.

Now we increase h to 3:

```r
data_model12 <- EmpireDynamics(h = 3)
AreaPlot(data_model12)
```

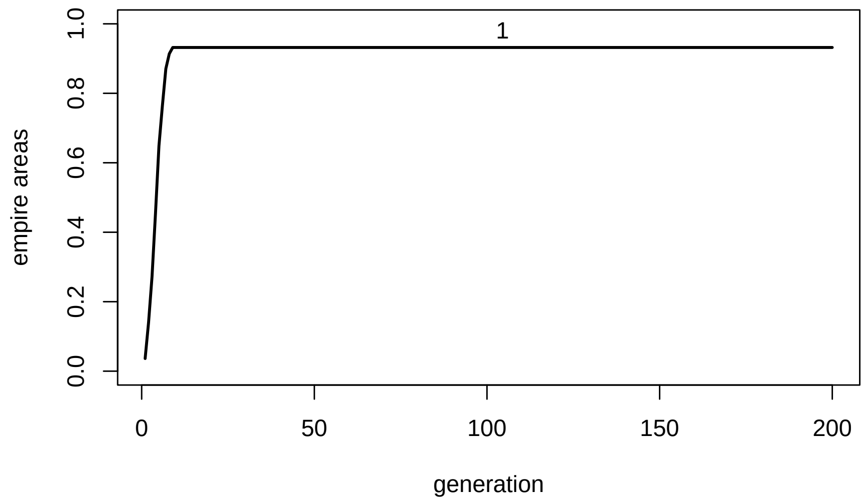


Here Empire 1 quickly fills the entire grid. The lack of any frontier regions means that asabiya then declines until it reaches S_{crit}, at which point it collapses. All groups become nonempire groups, and the lack of any frontiers means that no new empires emerge.

Varying h therefore indicates that the penalty to large empires of their frontier regions being far from the imperial centre is crucial in generating the oscillatory dynamics representative of real world empires. Too low and empires remain too small, too high and no empires persist at all.

Next we remove the changes in asabiya due to being a frontier or nonfrontier group. Setting $r_0 = 0$ and $\delta = 0$ does this.

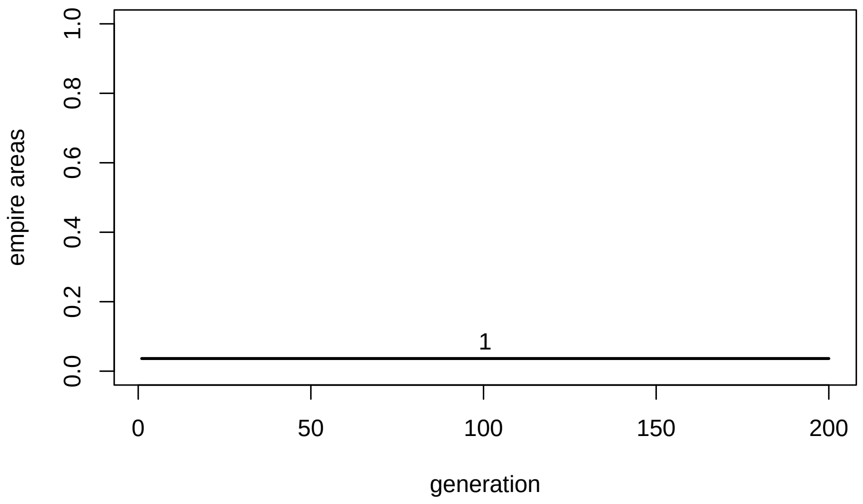
```r
data_model12 <- EmpireDynamics(delta = 0, r_0 = 0)
AreaPlot(data_model12)
```



Without frontier-related asabiya changes, the simulation reaches an equilibrium. Empire 1 increases to around 92% of the grid and stays that size. No new empires emerge. This occurs when Empire 1 is large enough such that the power of attackers and defenders (empire and nonempire groups) is equal. This equilibrium behaviour is not a realistic historical dynamic, and indicates that frontier-related changes in asabiya are crucial for generating oscillatory cycles.

Finally we can remove intergroup conflict by setting δ_P to a very large number. This means that the inequality $P_{att} - P_{def} > \delta_P$ is never satisfied, and groups never attack or conquer each other.

```r
data_model12 <- EmpireDynamics(delta_P = 10000)
AreaPlot(data_model12)
```



Without intergroup conflict, Empire 1 stays at size 16 for the entire simulation run. No expansion can occur, nor collapse, because groups cannot invade each other. Again, this equilibrium behaviour is historically unrealistic.

Summary

Model 12 simulated the rise and fall of empires throughout human history, recreating an agent-based model presented by Turchin (2003) which tests Turchin's cultural evolutionary theory of historical dynamics. This theory posits that empires emerge in frontier regions where different ethnic groups exist in close proximity. Such frontier regions are high in asabiya, a form of within-group cooperation. Asabiya provides an advantage in intergroup conflict. Empires emerge when frontier groups high in asabiya conquer neighbouring groups. As these multi-group empires grow larger, they gain power from their larger area and expand further. Large size, however, also reduces the empire's asabiya as the internal nonfrontier region grows larger. Large size also introduces distance penalties as frontier regions get more distant from the imperial centre of the empire. This weakens the empire, allowing new frontier groups high in asabiya to invade. These invaders become new empires, which expand, then weaken, and the cycle continues.

Model 12 simulated this theory with several assumptions about how the processes above work. We showed that when all processes are operating, then realistic oscillatory dynamics emerge with empires rising and falling continually over the simulation run. When one or more of these processes is removed - specifically, when we remove distance penalties of large empires, changes in asabiya due to being on a frontier, or intergroup conflict - then the dynamics disappear. Instead we see a lack of empires, or a single empire reaching equilibrium size. This lends support to Turchin's theory linking empire size, asabiya, frontier regions, and power in intergroup conflict.

As Turchin (2003) himself points out (p.71), however, we should be wary of accepting that a theory is correct just because it generates realistic dynamics. Those dynamics may be consistent with many other theories comprising different processes. However, Turchin goes on to test some of the predictions of the model. He shows, for example, that European empires from the years 0 - 1900 almost always emerge in frontier regions, consistent with the model. This lends empirical support to Turchin's theory, and shows how models can guide empirical research.

There are several 'black-boxed' assumptions in the model. For example, why does asabiya decline inside empires? Elsewhere, Turchin (2016) attributes this decline in cooperation to elite overproduction. When there are too many members of ruling elite classes (e.g. politicians, lawyers or scholars), these elites start competing amongst themselves for positions of power rather than working together for the society. This is not modelled here, but a more explicit multi-level selection model might capture this decline in group-level cooperation as a result of greater individual-level competition.

In subsequent work, Turchin et al. (2013) extended the spatially explicit model recreated here from an abstract N_{side} x N_{side} grid to a realistic map of Eurasia, and added mechanisms such as the diffusion of military technology that aids in intergroup conflict. This more geographically explicit model recreated specific historical patterns of empire dynamics in the region. Like Model 12, cultural group selection in the form of intergroup conflict, powered by within-group cooperation / asabiya, is crucial for these historical dynamics.

Unlike traditional historians, Turchin (2003) presents both analytic and agent-based models of his verbal theory of historical dynamics. While historians are typically sceptical of formal models, Turchin (2008) has argued for their value in allowing us to precisely state verbal historical explanations, test their internal logic and population-level dynamics in a way that the human brain is ill-equipped for, and provide specific quantitative predictions to then test with historical data. He calls this quantitative science of history 'cliodynamics', which can be seen as a branch of cultural evolution. In much of this work, Turchin has drawn on methods and concepts from population ecology (his former discipline). Formal models have been enormously useful in ecology, where population dynamics are often similarly complex yet can be usefully described by simple models. For example, the oscillatory cycles observed in Model 12 are similar to oscillatory predator-prey cycles observed for species such as lynxes and hares, as described by the famous Lotka-Volterra equations.

In terms of programming techniques, we drew here on the spatially-explicit methods of Model 10 (Polarisation). We use multiple matrices to store the empire and asabiya of each group in each cell of the square grid. Unlike previous models, Model 12 features quite a few equations. We have equations describing the increase and decline of asabiya, and equations for calculating the power of each group. While equations are more commonly seen in analytic models than agent-based simulations, we should not be afraid to use them in the latter. The more precise the relations are between different variables, the more straightforward will be our predictions. We can also choose specific functions (e.g. exponential, logistic) which have known properties (e.g. limitless growth vs reaching equilibrium) and may be appropriate to specific situations.

Exercises

1. Extend the number of timesteps to $t_{max} = 10000$, or some other large number. Do the same oscillatory dynamics persist or does the simulation ever reach equilibrium?

2. Vary the grid size via N_{side}, making it both smaller and larger than the default $N_{side} = 21$. What effect does this have on the dynamics of the model? What implications might we draw for how geography (e.g. small islands vs large continents) affects empire formation?

3. Modify the initial conditions of the model. What happens if Empire 1 is initially larger or smaller than 16 groups in size? What if there is more than one empire existing at the start?

4. Modify **EmpireDynamics** to generate a snapshot of the grid showing the empire id of each cell, similar to the **PolarizationPlot** and **PolarizationMultiplot** functions in Model 10 (Polarisation). Use numbers, different symbols or different colors to indicate different empires on the grid. Use these visualisations to test what Turchin (2003) calls the 'reflux effect'. This is where new empires that emerge at a border with an existing empire tend to expand initially backwards, into the nonempire region, rather than continuing to expand into the existing empire. This is because the existing empire is still quite strong due to its large size. Recreate Turchin's Fig 4.5 to illustrate the reflux effect.

5. Frontiers in Model 12 are one group wide and use a von Neumann neighbourhood. That is, a group's asabiya only increases if at least one of its four NSEW neighbours has a dissimilar empire id. Change this to (i) a

Moore neighbourhood, so that asabiya increases if at least one of the eight surrounding groups (including diagonals) have a dissimilar empire id, and (ii) make frontiers $n_f > 1$ groups wide, such that asabiya increases if at least one group in a n_f-group radius has a dissimilar empire id (currently $n_f = 1$). How do these extended frontiers affect the dynamics? Do these effects make sense given the assumptions of the model?

References

Turchin, P. (2003). Historical dynamics. Princeton University Press.

Turchin, P. (2008). Arise'cliodynamics'. Nature, 454(7200), 34-35.

Turchin, P. (2016). Ultrasociety: How 10,000 years of war made humans the greatest cooperators on earth. Chaplin, CT: Beresta Books.

Turchin, P., Currie, T. E., Turner, E. A., & Gavrilets, S. (2013). War, space, and the evolution of Old World complex societies. Proceedings of the National Academy of Sciences, 110(41), 16384-16389.

Model 13: Social contagion

Introduction

Model 13, Model 14 and Model 15 concern the linked topics of social contagion, social networks and opinion formation. These models and topics do not originate in the field of cultural evolution. They come instead from sociology and epidemiology. However, we will see how some of these concepts and models have direct parallels in the cultural evolution concepts and models we have already covered in this series. We will also see how they extend cultural evolution models in useful ways.

Social contagion models draw a parallel between the spread of diseases from person to person and the spread of ideas, beliefs, products, technologies and other cultural traits from person to person. Just as you can catch a disease like influenza or covid from another person, so too can you be 'infected' with their ideas, beliefs or habits through exposure to them in the form of observation or conversation.

While this analogy between diseases and ideas has merit, we should be wary of its limitations. For example, diseases can be caught with a single exposure to an infected individual, whereas many ideas, skills or beliefs require repeated exposure, such as when persuasion is needed to convince someone of an unusual belief or idea, or when a skill requires a lengthy apprenticeship to master. Nevertheless, contagion models from epidemiology provide useful insights into how cultural traits spread through society.

Model 13a: The SI model

Contagion models originating in epidemiology are called 'compartmental' models (Anderson & May 1992). They place individuals in one of a set of compartments. The simplest model compartmentalises individuals as either Susceptible (S) to acquiring a disease (or cultural trait), or already Infected (I) with the disease (or cultural trait). Consequently, these are called SI models.

Model 13a simulates the simplest possible SI model. We assume N agents, each of whom can be Susceptible or Infected. In each generation from 1 to t_{max}, every agent interacts with a single randomly chosen other agent in the population. If that other agent (the 'demonstrator') is Infected, then the focal agent becomes Infected with probability β. This parameter β represents the 'transmissability' of the cultural trait, with direct analogy to the transmissability of a disease. If the focal agent is already Infected, then nothing changes. The population is fixed, i.e. agents do not recover, die or migrate, and unstructured, i.e. any agent can potentially infect any other agent in the population in any timestep.

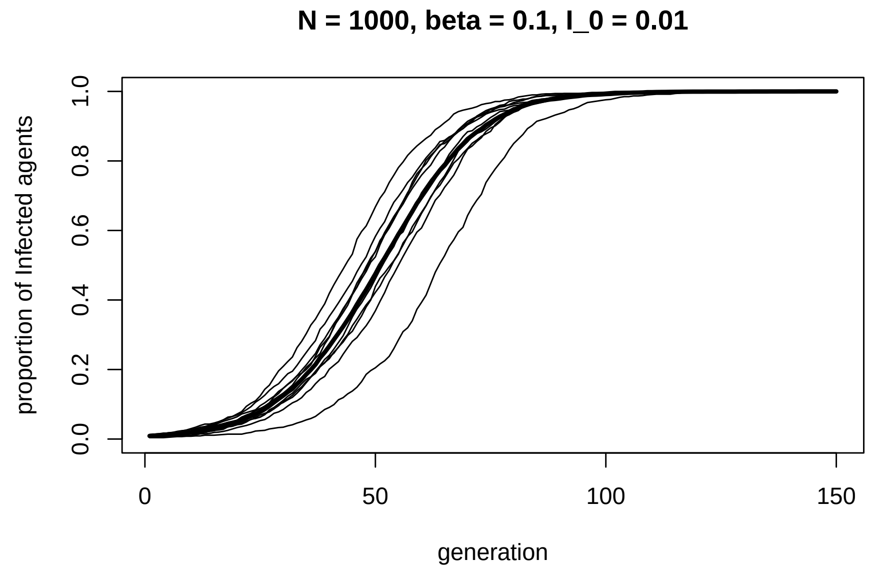
The function below implements this SI model. We specify an initial frequency of agents who are Infected with the parameter I_0, as in the absence of mutation the trait would otherwise never appear. We have r_{max} independent runs, and plot the proportion of Infected agents over time.

```
SImodel <- function(N, beta, I_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("I","S"), N, replace = TRUE,
                                       prob = c(I_0,1-I_0)))

    # add first generation's frequency of I to first row of column run
    output[1,r] <- sum(agent$trait == "I") / N

    for (t in 2:t_max) {

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # for each agent, pick a random agent from the previous generation
      # as demonstrator and store their trait
      demonstrator_trait <- sample(previous_agent$trait, N, replace = TRUE)

      # get N random numbers each between 0 and 1
      copy <- runif(N)

      # if agent is S, demonstrator is I and with probability beta, acquire I
      agent$trait[previous_agent$trait == "S" &
```

```
                  demonstrator_trait == "I" &
                  copy < beta] <- "I"

      # get frequency of I and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "I") / N

    }

  }

  # first plot a thick line for the mean proportion of I
  plot(rowMeans(output),
       type = 'l',
       ylab = "proportion of Infected agents",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N,
                    ", beta = ", beta,
                    ", I_0 = ", I_0,
                    sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function

}
```

Here is one run of the SI model:

```
data_model13a <- SImodel(N = 1000,
                         beta = 0.1,
                         I_0 = 0.01,
                         t_max = 150,
                         r_max = 10)
```



The SI process generates an S-shaped curve. Infections start off slowly, then increase rapidly, then level off as the entire population becomes Infected.

If this function and result looks familiar, it is because it is almost identical to the directly biased cultural transmission implemented in Model 3. Where Model 3 had traits A and B, here we have I and S. The strength of biased transmission s in Model 3 is equivalent to the transmissability parameter β here. It's the same process. This nicely illustrates one useful function of models: without modelling the two processes, we might not have realised that the two fields were talking about the same process, just with different notation and terminology.

Model 13b: The SIR model

The most common contagion model in epidemiology is the SIR model, which adds a third compartment of Recovered agents. As before, Susceptible agents become Infected with probability β, but now Infected agents Recover with probability γ. The latter is an asocial process: unlike infection, recovery does not require contact with any other individual to occur. Recovered individuals cannot become Infected again, assuming to have gained immunity.

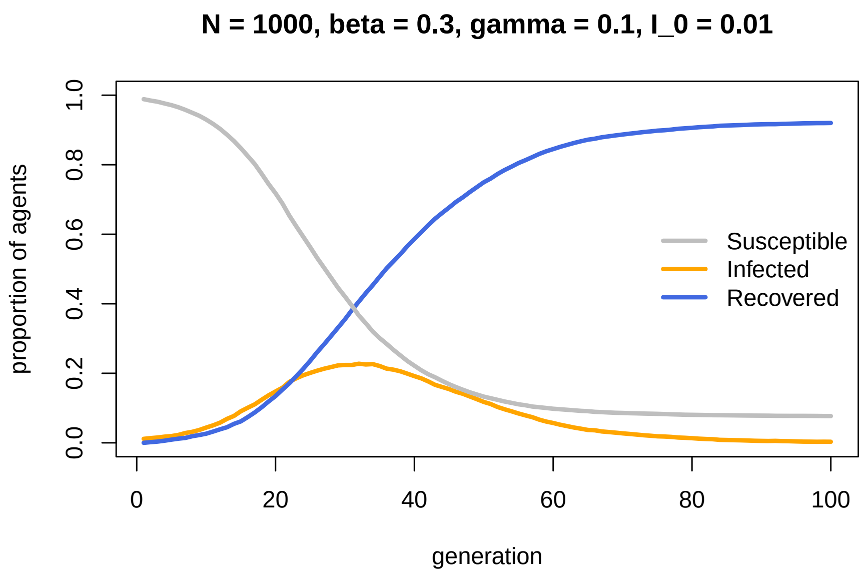
This is where we should be careful with the disease-idea analogy. While most diseases can be recovered from and be developed immunity to, the same cannot be said for most cultural traits. The best analogy might be with strong religious beliefs (and at the extreme, religious cults), which people join, sometimes leave, and never return to.

The following function adds some code to **SImodel** specifying that I individuals become R with probability γ. We now track the frequency of both I and R individuals, and plot all three types (the frequency of S is one minus the frequencies of the other two types, so no need to track that too).

```r
SIRmodel <- function(N, beta, gamma, I_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataframe
  output_I <- as.data.frame(matrix(NA, t_max, r_max))
  output_R <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output_I) <- paste("run", 1:r_max, sep="")
  names(output_R) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("I","S"), N, replace = TRUE,
                                       prob = c(I_0,1-I_0)))

    # add first generation's frequency of I/R to first row of column r
    output_I[1,r] <- sum(agent$trait == "I") / N
    output_R[1,r] <- sum(agent$trait == "R") / N

    for (t in 2:t_max) {

      # 1. biased transmission S to I

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # for each agent, pick a random agent from the previous generation
      # as demonstrator and store their trait
      demonstrator_trait <- sample(previous_agent$trait, N, replace = TRUE)

      # get N random numbers each between 0 and 1
      copy <- runif(N)

      # # if agent is S, demonstrator is I and with probability beta, acquire I
      agent$trait[previous_agent$trait == "S" &
                  demonstrator_trait == "I" &
                  copy < beta] <- "I"


      # 2. biased mutation I to R
      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get N random numbers each between 0 and 1
      mutate <- runif(N)

      # if agent was I, with prob gamma, flip to R
      agent$trait[previous_agent$trait == "I" & mutate < gamma] <- "R"


      # get frequency of I/R and put it into output slot for this generation t and run
      output_I[t,r] <- sum(agent$trait == "I") / N
      output_R[t,r] <- sum(agent$trait == "R") / N

    }

  }

  # first plot a thick line for the mean proportion of I
  plot(rowMeans(output_I),
       type = 'l',
       ylab = "proportion of agents",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       col = "orange",
       main = paste("N = ", N,
                    ", beta = ", beta,
                    ", gamma = ", gamma,
                    ", I_0 = ", I_0,
                    sep = ""))

  # add lines for R and S
  lines(rowMeans(output_R),
        type = 'l',
        lwd = 3,
        col = "royalblue")

  lines(1 - rowMeans(output_R) - rowMeans(output_I),
        type = 'l',
        lwd = 3,
        col = "grey")

  # add legend
  legend("right",
         legend = c("Susceptible",
                    "Infected",
                    "Recovered"),
         lty = 1,
         lwd = 3,
         col = c("grey", "orange", "royalblue"),
         bty = "n")

  list(output_I = output_I, output_R = output_R)  # export data from function

}
```

Here is one representative run of the SIR model:

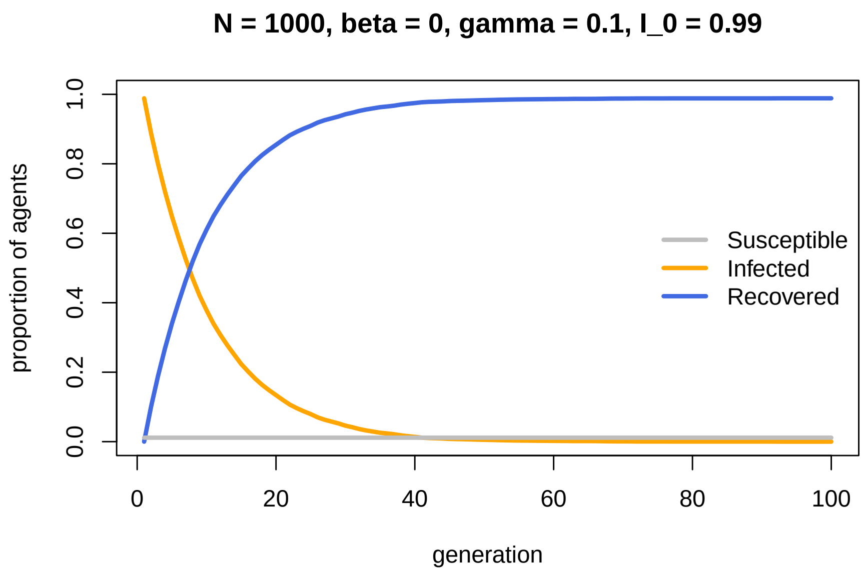
```
data_model13b <- SIRmodel(N = 1000,
                          beta = 0.3,
                          gamma = 0.1,
                          I_0 = 0.01,
                          t_max = 100,
                          r_max = 10)
```



Susceptible agents gradually become Infected, then Infected agents become Recovered. The end state is where every agent is either Susceptible or Recovered, with no Infecteds left to infect any remaining Susceptibles.

As hinted in the **SIRmodel** comments, the process by which Infected agents become Recovered is identical to biased mutation as seen in Model 2b. Both are individual-level processes occurring independently of other agents. The parameter γ in Model 13b is equivalent to μ_b in Model 2b. To confirm this, we can set $I_0 = 0.99$ and see how γ alone changes the frequency of I:

```
data_model13b <- SIRmodel(N = 1000,
                          beta = 0,
                          gamma = 0.1,
                          I_0 = 0.99,
                          t_max = 100,
                          r_max = 10)
```



Just as in Model 2b, γ generates an r-shaped diffusion curve, in contrast to the S-shaped curve seen in **SImodel** above and biased transmission in Model 3.

Model 13c: The SIS model

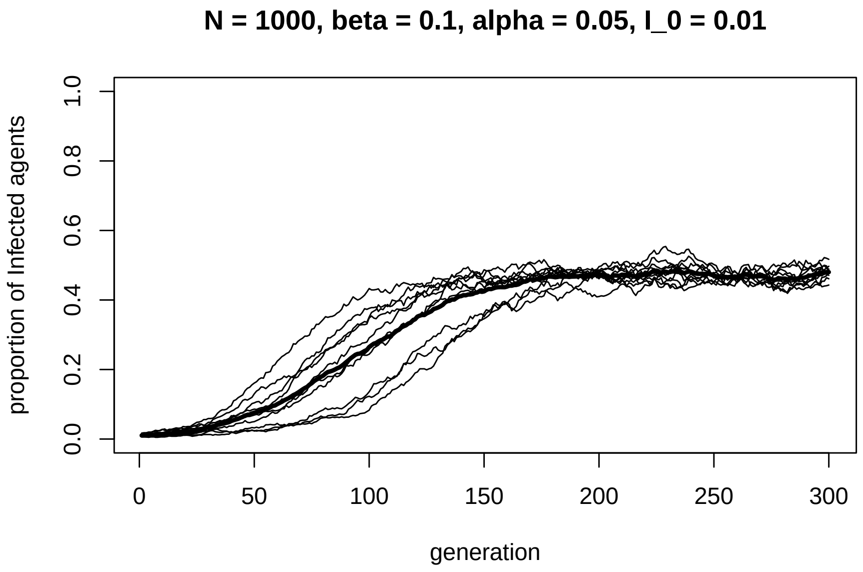
The SIS model involves Susceptible individuals becoming Infected, as in the SI model, but now Infected individuals can revert to being Susceptible again. In a disease context, this captures a virus like the common cold that one can catch repeatedly. In a social context, it captures cultural traits like jogging or vegetarianism that one might adopt then abandon then adopt again several times during one's life.

The following function **SISmodel** is a combination of **SImodel** and **SIRmodel**, with biased transmission favouring a switch from S to I according to β, and biased mutation favouring a switch from I to S according to parameter α.

```
SISmodel <- function(N, beta, alpha, I_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataframe
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("I","S"), N, replace = TRUE,
                                       prob = c(I_0,1-I_0)))

    # add first generation's frequency of I to first row of column run
    output[1,r] <- sum(agent$trait == "I") / N

    for (t in 2:t_max) {

      # 1. biased transmission S to I

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # for each agent, pick a random agent from the previous generation
      # as demonstrator and store their trait
      demonstrator_trait <- sample(previous_agent$trait, N, replace = TRUE)

      # get N random numbers each between 0 and 1
      copy <- runif(N)

      # if agent is S, demonstrator is I and with probability beta, acquire I
      agent$trait[previous_agent$trait == "S" &
                  demonstrator_trait == "I" &
                  copy < beta] <- "I"

      # 2. biased mutation I to S

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get N random numbers each between 0 and 1
      mutate <- runif(N)

      # if agent was I, with prob gamma, flip to R
      agent$trait[previous_agent$trait == "I" & mutate < alpha] <- "S"


      # get frequency of I and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "I") / N

    }

  }

  # first plot a thick line for the mean proportion of I
  plot(rowMeans(output),
       type = 'l',
       ylab = "proportion of Infected agents",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N,
                    ", beta = ", beta,
                    ", alpha = ", alpha,
                    ", I_0 = ", I_0,
                    sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  output  # export data from function

}
```

Here is one run of the SIS model with $\alpha = 0.05$, half that of β:

```
data_model13c <- SISmodel(N = 1000,
                          beta = 0.1,
                          alpha = 0.05,
                          I_0 = 0.01,
                          t_max = 300,
                          r_max = 10)
```



Unlike the previous models, the SIS model produces a stable equilibrium mix of S and I. This occurs when the number of Is being converted into Ss equals the number of Ss being converted into Is. Again, we see how the SIS model can be expressed in terms of cultural evolution concepts from previous models: biased transmission governs the spread of I, whereas biased mutation describes the switch back to S.

Model 13d: The SISa model

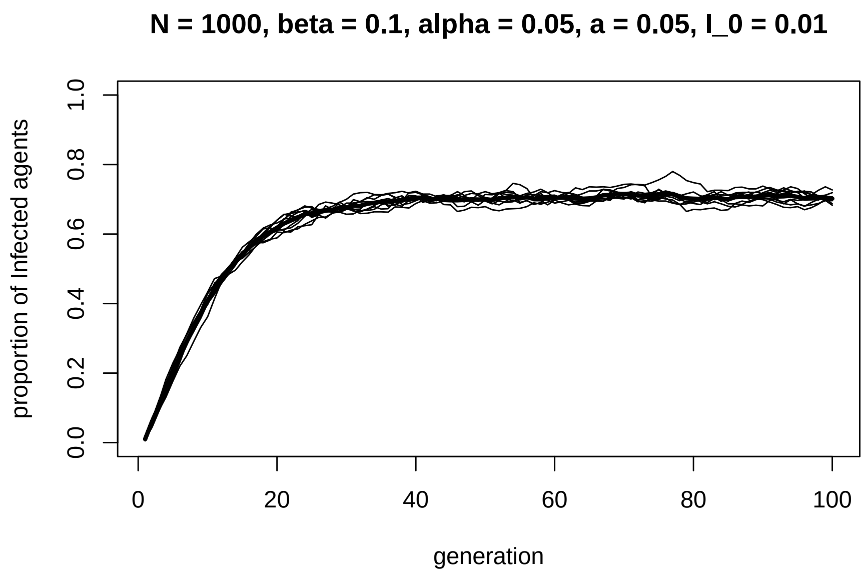
Finally, Hill et al. (2010) have proposed the SISa model, which they argue is more representative of social contagion compared to disease contagion. Hill et al. assumed that infection can occur not only via social transmission from others, but also *a*socially (or '*a*utomatically' as per Hill et al., but asocially makes more sense). People can't invent diseases, but they can invent new ideas, beliefs, products or technologies, or generally decide to adopt a behaviour independently of anyone else. Hill et al. use the example of obesity, which can be acquired both asocially (deciding not to exercise) and socially (copying the exercise or eating habits of friends).

SISa_model below adds a probability a that agents mutate from S to I. We also follow Hill et al. (2010) and add n, the number of demonstrators per timestep that agents potentially learn from during biased transmission of S to I. This is done by repeating the code implementing biased transmission n times. With $n = 1$, this is the same as **SISmodel**.

```
SISa_model <- function(N, beta, alpha, a, n, I_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation
    agent <- data.frame(trait = sample(c("I","S"), N, replace = TRUE,
                                       prob = c(I_0,1-I_0)))

    # add first generation's frequency of I to first row of column run
    output[1,r] <- sum(agent$trait == "I") / N

    for (t in 2:t_max) {

      # 1. biased transmission S to I

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # for n demonstrators
      for (i in 1:n) {

        # for each agent, pick a random agent from the previous generation
        # as demonstrator and store their trait
        demonstrator_trait <- sample(previous_agent$trait, N, replace = TRUE)

        # get N random numbers each between 0 and 1
        copy <- runif(N)

        # if agent is S, demonstrator is I and with probability beta, acquire I
        agent$trait[previous_agent$trait == "S" &
                    demonstrator_trait == "I" &
                    copy < beta] <- "I"

      }

      # 2. biased mutation I to S

      # copy agent to previous_agent dataframe
      previous_agent <- agent
      # get N random numbers each between 0 and 1
      mutate <- runif(N)

      # if agent was I, with prob gamma, flip to R
      agent$trait[previous_agent$trait == "I" & mutate < alpha] <- "S"


      # 3. biased mutation S to I

      # copy agent to previous_agent dataframe
      previous_agent <- agent

      # get N random numbers each between 0 and 1
      mutate <- runif(N)

      # if agent was S, with prob a, flip to I
      agent$trait[previous_agent$trait == "S" & mutate < a] <- "I"


      # get frequency of I and put it into output slot for this generation t and run r
      output[t,r] <- sum(agent$trait == "I") / N

    }

  }

  # first plot a thick line for the mean proportion of I
  plot(rowMeans(output),
       type = 'l',
       ylab = "proportion of Infected agents",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N,
                    ", beta = ", beta,
                    ", alpha = ", alpha,
                    ", a = ", a,
                    ", I_0 = ", I_0,
                    sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')
  }

  output  # export data from function

}
```

And one run of the SISa model:

```
data_model13d <- SISa_model(N = 1000,
                            beta = 0.1,
                            alpha = 0.05,
                            a = 0.05,
                            n = 1,
                            I_0 = 0.01,
                            t_max = 100,
                            r_max = 10)
```



Compared to the SIS model, the SISa model generates a higher equilibrium frequency of I agents, and reaches this equilibrium faster, due to the addition of mutation from S to I. The diffusion is also more r-shaped than S-shaped, reflecting the greater role of mutation.

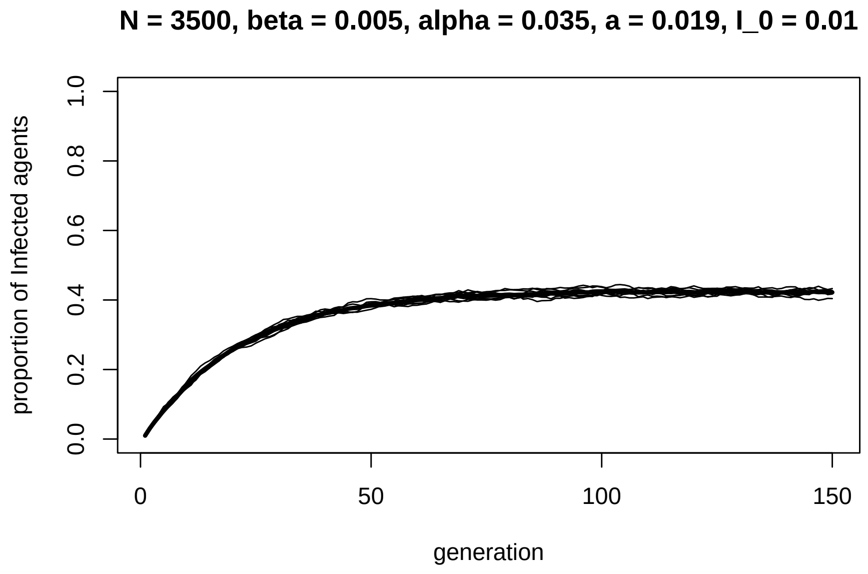
Summary

Social contagion models apply models of disease contagion from epidemiology to culturally transmitted behavioural traits. Naive individuals lacking the trait are Susceptible to becoming Infected with (i.e. socially learning) the trait. Different models provide additional assumptions: SIR models assume individuals can Recover, no longer bearing the trait nor able to learn it again; SIS models assume individuals can revert to being Susceptible after Infection and thus subsequently re-acquire the trait; SISa models assume individuals can asocially learn the trait as well as copy it from Infected individuals. Each of these fits a different kind of cultural trait.

A key take-away is that social contagion models are identical in their underlying mechanics to some of the cultural evolution models we have already covered in this series. The 'contagion' component (from S to I) is equivalent to directly biased transmission seen in Model 3. The asocial components (from I to R or S in SIR and SIS, and from S to I in SISa) are equivalent to biased mutation seen in Model 2b. The As and Bs in those models just need to be replaced with Is and Ss. One value of modelling is to reveal these parallels, so that researchers from different fields can identify common ground despite different terminologies, better learn from each others' efforts, and avoid reinventing the wheel.

The ultimate value of any model, including social contagion models, is whether it fits real-world data and allows us to predict future cultural dynamics. Hill et al. (2010) showed that the SISa model adequately describes changes in obesity in the Framingham Heart Study Network, a large, long-term health database. From their data, they estimated $\beta = 0.005$, $\alpha = 0.035$, $a = 0.019$ and $n = 3$ per year/timestep, for a dataset of $N = 3500$. We can plug these values into **SISa_model** to predict what will happen in the long-term, and calculate the expected equilibrium frequency of obesity:

```r
data_model13d <- SISa_model(N = 3500,
                            beta = 0.005,
                            alpha = 0.035,
                            a = 0.019,
                            n = 3,
                            I_0 = 0.01,
                            t_max = 150,
                            r_max = 10)
```



```
rowMeans(data_model13d[150,])
```

```
##       150
## 0.4220286
```

This recreates Hill et al.'s (2010) Figure 5A. The actual rate of obesity in the Framingham sample was 14% in the 1970s and 30% in 2000. This model predicts that obesity will eventually reach an equilibrium frequency of 42%. As Hill et al. remark, "While not great, this is a much more optimistic estimate than 100%". They also show that, despite β being smaller than γ and a, changing β has a larger effect on decreasing obesity per unit decrease in rate than the other parameters. This suggests that interventions aimed at reducing the social transmission of obesity will be more effective than interventions focused on asocial factors.

Other work has extended contagion models to further suit a cultural, rather than disease, context. Bettencourt et al. (2006) proposed and analysed an SEIZ model, where Susceptible individuals become Exposed (E) to an idea before eventually either becoming Infected (i.e. adopting it), or rejecting it (Z). They showed that this SEIZ model better captures the spread of the use of Feynman diagrams amongst post-war physicists, compared to simpler models lacking either E or Z classes. Walters & Kendal (2013), meanwhile, adapted an SIS model to make the social transmission from S to I conformist (see Model 5) rather than directly biased, with possible applications to the spread of binge drinking.

Exercises

1. Make a list of cultural traits that might be suitably described by the four different models implemented above (SI, SIR, SIS and SISa). How else might you modify the basic SI model to better suit the dynamics of other cultural traits not in any of your lists?

2. The SISa model assumes biased transmission from S to I, and biased mutation of both S and I. In theory, biased transmission could also drive the switch from I back to S. For example, one might copy the healthy eating or exercise habits of a friend to lose weight and no longer be obese. Add this biased transmission to **SISa_model**. Explore the dynamics generated by varying the four parameters (β, α, a and your new parameter).

3. Following Walters & Kendal (2013), and using code from Model 5, replace the directly biased social learning of **SISmodel** with conformist social learning. Explore how increasing the conformity parameter D affects the equilibrium value of I.

References

Anderson, R. M., & May, R. M. (1992). Infectious diseases of humans: dynamics and control. Oxford University Press.

Bettencourt, L. M., Cintrón-Arias, A., Kaiser, D. I., & Castillo-Chávez, C. (2006). The power of a good idea: Quantitative modeling of the spread of ideas from epidemiological models. Physica A, 364, 513-536.

Hill, A. L., Rand, D. G., Nowak, M. A., & Christakis, N. A. (2010). Infectious disease modeling of social contagion in networks. PLOS Computational Biology, 6(11), e1000968.

Walters, C. E., & Kendal, J. R. (2013). An SIS model for cultural trait transmission with conformity bias. Theoretical Population Biology, 90, 56-63.

Model 14: Social networks

Introduction

Most of the previous models assume unstructured populations in which every agent can potentially learn from any other agent. Exceptions are Model 7 (migration) and Model 11 (cultural group selection), in which agents can only learn from members of their own sub-population, and Model 10 (polarisation) in which agents are in a spatial grid and can only learn from their 4 or 8 neighbours.

Here we will extend this to explicitly model social networks of agents. Social networks specify the exact links between every agent. For example, a friendship network might capture who is friends with who within a group; a kin network might capture who is related to whom in a society; and a citation network might capture which scientists cite which other scientists in their papers. Social network analysis originates in sociology, but has become increasingly used in studies of cultural evolution in recent years.

Terminology

In the terminology of social network analysis, individuals or agents are called 'nodes' or 'vertices', and the connections between them are called 'links' or 'edges'. Networks can be undirected, which is when every edge from node A to node B is mirrored by an equivalent edge from node B to node A. For example, in a kin network, if person A is related to person B, then by definition person B will also be related to person A. In contrast, a directed network can have directional links. For example, in a citation network, scientist A might cite scientist B, but scientist B might never cite scientist A. Networks can be unweighted, in which each edge is of equal strength or intensity, or weighted, in which each edge is of different strength or intensity. For example, a citation network might weight edges by the number of citations, such that if scientist A cites scientist B 10 times and scientist B cites scientist A 100 times, the latter edge is weighted higher.

Adjacency matrices

First we will write some code to generate social networks, before looking at how information spreads on those networks.

Social networks are typically represented with an adjacency matrix. For a network of N individuals, this is an N x N matrix. The N rows represent nodes from which edges originate, while the N columns represent nodes at which edges terminate. For an unweighted network, each cell contains 0 if there is no edge connecting the row and column nodes, or 1 if there is an edge between those nodes. For example, a 1 in row 3, column 5 indicates an edge coming from node 3 to node 5. Assuming nodes cannot connect to themselves, the diagonal is always full of zeroes.

This code creates an empty N x N matrix called *network* full of zeroes.

```
N <- 10

# create empty adjacency matrix
network <- matrix(0, nrow = N, ncol = N)

network
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    0    0    0    0    0    0    0     0
##  [2,]    0    0    0    0    0    0    0    0    0     0
##  [3,]    0    0    0    0    0    0    0    0    0     0
##  [4,]    0    0    0    0    0    0    0    0    0     0
##  [5,]    0    0    0    0    0    0    0    0    0     0
##  [6,]    0    0    0    0    0    0    0    0    0     0
##  [7,]    0    0    0    0    0    0    0    0    0     0
##  [8,]    0    0    0    0    0    0    0    0    0     0
##  [9,]    0    0    0    0    0    0    0    0    0     0
## [10,]    0    0    0    0    0    0    0    0    0     0
```

Technically this isn't a network, as no-one is connected to anyone else. Let's add some connections:

```
edges <- data.frame(nodeA = c(1,4,7,8,9), nodeB = c(2,7,10,1,1))

for (i in 1:5) {

  network[edges$nodeA[i], edges$nodeB[i]] <- 1
  network[edges$nodeB[i], edges$nodeA[i]] <- 1

}
```

```
network
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    1    0    0    0    0    0    1    1     0
##  [2,]    1    0    0    0    0    0    0    0    0     0
##  [3,]    0    0    0    0    0    0    0    0    0     0
##  [4,]    0    0    0    0    0    0    1    0    0     0
##  [5,]    0    0    0    0    0    0    0    0    0     0
##  [6,]    0    0    0    0    0    0    0    0    0     0
##  [7,]    0    0    0    1    0    0    0    0    0     1
##  [8,]    1    0    0    0    0    0    0    0    0     0
##  [9,]    1    0    0    0    0    0    0    0    0     0
## [10,]    0    0    0    0    0    0    1    0    0     0
```

Here we have added five undirected edges between the pairs of nodes specified in *edges*. These are undirected because for every edge from node A to node B, we also create an edge between node B and node A. This makes the matrix, and every matrix containing only undirected edges, symmetrical. This can be verified using the **isSymmetric** command:

```
isSymmetric(network)
```

```
## [1] TRUE
```

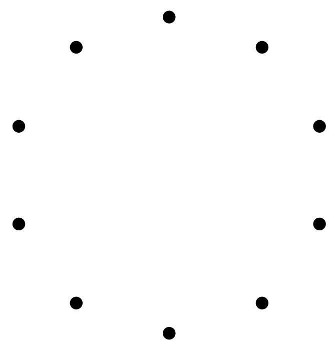
Weighted networks contain values other than 1, denoting the strength of the edge.

Small world networks

Watts & Strogatz (1998) introduced a simple way of generating realistic social networks. Imagine a ring of N nodes, each representing one agent. The end of the ring joins back to the first node. Think of this arrangement as the physical proximity or spatial arrangement of agents, like neighbouring houses in a big cul-de-sac.

It helps to draw this ring arrangement. Here is some code to draw N points in a ring. We first create an empty plot with no labels or axes and a square aspect ratio. Then the parametric form of the equation of a circle is used to draw N evenly spaced points around the origin (0,0) (note that angles in the equation are expressed in radians not degrees).

```r
# N agents around the origin in a big circle
plot(NULL,
     xlim = c(-5.5,5.5),
     ylim = c(-5.5,5.5),
     xlab = "",
     ylab = "",
     axes = FALSE,
     asp = 1)

for (i in 1:N) {

  points(5*sin((i-1)*2*pi/N), 5*cos((i-1)*2*pi/N),
         pch = 16,
         cex = 1.2)

}
```



We now add edges to represent social ties within this spatial arrangement. Watts & Strogatz (1998) assume first that each node is connected to k neighbouring nodes, $k/2$ to the left and $k/2$ to the right (k must therefore be an even number).

The following code generates an adjacency matrix for $k = 4$. We cycle through each row / node of the matrix, and for each one assign 1 to its $k/2$ neighbours to the right, and $k/2$ neighbours to the left. Because the ring joins round back to the beginning, if the neighbour column is greater than N then we subtract

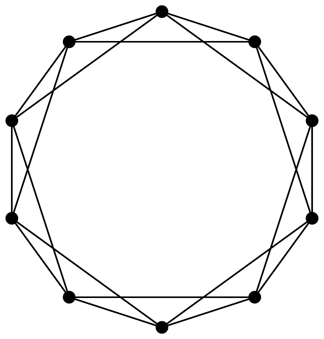
N, and if it's less than 1 then we add N, to wrap around to the other side.

```
k <- 4

# create empty adjacency matrix
network <- matrix(0, nrow = N, ncol = N, )

# create ring lattice network
for (Row in 1:N) {

  # k/2 neighbours to the right
  Col <- (Row+1):(Row+k/2)
  Col[which(Col > N)] <- Col[which(Col > N)] - N
  network[Row, Col] <- 1

  # k/2 neighbours to the left
  Col <- (Row-k/2):(Row-1)
  Col[which(Col < 1)] <- Col[which(Col < 1)] + N
  network[Row, Col] <- 1

}

network
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    1    1    0    0    0    0    0    1     1
##  [2,]    1    0    1    1    0    0    0    0    0     1
##  [3,]    1    1    0    1    1    0    0    0    0     0
##  [4,]    0    1    1    0    1    1    0    0    0     0
##  [5,]    0    0    1    1    0    1    1    0    0     0
##  [6,]    0    0    0    1    1    0    1    1    0     0
##  [7,]    0    0    0    0    1    1    0    1    1     0
##  [8,]    0    0    0    0    0    1    1    0    1     1
##  [9,]    1    0    0    0    0    0    1    1    0     1
## [10,]    1    1    0    0    0    0    0    1    1     0
```

We can update our plotting code to add these edges, again using the equation of a circle. We also wrap it in a self-contained function to re-use later.

```
DrawNetwork <- function(network) {

  # get N from network matrix
  N <- ncol(network)

  # N agents around the origin in a big circle
  plot(NULL,
       xlim = c(-5.5,5.5),
       ylim = c(-5.5,5.5),
       xlab = "",
       ylab = "",
       axes = FALSE,
       asp = 1)

  for (i in 1:N) {

    points(5*sin((i-1)*2*pi/N), 5*cos((i-1)*2*pi/N),
           pch = 16,
           cex = 1.2)

  }

  # lines representing edges
  for (i in 1:N) {

    for (j in which(network[i,] == 1)) {

      lines(x = c(5*sin((i-1)*2*pi/N), 5*sin((j-1)*2*pi/N)),
            y = c(5*cos((i-1)*2*pi/N), 5*cos((j-1)*2*pi/N)))

    }

  }

}

DrawNetwork(network)
```



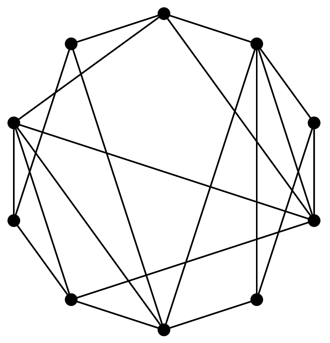
So far so straightforward: each node is connected to its four immediate neighbours. Real social networks, however, have additional connections. Some people might be friends with someone across the other side of the cul-de-sac, and never speak to their physical neighbours. Some scientists might collaborate with other scientists in different countries, or different academic disciplines.

Therefore with probability p the above *network* is 'rewired'. We cycle through each node from 1 to N. For each one, we take the first neighbour to its right, which given that $k >= 2$ it will be connected to with an edge. Then, with probability p, we pick a *new_neighbour* randomly from the set of all nodes excluding self and nodes to which connections already exist. The existing edge is removed, and a new edge is drawn to the *new_neighbour*. Remember, this is an undirected network, so every change made from node A to node B needs to be repeated for node B to node A. This keeps the matrix symmetrical. Finally, this whole process is repeated for the next connected neighbour, up to the last, so $k/2$ times in total.

```
p <- 0.2

# rewiring via p

for (j in 1:(k/2)) {

  for (i in 1:N) {

    if (runif(1) < p) {
      # pick jth clockwise neighbour
      neighbour <- i + j
      if (neighbour > N) neighbour <- neighbour - N

      # pick random new neighbour, excluding self and duplicate edges
      new_neighbour <- which(network[i,] == 0)
      new_neighbour <- new_neighbour[new_neighbour != i]
      new_neighbour <- sample(new_neighbour, 1)

      # remove edge to old neighbour
      network[i,neighbour] <- 0
      network[neighbour,i] <- 0

      # make edge to new neighbour
      network[i, new_neighbour] <- 1
      network[new_neighbour, i] <- 1

    }

  }

}

network
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    1    0    1    0    0    0    0    1     1
##  [2,]    1    0    1    1    1    1    0    0    0     0
##  [3,]    0    1    0    1    1    0    0    0    0     0
##  [4,]    1    1    1    0    0    0    1    0    1     0
##  [5,]    0    1    1    0    0    1    0    0    0     0
##  [6,]    0    1    0    0    1    0    1    0    1     1
##  [7,]    0    0    0    1    0    1    0    1    1     0
##  [8,]    0    0    0    0    0    0    1    0    1     1
##  [9,]    1    0    0    1    0    1    1    1    0     0
## [10,]    1    0    0    0    0    1    0    1    0     0
```

```
DrawNetwork(network)
```



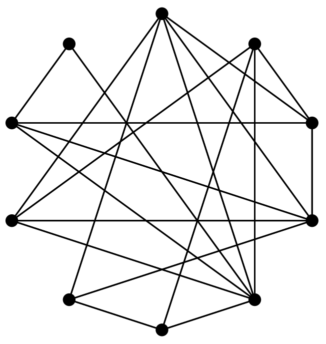
The resulting network is still mostly clustered, like the original network, but has a few long edges connecting otherwise distant nodes. Watts & Strogatz (1998) called this a 'small world network', after the phenomenon by which surprisingly few connections link any two people in a typical social network. The most famous example is the six degrees of Kevin Bacon, where any actor can be connected to Kevin Bacon via shared movie credits in six steps or less. Watts & Strogatz (1998) showed formally that their small world network algorithm adequately captures the characteristics of various real-world networks, including film actors.

We wrap the above code in a function to more easily re-run it with different parameter values, and re-use it later.

```
SmallWorld <- function(N, k, p, draw_plot = TRUE) {

  # 1. create empty adjacency matrix
  network <- matrix(0, nrow = N, ncol = N, )

  # 2. create ring lattice network
  for (Row in 1:N) {

    # k/2 neighbours to the right
    Col <- (Row+1):(Row+k/2)
    Col[which(Col > N)] <- Col[which(Col > N)] - N
    network[Row, Col] <- 1
    # k/2 neighbours to the left
    Col <- (Row-k/2):(Row-1)
    Col[which(Col < 1)] <- Col[which(Col < 1)] + N
    network[Row, Col] <- 1

  }

  # 3. rewiring via p

  for (j in 1:(k/2)) {

    for (i in 1:N) {

      if (runif(1) < p) {

        # pick jth clockwise neighbour
        neighbour <- i + j
        if (neighbour > N) neighbour <- neighbour - N

        # pick random new neighbour, excluding self and duplicate edges
        new_neighbour <- which(network[i,] == 0)
        new_neighbour <- new_neighbour[new_neighbour != i]
        new_neighbour <- sample(new_neighbour, 1)

        # remove edge to old neighbour
        network[i,neighbour] <- 0
        network[neighbour,i] <- 0

        # make edge to new neighbour
        network[i, new_neighbour] <- 1
        network[new_neighbour, i] <- 1

      }

    }

  }

  # 4. draw network if draw_network == TRUE

  if (draw_plot == TRUE) {

    DrawNetwork(network)

  }
  # output network from function
  network

}
```

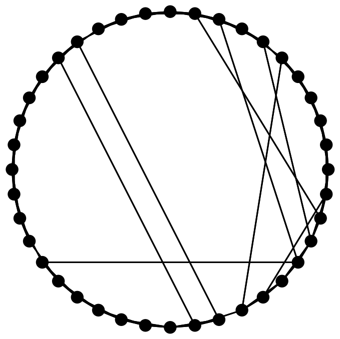
Increasing p to 1 gives a fully random network, with every original edge rewired:

```
data_model14 <- SmallWorld(N = 10,
                           k = 4,
                           p = 1)
```



Increasing N to 40 and setting $p = 0.1$ gives a typical small world network:

```
data_model14 <- SmallWorld(N = 40,
                           k = 4,
                           p = 0.1)
```



Most edges are still connecting adjacent nodes in the ring, but a small number of long ties traverse the ring connecting otherwise-distant nodes.

Network properties: Path length and clustering

The reason small world networks with a small but non-zero p like the one above are interesting is because they are clustered yet also have short path lengths. A path between two nodes is the set of edges linking those nodes. Path length is the number of edges in that path. The shortest path length is the minimum number of edges connecting two nodes. Two nodes directly connected by an edge have a shortest path length of 1; two nodes not directly connected but connected via a third node have shortest path length of 2; and so on.

Small world networks have much lower shortest path lengths than fully clustered $p = 0$ networks because of the long ties linking distant parts of the network. This is of importance to social learning and cultural evolution, because if we assume that information flows along edges, then small world networks will be much more efficient at spreading information across the whole network than fully clustered ($p = 0$) networks. We will explore this formally later, but for now let's quantitatively measure path length to confirm the above claims.

The following function **get_path_length** calculates, for a given *network*, the shortest path length between node A and node B. The code is a little opaque, but essentially it maintains a set v of nodes that are increasing degrees to A (degree 0 is node A itself, degree 1 are the nodes connected to A by one edge, degree 2 the nodes connected to A by two edges, etc.), and if any of the nodes in v are B, the loop **break**s and the current *path_length* is returned.

```
get_path_length <- function(network, A, B) {

  path_length <- NA
  N <- ncol(network)

  # start with vertex v = A
  v <- A

  for (i in 1:(N-1)) {

    # if any vertices v are connected to B, set path_length and break
    if (any(network[v,B] == 1)) {

      path_length <- i
      break

    }

    # otherwise, new v is the vertices connected to previous v
    v <- which(as.matrix(network[,v] == 1), arr.ind = T)[,1]

  }

  # return path_length
  path_length

}
```

Some examples for the previous network generated above:

```
get_path_length(data_model14, 1, 2)
```

```
## [1] 1
```

```
get_path_length(data_model14, 1, 12)
```

```
## [1] 3
```

```
get_path_length(data_model14, 1, 39)
```

```
## [1] 1
```

The function **get_mean_path_length** below calculates the average path length between every possible pair of nodes, and calculates this for *network*:

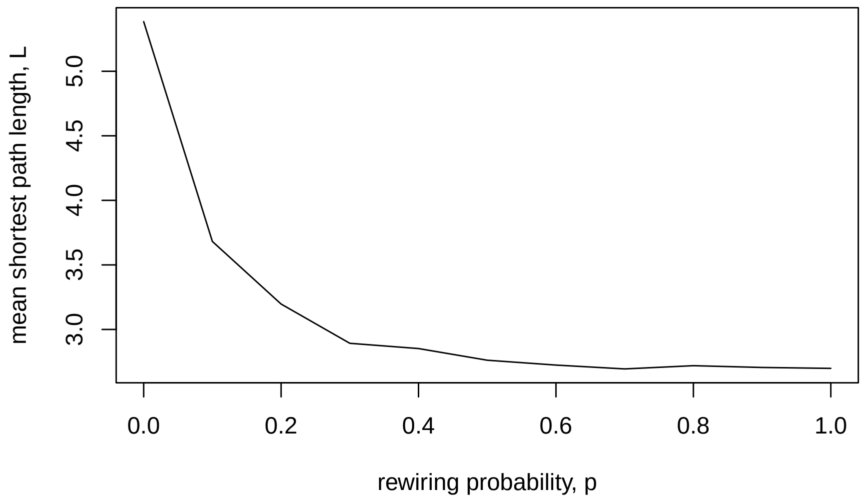
```
get_mean_path_length <- function(network) {

  mean_path_length <- 0
  N <- ncol(network)

  for (i in 1:(N-1)) {

    for (j in (i+1):N) {

      mean_path_length <- mean_path_length + get_path_length(network,i,j)

    }

  }

  mean_path_length / (N*(N-1)/2)

}

get_mean_path_length(network)
```

```
## [1] 1.644444
```

Let's use this function to plot mean path length L against p, averaging across r_{max} runs for each value of p:

```
p <- seq(0, 1, 0.1)
L <- rep(0, length(p))
r_max <- 10

for (r in 1:r_max) {

  for (i in 1:length(p)) {

    L[i] <- L[i] + get_mean_path_length(SmallWorld(40, 4, p[i], draw_plot = F))

  }

}

# average over all runs
L <- L / r_max
```

```r
# plot L vs p
plot(x = p,
     y = L,
     type = 'l',
     ylab = "mean shortest path length, L",
     xlab = "rewiring probability, p")
```



As expected, as networks increase in long ties, mean path length gets shorter, from 5.38 at $p = 0$ to around 2.7 for $p = 0.5$ and above. Note that the biggest drop occurs for small values of p. Only a small amount of rewiring ($p = 0.1$) is needed to cause mean path length to reduce considerably. In contrast, increasing p from 0.5 to 1 has little effect. Once there are a few long ties connecting different parts of the ring, further long ties are redundant.

Clustering refers to the cliqueishness of a particular neighbourhood within the network. The clustering coefficient for node i is defined as, for all neighbours directly connected to node i, the proportion of all possible edges linking those neighbours to each other that are actually present. For example, if node i has edges to $k_i = 3$ neighbours, there are a maximum of $k_i(k_i - 1)/2 = 3$ potential edges between those three neighbours (between neighbour A & B, A & C, and B & C). Say only two of these edges are actually present (e.g. between A & B and A & C), then the clustering coefficient is 2/3.

The function **get_clustering_coef** below calculates the clustering coefficient for a specific node in the adjacency matrix *network*. First we get the set of directly connected *neighbours*. Then we create a mini-adjacency-matrix containing just those neighbours. Then we count the number of edges between them, and divide by all possible edges. Note that we use the function **upper.tri()** to get the sum of all edges in only one half of the matrix, given that our network is undirected. The clustering coefficient for node 10 is shown as an example.

```
get_clustering_coef <- function(network, node) {

  # get direct neighbours of node
  neighbours <- which(network[node,] == 1)

  # number of neighbours
  k_i <- length(neighbours)

  # create mini-matrix representing neighbours
  neighbours <- network[neighbours, neighbours]

  # count edges, ignoring duplicates given that network is undirected
  # and divide by all possible edges
  sum(neighbours[upper.tri(neighbours)] == 1) / (k_i*(k_i-1)/2)

}

get_clustering_coef(network, 10)
```

```
## [1] 0
```

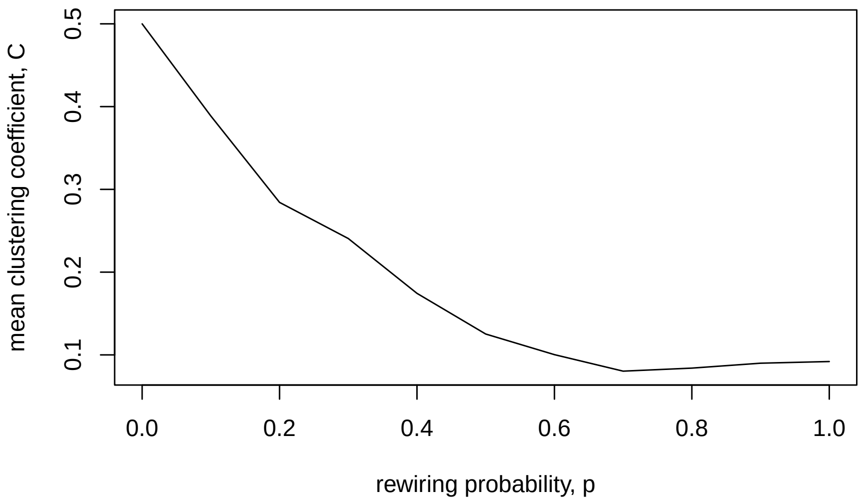
Now we can get the mean clustering coefficient across all N nodes using another function, and use it to calculate this for *network*:

```
get_mean_clustering_coef <- function(network) {

  mean_clustering_coef <- 0
  N <- ncol(network)

  for (i in 1:N) {

    mean_clustering_coef <- mean_clustering_coef + get_clustering_coef(network, i)

  }

  # return mean clustering coefficient
  mean_clustering_coef / N
}

get_mean_clustering_coef(network)
```

```
## [1] 0.39
```

And finally, as for path length, we can plot clustering coefficient C for a range of p:

```
p <- seq(0, 1, 0.1)
C <- rep(0, length(p))
r_max <- 10

for (r in 1:r_max) {

  for (i in 1:length(p)) {

    C[i] <- C[i] + get_mean_clustering_coef(SmallWorld(40, 4, p[i], draw_plot = F))

  }

}

# average over all runs
C <- C / r_max

# plot L vs p
plot(x = p,
     y = C,
     type = 'l',
     ylab = "mean clustering coefficient, C",
     xlab = "rewiring probability, p")
```



As for path length, the clustering coefficient drops as p increases and networks get more randomised. This makes sense, as randomisation will break up clusters of densely connected neighbours. Note however that the clustering coefficient drops less sharply than path length. This means that, for reasonably small values of p (e.g. $p = 0.1$ for the N and k shown above), i.e. in the 'small world' region in between fully clustered and fully random, the network retains most of its clustering but has much reduced path length.

Model 14

Now we have a way of generating social networks, we can simulate the spread of information in those networks. We adapt the **SImodel** from Model 13a in which N initially Susceptible (S) agents are seeded with a small number of Infected (I) agents. Whereas previously the Infection (i.e. the cultural trait) spread through an unstructured population, now it spreads along the edges in a network created using **SmallWorld**.

First we create a vector of N agents all initially with trait S. For reasons that will become apparent later, rather than randomly setting a proportion I_0 of agents to I as in **SImodel**, we make the first I_0 agents in *agent* I. Note that I_0 is now a positive integer, rather than a probability.

```
N <- 40
I_0 <- 2
```

```
# create first generation of agents
agent <- rep("S", N)

# add I_0 adjacent infected agents
agent[1:I_0] <- "I"

agent
```

```
##  [1] "I" "I" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S"
## [20] "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S"
## [39] "S" "S"
```

We also create a network for our N agents using **SmallWorld**:

```
k <- 4
p <- 0.1

# create network for agent
network <- SmallWorld(N, k, p, draw_plot = F)
```

Now we pick focal S agents in random order to potentially be infected by (i.e. socially learn from) I individuals. For each focal, we get that focal's neighbours to which they are connected via edges. Then, if at least n of those neighbours are I, the focal becomes I. For now, we set $n = 1$, so only one neighbour needs to be I in order to spread the trait.

```
n <- 1

# randomly ordered susceptible focals
focal <- sample(which(agent == "S"))

# cycle through focals
for (i in focal) {

  # get i's neighbours
  neighbours <- which(network[i,] == 1)

  # if at least one neighbour is I, adopt I
  if (sum(agent[neighbours] == "I") >= n) {

    agent[i] <- "I"

  }
}

agent
```

```
##  [1] "I" "I" "I" "I" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "I" "S"
## [20] "I" "S" "S" "S" "I" "I" "I" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S" "S"
## [39] "I" "I"
```

You should see that several agents have flipped to *I* as a result of contagion on the network.

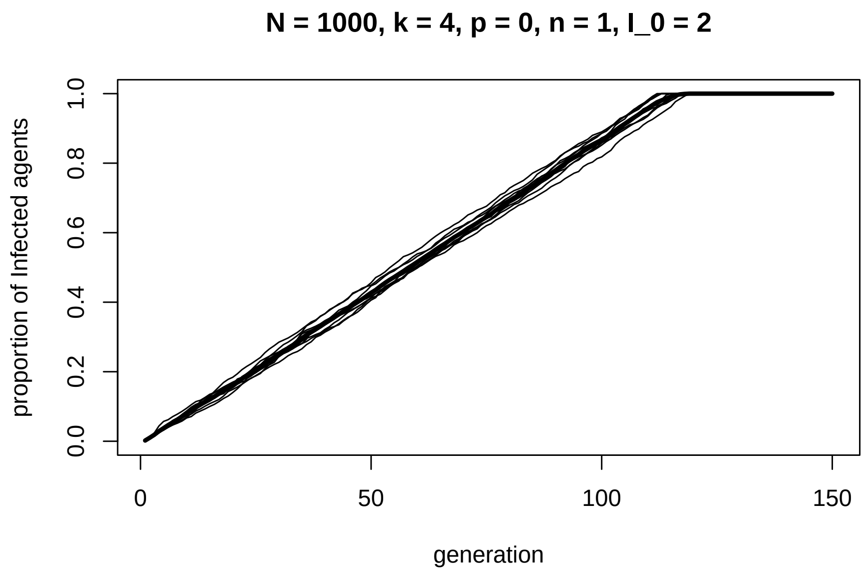
Let's wrap up the above code into a function **Contagion** which contains loops for timesteps and independent runs, records the frequency of *I* in an *output* dataframe, and plots this frequency.

```
Contagion <- function(N, k, p, n, I_0, t_max, r_max) {

  # create a matrix with t_max rows and r_max columns, fill with NAs, convert to dataf
  output <- as.data.frame(matrix(NA, t_max, r_max))

  # purely cosmetic: rename the columns with run1, run2 etc.
  names(output) <- paste("run", 1:r_max, sep="")

  for (r in 1:r_max) {

    # create first generation of agents
    agent <- rep("S", N)

    # add I_0 adjacent infected agents
    agent[1:I_0] <- "I"

    # add first generation's p to first row of column run
    output[1,r] <- sum(agent == "I") / N

    # create network for agent
    network <- SmallWorld(N, k, p, draw_plot = F)

    for (t in 2:t_max) {

      # randomly ordered susceptible focals
      focal <- sample(which(agent == "S"))

      # cycle through focals
      for (i in focal) {
        # get i's neighbours
        neighbours <- which(network[i,] == 1)

        # if at least one neighbour is I, adopt I
        if (sum(agent[neighbours] == "I") >= n) {

          agent[i] <- "I"

        }

      }

      output[t,r] <- sum(agent == "I") / N

    }

  }

  # first plot a thick line for the mean proportion of I
  plot(rowMeans(output),
       type = 'l',
       ylab = "proportion of Infected agents",
       xlab = "generation",
       ylim = c(0,1),
       lwd = 3,
       main = paste("N = ", N,
                    ", k = ", k,
                    ", p = ", p,
                    ", n = ", n,
                    ", I_0 = ", I_0,
                    sep = ""))

  for (r in 1:r_max) {

    # add lines for each run, up to r_max
    lines(output[,r], type = 'l')

  }

  # export output
  output

}
```

Because information spreads too fast to be interesting on small networks of

$N = 40$, we increase the population size to $N = 1000$. First, what happens in a completely clustered $p = 0$ network with no long ties?

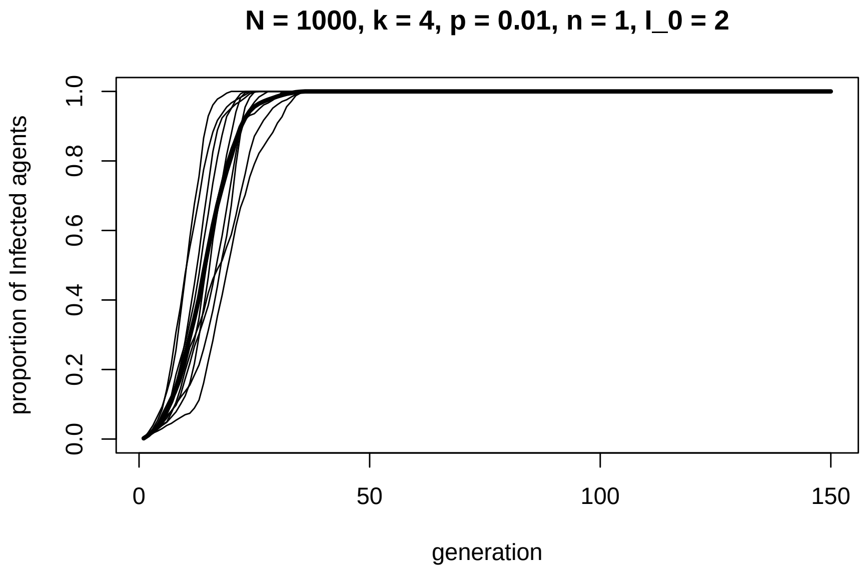
```
data_model14 <- Contagion(N = 1000,
                          k = 4,
                          p = 0,
                          n = 1,
                          I_0 = 2,
                          t_max = 150,
                          r_max = 10)
```



In a clustered network information spreads at a constant rate node-by-node around the ring network until it reaches every node at around timestep 120.

Now let's try a small world network, $p = 0.01$

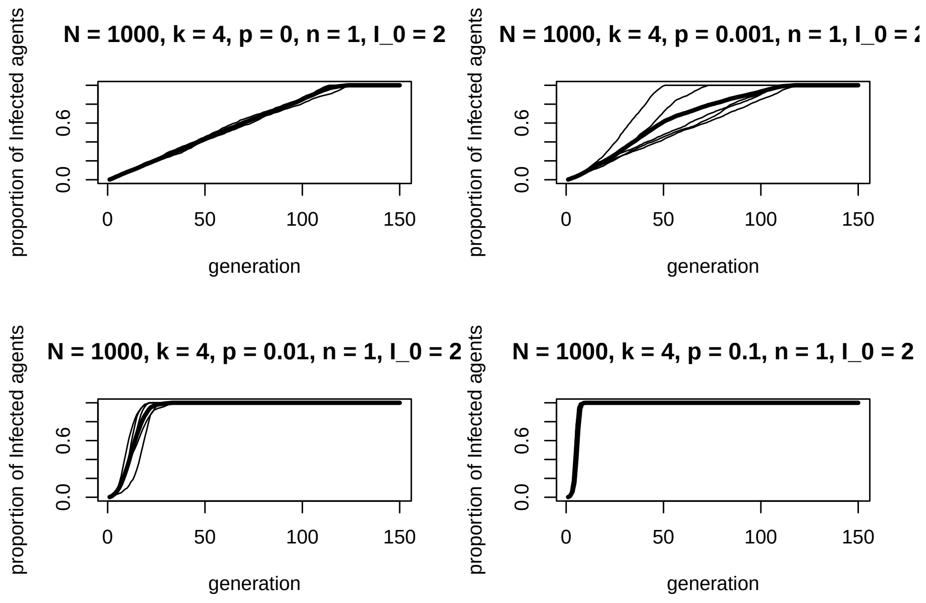
```
data_model14 <- Contagion(N = 1000,
                          k = 4,
                          p = 0.01,
                          n = 1,
                          I_0 = 2,
                          t_max = 150,
                          r_max = 10)
```



In a small world network with just a small number of long ties (approximately $Np = 10$), information spreads much, much faster. It flows along the shortcuts across the ring, not needing to travel node-by-node around the ring.

We can see this more clearly in the following four-panel plot, comparing $p = 0$, 0.001, 0.01 and 0.1:

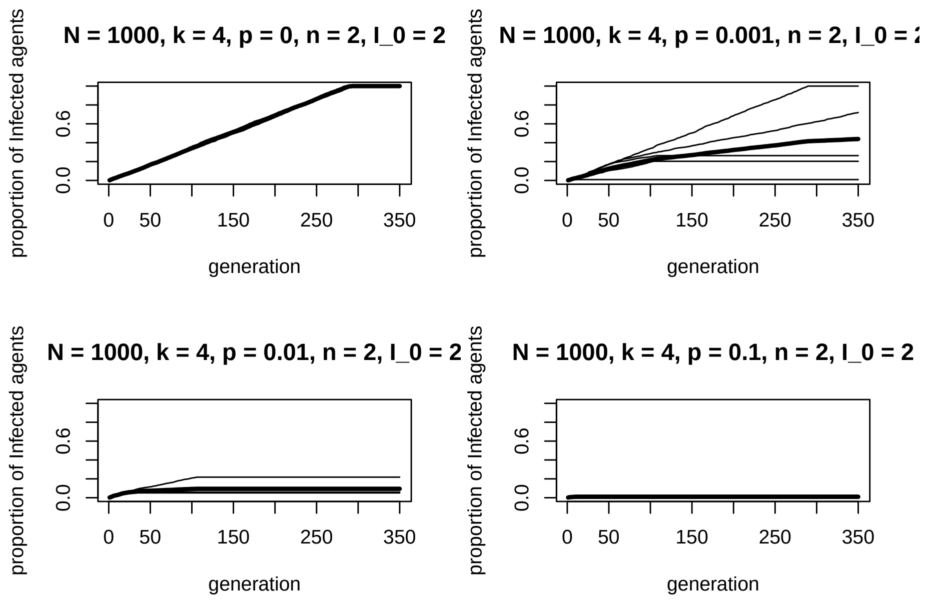
```
# 2 rows, 2 columns
par(mfrow=c(2,2))

# p = 0
panel1 <- Contagion(N = 1000,
                    k = 4,
                    p = 0,
                    n = 1,
                    I_0 = 2,
                    t_max = 150,
                    r_max = 5)

# p = 0.001
panel2 <- Contagion(N = 1000,
                    k = 4,
                    p = 0.001,
                    n = 1,
                    I_0 = 2,
                    t_max = 150,
                    r_max = 5)

# p = 0.01
panel3 <- Contagion(N = 1000,
                    k = 4,
                    p = 0.01,
                    n = 1,
                    I_0 = 2,
                    t_max = 150,
                    r_max = 5)

# p = 0.1
panel4 <- Contagion(N = 1000,
                    k = 4,
                    p = 0.1,
                    n = 1,
                    I_0 = 2,
                    t_max = 150,
                    r_max = 5)
```



As p increases by each order of magnitude, information spreads ever faster.

What happens if we increase n from 1 to 2? Now, at least two neighbours need to be I in order for the focal S agent to convert to I.

```r
# 2 rows, 2 columns
par(mfrow=c(2,2))

# p = 0
panel1 <- Contagion(N = 1000,
                    k = 4,
                    p = 0,
                    n = 2,
                    I_0 = 2,
                    t_max = 350,
                    r_max = 5)

# p = 0.001
panel2 <- Contagion(N = 1000,
                    k = 4,
                    p = 0.001,
                    n = 2,
                    I_0 = 2,
                    t_max = 350,
                    r_max = 5)

# p = 0.01
panel3 <- Contagion(N = 1000,
                    k = 4,
                    p = 0.01,
                    n = 2,
                    I_0 = 2,
                    t_max = 350,
                    r_max = 5)

# p = 0.1
panel4 <- Contagion(N = 1000,
                    k = 4,
                    p = 0.1,
                    n = 2,
                    I_0 = 2,
                    t_max = 350,
                    r_max = 5)
```



Increasing p when $n = 2$ has the opposite effect to $n = 1$: the proportion of I agents decreases as p increases. For $p = 0.1$, the trait hardly spreads at all.

This is because long ties generated when $p > 0$ break up clusters of I nodes that are needed when $n = 2$. In a highly clustered network, groups of I nodes emerge, each one providing at least two I neighbours for neighbouring S nodes to be converted. When some of those local ties are diverted to another part of the ring via rewiring, these clusters of I are less likely to remain. Ties to another part of the network are no good because they will divert to a region full of Ss [1].

This phenomenon was called 'complex contagion' in an influential paper by Centola et al. (2007), in contrast to 'simple contagion' when $n = 1$. Simple contagions that require just a single demonstrator to spread are facilitated by the long ties in small world networks. Examples of simple contagions might be the spread of news stories ('JFK has been shot'), sports results ('Watford beat Liverpool') or job openings ('the coffee shop is hiring'). Simple contagion is also most similar to disease contagion: exposure to just one person with covid can make you catch covid.

Complex contagions, however, require exposure to multiple demonstrators to successfully spread. Examples might be political or social movements, unproven new technologies or methods of contraception. Centola (2010) demonstrated experimentally that health behaviours spread further and faster on clustered

[1] Note that this is why initially we make I_0 adjacent nodes I rather than a proportion I_0, because it is highly unlikely that the latter will lead to two or more adjacent I nodes, and when $n = 2$ the trait will never spread even for $p = 0$.

networks than random networks with lots of long ties because of the need for exposure to multiple demonstrators.

Summary

Social networks capture the social ties between each individual in a population, going beyond the unstructured or weakly structured populations of previous models. Here we explored small world networks, a class of network that vary from highly clustered to highly random. In between these two extremes are networks that remain clustered but contain a small number of long ties connecting distant parts of the network. These small world networks are representative of many real world human social networks. Most of our friends and colleagues are local, but we also probably have a few long-distance acquaintances who connect us to more distant individuals.

For simple contagions which require minimal exposure to a single demonstrator to cause the adoption of a trait, small world networks with more long ties are beneficial. These long ties quickly propel information to all parts of the network. For complex contagions which require exposure to multiple demonstrators, however, long ties can prevent transmission by breaking up local clusters. This shows an important interaction between network topology and cultural trait transmission dynamics.

For further work that explicitly links cultural evolution and social networks, see Cantor et al. (2021), who examined how the diffusion of cultural traits is affected by several other kinds of networks beyond the small world networks examined above; Migliano et al. (2017), who measured the social networks of Agta and BaYaka hunter gatherers and explored how these networks affect cultural diffusion; and Smolla & Akçay (2019), who modelled how cultural selection for skill specialisation can shape social network structure.

In terms of programming, we learned how to use adjacency matrices to capture the edges between each node in a network. This allowed us to use R's built-in matrix functions such as **isSymmetric** to test whether a network is undirected, and **lower.tri** / **upper.tri** to select just the in or out nodes. We also learned how to draw a network using the standard **plot**, **points** and **lines** functions plus a bit of circle geometry, and use adjacency matrices to calculate path lengths and clustering coefficients. There are R packages that do all these things and much more, the most popular being **igraph**. While it is absolutely fine to use such packages, it is often easier to understand exactly what a measure means, or what an algorithm is doing, if we code it from scratch.

Exercises

1. Play around with different values of N, k, p, I_0 and n in **Contagion**. Under what combinations of parameter values does the 'complex contagion' effect occur, i.e. increasing p reduces diffusion of the trait when $n > 1$? Why?

2. Rewrite the **DrawNetwork** function such that filled points indicate I agents and unfilled points indicate S agents. Similar to how we did for Model 10 (polarisation), modify **Contagion** to draw the network at pre-specified timesteps (e.g. $t = 1, 50, 100, 150$) to visualise how the I trait spreads.

3. Rewrite **SmallWorld** to create a network on a square lattice rather than a ring lattice. Parameter k should now be the number of Moore or von Neumann neighbours to whom edges connect. How do simple and complex contagion differ on a square lattice?

References

Cantor, M., Chimento, M., Smeele, S. Q., He, P., Papageorgiou, D., Aplin, L. M., & Farine, D. R. (2021). Social network architecture and the tempo of cumulative cultural evolution. Proceedings of the Royal Society B, 288(1946), 20203107.

Centola, D., & Macy, M. (2007). Complex contagions and the weakness of long ties. American Journal of Sociology, 113(3), 702-734.

Centola, D. (2010). The spread of behavior in an online social network experiment. Science, 329(5996), 1194-1197.

Migliano, A. B., Page, A. E., Gómez-Gardeñes, J., Salali, G. D., Viguier, S., Dyble, M., ... & Vinicius, L. (2017). Characterization of hunter-gatherer networks and implications for cumulative culture. Nature Human Behaviour, 1(2), 1-6.

Smolla, M., & Akçay, E. (2019). Cultural selection shapes network structure. Science Advances, 5(8), eaaw0609.

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. Nature, 393(6684), 440-442.

Model 15: Opinion formation

Introduction

The final model in this mini-series of three adds opinion formation to social contagion (Model 13) and social networks (Model 14). We will recreate a model by Salathé & Bonhoeffer (2008) focussing on vaccination beliefs, a topic of considerable real-world importance.

Salathé & Bonhoeffer (2008) note that many countries suffer disease outbreaks despite ready access to vaccines that prevent those diseases. These outbreaks occur due to the spread of anti-vaccine opinions. These opinions cause people to refuse to vaccinate themselves or their children.

Salathé & Bonhoeffer (2008) assume that anti-vaccine opinions spread via social contagion (see Model 13) on small-world social networks (see Model 14). Rather than simple or complex contagion (Model 14), they assume that an individual switches opinion from pro- to anti-vaccine, or from anti- to pro-vaccine, with probability equal to the proportion of connected neighbours who have a dissimilar opinion multiplied by a parameter Ω that determines the strength of opinion formation. This is a form of unbiased cultural transmission (see Model 1), but restricted to an agent's neighbours rather than the entire population.

For example, if pro-vaccine agent i has ten neighbours connected to them via edges in the small world network, and six of those neighbours are anti-vaccine, then when $\Omega = 1$ agent i has a 0.6 chance of becoming anti-vaccine. When $\Omega = 0.5$ they have a 0.3 chance of flipping, while if $\Omega = 0$ they never switch opinion. Ω therefore generates opinion clustering, with groups of like-minded individuals causing individuals with dissimilar opinions to switch to their view.

Pro-vaccine agents then get vaccinated and become immune, while anti-vaccine agents remain susceptible. A single infected agent is introduced at time $t = 0$, and infection is allowed to proceed for $t_{max} = 300$ timesteps. Note that this is now *disease* contagion rather than social contagion. The question is: how does opinion clustering, determined by Ω, affect the number of subsequent infections? Model 15 will reveal the answer.

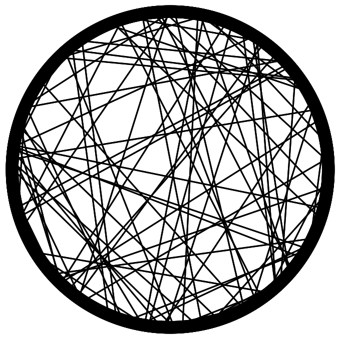
Model 15

First we generate a small world network using the **SmallWorld** function from Model 14. If you haven't already got it loaded, it's repeated below, along with **DrawNetwork** which it uses.

```r
SmallWorld <- function(N, k, p, draw_plot = TRUE) {

  # 1. create empty adjacency matrix
  network <- matrix(0, nrow = N, ncol = N, )

  # 2. create ring lattice network
  for (Row in 1:N) {

    # k/2 neighbours to the right
    Col <- (Row+1):(Row+k/2)
    Col[which(Col > N)] <- Col[which(Col > N)] - N
    network[Row, Col] <- 1

    # k/2 neighbours to the left
    Col <- (Row-k/2):(Row-1)
    Col[which(Col < 1)] <- Col[which(Col < 1)] + N
    network[Row, Col] <- 1

  }

  # 3. rewiring via p

  for (j in 1:(k/2)) {

    for (i in 1:N) {

      if (runif(1) < p) {

        # pick jth clockwise neighbour
        neighbour <- i + j
        if (neighbour > N) neighbour <- neighbour - N

        # pick random new neighbour, excluding self and duplicate edges
        new_neighbour <- which(network[i,] == 0)
        new_neighbour <- new_neighbour[new_neighbour != i]
        new_neighbour <- sample(new_neighbour, 1)

        # remove edge to old neighbour
        network[i,neighbour] <- 0
        network[neighbour,i] <- 0

        # make edge to new neighbour
        network[i, new_neighbour] <- 1
        network[new_neighbour, i] <- 1

      }

    }

  }

  # 4. draw network if draw_network == TRUE

  if (draw_plot == TRUE) {

    DrawNetwork(network)

  }

  # output network from function
  network

}

DrawNetwork <- function(network) {

  # get N from network matrix
  N <- ncol(network)

  # N agents around the origin in a big circle
  plot(NULL,
       xlim = c(-5.5,5.5),
       ylim = c(-5.5,5.5),
       xlab = "",
       ylab = "",
       axes = FALSE,
       asp = 1)

  for (i in 1:N) {
    points(5*sin((i-1)*2*pi/N), 5*cos((i-1)*2*pi/N),
           pch = 16,
           cex = 1.2)

  }

  # lines representing edges
  for (i in 1:N) {

    for (j in which(network[i,] == 1)) {

      lines(x = c(5*sin((i-1)*2*pi/N), 5*sin((j-1)*2*pi/N)),
            y = c(5*cos((i-1)*2*pi/N), 5*cos((j-1)*2*pi/N)))

    }

  }

}
```

Salathé & Bonhoeffer (2008) used $N = 2000$ nodes (i.e. agents) with $k = 10$ edges (connections) per node and rewiring probability $p = 0.01$. We create this below, along with a visualisation of the ring lattice network and a 10x10 snippet of the 2000x2000 adjacency matrix to remind us what small world networks look like.

```
N <- 2000
k <- 10
p <- 0.01

network <- SmallWorld(N, k, p)
```



```
network[1:10, 1:10]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    1    1    1    1    1    0    0    0     0
##  [2,]    1    0    1    1    1    1    1    0    0     0
##  [3,]    1    1    0    0    1    1    1    1    0     0
##  [4,]    1    1    0    0    1    1    1    1    1     0
##  [5,]    1    1    1    1    0    1    1    1    1     1
##  [6,]    1    1    1    1    1    0    1    1    1     1
##  [7,]    0    1    1    1    1    1    0    1    1     1
##  [8,]    0    0    1    1    1    1    1    0    1     1
##  [9,]    0    0    0    1    1    1    1    1    0     1
## [10,]    0    0    0    0    1    1    1    1    1     0
```

Next we assign opinions to each of the N agents, stored in a vector *opinion*. With probability c, agents are pro-vaccine. With probability $1 - c$ they are anti-vaccine. We use 1 to indicate pro-vaccine, and 0 to indicate anti-vaccine. This is more efficient than using a string such as "pro" or "anti", and we can get the proportion of pro-vaccination by taking the mean of *opinion*. We start with $c = 0.5$, half pro- and half anti-vaccine.

```
c <- 0.5

opinion <- sample(c(1,0), N, prob = c(c,1-c), replace = T)
```

```
head(opinion)
```

```
## [1] 0 1 0 1 1 0
```

```
mean(opinion)
```

```
## [1] 0.496
```

Now we simulate opinion formation, incorporating clustering. First we need to get *d*, the proportion of an agent's neighbours who have dissimilar opinions to that agent. Because we'll need to do this in a few different places in the simulation, we write a function **get_d** which returns *d* for a set of *nodes* from a *network* of agents with vaccine-related *opinion*s. The default value of *nodes* is the entire network, but this can be overridden to get *d* for specific nodes, as shown.

```
get_d <- function(network, opinion, nodes = 1:length(opinion)) {

  d <- rep(NA, length(nodes))

  for (i in nodes) {

    # get i's neighbours from network matrix
    neighbours <- which(network[i,] == 1)

    # proportion with differing opinions
    d[which(nodes == i)] <- sum(opinion[neighbours] != opinion[i]) / length(neighbours

  }

  # return d
  d

}

get_d(network, opinion, 1)
```

```
## [1] 0.7
```

```
get_d(network, opinion, 10:15)
```

```
## [1] 0.4545455 0.8000000 0.2727273 0.7000000 0.4000000 0.7000000
```

Right now we need d for all nodes:

```
d <- get_d(network, opinion)
```

Next we cycle through each node in random order, and for each focal node i, with probability $d_i\Omega$, node i switches opinion. This is done with the **ifelse** command, which is useful for assigning one value if a condition is true, and a different value if it's false. We set $\Omega = 1$ so that we can see the effect of clustering later on.

Salathé & Bonhoeffer (2008) made the additional assumption that for every agent that switches (e.g. from pro to anti), another agent switches in the opposite direction (e.g from anti to pro). While artificial, this keeps the proportion of pro- and anti-vaccine agents constant. Consequently, we know that any results we find are due to opinion clustering, rather than overall opinion frequency.

The code below therefore recalculates d for focal i and its neighbours (this is where having a function **get_d** is useful), then repeatedly samples another agent with the same opinion as i now has, and with probability $d_j\Omega$ switches j to i's original pre-switch opinion. We then recalculate d in preparation for the next focal.

```
omega <- 1

# random order of focal nodes
focal <- sample(N)

for (i in focal) {

  # with prob d*omega
  if (runif(1) < d[i] * omega) {

    # i changes its opinion
    opinion[i] <- ifelse(opinion[i], 0, 1)

    # recalculate d for i and its neighbours
    nodes <- c(i, which(network[i,] == 1))
    d[nodes] <- get_d(network, opinion, nodes)

    # get other nodes with the new opinion, excluding self
    same_opinion <- which(opinion == opinion[i])
    same_opinion <- same_opinion[same_opinion != i]

    # pick a random node j and switch with prob d*omega
    # repeat until successful
    repeat {
      j <- sample(same_opinion, 1)

      if (runif(1) < d[j] * omega) {

        # j changes its opinion
        opinion[j] <- ifelse(opinion[j], 0, 1)

        # recalculate d for j and its neighbours
        nodes <- c(j, which(network[j,] == 1))
        d[nodes] <- get_d(network, opinion, nodes)

        break

      }
    }
  }
}
```

As a check, we can make sure that the number of pro- vs anti-vaccine agents hasn't changed:

```
mean(opinion)
```

```
## [1] 0.496
```

Next we create a vector to store the disease status of each agent. We use the SIR notation from Model 13. All agents are initially susceptible (S). We then assume that all agents with a pro-vaccine opinion get vaccinated. These agents all become R, for Recovered (this might seem odd given that they never got the disease, but effectively they 'recovered' from the vaccine and now cannot become infected, just like someone who recovered from the actual disease and gained natural immunity). Note that c is therefore not only the probability of pro-vaccine opinions, but also the proportion of the population who are vaccinated and immune.

```
agent <- rep("S", N)

agent[opinion == 1] <- "R"
```

Next we infect a single random S individual. The **if** statement is there in case there are no S agents, e.g. when $c = 1$, otherwise we get an error message.

```
if (any(agent == "S")) {

      agent[sample(which(agent == "S"), 1)] <- "I"

}
```

Now we start the t-loop, cycling over $t_{max} = 300$ timesteps to model the spread (or not) of the infection. Infection occurs for each S agent with probability $1 - \exp(-\beta I_n)$, where β is the rate of transmission (fixed at $\beta = 0.05$) and I_n is the number of that agent's neighbours that are I. Meanwhile, I nodes recover to become R with probability $\gamma = 0.1$ each timestep.

We are interested in tracking the number of infections that occur beyond the 'patient zero' that was seeded above. For this we use a vector *outbreak* to which the number of new infections are added each timestep. Finally, to save cycling through the t-loop pointlessly and wasting time, we add a **break** clause at the top. If there are either no susceptibles left, or no infecteds left, then the t-loop stops early. Infection cannot occur if there are no susceptibles to become infected, nor if there are no infected agents to infect them.

```
t_max <- 300
beta <- 0.05
gamma <- 0.1
outbreak <- 0

# start t-loop
for (t in 1:t_max) {

  # if no susceptibles or infecteds left, break the loop
  if (!any(agent == "S") | !any(agent == "I")) break

  # get I_n, number of S's infected neighbours
  susceptibles <- which(agent == "S")
  I_n <- rep(NA, length(susceptibles))

  for (i in 1:length(susceptibles)) {

    neighbours <- which(network[susceptibles[i],] == 1)
    I_n[i] <- sum(agent[neighbours] == "I")

  }

  # probability of infection
  prob_infection <- 1 - exp(-beta * I_n)

  # probs to compare
  prob <- runif(length(susceptibles))

  # S agents are infected with prob_infection
  agent[agent == "S"][prob < prob_infection] <- "I"

  # record number of these follow-up infections
  outbreak <- outbreak + sum(prob < prob_infection)

  # recovery with prob gamma
  prob <- runif(sum(agent == "I"))
  agent[agent == "I"][prob < gamma] <- "R"

}
```

How many additional infections were there?

```
outbreak
```

```
## [1] 25
```

This number will vary from simulation to simulation, but with $c = 0.5$ (a low vaccination rate) and $\Omega = 1$ (maximum clustering) it is hopefully greater than zero, and perhaps greater than the threshold of 10 that Salathé & Bonhoeffer (2008) required to declare an 'outbreak'.

We would obviously like to run multiple simulations with the same parameters to obtain a distribution of *outbreak* values, rather than just one. The following function **OpinionFormation** wraps all the above code within an r-loop repeated r_{max} times. Now *outbreak* stores r_{max} values rather than just one. To make **OpinionFormation** standalone, we include **SmallWorld** and **get_d** at the beginning.

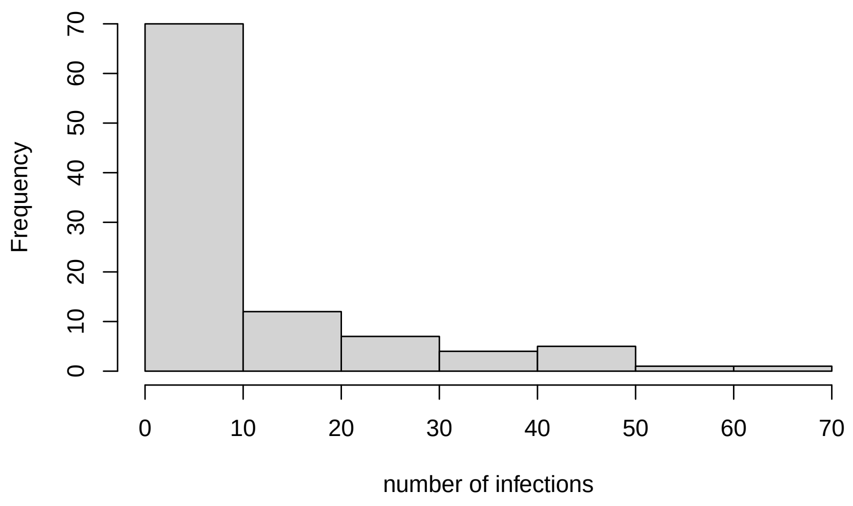
```
OpinionFormation <- function(N = 2000,
                             k = 10,
                             p = 0.01,
                             c,
                             omega,
                             t_max = 300,
                             r_max,
                             beta = 0.05,
                             gamma = 0.1) {

  # define functions

  SmallWorld <- function(N, k, p, draw_plot = TRUE) {
```

```
# 1. create empty adjacency matrix
network <- matrix(0, nrow = N, ncol = N, )

# 2. create ring lattice network
for (Row in 1:N) {

  # k/2 neighbours to the right
  Col <- (Row+1):(Row+k/2)
  Col[which(Col > N)] <- Col[which(Col > N)] - N
  network[Row, Col] <- 1

  # k/2 neighbours to the left
  Col <- (Row-k/2):(Row-1)
  Col[which(Col < 1)] <- Col[which(Col < 1)] + N
  network[Row, Col] <- 1

}

# 3. rewiring via p

for (j in 1:(k/2)) {

  for (i in 1:N) {

    if (runif(1) < p) {

      # pick jth clockwise neighbour
      neighbour <- i + j
      if (neighbour > N) neighbour <- neighbour - N

      # pick random new neighbour, excluding self and duplicate edges
      new_neighbour <- which(network[i,] == 0)
      new_neighbour <- new_neighbour[new_neighbour != i]
      new_neighbour <- sample(new_neighbour, 1)

      # remove edge to old neighbour
      network[i,neighbour] <- 0
      network[neighbour,i] <- 0

      # make edge to new neighbour
      network[i, new_neighbour] <- 1
      network[new_neighbour, i] <- 1

    }
    }

  }

  # 4. draw network if draw_network == TRUE

  if (draw_plot == TRUE) {

    DrawNetwork(network)

  }

  # output network from function
  network

}

get_d <- function(network, opinion, nodes = 1:length(opinion)) {

  d <- rep(NA, length(nodes))

  for (i in nodes) {

    # get i's neighbours from network matrix
    neighbours <- which(network[i,] == 1)

    # proportion with differing opinions
    d[which(nodes == i)] <- sum(opinion[neighbours] != opinion[i]) / length(neighbou

  }

  # return d
  d

}

# initialise output: number of follow-up infections
outbreak <- rep(0, r_max)

# begin r loop:
for (r in 1:r_max) {

  # 1. network generation
  network <- SmallWorld(N, k, p, draw_plot = FALSE)
  # 2. assignment of vaccination opinion
  opinion <- sample(c(1,0), N, prob = c(c,1-c), replace = T)

  # 3. opinion formation

  # skip if omega==0, as no opinion change is possible
  if (omega > 0) {

    # get d, proportion of differing neighbouring opinions
    d <- get_d(network, opinion)

    # random order of focal nodes
    focal <- sample(N)

    for (i in focal) {

      # with prob d*omega
      if (runif(1) < d[i] * omega) {

        # i changes its opinion
        opinion[i] <- ifelse(opinion[i], 0, 1)

        # recalculate d for i and its neighbours
        nodes <- c(i, which(network[i,] == 1))
        d[nodes] <- get_d(network, opinion, nodes)

        # get other nodes with the new opinion, excluding self
        same_opinion <- which(opinion == opinion[i])
        same_opinion <- same_opinion[same_opinion != i]

        # pick a random node j and switch with prob d*omega
        # repeat until successful
        repeat {

          j <- sample(same_opinion, 1)

          if (runif(1) < d[j] * omega) {

            # j changes its opinion
            opinion[j] <- ifelse(opinion[j], 0, 1)

            # recalculate d for j and its neighbours
            nodes <- c(j, which(network[j,] == 1))
            d[nodes] <- get_d(network, opinion, nodes)
            break

          }
        }
      }
    }
  }


  # 4. vaccination according to opinion
  agent <- rep("S", N)
  agent[opinion == 1] <- "R"

  # 5. infection of a random susceptible individual (if any are present)
  if (any(agent == "S")) {

    agent[sample(which(agent == "S"), 1)] <- "I"

  }

  # 6. spread of infection

  # start t-loop
  for (t in 1:t_max) {

    # if no susceptibles or infecteds left, break the loop
    if (!any(agent == "S") | !any(agent == "I")) break

    # get I_n, number of S's infected neighbours
    susceptibles <- which(agent == "S")
    I_n <- rep(NA, length(susceptibles))

    for (i in 1:length(susceptibles)) {

      neighbours <- which(network[susceptibles[i],] == 1)
      I_n[i] <- sum(agent[neighbours] == "I")

    }

    # probability of infection
    prob_infection <- 1 - exp(-beta * I_n)

    # probs to compare
    prob <- runif(length(susceptibles))
```

```
      # S agents are infected with prob_infection
      agent[agent == "S"][prob < prob_infection] <- "I"

      # record number of these follow-up infections
      outbreak[r] <- outbreak[r] + sum(prob < prob_infection)

      # recovery with prob gamma
      prob <- runif(sum(agent == "I"))
      agent[agent == "I"][prob < gamma] <- "R"

    }

  }

  # export outbreak
  outbreak

}
```

Note that all parameters except c and Ω are set to the default values that Salathé & Bonhoeffer (2008) used and did not change. Also r_{max}, as their $r_{max} = 2000$ can lead to long simulation times. With $c = 0.5$, $\Omega = 1$ and $r_{max} = 100$:

```
data_model15 <- OpinionFormation(c = 0.5, omega = 1, r_max = 100)

# histogram of infections
hist(data_model15,
     main = "",
     xlab = "number of infections")
```



```r
# mean number of infections
mean(data_model15)
```

```
## [1] 10.79
```

```r
# proportion of outbreaks (infections > 10)
sum(data_model15 >= 10) / length(data_model15)
```

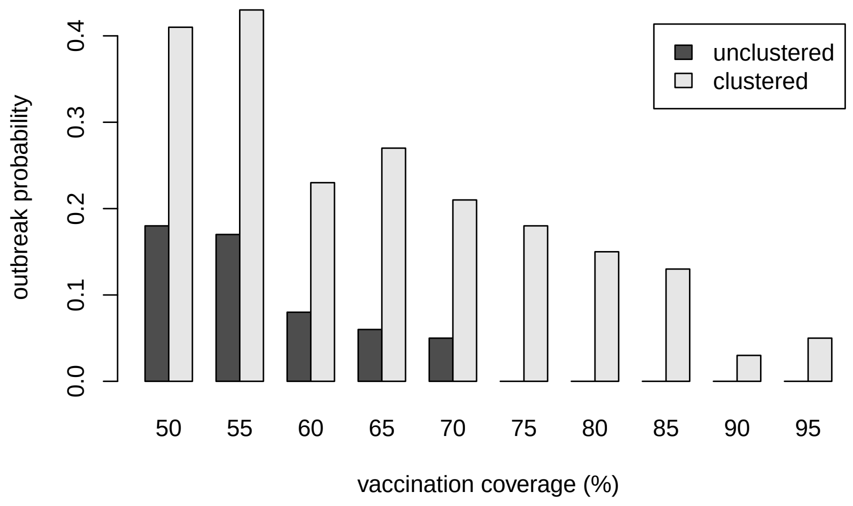
```
## [1] 0.32
```

The histogram shows a range of infection frequencies across the 100 runs, most commonly zero or near to zero, and less frequently greater than 10. The mean number of infections is 10.79, and 0.32 of the runs count as 'outbreaks' according to Salathé & Bonhoeffer (2008). This should be similar to the value of 0.4 found by Salathé & Bonhoeffer (2008; Figure 1c), although not exactly because they ran 2000 simulations rather than 100.

We can complete the recreation of Salathé & Bonhoeffer's Figure 1c by running simulations for a range of values of c, each of which is repeated for $\Omega = 0$ and $\Omega = 1$.

```r
c_values <- seq(0.5,0.95,0.05)
omega_values <- c(0,1)
```

```r
infections <- NULL
r_max <- 100

for (omega in omega_values) {

  for (c in c_values) {

    infections <- append(infections,
                         OpinionFormation(c = c,
                                          omega = omega,
                                          r_max = r_max))

  }

}

# create dataframe for barplot
bar_data <- data.frame(omega = rep(omega_values, each = r_max*length(c_values)),
                       c = rep(c_values, each = r_max),
                       infections)

# reformat based on outbreaks (>=10 infections)
bar_data <- by(bar_data$infections >= 10,
               list(bar_data$omega, bar_data$c*100),
               FUN = sum) / r_max

# plot barplot
barplot(bar_data,
        ylab = "outbreak probability",
        xlab = "vaccination coverage (%)",
        beside = TRUE,
        legend.text = c("unclustered", "clustered"))
```



Here we have recreated Salathé & Bonhoeffer's Figure 1c, albeit less perfectly given our smaller number of runs. The result should be qualitatively the same, however: outbreaks are less likely with greater vaccination coverage (larger c), but for each value of c, outbreaks are more likely when there is opinion clustering ($\Omega = 1$) than with no opinion clustering ($\Omega = 0$). At high levels of vaccination (>80%), an unclustered population has achieved herd immunity, whereas a clustered population still suffers outbreaks.

Summary

Model 15 recreated an important model of Salathé & Bonhoeffer (2008). This model combined unbiased transmission (Model 1), SIR contagion models (Model 13) and small world social networks (Model 14) to explore the effect of vaccine-related opinion clustering on the spread of diseases. Agents switch opinion, from pro- to anti-vaccine or anti- to pro-vaccine, in proportion to the number of neighbours who have dissimilar opinions, a form of unbiased cultural transmission. The clear finding is that opinion clustering increases the number of infections and makes outbreaks more likely, compared to a lack of clustering.

This finding has implications for controlling diseases for which vaccines are available. This is a timely topic given the global covid pandemic, as well as previous outbreaks caused by vaccine refusal such as MMR in the UK. People do not get their opinions from random others. They get them from their friends, family, neighbours and work colleagues. This generates self-reinforcing clusters of anti-vaccine opinions, and consequently disease outbreaks. Spending billions of pounds on vaccine development is pointless if anti-vaccine opinions mean that large clusters of people refuse to take it. Models like Salathé & Bonhoeffer's are important for suggesting interventions that might prevent opinion clustering, or break up clusters that have already formed. See Funk et al. (2010) for further discussion.

In terms of programming, we again saw the advantage of using functions. The previous function **SmallWorld** was re-used, and a new function **get_d** was created and used several times, preventing the repetition of the same code in different places. We also made use of the **break** command to speed up the simulation. If there are instances when you know nothing further will happen, such as when there are no Susceptibles left to be infected or Infecteds left to infect them, use **break** to exit the loop and save time.

Exercises

1. Recreate Salathé & Bonhoeffer's Figure 1b, showing the increase in outbreak probability for different values of c and Ω.

2. Modify **OpinionFormation** to plot the frequency of I, S and R over time as we did in Model 13. Examine the trajectories for different combinations of c and Ω.

3. Salathé & Bonhoeffer (2008) only varied c and Ω. Explore the effect on the number of infections of varying the other parameters: N, k, p, β and γ.

4. In real life, vaccination opinion is likely to be continuous rather than dichotomous. Rewrite **OpinionFormation** so that *opinion* is a probability from 0 to 1 specifying the likelihood that an agent gets vaccinated. Think about how to implement c, the probability of being pro-vaccine (e.g. as a distribution rather than a single probability), and d, now that neighbours will be unlikely to ever have an identical *opinion* (e.g. as a blending rule from Model 8).

5. Rewrite **OpinionFormation** replacing the $d_i\Omega$ switching rule with (i) simple contagion, where just one dissimilar neighbour causes opinion change; (ii) complex contagion, where a threshold of two or more neighbours are needed to cause opinion change; (iii) conformity, following Model 5, where agents adopt the majority opinion amongst its neighbours; and (iv) prestige bias, in which one or more agents are designated as 'celebrities' and

have a disproportionate influence on their neighbour's opinions. How do the number of infections generated by these transmission assumptions differ from Salathé & Bonhoeffer's (2008) original transmission rule?

References

Funk, S., Salathé, M., & Jansen, V. A. (2010). Modelling the influence of human behaviour on the spread of infectious diseases: a review. Journal of the Royal Society Interface, 7(50), 1247-1256.

Salathé, M., & Bonhoeffer, S. (2008). The effect of opinion clustering on disease outbreaks. Journal of The Royal Society Interface, 5(29), 1505-1508.

Model 16: Bayesian iterated learning

Introduction

Model 16 covers a slightly different approach to cultural evolution compared to the population-genetic-inspired models covered previously. These models draw instead from cognitive science, statistics and linguistics. They use the statistical tools of Bayesian inference to explore how cognitive representations or linguistic features evolve over time as they are passed from person to person, generation to generation. After introducing the concepts of Bayesian inference and iterated learning, Model 16 concerns the emergence of regularisation in language evolution.

Bayesian inference

Bayesian inference is a statistical approach to the problem of how to update one's beliefs in light of new data. Let's use an example to illustrate this, inspired by Perfors et al. (2011). Imagine that you develop a headache. This is the *data* you receive from the world, in this case from your body. Assume for the purposes of this simple example that you can only think of three possible causes for the headache: dehydration, a brain tumour, and indigestion. These are your *hypotheses*. Bayesian induction involves combining the *likelihood* of observing the data given each hypothesis with your *prior* beliefs about those hypotheses to generate *posterior* probabilities for each hypothesis, given the new data.

Likelihoods are denoted $P(d|h_i)$, which means the probability of the data d given hypothesis h_i. Given that both dehydration and brain tumours are highly likely to cause headaches while indigestion is not, we set our likelihoods as $P(d|h_{dehydration} = 0.9)$, $P(d|h_{tumour} = 0.8)$ and $P(d|h_{indigestion} = 0.1)$. (Note that these probabilities do not have to be explicitly or consciously held by anyone, they just represent people's estimates for modelling purposes without worrying about how they are implemented mechanistically in the brain. We do not need to assume that people are explicitly calculating Bayes' rule for Bayesian inference to be a good approximation of human decision making.) Priors, denoted $P(h_i)$, represent your estimate of the probability of each hypothesis before seeing any data. Because you often suffer from dehydration and indigestion, but have never had a brain tumour and which you know are (thankfully) rare in the population, your priors are $P(h_{dehydration} = 0.4)$, $P(h_{tumour} = 0.1)$ and $P(h_{indigestion} = 0.5)$. Note that these prior probabilities sum to one, because this is the full set of hypotheses that we are considering, and one of them must be responsible for the headache.

The posterior probabilities of each hypothesis h_i given the data, denoted $P(h_i|d)$, are then given by Bayes' rule:

$$P(h_i|d) = \frac{P(d|h_i)P(h_i)}{\sum_{h_j \in H} P(d|h_j)P(h_j)}$$

Bayes' rule says that the posterior probability of hypothesis h_i is equal to its likelihood multiplied by its prior probability. The denominator is simply the sum of the posterior probabilities for the set of all hypotheses (denoted H) and ensures that all posterior probabilities sum to one. Plugging in our hypothetical likelihoods and priors from above, we have:

$$P(h_{dehydration}|d) = \frac{0.9 * 0.4}{(0.9 * 0.4) + (0.8 * 0.1) + (0.1 * 0.5)} = 0.735$$

$$P(h_{tumour}|d) = \frac{0.8 * 0.1}{(0.9 * 0.4) + (0.8 * 0.1) + (0.1 * 0.5)} = 0.163$$

$$P(h_{indigestion}|d) = \frac{0.1 * 0.5}{(0.9 * 0.4) + (0.8 * 0.1) + (0.1 * 0.5)} = 0.102$$

Bayesian inference tells us that dehydration is a far more likely cause of our headache than either a brain tumour or indigestion. This is probably the conclusion you reached at the very beginning - obviously dehydration is the most likely explanation! That's good, as it indicates that a Bayesian approach adequately approximates your intuitive judgements. However, intuitions can't be put directly into a model, and Bayesian inference provides one way of quantifying how people balance the likelihood of explanations with their prior beliefs to reach a judgement. It's also useful for modelling more complex cases than our simple example which have more data and a larger hypothesis space, such as learning languages, as we will see later. Finally, unlike other approaches, Bayesian inference is useful because it is probabilistic rather than all-or-nothing. We do not seek to accept or reject hypotheses, rather we seek a probability distribution across all possible hypotheses to estimate our (un)certainty in each one.

Iterated learning

Iterated learning is the name given by some cognitive scientists (e.g. Griffiths et al. 2008) and linguists (e.g. Smith & Kirby 2008) to describe the repeated transmission of information from individual to individual along a linear chain of learners. This is much like the 'transmission chains' discussed previously that have been used to study transmission biases in the lab. However, iterated learning is usually placed within a Bayesian framework. The data d comes not from the physical world, nor one's own body (as in the headache example above), but from other individuals. Specifically, each individual in the chain generates some data d which is observed by the next individual in the chain. Each learner combines their priors with the likelihood of the culturally-transmitted data using Bayes' rule, to generate a posterior distribution across all potential hypotheses. This posterior is used to generate new data which is passed to the next individual in the chain, and so on along the chain.

Linguists have used iterated learning to study language evolution (e.g. Kirby et al. 2007; Smith & Kirby 2008). Here the data are utterances that contain information about vocabulary or grammatical rules. Subsequent language learners, such as infants learning their first language or adults learning a second language, must infer the grammatical rules of the language from the limited data they receive from other speakers. The priors in this context are innate, cognitively attractive or culturally acquired expectations about the possible hypothesis space of real languages. Cognitive scientists have used Bayesian iterated learning to address similar questions for non-linguistic culturally transmitted representations, such as object categorisation schemes or colour terminologies (e.g. Griffiths et al. 2008). The question addressed by all this research, as well as Model 16, is: how do languages or cognitive representations culturally evolve over time under the assumption of Bayesian iterated learning? One major claim that we will examine in these models is that languages and cognitive structures always come to reflect the priors (or 'inductive biases') of the learners, irrespective of the initial data observed by the first individual in the chain.

Model 16

Model 16 recreates a common model of Bayesian iterated learning examining the emergence of linguistic regularisation, as developed by Reali & Griffiths (2009), Ferdinand et al. (2014) and Navarro et al. (2018), amongst others. As always, this model is highly simplified compared to real languages and language learning. But as we have repeatedly seen, simple models are useful for understanding such complex phenomena precisely because of their simplicity.

Imagine a grammatical feature that can take one of two forms. For example, verbs can either follow the subject ("I go") or precede the subject ("go I"). (NB While I use a linguistic example here, technically this model applies to any socially transmitted trait that takes two forms, e.g. social conventions such as shaking hands with either the right or the left hand.) We define θ as the probability of one variant (say, subject-verb) being used in the language, and $1-\theta$ as the probability of the other variant (verb-subject) being used. Your task as a naive learner is to infer from a set of utterances the grammatical rule used in your society, i.e. to infer the value of θ for this language, which is unknown to you except via data received from other speakers.

In particular, we are interested in whether the language is regular or irregular. A regular language always uses the same variant (e.g. always subject-verb, or always verb-subject). Regular languages therefore have either $\theta = 0$ or $\theta = 1$. An irregular language uses some mix of variants (e.g. sometimes subject-verb, sometimes verb-subject). Irregular languages therefore have $0 < \theta < 1$. A highly irregular language has $\theta \approx 0.5$, such that one variant is used about half the time and the other variant used the other half.

Our hypothesis space is therefore the full range of possible θ values. Because θ is a probability, this goes from 0 to 1 on a continuous scale. We will call this hypothesis space p and code it as follows:

```
p <- seq(0, 1, by = 0.01)
p
```

```
##   [1] 0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
##  [16] 0.15 0.16 0.17 0.18 0.19 0.20 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29
##  [31] 0.30 0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.40 0.41 0.42 0.43 0.44
##  [46] 0.45 0.46 0.47 0.48 0.49 0.50 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59
##  [61] 0.60 0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69 0.70 0.71 0.72 0.73 0.74
##  [76] 0.75 0.76 0.77 0.78 0.79 0.80 0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88 0.89
##  [91] 0.90 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1.00
```

The granularity here is somewhat arbitrary. I've chosen intervals of 0.01, but you could use a smaller interval to get a more fine-grained hypothesis space and therefore more accurate results, albeit at the cost of longer run times.

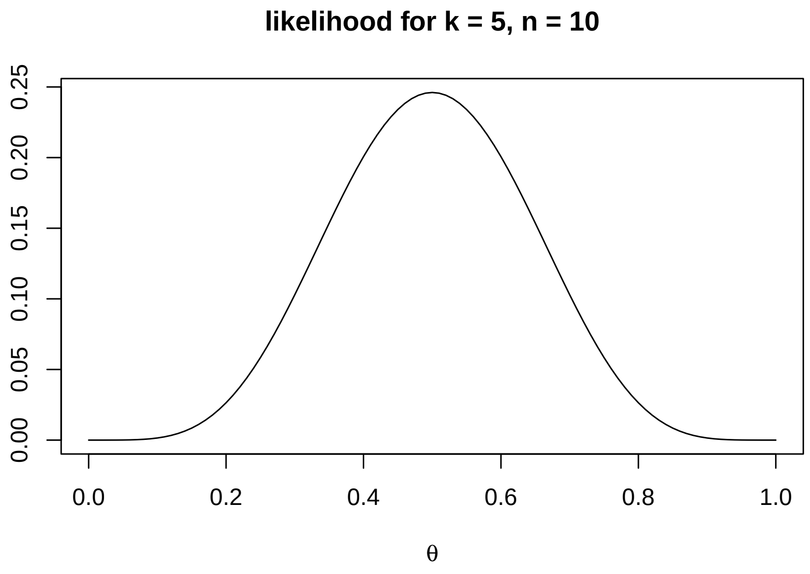
The data take the form of n independent observations of the grammatical rule. In the context of an iterated learning chain, this is like hearing n examples of the grammatical rule from the previous individual in the chain. We assume that k of these observations fit variant 1 (e.g. subject-verb) while $n-k$ fit variant 2 (e.g. verb-subject). In our simulations we will assume $n = 10$ observations with $k = 5$ of those fitting the first variant:

```
n <- 10
k <- 5
```

Note that while p is a continuous probability ranging from 0 to 1, k is an integer that varies from 0 to n. Our initial k implies a highly irregular language, given that half of the utterances fit variant 1 (e.g. subject-verb) and half fit variant 2 (e.g. verb-subject). However, this is a very small set of data, and may not accurately reflect the θ of the entire language. A θ of 0.5 could generate $k = 5$, but so could a range of other θ values. Conversely, $\theta = 0.5$ won't always generate 5/10 observations, just as flipping a coin 10 times doesn't always give 5 heads and 5 tails. This is where Bayesian inference comes in: we must infer θ from the likelihood of our data, as well as our priors.

First the likelihoods. The likelihood of k successes in n independent events with two outcomes, success/fail, is given by the binomial distribution. In our case a 'success' is an observation of grammatical variant 1 and a 'fail' is an observation of variant 2. The likelihoods for our data across all possible hypotheses in p can be obtained using the **dbinom** command:

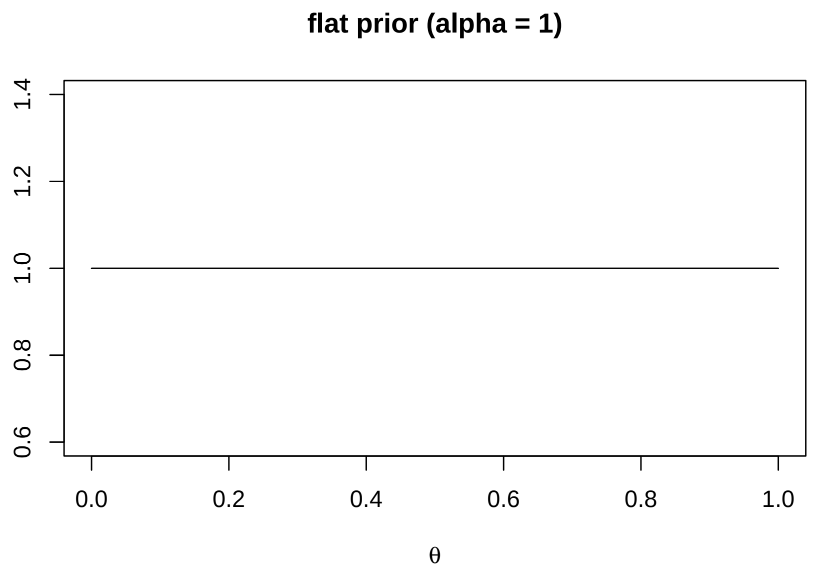
```
lik <- dbinom(k, n, p)

plot(lik,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "likelihood for k = 5, n = 10")
```



The plot shows, as we would expect, that a θ of 0.5 most likely generates $k = 5$ observations, but that a range of other values are also possible. Try different values of k to see how the likelihoods change.

Now the priors. Reali & Griffiths (2009) introduced the beta distribution for quantifying priors in this regularisation model. The beta distribution takes two parameters that determine its shape, α and β. For now we will assume $\alpha = \beta$ so we have one less parameter to worry about. Assuming $\alpha = \beta$ also gives symmetrical distributions which are easier to interpret. The following code shows a beta distribution with $\alpha = 1$:

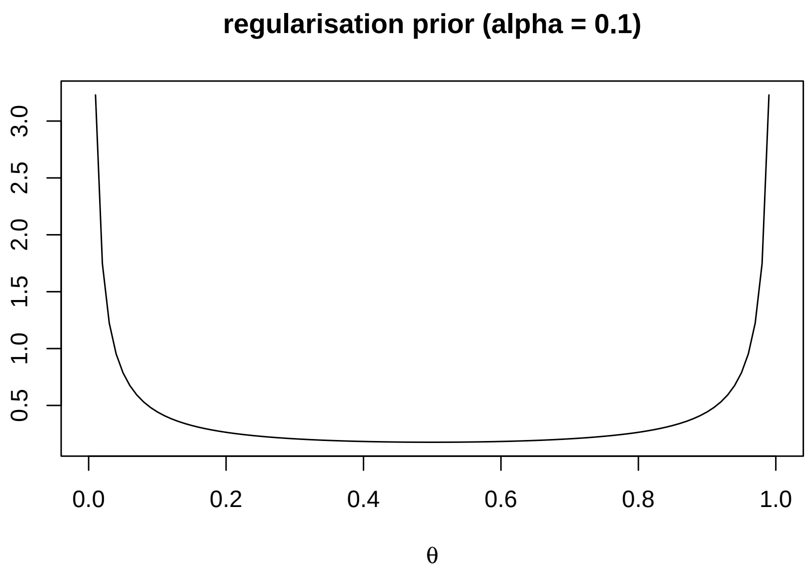
```r
alpha <- 1
beta <- alpha

prior <- dbeta(p, alpha, beta)

plot(prior,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "flat prior (alpha = 1)")
```



This gives a flat prior. The learner expects every value of θ to be equally likely and brings no particular expectation to the inference problem.

Reducing α to less than 1 gives a prior distribution concentrated at 0 and 1:

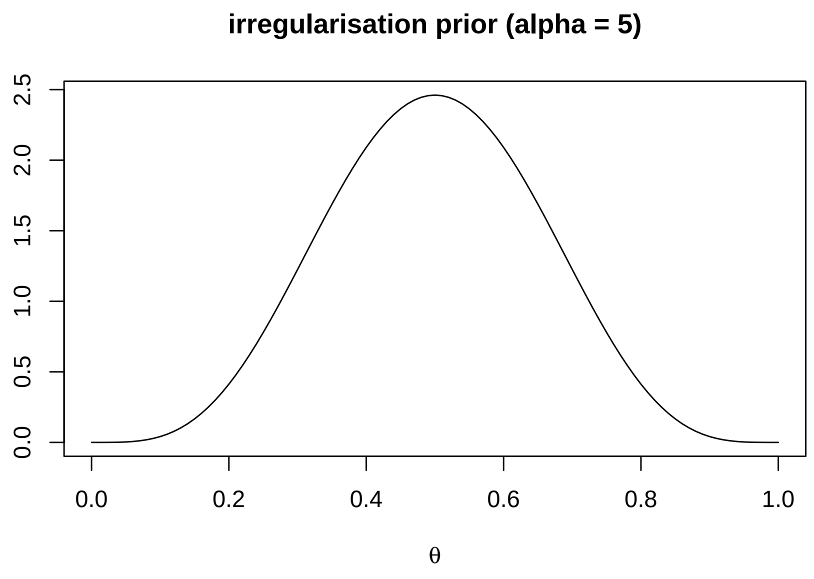
```
alpha <- 0.1
beta <- alpha

prior <- dbeta(p, alpha, beta)

plot(prior,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "regularisation prior (alpha = 0.1)")
```



This represents a regularisation prior. The learner expects a regular language, i.e. one which either always has feature 1 ($\theta = 1$) or always has feature 2 ($\theta = 0$).

Finally, increasing α above 1 generates a bell-shaped distribution much like a normal distribution:

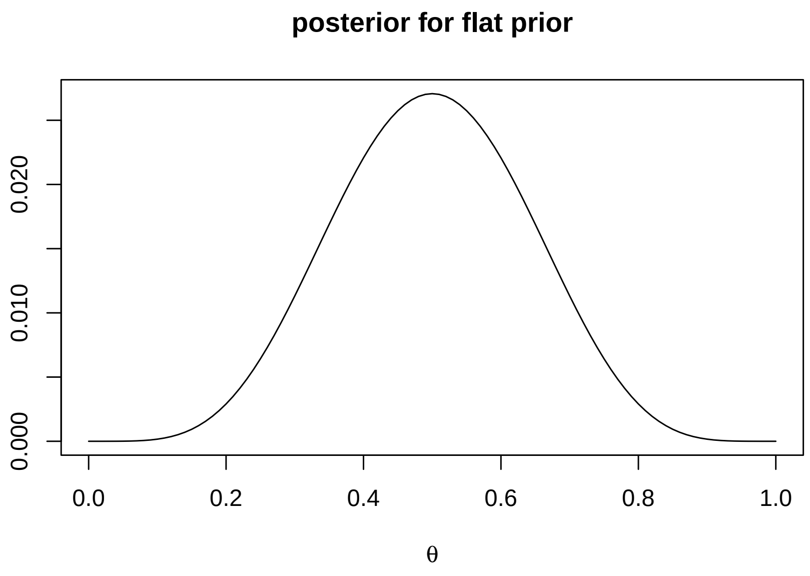
```
alpha <- 5
beta <- alpha

prior <- dbeta(p, alpha, beta)

plot(prior,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "irregularisation prior (alpha = 5)")
```



This represents an irregularisation prior. The learner expects an irregular language with a θ peaking at 0.5, meaning that both grammatical variants are equally likely to be observed in the language.

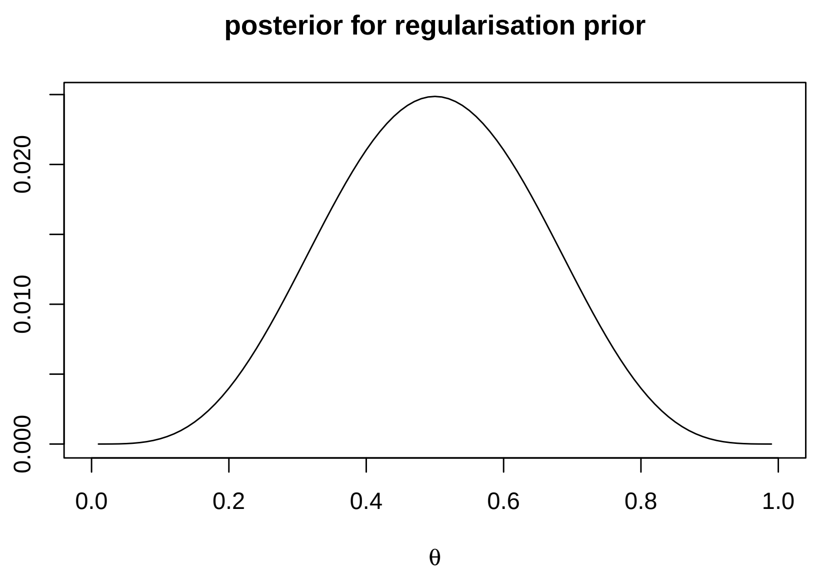
The posterior distribution is then obtained by multiplying the likelihood by the prior and dividing by the sum of all posterior probabilities, as per Bayes' rule. The following code does this for the flat priors:

```
alpha <- 1
beta <- alpha

prior <- dbeta(p, alpha, beta)

post <- lik*prior
post <- post / sum(post)

plot(post,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "posterior for flat prior")
```



With a flat prior, the posterior is entirely determined by the likelihood and centres on $\theta = 0.5$, just like the likelihood. What about a regularisation prior?

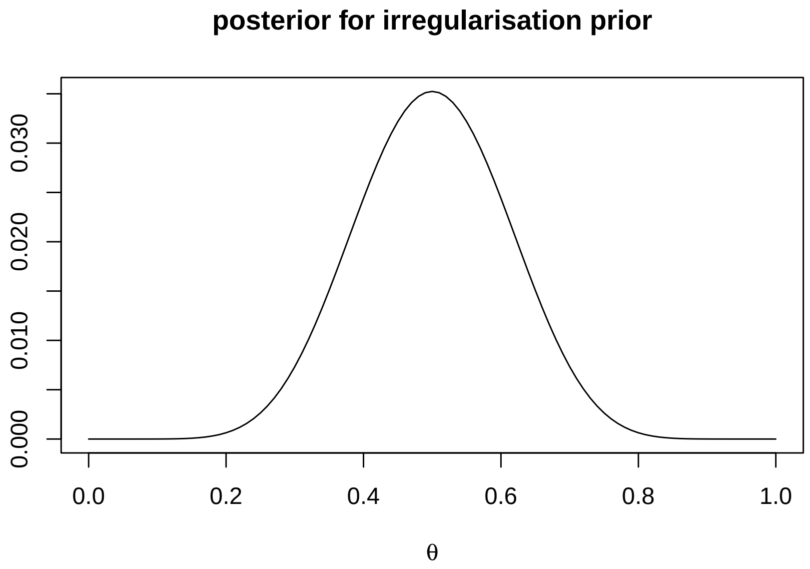
```
alpha <- 0.1
beta <- alpha

prior <- dbeta(p, alpha, beta)

post <- lik*prior
post <- post / sum(post, na.rm = TRUE)

plot(post,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "posterior for regularisation prior")
```



Here the prior seems to have very little effect on the posterior distribution, which again looks very similar to the likelihood.

Finally irregularisation priors:

```
alpha <- 5
beta <- alpha

prior <- dbeta(p, alpha, beta)

post <- lik*prior
post <- post / sum(post)

plot(post,
     x = p,
     xlab = expression(theta),
     ylab = "",
     type = 'l',
     main = "posterior for irregularisation prior")
```



Because the prior and likelihood agree in this case, the posterior looks narrower than either of them. Our confidence in $\theta = 0.5$ has increased. Try different values of k and α to explore how priors and likelihoods combine to generate posterior distributions.

Before going further, we can take advantage of a Bayesian trick to simplify and speed up our code. A binomial likelihood and a beta prior form what is known as a conjugate pair. The posterior distribution of such a pair is a beta distribution with parameters $k+\alpha$ and $n-k+\beta$. Don't worry about the details or derivation of this, it's best to just take it as a statistical convenience. The key thing is that we can use this fact to generate the posterior direct from k, α and β, without having to calculate the prior and likelihood and multiply them together. Let's compare the new conjugate method with the old step-by-step method:

```r
# old method
lik <- dbinom(k, n, p)
prior <- dbeta(p, alpha, beta)

post <- lik*prior
post <- post / sum(post)

head(post)
```

```
## [1] 0.000000e+00 8.438890e-15 3.943421e-12 1.382310e-10 1.677064e-09
## [6] 1.137132e-08
```

```
# new method
post_conjugate <- dbeta(p, k + alpha, n - k + beta)
post_conjugate <- post_conjugate / sum(post_conjugate)

head(post_conjugate)
```

```
## [1] 0.000000e+00 8.438890e-15 3.943421e-12 1.382310e-10 1.677064e-09
## [6] 1.137132e-08
```

I only show the first six values using **head()** for brevity, but you can see that they are identical. You can check yourself that the rest is identical. We'll use this more efficient method from now on.

So far we have generated a posterior distribution from some data. In an iterated learning chain, this occurs when an individual receives data from the previous individual in the chain. Now we need to simulate the next step, where this receiving individual generates new data from their posterior distribution to pass on to the next individual in the chain.

First we need to pick a value of θ from the posterior distribution. This will be the hypothesis used to generate the data.

There are two ways of doing this. The first is to randomly sample from the hypothesis space in proportion to the posterior probabilities. This is known as 'sampling'. Now that we know the conjugate shortcut to calculating the posterior, this can be done with a single **rbeta** command:

```
theta <- rbeta(1, k + alpha, n - k + beta)
theta
```

```
## [1] 0.4047704
```

θ here should be close to 0.5, but not exactly 0.5 because we are randomly sampling.

The second way of choosing a θ is to pick the hypothesis with the maximum posterior probability. This is known as the 'maximum a posteriori' (MAP) hypothesis. To obtain this we can use the formulae for the mode of a beta distribution (you can look this up on wikipedia):

```
if ((k + alpha) > 1 & (n - k + beta) > 1) {

  theta <- (k + alpha - 1) / (n + alpha + beta - 2)

} else if ((k + alpha) <= 1 & (n - k + beta) > 1) {
  theta <- 0

} else if ((k + alpha) > 1 & (n - k + beta) <= 1) {

  theta <- 1

}

theta
```

```
## [1] 0.5
```

θ here should be exactly 0.5 given that this is the maximum of this posterior distribution.

Now we need to use our θ to generate a new k to be the data for the next individual in the chain. This is another task for the binomial distribution, which converts probabilities into discrete picks. The following code uses **rbinom** to pick a random draw from a binomial with n trials and probability of success θ:

```
k <- rbinom(1, n, theta)
k
```

```
## [1] 4
```

The new k should be an integer around 5, but possibly not exactly 5, given that we are sampling from only $n = 10$ observations.

Finally we can generate a probability distribution for all possible values of k across all values of θ. We label this *prob_k*. You can think of this as the posterior transformed from a continuous probability from 0 to 1 into a discrete integer scale from 0 to n to give the probability of each new k, without the stochasticity introduced by random picks. The following code cycles through all integers from 0 to n and, for each one, multiplies the posterior across all p values by the binomial distribution for that integer (like we did above with the likelihood). We then sum across all p values and normalise to ensure *prob_k* sums to 1.

```
# get probability distribution of new picks
prob_k <- rep(NA, n+1)
names(prob_k) <- 0:n

for (j in 0:n) {

  prob_k[j+1] <- sum( dbinom(j, n, p) * post )
}

prob_k <- prob_k / sum(prob_k)
prob_k
```

```
##          0          1          2          3          4          5          6 
## 0.00461198 0.02427358 0.06675234 0.12565146 0.17866067 0.20009995 0.17866067 
##          7          8          9         10 
## 0.12565146 0.06675234 0.02427358 0.00461198
```

As expected, $k = 5$ is most likely, but other values are also quite likely to be picked too.

Now let's put all these bits of code into a function, **BayesianInference**, which takes data k and the other parameters as input and generates and exports both the new k value and the entire k probability distribution. Note (i) we set the default β to be equal to α so that if β is left unspecified it defaults to be equal to α; (ii) we add a couple of lines of code to avoid infinity in the posterior when the conjugate beta parameters are less than 1; and (iii) we define a variable *sampling* which implements sampling if *TRUE* (the default) and MAP if *FALSE*.

```
BayesianInference <- function(n = 10,
                              k = 5,
                              alpha = 1,
                              beta = alpha,
                              sampling = TRUE) {

  # hypothesis space
  p <- seq(0, 1, by = 0.01)

  # avoid infinity when beta distribution parameters are less than 1
  if ((k + alpha) < 1) p[1] <- 0.00001
  if ((n - k + beta) < 1) p[length(p)] <- 0.99999

  # generate posterior
  post_conjugate <- dbeta(p, k + alpha, n - k + beta)
  post_conjugate <- post_conjugate / sum(post_conjugate)

  if (sampling == TRUE) {

    # pick a random sample from the posterior
    theta <- rbeta(1, k + alpha, n - k + beta)

  } else {
```

```r
    # pick the maximum posterior probability (MAP)

    if ((k + alpha) > 1 & (n - k + beta) > 1) {

      theta <- (k + alpha - 1) / (n + alpha + beta - 2)

    } else if ((k + alpha) <= 1 & (n - k + beta) > 1) {

      theta <- 0

    } else if ((k + alpha) > 1 & (n - k + beta) <= 1) {

      theta <- 1

    }

  }

  # draw a new k from a binomial using theta
  k <- rbinom(1, n, theta)

  # get probability distribution of new picks
  prob_k <- rep(NA, n+1)
  names(prob_k) <- 0:n

  for (j in 0:n) {

    prob_k[j+1] <- sum( dbinom(j, n, p) * post_conjugate )

  }

  prob_k <- prob_k / sum(prob_k)

  # export new k and prob_k
  list(k = k, prob_k = prob_k)

}
```

Here is an example output with regularisation priors:

```r
BayesianInference(alpha = 0.1)
```

```
## $k
## [1] 8
```

```
##
## $prob_k
##          0          1          2          3          4          5          6
## 0.01055127 0.03816419 0.07996999 0.12513211 0.15979708 0.17277071 0.15979708
##          7          8          9         10
## 0.12513211 0.07996999 0.03816419 0.01055127
```

Now we can use **BayesianInference** to simulate an iterated learning chain of $N = 20$ individuals. Let's go step by step. The output will be each agents' k and *prob_k*. These are held in a vector and matrix respectively, initialised with zeroes:

```
N <- 20

# vector to hold N k values
k_vector <- rep(0, N)

# matrix to hold N probability distributions for k
k_matrix <- matrix(0, nrow = N, ncol = n+1)
```

For the actual iterated learning, we cycle through N agents, for each one getting the next generation's k and *prob_k* using **BayesianInference**. For now we use its default parameters, which specify flat priors ($\alpha = 1$). k and *prob_k* are stored in *k_vector* and *k_matrix* respectively. We also update the current value of k with the new one to pass to the next generation.

```
for (i in 1:N) {

  next_gen <- BayesianInference()

  # k for next generation
  k <- next_gen$k

  # store probability distribution and k
  k_vector[i] <- next_gen$k
  k_matrix[i,] <- next_gen$prob_k

}
```

A look at *k_vector* reveals a range of values:

```
k_vector
```

```
##  [1] 5 3 6 3 7 6 4 8 0 3 5 5 5 4 5 6 3 6 8 3
```

There is lots of stochasticity here, and ideally we would like to generalise across multiple independent iterated learning chains. Let's do this by putting the N loop above inside a loop over $r_{max} = 10000$ runs. As in previous models, we keep a running total in *k_vector* and *k_matrix* and then divide them by r_{max} at the end to get their mean values across all runs. We also need to reset k to the initial default value at the start of every run, otherwise each run would start with the last k of the previous run making them non-independent.

```r
r_max <- 10000
k <- 5

# vector to hold N k values
k_vector <- rep(0, N)

# matrix to hold N probability distributions for k
k_matrix <- matrix(0, nrow = N, ncol = n+1)

# record starting k
k_initial <- k

for (r in 1:r_max) {

  k <- k_initial

  for (i in 1:N) {

    next_gen <- BayesianInference()

    # k for next generation
    k <- next_gen$k

    # store probability distribution and k
    k_vector[i] <- k_vector[i] + next_gen$k
    k_matrix[i,] <- k_matrix[i,] + next_gen$prob_k

  }

}

# average across all runs
k_vector <- k_vector / r_max
k_matrix <- k_matrix / r_max
```

We can now clearly see that the mean *k_vector* across all $r_{max} = 10000$ runs is close to 5, as expected for a flat prior and starting $k = 5$:

```
k_vector
```

```
##  [1] 4.9881 5.0019 5.0016 4.9743 5.0045 4.9758 4.9527 5.0169 5.0105 4.9750
## [11] 5.0173 4.9469 5.0190 4.9984 4.9950 5.0199 5.0052 4.9677 4.9770 4.9819
```

As usual we wrap all this in a function which we call **IteratedLearning**. Note that we set all the parameter values in the **IteratedLearning** call and pass them to **BayesianInference**. We also export the parameters for use later in plotting functions.

```
IteratedLearning <- function(n = 10,
                             k = 5,
                             alpha = 1,
                             beta = alpha,
                             sampling = TRUE,
                             N = 20,
                             r_max = 10000) {

  # vector to hold N k values
  k_vector <- rep(0, N)

  # matrix to hold N probability distributions for k
  k_matrix <- matrix(0, nrow = N, ncol = n+1)

  # record starting k
  k_initial <- k

  for (r in 1:r_max) {

    # reset k
    k <- k_initial

    for (i in 1:N) {

      next_gen <- BayesianInference(n,
                                    k,
                                    alpha,
                                    beta,
                                    sampling)

      # k for next generation
      k <- next_gen$k

      # store probability distribution and k
      k_vector[i] <- k_vector[i] + next_gen$k
      k_matrix[i,] <- k_matrix[i,] + next_gen$prob_k

    }

  }

  # average across all runs
  k_vector <- k_vector / r_max
  k_matrix <- k_matrix / r_max

  # export k_vector, k_matrix and parameters
  list(k_vector = k_vector,
       k_matrix = k_matrix,
       parameters = c(n = n,
                      k = k_initial,
                      alpha = alpha,
                      beta = beta,
                      sampling = sampling,
                      N = N,
                      r_max = r_max))

}
```

To verify everything worked as expected we can check that the *k_vector* from the default values matches the ones shown above.

```
data_model16 <- IteratedLearning()

data_model16$k_vector
```

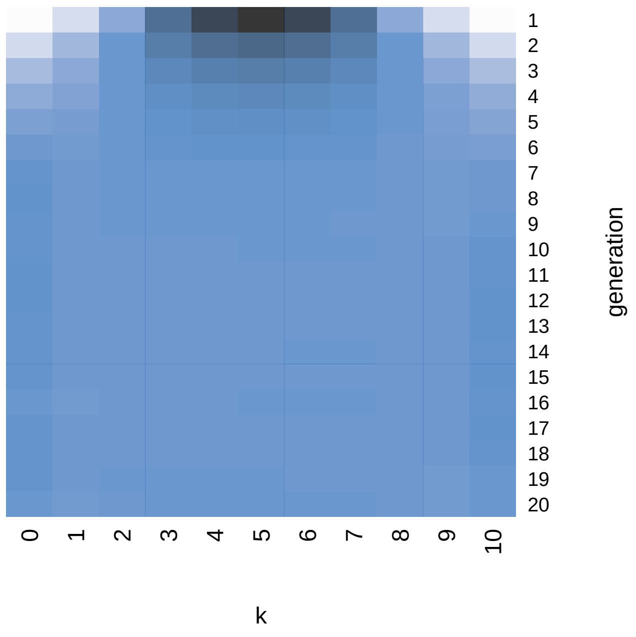
```
##  [1] 4.9897 4.9909 4.9956 4.9680 4.9465 4.9586 4.9521 4.9701 5.0009 4.9956
## [11] 5.0043 5.0132 5.0113 5.0188 5.0167 5.0108 5.0039 4.9738 4.9985 4.9946
```

If you inspect *k_vector* for $\alpha = 0.1$, $\alpha = 1$ and $\alpha = 5$ you'll see that they are all around 5. However this mean value obscures different underlying probability distributions. To see this, we can follow Reali & Griffiths (2009, Fig 1) and make a heat plot showing the distribution of k over the generations using the **heatmap** command, as in the following function:

```
IteratedHeatPlot <- function(data_model16) {

  k_matrix <- data_model16$k_matrix
  # heat map showing probability distribution over time
  heatmap(k_matrix,
          Rowv = NA,
          Colv = NA,
          labCol = 0:(ncol(k_matrix)-1),
          revC = TRUE,
          col = hcl.colors(100, palette = "Oslo", alpha = 0.8, rev = F),
          scale = "none",
          ylab = "generation",
          xlab = "k")

}
```

I'll leave you to play around with the different arguments of **heatmap** and read its help function, but essentially the command above creates a plot with generation on the vertical axis starting at the top, k on the horizontal axis, and the darkness of each cell indicating the probability of that k value in that generation. Darker colors indicate higher probabilities. For example, with the default $\alpha = 1$, a flat prior, we generate this heatmap:

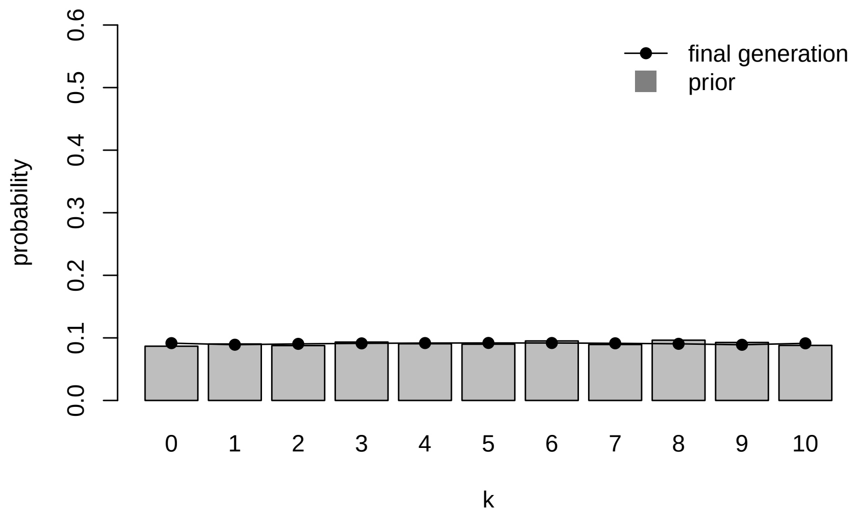
```
IteratedHeatPlot(data_model16)
```



While generation 1 at the top resembles the likelihood concentrated around $k = 5$, in just a few generations the distribution becomes uniform with roughly equal probability of each value of k. In other words, the distribution has converged on the flat prior.

To compare the final generation more precisely with the priors, we can generate a plot from Navarro et al. (2018, Fig 3). Here we simulate r_{max} draws of θ direct from the prior distribution, without any likelihoods or posteriors. We use this θ to generate a k value like we did before, then calculate the proportion of each k value from 0 to n. This is plotted as a bar plot using the **barplot** command, along with the final generation iterated learning probability distribution as a line.

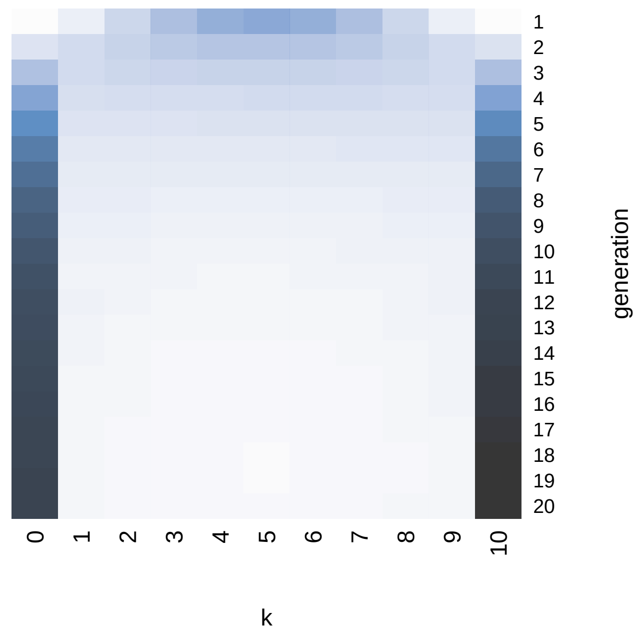
```
IteratedFinalPlot <- function(data_model16, ymax = 0.6) {

  # retrieve parameters
  n <- data_model16$parameters["n"]
  r_max <- data_model16$parameters["r_max"]
  alpha <- data_model16$parameters["alpha"]
  beta <- data_model16$parameters["beta"]
  k_matrix <- data_model16$k_matrix

  # simulate prior distribution
  prior_dist <- rep(NA, r_max)

  for (i in 1:r_max) {

      theta <- rbeta(1, alpha, beta)
      prior_dist[i] <- rbinom(1, n, theta)

  }

  prior_dist <- table(factor(prior_dist, levels = 0:n))
  prior_dist <- prior_dist / sum(prior_dist)

  # draw barplot for simulated priors
  b <- barplot(height = prior_dist,
               names = 0:(length(prior_dist)-1),
               xlab = "k",
               ylab = "probability",
               ylim = c(0,ymax))

  # add line for the final generation iterated learning chain
  lines(k_matrix[nrow(k_matrix),],
        x = b,
        pch = 19,
        type = 'o')
  # add legend
  legend("topright",
         legend=c("final generation","prior"),
         pch = c(19,15),
         col=c("black","grey50"),
         pt.cex=c(1,2),
         bty="n",
         lty=c(1,NA),
         lwd=c(1,1))

}

IteratedFinalPlot(data_model16)
```



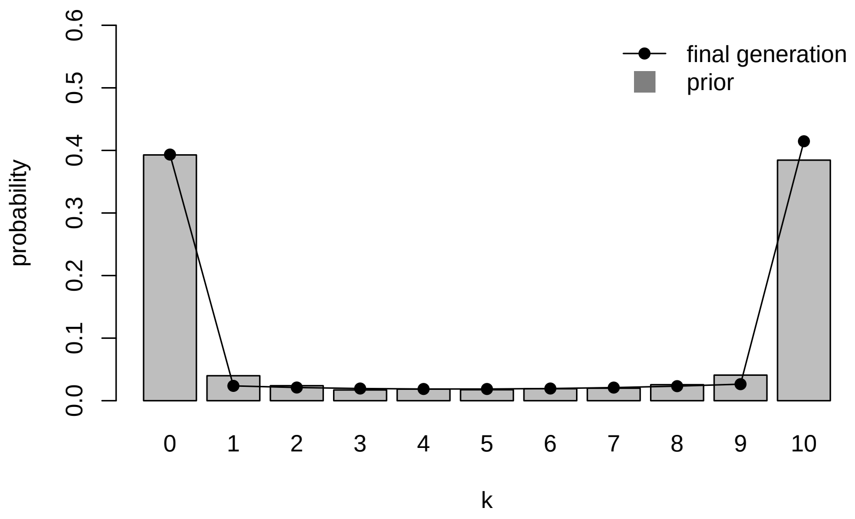
Here we confirm that after $N = 20$ generations the initial data ($k = 5$) is overridden by the flat prior.

Now we create plots for a regularisation prior:

```
data_model16 <- IteratedLearning(alpha = 0.1)

IteratedHeatPlot(data_model16)
```



```
IteratedFinalPlot(data_model16)
```

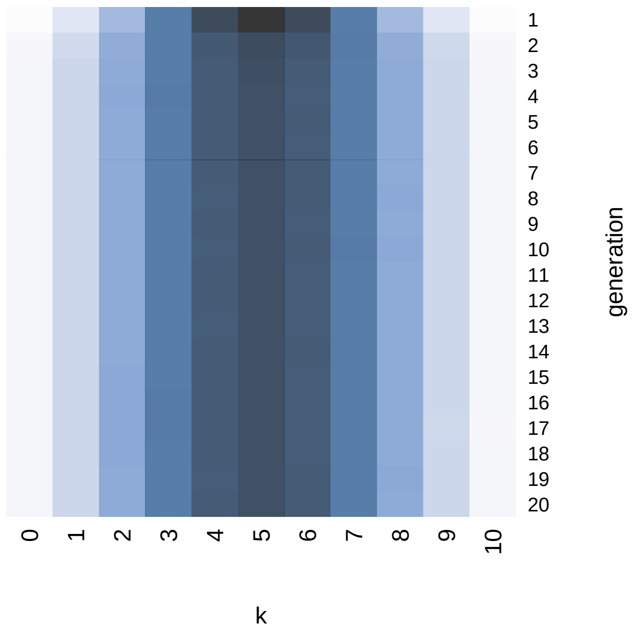


Strikingly, the heatmap shows that even though the first few generations retain the normal distribution of the likelihood, the chains soon converge on either

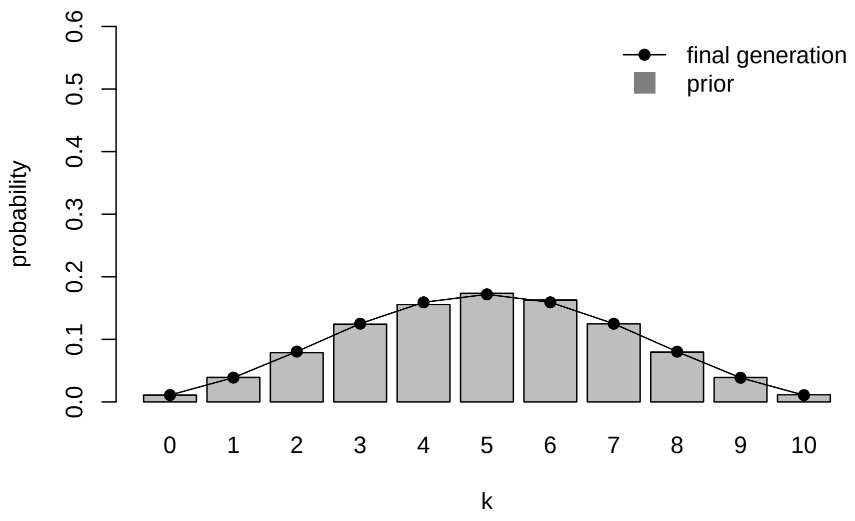
$k = 0$ or $k = 10$. The final distribution plot confirms that the chains have converged on the prior.

And finally for an irregularisation prior:

```r
data_model16 <- IteratedLearning(alpha = 5)

IteratedHeatPlot(data_model16)
```



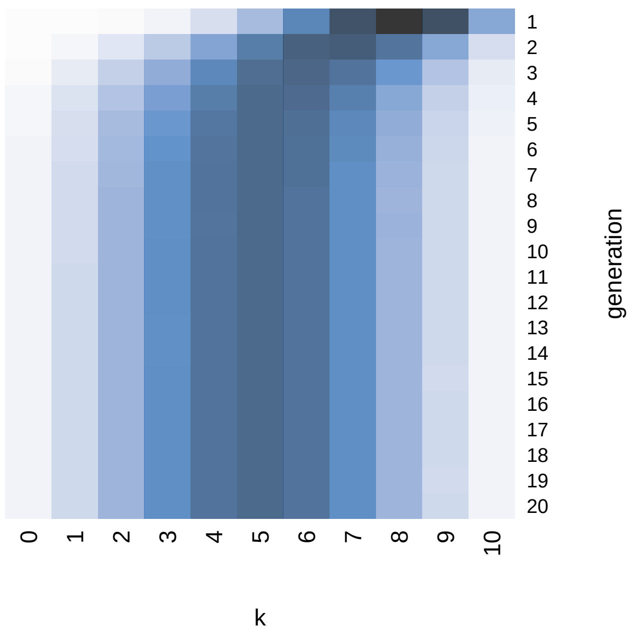
```r
IteratedFinalPlot(data_model16)
```



Again, the final generation converges on the irregularisation prior, favouring languages that have a mix of the two grammatical variants.

We can now state our interim conclusion: despite all of the posteriors looking roughly the same after one generation, resembling a normal distribution centred around $k/n = 0.5$, after 20 generations the population has converged on the prior distribution in each case. This is the conclusion reached by Reali & Griffiths (2009) and several other Bayesian iterated learning models and experiments. Try different values of k to confirm that this convergence on the priors is robust to any starting data. For example, a starting $k = 10$ with irregularising priors ($\alpha = 5$) converges on those normally-distributed priors fairly rapidly:

```r
data_model16 <- IteratedLearning(k = 10,
                                 alpha = 5)

IteratedHeatPlot(data_model16)
```



While this is an important result, there are cases when iterated learning chains do not converge on the prior. One such case was explored by Navarro et al. (2018). So far we have assumed that all agents have identical priors. However, in many real world situations it is unlikely that everyone has the same priors. People come to problems with different expectations and past experiences, while some people may be more confident or certain than others.

Navarro et al. (2018) therefore assumed two different prior distributions, i.e. two sets of prior parameters. Each agent in the iterated learning chain uses parameters α_1 and β_1 with probability h, and uses α_2 and β_2 with probability $1 - h$.

The following function **MixedIteratedLearning** modifies **IteratedLearning** to accept two sets of prior parameters. With probability h the first set is passed to **BayesianInference**, and with probability $1 - h$ the second set is passed. Otherwise everything is the same. Note the default parameter values are set so that if only α_1 and β_1 are set, these values are copied to α_2 and β_2.

```
MixedIteratedLearning <- function(n = 10,
                                  k = 5,
                                  alpha1 = 1,
                                  beta1 = alpha1,
                                  alpha2 = alpha1,
                                  beta2 = beta1,
                                  h = 0.5,
                                  sampling = TRUE,
                                  N = 20,
                                  r_max = 10000) {

# vector to hold N k values
k_vector <- rep(0, N)

# matrix to hold N probability distributions for k
k_matrix <- matrix(0, nrow = N, ncol = n+1)

# record starting k
k_initial <- k

for (r in 1:r_max) {

  # reset k
  k <- k_initial

  # N probabilities that learner uses alpha1/beta1 rather than alpha2/beta2
  h_prob <- runif(N)

  for (i in 1:N) {

    if (h_prob[i] < h) {

      # alpha1 and beta1
      next_gen <- BayesianInference(n,
                                    k,
                                    alpha = alpha1,
                                    beta = beta1,
                                    sampling)

    } else {

      # alpha2 and beta2
      next_gen <- BayesianInference(n,
                                    k,
                                    alpha = alpha2,
                                    beta = beta2,
                                    sampling)

    }

    # k for next generation
    k <- next_gen$k

    # store probability distribution and k
```

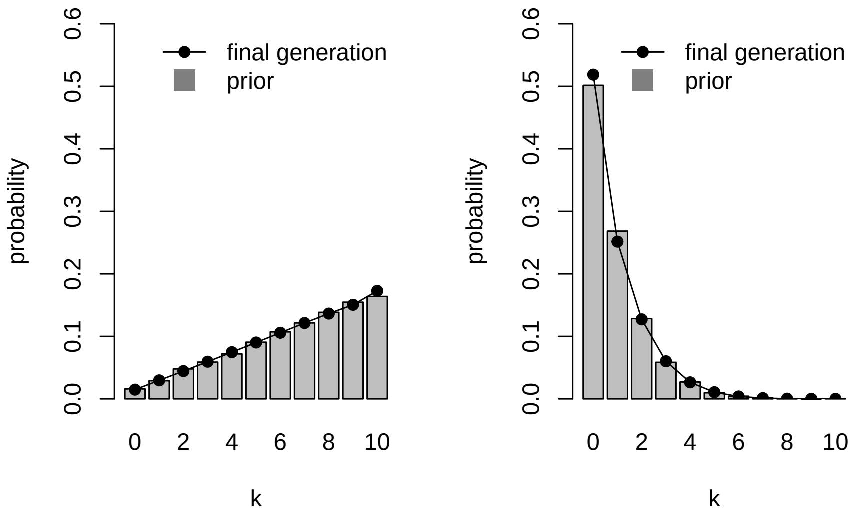
```
      k_vector[i] <- k_vector[i] + next_gen$k
      k_matrix[i,] <- k_matrix[i,] + next_gen$prob_k

    }

  }

  # average across all runs
  k_vector <- k_vector / r_max
  k_matrix <- k_matrix / r_max

  # export k_vector, k_matrix and parameters
  list(k_vector = k_vector,
       k_matrix = k_matrix,
       parameters = c(n = n,
                      k = k_initial,
                      alpha1 = alpha1,
                      beta1 = beta1,
                      alpha2 = alpha2,
                      beta2 = beta2,
                      h = h,
                      sampling = sampling,
                      N = N,
                      r_max = r_max))

}
```

We also update **IteratedFinalPlot** to calculate the prior bars using the additional set of prior parameters:

```
IteratedFinalPlot <- function(data_model16, ymax = 0.6) {

  # retrieve parameters
  n <- data_model16$parameters["n"]
  r_max <- data_model16$parameters["r_max"]
  alpha1 <- data_model16$parameters["alpha1"]
  beta1 <- data_model16$parameters["beta1"]
  alpha2 <- data_model16$parameters["alpha2"]
  beta2 <- data_model16$parameters["beta2"]
  h <- data_model16$parameters["h"]
  k_matrix <- data_model16$k_matrix

  # simulate prior distribution
  prior_dist <- rep(NA, r_max)
  h_prob <- runif(r_max)
  for (i in 1:r_max) {

    if (h_prob[i] < h) {

      theta <- rbeta(1, alpha1, beta1)
      prior_dist[i] <- rbinom(1, n, theta)

    } else {

      theta <- rbeta(1, alpha2, beta2)
      prior_dist[i] <- rbinom(1, n, theta)

    }

  }

  prior_dist <- table(factor(prior_dist, levels = 0:n))
  prior_dist <- prior_dist / sum(prior_dist)

  # draw barplot for simulated priors
  b <- barplot(height = prior_dist,
               names = 0:(length(prior_dist)-1),
               xlab = "k",
               ylab = "probability",
               ylim = c(0,ymax))

  # add line for the final generation iterated learning chain
  lines(k_matrix[nrow(k_matrix),],
        x = b,
        pch = 19,
        type = 'o')

  # add legend
  legend("topright",
         legend=c("final generation","prior"),
         pch = c(19,15),
         col=c("black","grey50"),
         pt.cex=c(1,2),
         bty="n",
         lty=c(1,NA),
         lwd=c(1,1))

}
```

Navarro et al. (2018) wanted to simulate a population of agents with priors that favour different values of k and priors that differ in strength. Specifically, where some agents have weak priors for large values of k ($\alpha_1 = 2, \beta_1 = 1$), and some agents have strong priors for small values of k ($\alpha_2 = 1, \beta_2 = 10$). First let's visualise these two sets of priors in homogenous populations, before simulating a heterogenous population. The following code first runs a simulation where all agents have weak priors for large values of k, then a simulation where all agents have strong priors for small values of k, plotting each side-by-side.

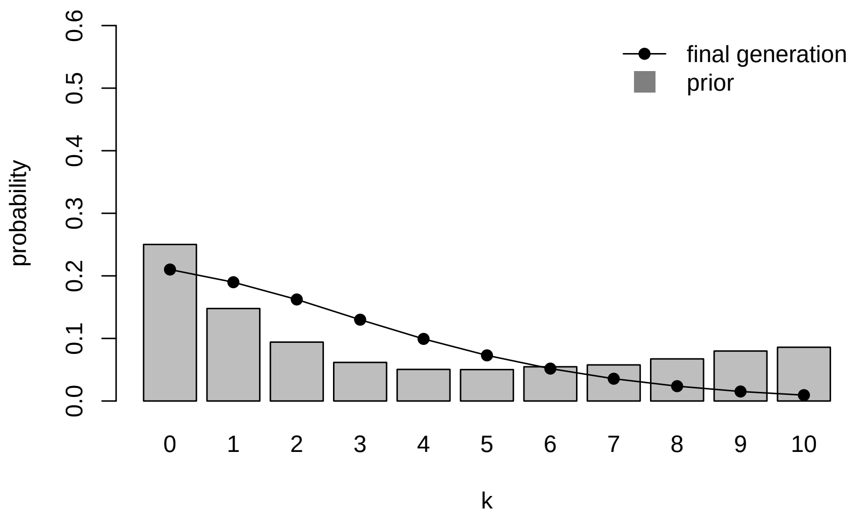
```
par(mfrow=c(1,2))

IteratedFinalPlot(MixedIteratedLearning(alpha1 = 2,
                                        beta1 = 1))

IteratedFinalPlot(MixedIteratedLearning(alpha1 = 1,
                                        beta1 = 10))
```



You can see how the left-hand prior distribution favours $k = 10$ and the right-hand prior distribution favours $k = 0$, and that the latter represents much more certainty in $k = 0$ (about $p = 0.5$) than the former has in $k = 10$ (about $p = 0.175$). As before, the final generation distributions almost exactly match the priors.

Now we run a mixed population where half ($h = 0.5$, the default) of the agents have weak priors and half have strong priors:

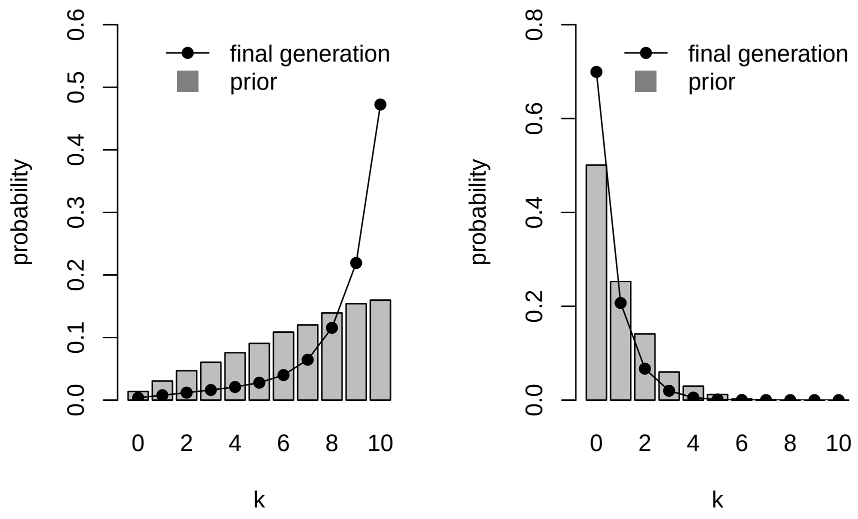
```
par(mfrow=c(1,1))

IteratedFinalPlot(MixedIteratedLearning(alpha1 = 2,
                                        beta1 = 1,
                                        alpha2 = 1,
                                        beta2 = 10))
```



Here we have recreated Navarro et al.'s (2018) Fig 3. Unlike homogenous populations, heterogenous populations do not converge on the average of the different priors (the grey bars). They also don't converge on either of the two prior distributions of the agents (shown in the previous two plots). Extreme values ($k = 0$ and $k > 6$) are under-represented, while low non-zero values are over-represented. This is because the (1,10) priors favouring low values are stronger than the weak (2,1) priors favouring high values. The greater certainty of the former exerts a disproportionate influence on the iterated learning chain.

Another case where iterated learning chains do not converge exactly on the priors is when MAP is used instead of sampling. Recall that MAP selects the value of θ with the maximum posterior probability, while sampling picks θ randomly in proportion to the posterior. This can be seen with the weak and strong homogenous priors we just simulated, changing *sampling* to FALSE.

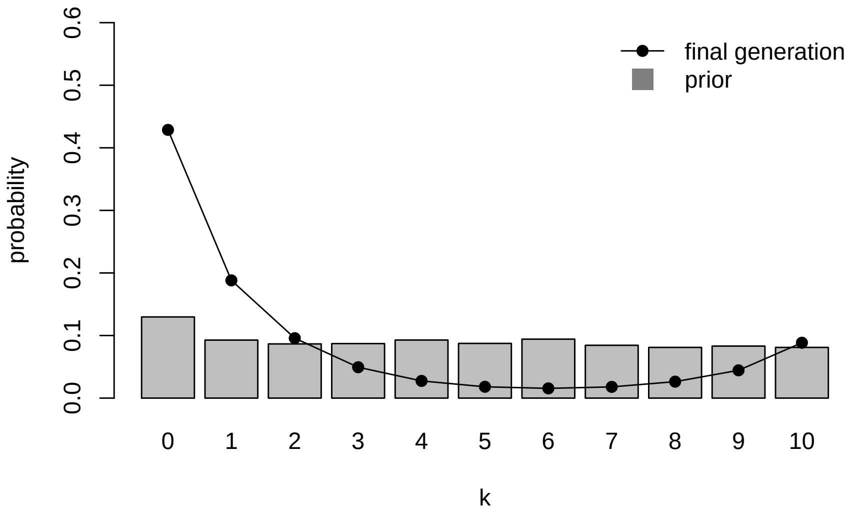
```
par(mfrow=c(1,2))

IteratedFinalPlot(MixedIteratedLearning(alpha1 = 2,
                                           beta1 = 1,
                                           sampling = FALSE))

IteratedFinalPlot(MixedIteratedLearning(alpha1 = 1,
                                        beta1 = 10,
                                        sampling = FALSE),
                  ymax = 0.8)
```



MAP exaggerates the prior distribution. For the weak priors favouring large values of k (the left hand plot), $k = 9$ and $k = 10$ are greatly over-represented in the final generation compared to the relatively smooth priors. For the strong priors favouring small values of k (the right hand plot), $k = 0$ is over-represented. This tendency for MAP to exaggerate the prior distribution has been reported in previous models (e.g. Kirby et al. 2007).

Finally, we can combine the previous two cases to simulate a mixed population of MAP agents where the vast majority (95%, i.e. $h = 0.95$) are unbiased with flat priors ($\alpha_1 = \beta_1 = 1$), and a small minority (5%) are extremely biased towards one of the variants ($\alpha_2 = 1, \beta_2 = 100$):

```
IteratedFinalPlot(MixedIteratedLearning(alpha1 = 1,
                                        beta1 = 1,
                                        alpha2 = 1,
                                        beta2 = 100,
                                        h = 0.95,
                                        sampling = FALSE))
```



Here the final generation distribution looks very different to the almost flat priors. The small minority who have extreme beliefs in one variant skew the unbiased majority towards that variant.

Summary

Model 16 combined Bayesian inference with iterated learning to explore the cultural evolution of linguistic regularisation. Bayesian inference provides a way of integrating the likelihood of some data with prior expectations to generate a posterior probability distribution over all possible hypotheses. Iterated learning places Bayesian learners in a chain, with each learner producing data for the next participant from their posterior distribution. While this was discussed in terms of regularisation in language, this general approach has been used to study the cultural evolution of a range of linguistic and cognitive features and offers an alternative to the population-genetics-inspired models of previous chapters.

The key result of Model 16 is that chains typically converge on individuals' priors irrespective of the starting data, as concluded by Reali & Griffiths (2009).

This is not obvious from a single learning episode, and only occurs after several generations of repeated learning and cultural transmission. This suggests that if in the real world we see languages that are mostly regular, then people possess priors that reflect this. In the model, this would be an α less than one. Indeed, Reali & Griffiths (2009) ran an iterated learning experiment resembling this model and found that participants' choices were consistent with them having $\alpha = 0.052$, a strong regularisation prior. In a similar experiment, Ferdinand et al. (2013) found $\alpha = 0.605$, not as extreme but still less than one, so again indicating a regularisation bias.

Where do these priors come from? It is possible that they are innate (i.e. genetically specified), supporting nativist approaches to language evolution commonly identified with Noam Chomsky. However, this is not necessarily the case: priors could also be the result of how human cognition or learning works in general, without being specific to language. One might imagine that regular grammatical rules are easier to learn and remember than irregular forms. Priors might also be themselves culturally acquired, which can be captured by hierarchical Bayesian models (Kemp et al. 2007).

This Bayesian iterated learning approach has been more generally applied than just regularisation (e.g. to compositionality: Kirby et al. 2007; Smith & Kirby 2008), and just language. Any culturally transmitted cognitive representation can be modelled in this way, from object categorisation to color terminologies (e.g. Griffiths et al. 2008). Can we apply Reali & Griffiths' (2009) conclusion that cultural forms always reflect individual priors to all of human culture? This is certainly consistent with 'cultural attraction' approaches to cultural evolution (Claidière & Sperber 2007), which holds that stable patterns of cultural variation can be explained in terms of individual-level cognitive biases.

However, we should remember that there were two cases where cultural evolution did not converge on the priors in Model 16. When agents had different priors, iterated learning did not converge on any single agent's priors, nor on the average of all agents' priors (Navarro et al. 2018). In the final case we considered, a small minority with extreme priors disproportionately influenced cultural dynamics when the rest of the population had flat priors. Such a case does not seem unrealistic in an age when a small number of people with extreme opinions are highly visible on social media and may disproportionately influence the silent majority.

The second case was when agents chose the 'maximum a posteriori' (MAP) hypothesis from the posterior rather than sampled randomly from the posterior. Iterated learning with MAP exaggerates the prior distribution. This result has been used to argue that the distinctive patterns of cultural variation we see in the real world can be generated by very weak biases at the individual level, given that MAP magnifies these biases (Kirby et al. 2007). Do real people use sampling or MAP? The evidence is mixed. Reali & Griffiths (2009) found that their participants better resembled samplers, and Bonawitz et al. (2014) summarise evidence for sampling in children, whereas Smith & Kirby (2008) show that MAP is more evolutionarily stable than sampling.

We should also remember that there is no selection or cultural accumulation in these models. In real life, cognitively intuitive ideas and practices such as miasma theory or creationism have been replaced with more accurate but less intuitive concepts such as germ theory or evolutionary theory. While language may be a special case, there are likely many other realms of cultural evolution where cultural forms do not resemble people's priors due to cultural selection (see Mesoudi 2021).

Model 16 introduced a few new tools to our modelling toolkit. We made heavy use of statistical distributions (the binomial and the beta), we saw how to implement Bayes' rule, and how to use conjugate priors to speed up code. We used the **heatmap** function to visualise the data probability distribution over time, and **barplot** to plot the priors. Finally, unlike previous simulation functions which incorporated plotting code, we created two separate plot functions that took the function output as input. A handy trick is to include the simulation function parameters in the output, when they are needed by the plotting functions.

Acknowledgements

For this model I am grateful to Simon Kirby for sharing his Masters course code, and Danielle Navarro for releasing the R code for Navarro et al. (2018) upon which Model 16 is based.

Exercises

1. Create a function that, for given values of n, k and α, uses **BayesianInference** to generate three plots one above the other, one showing the likelihood, one showing the prior, and one showing the posterior. Try different values of n, k and α to explore how priors and likelihoods combine to generate posterior distributions.

2. Try different values of α in **IteratedLearning** to confirm that $\alpha < 1$ generates regularisation and $\alpha > 1$ generates irregularisation. Also try setting β to different values than α to see whether iterated learning chains converge on asymmetrical priors.

3. Simulate a population that is highly polarised in their prior opinions, with 50% possessing $\alpha_1 = 1, \beta_1 = 100$ priors and 50% possessing $\alpha_2 = 100, \beta_2 = 1$ priors. Do the chains reach a consensus or do they remain as polarised as the priors? Does this differ from the patterns observed in Model 10 (Polarisation)? If so, why?

4. Rather than h being the probability of a single agent using the first set of priors, make h the probability of an entire chain using the first set of priors. How does this change the cultural dynamics of the iterated learning chains?

5. Another extension that potentially prevents convergence on the prior is having multiple individuals in each generation of the iterated learning chain (Ferdinand & Zuidema 2009). Create a new simulation function **MultiAgentIteratedLearning** which contains g agents per generation. Each agent generates n utterances, and the subsequent g agents in the next generation use all gn utterances as their input. Do chains with $g > 1$ recapitulate the priors?

References

Bonawitz, E., Denison, S., Griffiths, T. L., & Gopnik, A. (2014). Probabilistic models, learning algorithms, and response variability: sampling in cognitive development. Trends in cognitive sciences, 18(10), 497-500.

Claidière, N., & Sperber, D. (2007). The role of attraction in cultural evolution. Journal of Cognition and Culture, 7(1-2), 89-111.

Ferdinand, V., & Zuidema, W. (2009). Thomas' theorem meets Bayes' rule: A model of the iterated learning of language. In Proceedings of the Annual Meeting of the Cognitive Science Society (Vol. 31, No. 31).

Ferdinand, V., Kirby, S., & Smith, K. (2014). Regularization in language evolution: On the joint contribution of domain-specific biases and domain-general frequency learning. In Evolution of Language: Proceedings of the 10th International Conference (EVOLANG10) (pp. 435-436).

Griffiths, T. L., Christian, B. R., & Kalish, M. L. (2008). Using category structures to test iterated learning as a method for identifying inductive biases. Cognitive Science, 32(1), 68-107.

Kemp, C., Perfors, A. & Tenenbaum, J. B. (2007). Learning overhypotheses with hierarchical Bayesian models. Developmental Science, 10, 307–321.

Kirby, S., Dowman, M., & Griffiths, T. L. (2007). Innateness and culture in the evolution of language. Proceedings of the National Academy of Sciences, 104(12), 5241-5245.

Mesoudi, A. (2021). Cultural selection and biased transformation: two dynamics of cultural evolution. Philosophical Transactions of the Royal Society B, 376(1828), 20200053.

Navarro, D. J., Perfors, A., Kary, A., Brown, S. D., & Donkin, C. (2018). When extremists win: Cultural transmission via iterated learning when populations are heterogeneous. Cognitive Science, 42(7), 2108-2149.

Perfors, A., Tenenbaum, J. B., Griffiths, T. L., & Xu, F. (2011). A tutorial introduction to Bayesian models of cognitive development. Cognition, 120(3), 302-321.

Reali, F., & Griffiths, T. L. (2009). The evolution of frequency distributions: Relating regularization to inductive biases through iterated learning. Cognition, 111(3), 317-328.

Smith, K., & Kirby, S. (2008). Cultural evolution: implications for understanding the human language faculty and its evolution. Philosophical Transactions of the Royal Society B, 363(1509), 3591-3603.

Model 17: Reinforcement learning

Introduction

It's fair to say that the field of cultural evolution focuses far more on social learning (learning from others) than individual learning (learning on one's own). While we have modelled several different types of social learning (direct bias, indirect bias, conformity, vertical transmission etc.), individual learning has taken relatively simple forms, such as the cultural mutation of Model 2. Here individuals invent a new cultural trait or switch to a different cultural trait with a certain probability. This is a reasonable starting point but can be much improved to give a more psychologically informed model of individual learning. After all, without new variation introduced by individual learning, cultural evolution cannot occur, so individual learning deserves more attention.

Model 17 will implement reinforcement learning, a popular method of modelling individual learning across several disciplines. Reinforcement learning has its roots in psychology, where early behaviourists like Edward Thorndyke, Ivan Pavlov and BF Skinner argued that behaviours which get rewarded (or 'reinforced') increase in frequency, while behaviours that are unrewarded are performed less often. More recently, reinforcement learning has become popular in machine learning and artificial intelligence (Sutton & Barto 2018). In these fields reinforcement learning is defined as learning from the environment what actions or choices maximise some numerical reward. This reward might be points in a game, money in a transaction, even happiness if we are willing to assign it a numerical value. Researchers studying reinforcement learning devise algorithms - models, essentially - that can achieve this maximisation.

In the field of cultural evolution, reinforcement learning models have been used in lab experiments to model participants' choices in decision-making tasks, combined with the social learning biases that we have already come across (e.g. McElreath et al. 2008; Barrett et al. 2017; Toyokawa et al. 2019; Deffner et al. 2020). In Model 17 we will recreate simple versions of these studies' models to see how to simulate individual learning as reinforcement learning and how to combine it with social learning.

Also, in a 'Statistical Appendix' we will see how to retrieve parameters of our reinforcement-plus-social-learning models from simulation data. In other words, we run our agent-based simulation of reinforcement and social learning with various known parameters, such as a probability $s = 0.3$ that an agent engages in social learning rather than reinforcement learning. This gives us our familiar *output* dataframe containing all the traits of all agents over all trials. We then feed this *output* dataframe into the statistical modelling platform Stan, without the original parameter values, to try to retrieve those parameter values. In the example above we would hope to obtain $s = 0.3$ or thereabouts, rather than $s = 0.1$ or $s = 0.9$. This might seem somewhat pointless with simulated data when we already know what the parameters were. But in experiments and analyses of real world data we won't know in advance what real people's tendencies to engage in social learning are, or their propensity for conformity, etc. By testing statistical analyses on agent-based simulation data, we can confirm that our statistical methods adequately estimate such parameters. This is another valuable use for agent-based models.

Model 17a: Reinforcement learning with a single agent

Before simulating multiple agents who can potentially learn from one another, we start with a single agent. This is all we need for individual learning and will help to better understand how reinforcement learning works.

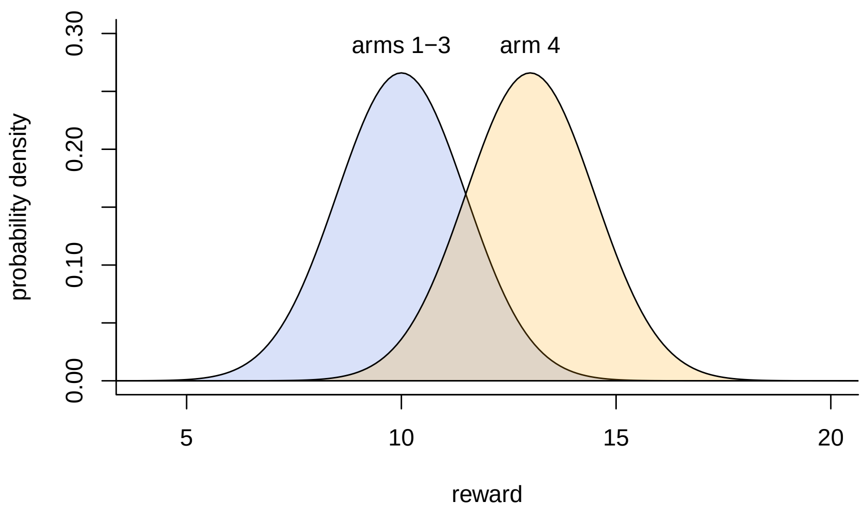
In models of reinforcement learning agents typically face what is known as a multi-armed or k-armed bandit task. A 'one-armed bandit' is a slot machine like you would find in a casino. You pull the lever and hope to get a reward. A k-armed bandit is a set of k of these slot machines. On each trial the agent pulls one of the k arms and gets a reward. While all arms give rewards, they differ in the average size of the reward. However, the agent doesn't know these average rewards in advance. The agent's task is therefore to learn which of the k arms gives the highest average reward by pulling the arms and observing the rewards.

To make things difficult, there is random noise in the exact size of the reward received on each trial. The agent can't just try each arm and immediately know which is best, they must learn which is best over successive trials each of which gives noisy data.

We can implement this by assuming that each arm's reward is drawn randomly from a normal distribution with a different mean for each arm and a common standard deviation σ. We will use a simplified version of the reward environment implemented in an experiment by Deffner et al. (2020). They had $k = 4$ arms.

One arm gave a higher mean reward than the other three, which all had the same mean reward. We assume that the optimal arm (arm 4) has a mean reward of 13 points, and the others (arms 1-3) have a mean reward of 10 points. All arms have a standard deviation $\sigma = 1.5$. Let's plot these reward distributions:

```
# define parameters of the 4-armed bandit task
arm_means <- c(10,10,10,13)
arm_sd <- 1.5

# empty plot
plot(NA,
     ylab = "probability density",
     xlab = "reward",
     type = 'n',
     bty='l',
     xlim = c(4, 20),
     ylim = c(0, 0.3))

# plot each arm's normally distributed reward
x <- seq(-10, 100, 0.1)
polygon(x, dnorm(x, arm_means[1], arm_sd), col = adjustcolor("royalblue", alpha.f = 0.2))
text(10, 0.29, "arms 1-3")
polygon(x, dnorm(x, arm_means[4], arm_sd), col = adjustcolor("orange", alpha.f = 0.2))
text(13, 0.29, "arm 4")
```



While arm 4 gives a higher reward on average, there is overlap with the other three arms' reward distributions. This is made more concrete by picking 10 values from each arm's reward distribution:

```
m <- matrix(NA, nrow = 4, ncol = 10,
            dimnames = list(c("arm1","arm2","arm3","arm4"), NULL))

for (i in 1:4) m[i,] <- round(rnorm(10, arm_means[i], arm_sd))

m
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## arm1   10   10   11   11   10    8   11   11   13     9
## arm2    9    9    8    9    8    8    8   10    7     9
## arm3    9    9    7   11   10    8   10   13   12    10
## arm4   15   11   14   12   11   16   15   13   13    12
```

Think of each of the 10 columns as an agent trying all four arms once to see which gives the highest reward. While most of the 10 columns will show that arm 4 gives a higher reward than the other arms, some won't. This is the challenge that the agent faces. Over successive trials, they must maximise rewards by picking from the arm that gives the highest mean reward without knowing which arm does this in advance, and with noisy feedback from the environment. (NB the parameter σ controls this uncertainty; try increasing it to $\sigma = 3$ and rerunning the two code chunks above, to see how the task becomes even more difficult.)

A fundamental problem which arises in tasks such as the k-armed bandit is an exploitation vs exploration tradeoff. Imagine an agent picks each of the above four arms once on four successive trials. There is a chance that arm 4 gives the highest reward, but it's far from certain (in how many of the 10 columns in the rewards output above does arm 4 give the single highest reward?). Imagine instead that arm 2 gives the highest reward. A highly exploitative agent will proceed to pick ('exploit') arm 2 until the end of the simulation, believing (falsely) that arm 2 maximises its rewards. At the other extreme, a highly exploratory agent will continue to try all of the other arms to gain as much information about the reward distributions as possible. Both agents lose out: the exploitative agent is losing out on the rewards it would have got if it had explored more and discovered that arm 4 is actually the best, while the exploratory agent is losing out because it never settles on arm 4, continuing to pick arms 1-3 until the end.

What is needed is an algorithm that balances exploitation and exploration to explore enough to find the best option, but not too much so that the best option is exploited as much as possible. That's what we'll implement in Model 17a.

First we need to numerically represent the agent's estimate of the value of each arm. These estimates are called Q values (or sometimes 'attractions', denoted

A). The agent has one Q value for each arm. Q values can take any value - fractions, zeroes, negative numbers, etc. - as long as they numerically specify the relative value of each action.

We create a variable *Q_values* to hold the agent's Q values for each arm on the current trial:

```r
# for storing Q values for 4 arms on current trial
Q_values <- rep(0, 4)
names(Q_values) <- paste("arm", 1:4, sep="")

Q_values
```

```
## arm1 arm2 arm3 arm4
##    0    0    0    0
```

On the first trial ($t = 1$) we'll set each Q value to zero, as shown above. This indicates that the agent has no initial preference for any of the arms. If we had prior knowledge about the arms we might set different values, but for now equal preferences is a reasonable assumption.

Now we convert these values into probabilities, specifying the probability that the agent chooses each arm on the next trial. This is where the exploitation-exploration trade-off is addressed. We use what is called a 'softmax' function, which converts any set of values into probabilities that sum to one (as probabilities must) and weights the probabilities according to those values. The higher the relative value, the higher the relative probability.

The softmax function contains a parameter β, often called 'inverse temperature', that determines how exploitative the agent is (sometimes expressed as how 'greedy' the agent is). When $\beta = 0$, arms are chosen entirely at random with no influence of the Q values. This is super exploratory, as the agent continues to choose all arms irrespective of observed rewards. As β increases, there is a higher probability of picking the arm with the highest Q value. This is increasingly exploitative (or 'greedy'). When β is very large then only the arm that currently has the highest Q value will be chosen, even if other arms might actually be better.

Mathematically, the softmax function looks like this:

$$p_i = \frac{e^{\beta Q_i}}{\sum_{j=1}^{k} e^{\beta Q_j}} \tag{17.1}$$

On the left hand side p_i means the probability of choosing arm i. This is equal to the exponential of the product of the parameter β and the Q value of arm i, divided by the sum of this value for all arms (Q_j for $j = 1$ to $j = k$, where $k = 4$ in our case).

Some code will help to understand this equation. The softmax function can easily be written in R as follows, defining $\beta = 0$ for now:

```
beta <- 0

exp(beta * Q_values) / sum(exp(beta * Q_values))
```

```
## arm1 arm2 arm3 arm4
## 0.25 0.25 0.25 0.25
```

Given that our initial Q values are all zero, each arm has an equal probability of being chosen. β is irrelevant here as no arm has the highest reward yet. Let's try a different set of Q values, representing a hypothetical set of four trials, one per arm, like one of the 10 columns in the matrix above:

```
Q_values[1:4] <- c(10,14,9,13)

exp(beta * Q_values) / sum(exp(beta * Q_values))
```

```
## arm1 arm2 arm3 arm4
## 0.25 0.25 0.25 0.25
```

Because $\beta = 0$, the new Q values make no difference. The agent still picks each arm with equal probability. Now we increase β to 0.3 (rounding the display to 2 decimal places for ease of reading using the **round** function):

```
beta <- 0.3

round(exp(beta * Q_values) / sum(exp(beta * Q_values)), 2)
```

```
## arm1 arm2 arm3 arm4
## 0.13 0.44 0.10 0.33
```

Now arm 2 is more likely to be chosen, given that it has the highest reward. But it's still possible that the other arms are chosen, particularly arm 4 which has an only slightly lower reward than arm 2. This seems like a good value of β to balance exploitation and exploration: the currently highest valued arm is most likely to be chosen, but there is a chance of exploring other arms, particularly other relatively high-scoring arms.

To check this, we increase β to a much higher value:

```
beta <- 5

round(exp(beta * Q_values) / sum(exp(beta * Q_values)), 2)
```

```
## arm1 arm2 arm3 arm4
## 0.00 0.99 0.00 0.01
```

It's now almost certain that arm 2 is chosen. This β value is probably too large. It will greedily exploit arm 2 without ever exploring the other arms, and never discover that arm 4 is better.

For now we'll revert to our initial all-equal Q values and a reasonable $\beta = 0.3$. Our agent stores the softmax probabilities in *probs* then uses this in **sample** to choose one of the four arms. This choice is stored in *choice*.

```
Q_values[1:4] <- rep(0,4)
beta <- 0.3

probs <- exp(beta * Q_values) / sum(exp(beta * Q_values))

choice <- sample(1:4, 1, prob = probs)
choice
```

```
## [1] 2
```

The arm number above will vary on different simulations, given that it is essentially a random choice. Next the agent pulls the chosen arm and gets a *reward*, using the command **rnorm** to pick a random number from one of the normal distributions specified above. Note that we **round** the reward to the nearest integer to avoid unwieldy decimals.

```
reward <- round(rnorm(1, mean = arm_means[choice], sd = arm_sd))
reward
```

```
## [1] 11
```

This reward will also vary on different simulations, as it's a random draw from a normal distribution with a randomly selected mean.

Finally, the agent updates its Q values given the reward. The extent to which Q values are updated is controlled by a second parameter, α, which ranges from 0 to 1. When $\alpha = 0$ there is no updating and the reward has no effect on Q values. When $\alpha = 1$ the Q value for the rewarded arm increases by the full reward amount. When $0 < \alpha < 1$ the Q value for the rewarded arm increases by a proportion α of the reward. There is again a trade-off here. Obviously $\alpha = 0$ is no good because the agent is not learning. However, if α is too large then the agent can prematurely focus on an arm that gives a high reward early on, without exploring other arms.

The following code updates the Q values according to $\alpha = 0.7$ and the *reward*:

```
alpha <- 0.7

Q_values[choice] <- Q_values[choice] + alpha * (reward - Q_values[choice])

Q_values
```

```
## arm1 arm2 arm3 arm4
##  0.0  7.7  0.0  0.0
```

You can see that the chosen arm's Q value has increased by α times the *reward* shown above. The other arms remain zero.

We then repeat these steps (get probabilities, make choice, get reward, update Q values) over each of $t_{max} = 100$ timesteps. But let's add a final twist. Every 25 timesteps we will change which arm gives the higher reward. This lets us see whether our reinforcement algorithm can handle environmental change, rather than blindly sticking with arm 4 even after it is no longer optimal.

The following function **RL_single** does all this, using a dataframe *output* to store the Q values, choice and reward for each trial. For simplicity, we hard code the number of arms (k), the arm means and standard deviations, and the number of timesteps (t_{max}) rather than making them user-defined variables.

```
RL_single <- function(alpha, beta) {

  # set up arm rewards
  arm_means <- data.frame(p1 = c(10,10,10,13),
                          p2 = c(10,13,10,10),
                          p3 = c(13,10,10,10),
                          p4 = c(10,10,13,10))
  arm_sd <- 1.5

  # for storing Q values for 4 arms on current trial, initially all zero
  Q_values <- rep(0, 4)

  # for storing Q values, choices, rewards per trial
  output <- as.data.frame(matrix(NA, 100, 6))
  names(output) <- c(paste("arm", 1:4, sep=""), "choice", "reward")
  # t-loop
  for (t in 1:100) {

    # get softmax probabilities from Q_values and beta
    probs <- exp(beta * Q_values) / sum(exp(beta * Q_values))

    # choose an arm based on probs
    choice <- sample(1:4, 1, prob = probs)

    # get reward, given current time period
    if (t <= 25) reward <- round(rnorm(1, mean = arm_means$p1[choice], sd = arm_sd))
    if (t > 25 & t <= 50) reward <- round(rnorm(1, mean = arm_means$p2[choice], sd = arm_sd))
    if (t > 50 & t <= 75) reward <- round(rnorm(1, mean = arm_means$p3[choice], sd = arm_sd))
    if (t > 75) reward <- round(rnorm(1, mean = arm_means$p4[choice], sd = arm_sd))

    # update Q_values for choice based on reward and alpha
    Q_values[choice] <- Q_values[choice] + alpha * (reward - Q_values[choice])

    # store all in output dataframe
    output[t,1:4] <- Q_values
    output$choice[t] <- choice
    output$reward[t] <- reward

  }

  # record whether correct
  output$correct <- output$choice == c(rep(4,25),rep(2,25),rep(1,25),rep(3,25))

  # export output dataframe
  output

}
```

The core code regarding Q values, probabilities and choices is as it was earlier. However, instead of a single set of *arm_rewards*, we now create a dataframe with the different rewards for the four time periods (p1 to p4). These are then used later to calculate the reward given the current time period. The *output* dataframe records each Q value, choice and reward for each timestep. After the t-loop, a variable is created within *output* called *correct* which records whether the agent picked the optimal arm for that timestep.

Here is one run of **RL_single** with the parameter values from above, showing the first period of 25 timesteps when arm 4 is optimal:

```
data_model17a <- RL_single(alpha = 0.7, beta = 0.3)

data_model17a[1:25,]
```

```
##    arm1  arm2      arm3 arm4 choice reward correct
## 1     0  0.00  7.700000    0      3     11   FALSE
## 2     0  0.00  7.210000    0      3      7   FALSE
## 3     0  0.00  7.763000    0      3      8   FALSE
## 4     0  0.00  7.928900    0      3      8   FALSE
## 5     0  0.00  9.378670    0      3     10   FALSE
## 6     0  0.00 10.513601    0      3     11   FALSE
## 7     0  0.00 10.854080    0      3     11   FALSE
## 8     0  0.00 11.656224    0      3     12   FALSE
## 9     0  0.00 10.496867    0      3     10   FALSE
## 10    0  0.00 11.549060    0      3     12   FALSE
## 11    0  0.00  9.764718    0      3      9   FALSE
## 12    0  0.00  9.929415    0      3     10   FALSE
## 13    0  0.00  9.278825    0      3      9   FALSE
## 14    0  0.00 10.483647    0      3     11   FALSE
## 15    0  0.00  8.045094    0      3      7   FALSE
## 16    0  0.00  9.413528    0      3     10   FALSE
## 17    0  0.00  9.824058    0      3     10   FALSE
## 18    0  0.00  9.947218    0      3     10   FALSE
## 19    0  8.40  9.947218    0      2     12   FALSE
## 20    0  8.40  9.284165    0      3      9   FALSE
## 21    0  8.40  9.085250    0      3      9   FALSE
## 22    0  8.40  9.725575    0      3     10   FALSE
## 23    0  8.40 12.017672    0      3     13   FALSE
## 24    0  8.40 10.605302    0      3     10   FALSE
## 25    0 10.22 10.605302    0      2     11   FALSE
```

Each run will be different, but you can see the Q values gradually updating over these 25 timesteps. Perhaps arm 4 ends the period with the highest Q value. Perhaps not. Possibly not all arms got explored. The task is difficult and this learning algorithm is not perfect. But some learning is occurring. What about the final 25 timesteps, where arm 3 is optimal?

```
data_model17a[76:100,]
```

```
##         arm1      arm2      arm3     arm4 choice reward correct
## 76  10.96900 10.422928  9.283552 9.304657      1     10   FALSE
## 77  10.96900  9.426878  9.283552 9.304657      2      9   FALSE
## 78  10.96900  9.128063  9.283552 9.304657      2      9   FALSE
## 79  10.96900  9.128063 11.185066 9.304657      3     12    TRUE
## 80   8.19070  9.128063 11.185066  9.304657      1      7   FALSE
## 81   8.19070  9.128063 11.055520  9.304657      3     11    TRUE
## 82   8.19070  9.128063 11.716656  9.304657      3     12    TRUE
## 83   8.19070  9.128063 14.714997  9.304657      3     16    TRUE
## 84   8.19070  9.128063 12.814499  9.304657      3     12    TRUE
## 85   8.19070 10.438419 12.814499  9.304657      2     11   FALSE
## 86   8.19070 10.831526 12.814499  9.304657      2     11   FALSE
## 87   8.19070 10.831526 12.244350  9.304657      3     12    TRUE
## 88  10.15721 10.831526 12.244350  9.304657      1     11   FALSE
## 89  10.15721 10.831526 14.173305  9.304657      3     15    TRUE
## 90  10.15721  8.849458 14.173305  9.304657      2      8   FALSE
## 91  10.15721  8.849458 12.651991  9.304657      3     12    TRUE
## 92  10.15721  8.849458 12.651991 10.491397      4     11   FALSE
## 93  10.15721  8.849458 14.995597 10.491397      3     16    TRUE
## 94  10.15721  8.849458 12.198679 10.491397      3     11    TRUE
## 95  10.15721  8.954837 12.198679 10.491397      2      9   FALSE
## 96  10.15721  8.986451 12.198679 10.491397      2      9   FALSE
## 97  10.15721  9.695935 12.198679 10.491397      2     10   FALSE
## 98  10.15721 10.608781 12.198679 10.491397      2     11   FALSE
## 99  10.15721 10.608781 12.198679  8.747419      4      8   FALSE
## 100 10.15721 10.608781 12.759604  8.747419      3     13    TRUE
```

Again, each run will be different, but arm 3 should have the highest Q value or thereabouts. By the final period, the agent will have had a chance to explore all of the arms.

Model 17b: Reinforcement learning with multiple agents

It is relatively straightforward to add multiple agents to **RL_single**. This serves three purposes. First, we are effectively adding multiple independent runs to **RL_single** as we have done with previous models to better understand the stochasticity in our results. Second, this allows us in Model 17c to build in social learning between the multiple agents.

Third, we can introduce agent heterogeneity. In previous models we have assumed that all agents have the same global parameter values. For example, in Model 5 (conformity), all N agents have the same conformity parameter D in any one simulation. Here instead we will introduce heterogeneity (sometimes called 'individual variation' or 'individual differences') amongst the agents in parameter values α and β. This simulates a situation where agents differ in their reinforcement learning parameters. The latter might better resemble reality, where some people are more exploratory than others, some more conservative than others.

Each of N agents therefore has an α and β drawn from a normal distribution with mean α_μ and β_μ, and standard deviation α_σ and β_σ, respectively. When the standard deviations are zero, all agents have identical α and β, and we are doing independent runs with identical parameters for all agents. When the standard deviations are greater than zero, we are simulating heterogeneity amongst agents. Below is an example of agent heterogeneity. Note that we add some code to bound α between zero and one, and β greater than zero, otherwise our reinforcement learning algorithm won't work properly.

```
N <- 10

alpha_mu <- 0.7
alpha_sd <- 0.1

beta_mu <- 0.3
beta_sd <- 0.1

beta <- rnorm(N, beta_mu, beta_sd)  # inverse temperatures
alpha <- rnorm(N, alpha_mu, alpha_sd)  # learning rates
alpha[alpha < 0] <- 0  # ensure all alphas are >0
alpha[alpha > 1] <- 1  # ensure all alphas are <1
beta[beta < 0] <- 0  # ensure all betas are >0

data.frame(agent = 1:N, alpha, beta)
```

```
##    agent     alpha       beta
## 1      1 0.6917780 0.27420170
## 2      2 0.6627881 0.30600731
## 3      3 0.5996392 0.32566394
## 4      4 0.7854937 0.41019545
## 5      5 0.5685313 0.37557758
## 6      6 0.7709086 0.38936667
## 7      7 0.5862667 0.09629515
## 8      8 0.6886603 0.42488083
## 9      9 0.6566551 0.35926328
## 10    10 0.7456959 0.22728680
```

Now that we have N agents, *Q_values* must store the four Q values for each agent, not just one. We make it an N x 4 matrix, initialised with zeroes as before:

```
# for storing Q values for 4 arms on current trial, initially all zero
Q_values <- matrix(data = 0,
                   nrow = N,
                   ncol = 4,
                    dimnames = list(NULL, paste("arm", 1:4, sep="")))

Q_values
```

```
##       arm1 arm2 arm3 arm4
##  [1,]    0    0    0    0
##  [2,]    0    0    0    0
##  [3,]    0    0    0    0
##  [4,]    0    0    0    0
##  [5,]    0    0    0    0
##  [6,]    0    0    0    0
##  [7,]    0    0    0    0
##  [8,]    0    0    0    0
##  [9,]    0    0    0    0
## [10,]    0    0    0    0
```

Arms are represented along the columns as before, but now each row is a different agent.

Next we calculate the probabilities of each arm using the softmax function, this time for each agent. Note that for this and the next few commands, we use vectorised code to calculate *probs*, *choice*, *reward* and updated *Q_values* for all agents simultaneously, rather than looping over each agent. As always, vectorised code is much quicker than loops.

```
probs <- exp(beta * Q_values) / rowSums(exp(beta * Q_values))
probs
```

```
##       arm1 arm2 arm3 arm4
##  [1,] 0.25 0.25 0.25 0.25
##  [2,] 0.25 0.25 0.25 0.25
##  [3,] 0.25 0.25 0.25 0.25
##  [4,] 0.25 0.25 0.25 0.25
##  [5,] 0.25 0.25 0.25 0.25
##  [6,] 0.25 0.25 0.25 0.25
##  [7,] 0.25 0.25 0.25 0.25
##  [8,] 0.25 0.25 0.25 0.25
##  [9,] 0.25 0.25 0.25 0.25
## [10,] 0.25 0.25 0.25 0.25
```

As before, with initially all-zero Q values, each arm is equally likely for all agents. Note that unlike in Model 17a, *probs* is now divided by **rowSums**. This gives the sum of each agent's exponentiated and multiplied-by-β Q value which are stored on each row.

Next we choose an arm based on *probs*:

```
choice <- apply(probs, 1, function(probs) sample(1:4, 1, prob = probs))
choice
```

```
##  [1] 4 1 3 2 2 3 4 3 3 3
```

This returns *N* choices, one per agent. Here, *choice* is obtained using **apply**, a command which applies a function to each row (indicated with a '1'; columns would be '2') of the *probs* matrix. The function it applies is **sample** which picks an arm in proportion to that agent's *probs* as in Model 17a.

Next we get a *reward* from each *choice*:

```
reward <- round(rnorm(N, mean = arm_means[choice], sd = arm_sd))
reward
```

```
##  [1] 14 10  9 11 11 14 13 10  9  8
```

Note that *reward* is now *N* random draws from a normal distribution, one for each agent, rather than one draw for one agent like it was in Model 17a.

Finally we update *Q_values* based on these rewards. We do this one arm at a time, for each arm identifying which agents chose that arm using a variable *chosen*, then updating those Q values of the *chosen* agents' *arm* as before.

```
for (arm in 1:4) {

  chosen <- which(choice==arm)

  Q_values[chosen, arm] <- Q_values[chosen, arm] +
      alpha[chosen] * (reward[chosen] - Q_values[chosen, arm])

}

Q_values
```

```
##            arm1     arm2      arm3     arm4
##  [1,] 0.000000 0.000000  0.000000 9.684892
##  [2,] 6.627881 0.000000  0.000000 0.000000
##  [3,] 0.000000 0.000000  5.396752 0.000000
##  [4,] 0.000000 8.640431  0.000000 0.000000
##  [5,] 0.000000 6.253844  0.000000 0.000000
##  [6,] 0.000000 0.000000 10.792720 0.000000
##  [7,] 0.000000 0.000000  0.000000 7.621467
##  [8,] 0.000000 0.000000  6.886603 0.000000
##  [9,] 0.000000 0.000000  5.909896 0.000000
## [10,] 0.000000 0.000000  5.965567 0.000000
```

These updated values will all be different, as each agent has a different choice, reward and α.

Now we can put these steps into a t-loop as before and run over 100 trials, all within a function called **RL_multiple**. Note that we need to modify the *output* dataframe to hold the choices and rewards of all agents. We do this by adding an 'agent' column which indicates the agent id from 1 to N. We also add a 'trial' column now that there are multiple rows per trial. For brevity we omit Q values from *output*.

```
RL_multiple <- function(N, alpha_mu, alpha_sd, beta_mu, beta_sd) {

  # set up arm rewards
  arm_means <- data.frame(p1 = c(10,10,10,13),
                          p2 = c(10,13,10,10),
                          p3 = c(13,10,10,10),
                          p4 = c(10,10,13,10))
  arm_sd <- 1.5

  # draw agent beta and alpha from overall mean and sd
  beta <- rnorm(N, beta_mu, beta_sd)  # inverse temperatures
  alpha <- rnorm(N, alpha_mu, alpha_sd)  # learning rates
  alpha[alpha < 0] <- 0  # ensure all alphas are >0
  alpha[alpha > 1] <- 1  # ensure all alphas are <1
  beta[beta < 0] <- 0  # ensure all betas are >0

  # for storing Q values for k arms on current trial, initially all zero
  Q_values <- matrix(data = 0,
                     nrow = N,
                     ncol = 4)

  # for storing choices and rewards per agent per trial
  output <- data.frame(trial = rep(1:100, each = N),
                       agent = rep(1:N, 100),
                       choice = rep(NA, 100*N),
                       reward = rep(NA, 100*N))

  # t-loop
  for (t in 1:100) {

    # get softmax probabilities from Q_values and beta
    probs <- exp(beta * Q_values) / rowSums(exp(beta * Q_values))

    # choose an arm based on probs
    choice <- apply(probs, 1, function(probs) sample(1:4, 1, prob = probs))
    # get reward
    if (t <= 25) reward <- round(rnorm(N, mean = arm_means$p1[choice], sd = arm_sd))
    if (t > 25 & t <= 50) reward <- round(rnorm(N, mean = arm_means$p2[choice], sd = ar
    if (t > 50 & t <= 75) reward <- round(rnorm(N, mean = arm_means$p3[choice], sd = ar
    if (t > 75) reward <- round(rnorm(N, mean = arm_means$p4[choice], sd = arm_sd))

    # update Q values one arm at a time
    for (arm in 1:4) {

      chosen <- which(choice==arm)

      Q_values[chosen,arm] <- Q_values[chosen,arm] +
        alpha[chosen] * (reward[chosen] - Q_values[chosen,arm])

    }

    # store choice and reward in output
    output[output$trial == t,]$choice <- choice
    output[output$trial == t,]$reward <- reward

  }

  # record whether correct
  output$correct <- output$choice == c(rep(4,25*N),rep(2,25*N),rep(1,25*N),rep(3,25*N)

  # export output dataframe
  output

}
```

Here are the first 6 rows of *output* using the parameters from above, and $N = 200$:

```
data_model17b <- RL_multiple(N = 200,
                             alpha_mu = 0.7,
                             alpha_sd = 0.1,
                             beta_mu = 0.3,
                             beta_sd = 0.1)

head(data_model17b)
```

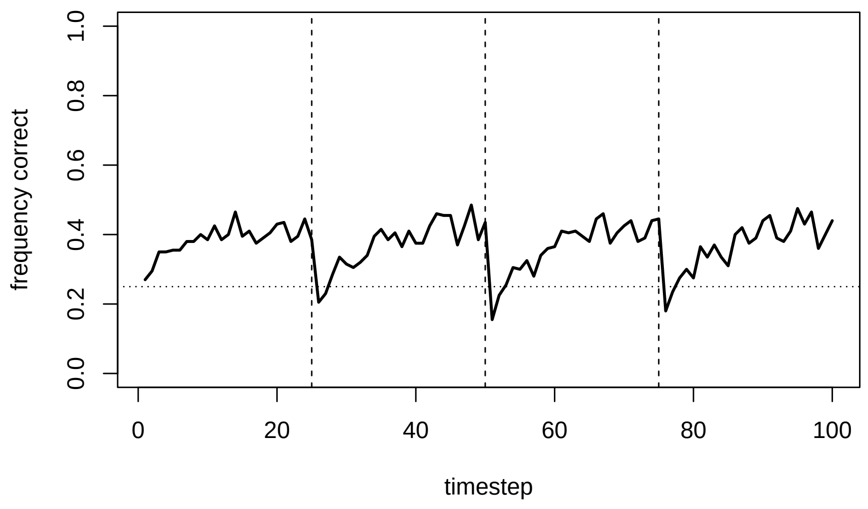
```
##   trial agent choice reward correct
## 1     1     1      4     13    TRUE
## 2     1     2      4     12    TRUE
## 3     1     3      1     11   FALSE
## 4      1      4       1      10    FALSE
## 5      1      5       3      12    FALSE
## 6      1      6       3      12    FALSE
```

As always, it's easier to understand the output by plotting it. We are most interested in whether the agents have learned the optimal arm during each period. The following function **plot_correct** plots the proportion of N agents who choose the optimal in each timestep. This is stored in the variable *plot_data* which simply takes the mean of *output$correct* which, given that correct choices are TRUE (which R reads as 1) and incorrect choices are FALSE (which R reads as 0), results in the proportion correct. We also add dotted vertical lines indicating the timesteps when optimal arm changes, and a dashed horizontal line indicating random chance (25%).

```r
plot_correct <- function(output) {

  plot_data <- rep(NA, 100)
  for (t in 1:100) plot_data[t] <- mean(output$correct[output$trial == t])

  plot(x = 1:100,
       y = plot_data,
       type = 'l',
       ylab = "frequency correct",
       xlab = "timestep",
       ylim = c(0,1),
       lwd = 2)

  # dotted vertical lines indicate changes in optimal
  abline(v = c(25,50,75),
         lty = 2)

  # dotted horizontal line indicates chance success rate
  abline(h = 0.25,
         lty = 3)

}
```

First we can plot $N = 200$ with no agent heterogeneity and the usual values of α and β. This allows us to effectively run multiple independent runs of the simulation above with **RL_single**.

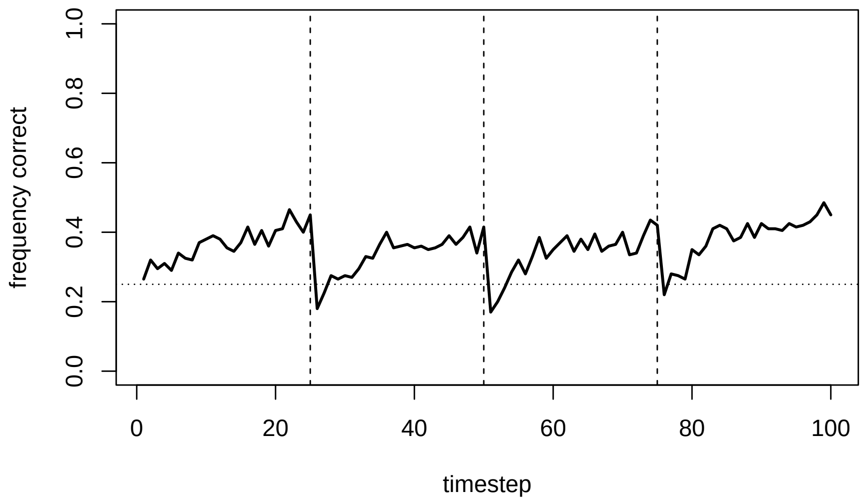
```r
data_model17b <- RL_multiple(N = 200,
                             alpha_mu = 0.7,
                             alpha_sd = 0,
                             beta_mu = 0.3,
                                    beta_sd = 0)

plot_correct(data_model17b)
```



In each time period, the frequency of correct arm choices increases from chance (25%) to around 40%. While not amazing, some agents at least are successfully learning the optimal arm. And they are doing this despite environmental change and noisy rewards.

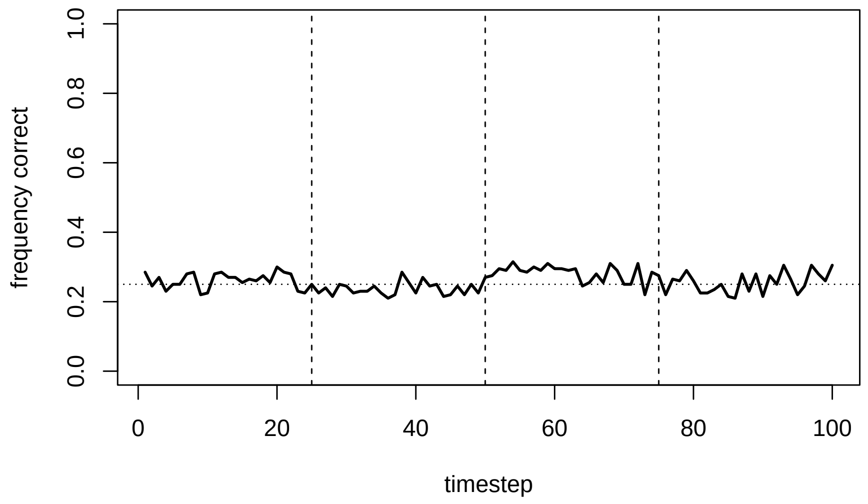
We now add heterogeneity via α_σ and β_σ:

```
data_model17b <- RL_multiple(N = 200,
                             alpha_mu = 0.7,
                             alpha_sd = 0.1,
                             beta_mu = 0.3,
                             beta_sd = 0.1)

plot_correct(data_model17b)
```



There is not too much difference here with just a small amount of random variation in the parameters. However, adding too much heterogeneity has a negative effect:

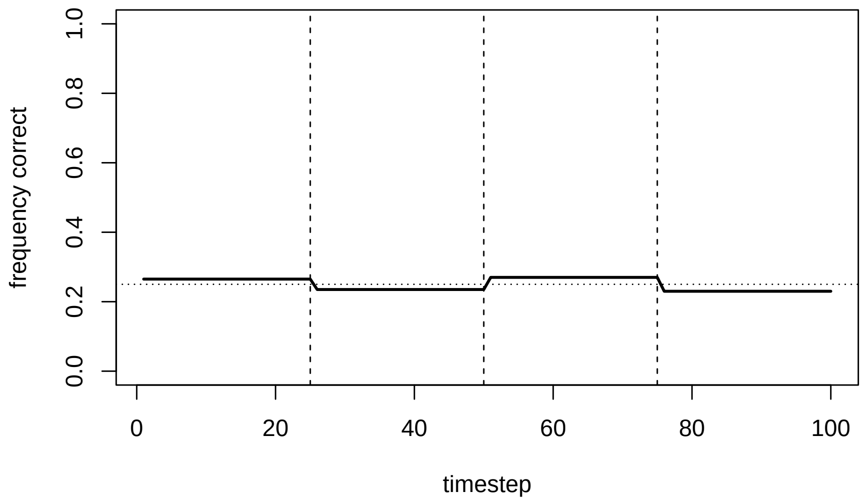
```r
data_model17b <- RL_multiple(N = 200,
                             alpha_mu = 0.7,
                             alpha_sd = 1,
                             beta_mu = 0.3,
                             beta_sd = 1)

plot_correct(data_model17b)
```



Now the proportion of optimal choices remain at chance. With too much heterogeneity we lose the exploration-exploitation balance that the values $\alpha = 0.7$ and $\beta = 0.3$ give us. Instead, most agents are either too exploitative or too exploratory and fail to learn.

To illustrate this further, we can remove heterogeneity and increase β_μ:

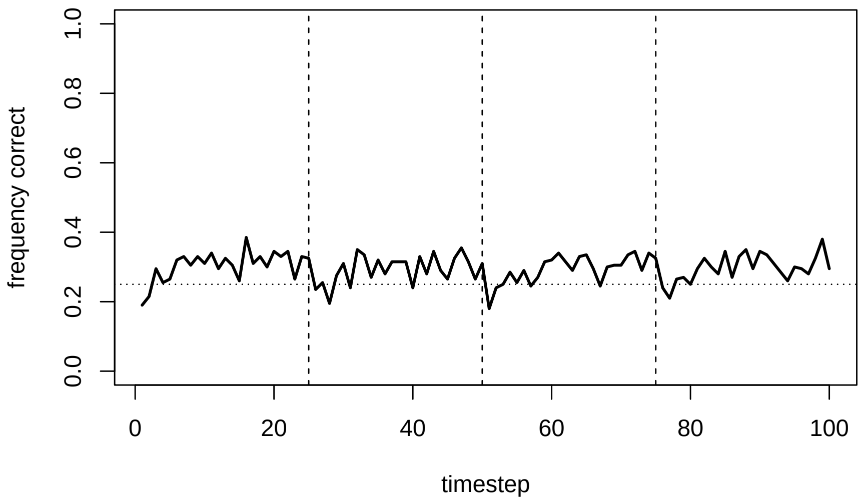
```r
data_model17b <- RL_multiple(N = 200,
                             alpha_mu = 0.7,
                             alpha_sd = 0,
                             beta_mu = 5,
                             beta_sd = 0)

plot_correct(data_model17b)
```



Now agents are too greedy, never exploring other arms and never learning the optimal arm. Indeed, the flat lines indicate that agents never switch arms at all.

Making β_μ too low also eliminates learning, as agents are too exploratory and fail to exploit the optimal:

```r
data_model17b <- RL_multiple(N = 200,
                             alpha_mu = 0.7,
                             alpha_sd = 0,
                             beta_mu = 0.1,
                             beta_sd = 0)

plot_correct(data_model17b)
```



Rather than flat lines, this gives fluctuating lines as agents continually switch arms, never settling on the optimal.

Here then we have more robustly confirmed the results obtained with **RL_single**, demonstrating that this reinforcement learning algorithm with suitable values of α and β leads on average to the learning of an optimal choice despite environmental change and noisy rewards.

Model 17c: Reinforcement learning plus social learning

Now that we have multiple agents all learning alongside each other, we can add social learning. Agents will be able to update their Q values not only using their own personal rewards, but also the choices of the other agents.

To implement social learning we use two new parameters, s and f. s is defined as the probability on each trial that an agent engages in social learning, rather than the reinforcement learning as modelled above. When $s = 0$, social learning is never used, and we revert to Model 17b. As s increases, social learning is more likely.

If social learning is used, it takes the form of unbiased cultural transmission (see Model 1) or conformist cultural transmission (see Model 5). Unlike in Model 5, we use a slightly different way of implementing conformist transmission. This way is less intuitive to understand, but easier to implement in this context.

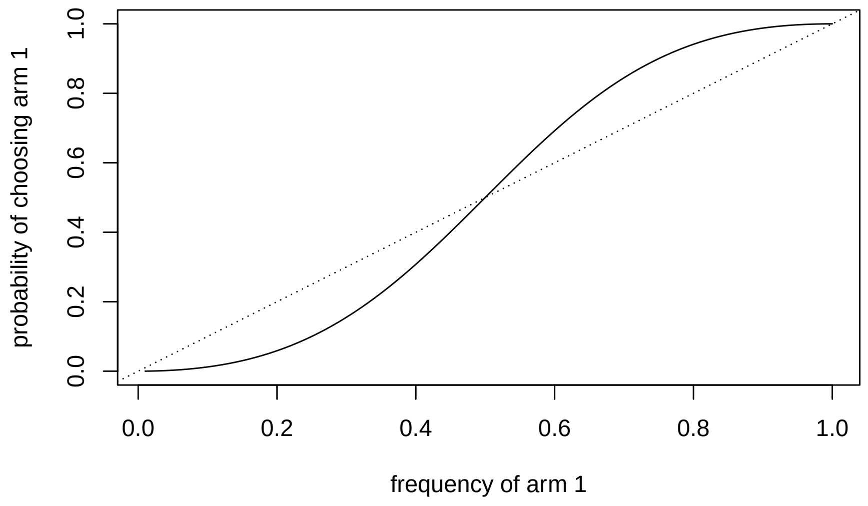
We assume that the probability that an agent chooses arm i, p_i, is given by:

$$p_i = \frac{n_i^f}{\sum_{j=1}^{k} n_j^f} \tag{17.2}$$

where n_i is the number of agents in the population who chose arm i on the previous trial and f is a conformity parameter. This f parameter is technically different to the D used in Model 5, but has the same effect of controlling the strength of conformity. The denominator is the sum of n^f for all arms from 1 to k, where in our case $k = 4$.

When $f = 1$, transmission is unbiased. Anything to the power of 1 is unchanged, so the equation above simply gives the frequency of arm i in the population. When $f > 1$, there is conformity in the sense defined in Model 5, i.e. more popular arms are disproportionately more likely to be chosen relative to unbiased transmission. To see this, here is a plot for two arms ($k = 2$), comparable to the two traits we simulated in Model 5, with $N = 100$ and $f = 2$:

```r
N <- 100  # number of agents
f <- 2 # conformity strength
n <- 1:N  # number of N agents who picked arm 1; arm 2 is then N-n
freq <- n / N  # frequencies of arm 1
prob <- n^f / (n^f + (N-n)^f)  # probabilities according to equation above

plot(freq,
     prob,
     type = 'l',
     xlab = "frequency of arm 1",
     ylab = "probability of choosing arm 1")

abline(a = 0, b = 1, lty = 3)  # dotted line for unbiased transmission baseline
```



Here we see the familiar S-shaped conformity curve from Model 5. When a majority of agents pick arm 1 (frequency > 0.5), there is a greater than expected probability of picking arm 1 relative to unbiased transmission. When arm 1 is in the minority (frequency < 0.5), there is a less than expected probability. Just like D, f acts to magnify the adoption of more-common traits. However, this formulation with f allows multiple traits - here, multiple arms - whereas the D formulation only permits two options.

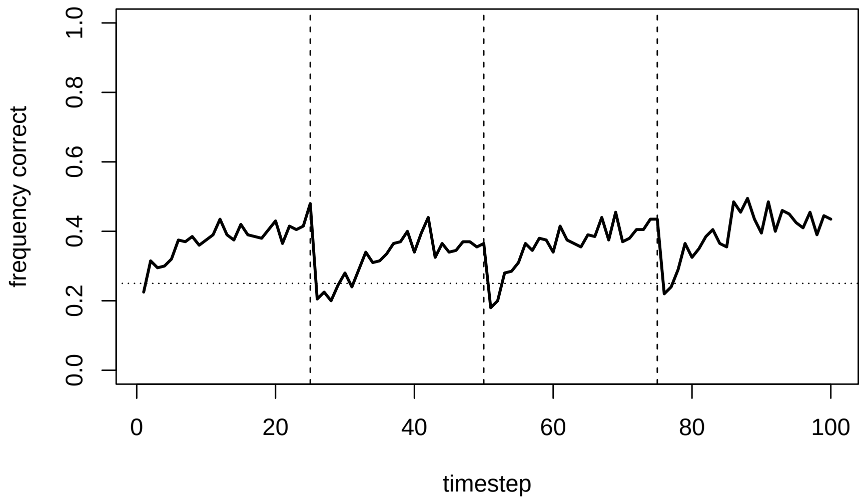
Let's start by assuming that all agents have identical s and f values, with α and β still potentially varying across agents. The function **RL_social** below is largely identical to **RL_multiple**. However, instead of *probs* being the softmax probabilities derived from Q values, we instead store these softmax probabilities in a new variable *p_RL* (probabilities for reinforcement learning). On trial 1 ($t == 1$), *probs* is just *p_RL*, because there haven't been any choices yet to use for social learning. On subsequent trials ($t > 1$), we also calculate *p_SL* (probabilities for social learning). First we get $n[arm]$, the number of agents who chose each arm on trial $t - 1$. Then we apply our equation above to get *p_SL*. After converting *p_SL* into a matrix to make it comparable with *p_RL*, we combine *p_RL* and *p_SL* according to s to get overall *probs*. The rest is the same as before.

```
RL_social <- function(N, alpha_mu, alpha_sd, beta_mu, beta_sd, s, f) {

  # set up arm rewards
  arm_means <- data.frame(p1 = c(10,13,10,10),
                          p2 = c(10,10,10,13),
                          p3 = c(13,10,10,10),
                          p4 = c(10,10,13,10))
  arm_sd <- 1.5

  # draw agent beta, alpha, s and f from overall mean and sd
  beta <- rnorm(N, beta_mu, beta_sd)  # inverse temperatures
  alpha <- rnorm(N, alpha_mu, alpha_sd)  # learning rates

  # avoid impossible values
  alpha[alpha < 0] <- 0  # ensure all alphas are >0
  alpha[alpha > 1] <- 1  # ensure all alphas are <1
  beta[beta < 0] <- 0  # ensure all betas are >0

  # for storing Q values for 4 arms on current trial, initially all zero
  Q_values <- matrix(data = 0,
                     nrow = N,
                     ncol = 4)

  # for storing choices and rewards per agent per trial
  output <- data.frame(trial = rep(1:100, each = N),
                       agent = rep(1:N, 100),
                       choice = rep(NA, 100*N),
                       reward = rep(NA, 100*N))

  # vector to hold frequencies of choices for conformity
  n <- rep(NA, 4)

  # t-loop
  for (t in 1:100) {

    # get asocial softmax probabilities p_RL from Q_values
    p_RL <- exp(beta * Q_values) / rowSums(exp(beta * Q_values))

    # get social learning probabilities p_SL from t=2 onwards
    if (t == 1) {

      probs <- p_RL

    } else {

      # get number of agents who chose each option
      for (arm in 1:4) n[arm] <- sum(output[output$trial == (t-1),]$choice == arm)

      # conformity according to f
      p_SL <- n^f / sum(n^f)
```

```
      # convert p_SL to N-row matrix to match p_RL
      p_SL <- matrix(p_SL, nrow = N, ncol = 4, byrow = T)

      # update probs by combining p_RL and p_SL according to s
      probs <- (1-s)*p_RL + s*p_SL

    }

    # choose an arm based on probs
    choice <- apply(probs, 1, function(x) sample(1:4, 1, prob = x))

    # get reward
    if (t <= 25) reward <- round(rnorm(N, mean = arm_means$p1[choice], sd = arm_sd))
    if (t > 25 & t <= 50) reward <- round(rnorm(N, mean = arm_means$p2[choice], sd = a
    if (t > 50 & t <= 75) reward <- round(rnorm(N, mean = arm_means$p3[choice], sd = a
    if (t > 75) reward <- round(rnorm(N, mean = arm_means$p4[choice], sd = arm_sd))

    # update Q values one arm at a time
    for (arm in 1:4) {

      chosen <- which(choice==arm)

      Q_values[chosen,arm] <- Q_values[chosen,arm] +
        alpha[chosen] * (reward[chosen] - Q_values[chosen,arm])

    }

    # store choice and reward in output
    output[output$trial == t,]$choice <- choice
    output[output$trial == t,]$reward <- reward

  }

  # record whether correct
  output$correct <- output$choice == c(rep(2,25*N),rep(4,25*N),rep(1,25*N),rep(3,25*N)

  # export output dataframe
  output

}
```

If we set $s = 0$ - no social learning - we should recapitulate **RL_multiple**. Note that when $s = 0$, f is irrelevant as conformity cannot occur when there is no social learning.

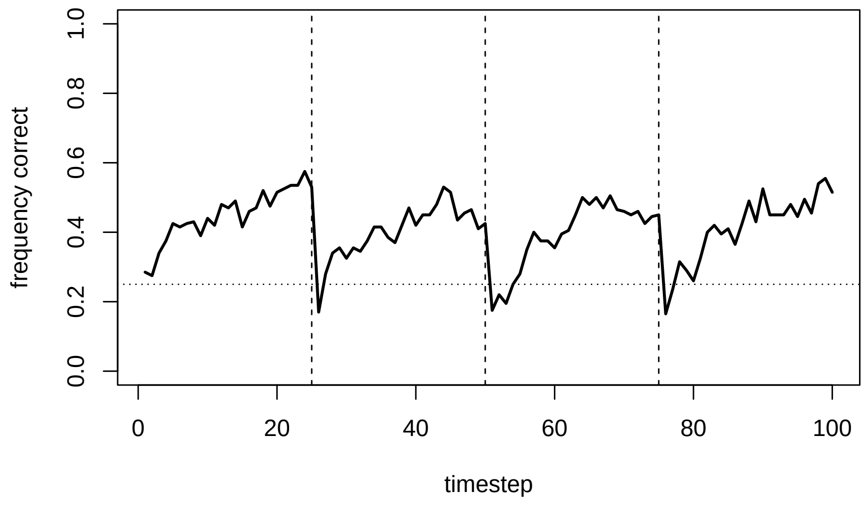
```
data_model17c <- RL_social(N = 200,
                           alpha_mu = 0.7,
                           alpha_sd = 0.1,
                           beta_mu = 0.3,
                           beta_sd = 0.1,
                           s = 0,
                           f = 1)

plot_correct(data_model17c)
```



As before, the frequency of optimal arm choices increases in each time period to around 40%.

What happens if we add some social learning, $s = 0.3$?

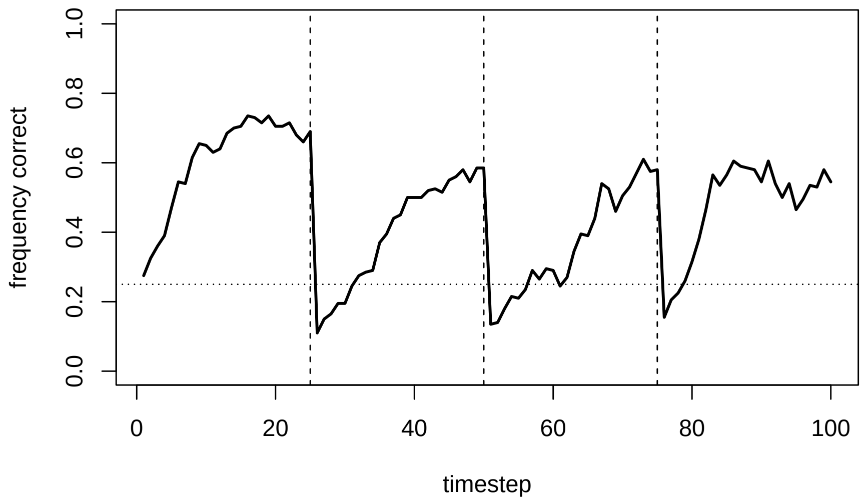
```
data_model17c <- RL_social(N = 200,
                           alpha_mu = 0.7,
                           alpha_sd = 0.1,
                           beta_mu = 0.3,
                           beta_sd = 0.1,
                           s = 0.3,
                           f = 1)

plot_correct(data_model17c)
```



Adding some social learning improves the performance of our agents slightly. The proportion of correct choices now exceeds 40% by the end of most time periods. Social learning is magnifying the effect of reinforcement learning. Those agents who are engaging in reinforcement learning are on average learning and choosing the optimal arm, and agents who are engaging in social learning are therefore more likely to copy this more-frequent arm than the other less-frequent arms.

What about conformity, say $f = 2$?

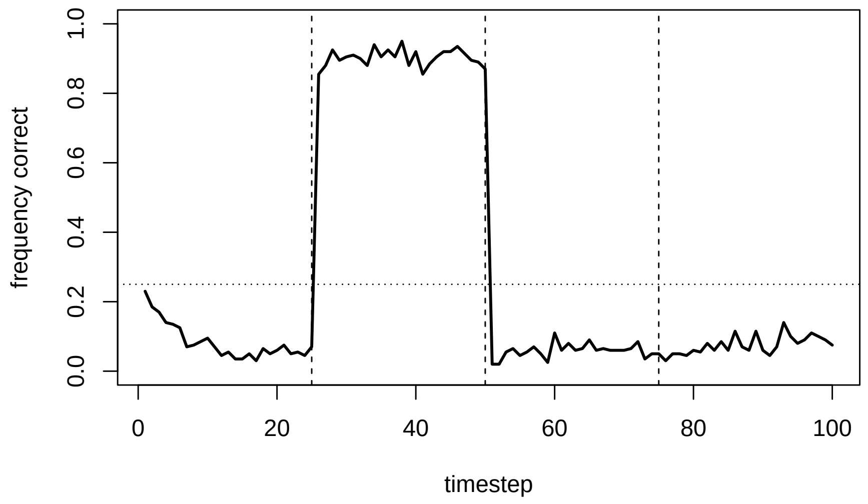
```r
data_model17c <- RL_social(N = 200,
                           alpha_mu = 0.7,
                           alpha_sd = 0.1,
                           beta_mu = 0.3,
                           beta_sd = 0.1,
                           s = 0.3,
                           f = 2)

plot_correct(data_model17c)
```



This is even better, with 60-70% of agents now making optimal choices by the end of each time period. Conformity magnifies reinforcement learning even more than unbiased social learning, given that reinforcement learning makes the optimal arm the most-common arm and conformity disproportionately increases this arm's frequency.

However, too much social learning is not good. Here we increase s to 0.8:

```r
data_model17c <- RL_social(N = 200,
                           alpha_mu = 0.7,
                           alpha_sd = 0.1,
                           beta_mu = 0.3,
                           beta_sd = 0.1,
                           s = 0.8,
                           f = 2)

plot_correct(data_model17c)
```



The output should show one time period at almost 100% correct choices, and the other three at almost 0%. With too much social learning, agents mostly make choices based on what others are choosing. Too few agents are using reinforcement / individual learning to discover the optimal arm. Instead, one arm goes to fixation due to cultural drift. In one time period it happens to be optimal, in the others it isn't, and which one is purely due to chance.

Finally, we can add agent heterogeneity in both s and f, just as we did for α and β. It seems reasonable that real people vary in both their propensity to copy others and their propensity for conformity, so we should be able to simulate these individual differences.

The following updated **RL_social** function draws s and f from normal distributions with means s_μ and f_μ and standard deviations s_σ and f_σ respectively. Note the new conformity code, which creates a matrix of n for each agent, raises it to the power of that agent's f value, and divides by the sum of all the n^f values. This is a bit opaque, but as before this vectorised code is much faster than looping through all agents.

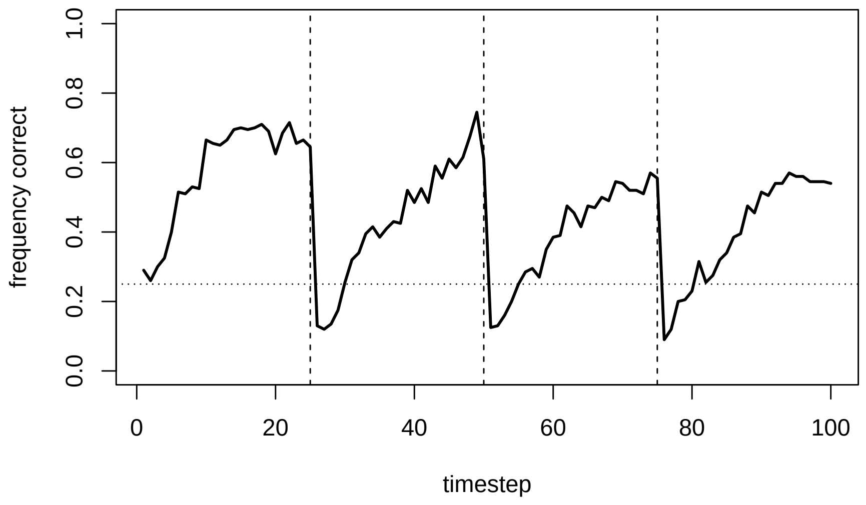
```
RL_social <- function(N,
                      alpha_mu, alpha_sd,
                      beta_mu, beta_sd,
                      s_mu, s_sd,
                      f_mu, f_sd) {

  # set up arm rewards
```

```
arm_means <- data.frame(p1 = c(10,10,10,13),
                        p2 = c(10,13,10,10),
                        p3 = c(13,10,10,10),
                        p4 = c(10,10,13,10))
arm_sd <- 1.5

# draw agent beta, alpha, s and f from overall mean and sd
beta <- rnorm(N, beta_mu, beta_sd)  # inverse temperatures
alpha <- rnorm(N, alpha_mu, alpha_sd)  # learning rates
s <- rnorm(N, s_mu, s_sd)  # social learning tendency
f <- rnorm(N, f_mu, f_sd)  # conformity parameter

# avoid impossible values
alpha[alpha < 0] <- 0  # ensure all alphas are >0
alpha[alpha > 1] <- 1  # ensure all alphas are <1
beta[beta < 0] <- 0  # ensure all betas are >0
s[s < 0] <- 0  # s between 0 and 1
s[s > 1] <- 1
f[f < 0] <- 0  # f > 0

# for storing Q values for 4 arms on current trial, initially all zero
Q_values <- matrix(data = 0,
                   nrow = N,
                   ncol = 4)

# for storing choices and rewards per agent per trial
output <- data.frame(trial = rep(1:100, each = N),
                     agent = rep(1:N, 100),
                     choice = rep(NA, 100*N),
                     reward = rep(NA, 100*N))

# vector to hold frequencies of choices for conformity
n <- rep(NA, 4)

# t-loop
for (t in 1:100) {

  # get asocial softmax probabilities p_RL from Q_values
  p_RL <- exp(beta * Q_values) / rowSums(exp(beta * Q_values))

  # get social learning probabilities p_SL from t=2 onwards
  if (t == 1) {

    probs <- p_RL
  } else {

    # get number of agents who chose each option
    for (arm in 1:4) n[arm] <- sum(output[output$trial == (t-1),]$choice == arm)

    # conformity according to f, allowing for different agent f's if f_sd>0
    p_SL <- matrix(rep(n, times = N)^rep(f, each = 4), nrow = N, ncol = 4, byrow = T
    p_SL_rowsums <- matrix(rep(rowSums(p_SL), each = 4), nrow = N, ncol = 4, byrow =
    p_SL <- p_SL / p_SL_rowsums

    # update probs by combining p_RL and p_SL according to s
    probs <- (1-s)*p_RL + s*p_SL

  }

  # choose an arm based on probs
  choice <- apply(probs, 1, function(x) sample(1:4, 1, prob = x))

  # get reward
  if (t <= 25) reward <- round(rnorm(N, mean = arm_means$p1[choice], sd = arm_sd))
  if (t > 25 & t <= 50) reward <- round(rnorm(N, mean = arm_means$p2[choice], sd = a
  if (t > 50 & t <= 75) reward <- round(rnorm(N, mean = arm_means$p3[choice], sd = a
  if (t > 75) reward <- round(rnorm(N, mean = arm_means$p4[choice], sd = arm_sd))

  # update Q values one arm at a time
  for (arm in 1:4) {

    chosen <- which(choice==arm)

    Q_values[chosen,arm] <- Q_values[chosen,arm] +
      alpha[chosen] * (reward[chosen] - Q_values[chosen,arm])

  }

  # store choice and reward in output
  output[output$trial == t,]$choice <- choice
  output[output$trial == t,]$reward <- reward

}

# record whether correct
output$correct <- output$choice == c(rep(4,25*N),rep(2,25*N),rep(1,25*N),rep(3,25*N)

# export output dataframe
output
```

```
}
```

We can confirm that the last output above is recreated with agent heterogeneity in α, β, s and f:

```
data_model17c <- RL_social(N = 200,
                           alpha_mu = 0.7,
                           alpha_sd = 0.1,
                           beta_mu = 0.3,
                           beta_sd = 0.1,
                           s_mu = 0.3,
                           s_sd = 0.1,
                           f_mu = 2,
                           f_sd = 0.1)

plot_correct(data_model17c)
```



While it may seem trivial to show that adding a small amount of random variation to parameters gives the same output as not having any variation, it is useful in principle to be able to simulate agent heterogeneity in model parameters. In reality individual heterogeneity in learning and behaviour is the norm. In the Statistical Appendix below we will see how to recover these individually-varying parameters from our simulated data. This serves as a validation check for experiments and real-world data, where we might want to estimate these parameters from actual, individually-varying people.

Summary

In Model 17 we saw how to implement reinforcement learning. Cultural evolution models require some kind of individual learning to introduce variation and track environmental change. Reinforcement learning, extensively used in artificial intelligence and computer science, is a richer form of individual learning than something like cultural mutation seen in Model 2.

Our reinforcement learning algorithm is designed to balance exploitation and exploration in the multi-armed bandit task with hidden optima and noisy reward feedback. Too much exploitation means that the agent might prematurely settle on a non-optimal choice. Too much exploration and the agent will continue to try all options, never settling on the optimal choice. The parameters α (the learning rate) and β (inverse temperature) can be used to balance this trade-off.

We also saw how social learning can magnify reinforcement learning, particularly conformist social learning. Conformity boosts the frequency of the most common option. If reinforcement learning has increased the frequency of the optimal option even slightly above chance, then conformity can boost the frequency of this optimal option further. However, too much social learning is bad. If there is not enough reinforcement learning going on, the optimal option is not discovered, and so a random arm is magnified.

Reinforcement learning has commonly been used in model-driven cultural evolution lab experiments such as McElreath et al. (2008), Barrett et al. (2017), Toyokawa et al. (2019) and Deffner et al. (2020). These experiments get participants to play a multi-armed bandit task of the kind simulated above. Rather than using generic statistical models such as linear regression to analyse the participant's data, these studies use the reinforcement and social learning models implemented above. Participant data is fit to these models, and model parameters estimated both for the whole sample and for each participant. Some participants might be particularly exploratory, reflected in a low estimated β. Some might be particularly prone to conformity, reflected in a large estimated f. This is why we simulated agent heterogeneity - to reflect this real life heterogeneity. Overall, these experiments have shown that real participants are often quite effective at solving the multi-armed bandit task, both in balancing the exploitation-exploration trade-off and in using conformist social learning.

One might wonder about the ecological validity of multi-armed bandit tasks. Most of us rarely find ourselves faced with the task of choosing one of k slot machines in a casino. However, the general structure of the task probably resembles many real world problems beyond the windowless walls of a casino. For example, we might want to decide which of several restaurants have the best food; which beach has the fewest people; or which website gives the cheapest travel tickets. In each case, you do not know in advance which option is best; feedback is noisy (e.g. you might visit a restaurant on a day when they've run out of your favourite dish); and it's possible to learn from others' choices (e.g. by seeing how busy the restaurant is). The exploitation-exploration trade-off is inherent in all these real world examples of information search.

In terms of programming techniques, one major innovation is the introduction of agent heterogeneity in model parameters. Rather than assuming that every agent has the same, say, conformity parameter, we assume each agent differs. Formally, agent parameter values are drawn randomly from a normal distribution, with the standard deviation of this normal distribution controlling the amount of heterogeneity. However there is no reason that this has to be a normal distribution. Other distributions might better reflect real world individual variation.

We also saw a different way of modelling conformity to the one introduced in Model 5. While less intuitive than the 'mating tables' approach used to derive equation 5.1, equation 17.2 has the advantage of allowing any number of choices rather than just two. That makes it useful for the multi-armed bandit task when $k > 2$.

Acknowledgements

The following resources were invaluable in compiling Model 17: Raphael Geddert's *Reinforcement Learning Resource Guide & Tutorial* (https://github.com/rmgeddert/Reinforcement-Learning-Resource-Guide/blob/master/RLResourceGuide.md), Lei Zhang's *Bayesian Statistics and Hierarchical Bayesian Modeling for Psychological Science* (https://github.com/lei-zhang/BayesCog_Wien) course, and Dominik Deffner's analysis code for his 2020 study (https://github.com/DominikDeffner/Dynamic-Social-Learning). Many thanks also to Dominik Deffner who helped with the Stan code in the Statistical Appendix.

Exercises

1. Use **RL_single** to explore different values of α and β. Using code from previous models, write a function to generate a heatmap showing the proportion correct arm for a range of values of each parameter.

2. In Model 17 we started the agents with Q values all initially zero. As rewards are obtained, these Q values then increased. Try instead very high starting values, e.g. all 100. These will then decrease. Does this change the dynamics of **RL_multiple** shown in the plots? (see Sutton & Barto 2018 for discussion)

3. Use **RL_social** to explore different values of s and f, using the heatmap function from (1) with fixed α and β. What happens when $s = 1$? What about when $s = 0.3$ and $f = 10$? Explain these dynamics in terms of the findings regarding unbiased transmission in Model 1 and conformity in Model 5.

4. Modify the Model 17 functions to make the number of arms k, the standard deviation of the arm reward distribution arm_sd, and the number of timesteps t_{max}, user-defined parameters. Instead of changing the optimal arm at fixed intervals, add another parameter which specifies the probability on each timestep of a new random arm becoming the optimal. Explore how the reinforcement learning algorithm handles different numbers of arms, different arm noisinesses, and different rates of environmental change.

Statistical Appendix

As discussed above, it is useful to be able to recover the model parameters (in our case α, β, s and f) from the output simulation data. This serves as a validity check when running the same analyses on experimental or observational data generated by real people (or other animals), where we don't know the parameter values.

Here we follow studies such as McElreath et al. (2008), Barrett et al. (2017), Toyokawa et al. (2019) and Deffner et al. (2020) and use multi-level Bayesian statistical models run using the platform Stan (https://mc-stan.org/). McElreath (2020) provides an excellent overview of Bayesian statistics in R and Stan. I won't discuss the statistical methods used here but I recommend working through McElreath's book to understand them fully.

We will use the `cmdstanr` package to interface with Stan. Follow the instructions to install `cmdstanr` at https://mc-stan.org/cmdstanr/articles/cmdstanr.html.

You will also need to install CmdStan, the command line interface to Stan, and check you have a suitable C++ toolchain (don't worry, you don't need to understand what this is, just follow the instructions via the link). We will also use the `bayesplot` package to generate some plots later. This can be installed via CRAN like most other packages.

```
library(cmdstanr)
library(bayesplot)
```

First we try to recover α and β from a run of **RL_single**. We run the simulation and remind ourselves of the output:

```
data_model17a <- RL_single(alpha = 0.7, beta = 0.3)

head(data_model17a)
```

```
##   arm1     arm2 arm3 arm4 choice reward correct
## 1    0  7.70000    0    0      2     11   FALSE
## 2    0 10.01000    0    0      2     11   FALSE
## 3    0 10.70300    0    0      2     11   FALSE
## 4    0 11.61090    0    0      2     12   FALSE
## 5    0 11.18327    0    0      2     11   FALSE
## 6    0 11.05498    0    0      2     11   FALSE
```

Our Stan model will only use *choice*s and *reward*s from this output. These need to be in the form of a list rather than a dataframe, which we call *dat*:

```
# create data list
dat <- list(choice = data_model17a$choice,
            reward = data_model17a$reward)

dat
```

```
## $choice
##   [1] 2 2 2 2 2 2 4 4 3 4 3 3 4 4 4 3 2 2 4 4 2 2 4 4 4 4 2 2 2 2 3 2 2 2 2 2 3
##  [38] 3 4 2 2 4 4 2 2 2 2 4 4 2 4 1 3 1 1 2 2 3 1 1 2 1 2 4 1 1 2 4 3 1 2 1 1 1
##  [75] 1 4 1 4 1 1 3 3 3 3 3 4 3 3 2 3 3 1 4 3 1 1 3 2 1 1
##
## $reward
##   [1] 11 11 11 12 11 11 13 13 11 13  9  8 13 15 13 10  9  9 14 13 12 12 14 15 15
##  [26]  7 14 16 14 16  8 13 12 14 14 14 10 10 13 13 14 11 10 14 11 14 13 10  6 14
##  [51] 10 12 10 12 14 13 10 11 14 14 11 13  9 12 14 13  7  9 10 11  7 15 11 10 11
##  [76]  9 12 11 12  7 12 12 13 14 12  9 13 15  8 13 11 10 10 14  9 10 14 10 13  9
```

Stan requires models written in the Stan programming language. This is slightly different to R. Usually the Stan model is saved as a separate text file with the extension `.stan`. This `.stan` file is then called within R to actually run the model. We will follow this practice. All the `.stan` files we will use are in the folder `model17_stanfiles`. The first one is called `model17_single.stan`. I will reproduce chunks of these stan files here but include `eval=FALSE` in the code chunks so that they are not run if you knit the RMarkdown file. If you are following along, don't run these code chunks (you'll just get an error).

Stan files require at least three sections. The first, labelled `data`, specifies the data that is required by the model. This should match the *dat* list from above. The `data` section of `model17_single.stan` is reproduced below:

```
data {
  array[100] int<lower=1,upper=4> choice;  //arm choice
  array[100] int reward;  // reward
}
```

Here we have our *reward* and *choice* variables. There are a few differences in this Stan code compared to R code. First, all statements must end in a semicolon. Second, comments are marked with `//` rather than `#`. Third, Stan requires the programmer to explicitly declare what type each variable is, i.e. whether it is an integer, real (continuous) number, vector, matrix or whatever (such languages are called 'statically typed', as opposed to R which is 'dynamically typed'). The possible range of each variable can also be specified. The advantage of this explicit type declaration is that Stan throws an error if you try to do a calculation with, say, an integer when a real number is required, or mistakenly enter a value of 100 for a variable that is constrained to be between 1 and 10.

Hence *choice* and *reward* are both declared as arrays of 100 integers (`int`). An array is like a vector, but vectors in Stan must be `real`, while our variables are `int`, so we use arrays instead. *choice* is additionally constrained between 1 and 4 using the code `<lower=1,upper=4>` given that we only have four arms.

The second Stan block is `parameters`, where we declare the parameters of the model that we want Stan to estimate. For **RL_single** these are α and β:

```
parameters {
  real<lower = 0, upper = 1> alpha; // learning rate
  real<lower = 0> beta; // inverse temperature
}
```

Both our parameters are `real`, i.e. continuous. We constrain α to be between 0 and 1, and β to be positive.

The final block in the Stan file is the `model`. This essentially recreates the simulation contained in **RL_single**, but instead of generating choices and rewards from *probs* and *arm_means* respectively, we use the already-generated *choice* and *reward* data in *dat*. We also omit *output*, because that's already been generated.

```
model {
  vector[4] Q_values; // Q values for each arm
  vector[4] probs; // probabilities for each arm

  // priors
  alpha ~ beta(1,1);
  beta ~ normal(0,3);

  // initial Q values all zero
  Q_values = rep_vector(0,4);

  // trial loop
  for (t in 1:100) {

    // get softmax probabilities from Q_values
    probs = softmax(beta * Q_values);

    // choose an arm based on probs
    choice[t] ~ categorical(probs);

    //update Q_values
    Q_values[choice[t]] = Q_values[choice[t]] +
      alpha * (reward[t] - Q_values[choice[t]]);

  }

}
```

The first two lines above declare *Q_values* and *probs* as vectors (hence real numbers) of length four, one for each arm. We then specify priors for α and β. Recall from Model 16 that Bayesian analyses require a prior distribution, which is updated given data to generate a posterior distribution. Because α must range from 0 to 1 and we don't have strong expectations about what it might be (remember, we are assuming that we haven't already seen the simulated output, just like we wouldn't know what the experimental participants' αs are), we specify a flat beta distribution. Recall again from Model 16 that a beta distribution must range from 0 to 1, and with parameters (1,1) this gives a flat uniform distribution. For β we can expect that relatively low values are more likely; β has no upper bound, but something like 0.5 or 1 is more likely than 1000 or 10000. We therefore specify a normal distribution peaking at 0. Remember that β was constrained to be positive in the `parameters` section so the prior is cut at zero and can't be negative.

Then like in **RL_single** we set initial *Q_values* to all be zero and start the t-loop. *probs* is calculated as in **RL_single** but using Stan's built-in **softmax** function rather than explicitly writing out the softmax formula. The *choice* for this trial is then modelled using the categorical distribution, and *Q_values* are updated as before using the *reward* for this trial.

Now we have all three blocks of the Stan file we can load the model into R using the **cmdstan_model** command, specifying the Stan file location. At this point, any programming errors or inconsistencies in your Stan file might be flagged. We then sample the model using the **sample** command, specifying the data list *dat*, the number of chains and number of warmup and sampling iterations. Chains refers to the number of Markov chains, and iterations to the length of those chains. Warmup chains are not used to derive estimates of your parameters, which only come from the sampling iterations. See McElreath (2020) for an explanation of these terms. Essentially though, the more chains and the more iterations, the more robust will be the model estimates. On the other hand, the more chains and iterations, the longer the model will take to run. Good advice is to run several short chains to ensure that the chains converge well and the model is working properly, then one long chain for the final analysis. Here we will run 1 chain with 500 warmup and 500 sampling iterations. Try more chains and iterations to compare.

```r
# load stan model from file
RL_single_stan <- cmdstan_model("model17_stanfiles/model17_single.stan")

# run model
fit_RL_single <- RL_single_stan$sample(
  data = dat,
  chains = 1,
  iter_warmup = 500,
  iter_sampling = 500)
```

```
## Running MCMC with 1 chain...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 0.6 seconds.
```

You will see a series of progress messages indicating the number of iterations completed and the time taken. To see the output, we use the **print** command with the Stan model and the parameters:

```
fit_RL_single$print(variables = c("alpha", "beta"))
```

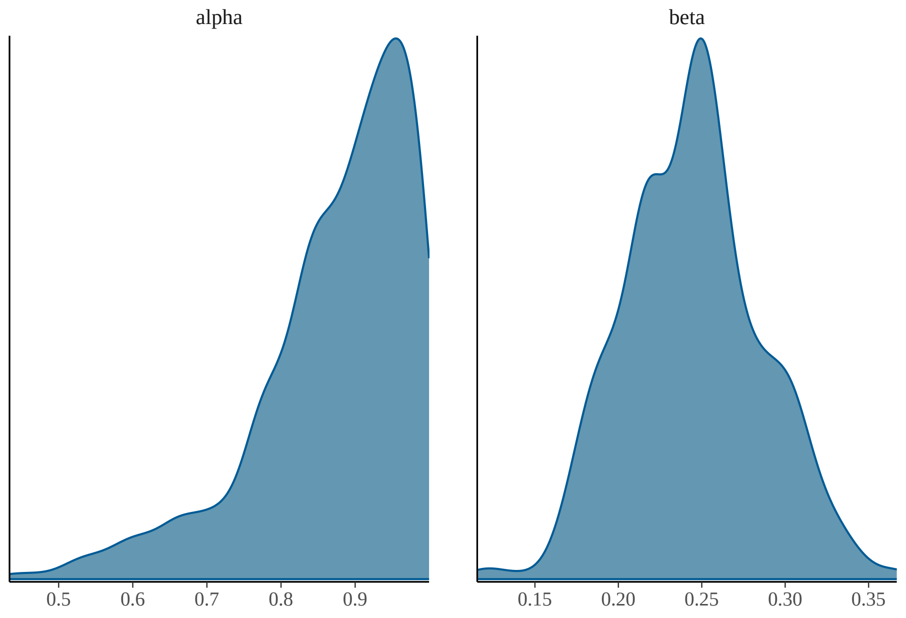
```
##  variable mean median   sd  mad   q5  q95 rhat ess_bulk ess_tail
##     alpha 0.87   0.90 0.11 0.09 0.65 0.99 1.00      220       69
##     beta  0.24   0.25 0.04 0.04 0.18 0.31 1.00      354      379
```

The key bits here are the estimated means of α and β in the first column of the table. Hopefully our Stan model has estimated our parameters well and they are close to $\alpha = 0.7$ and $\beta = 0.3$. Given that there is only one agent and a relatively small number of timesteps, they are unlikely to be super accurate, and the 95% confidence intervals (q5 and q95) are probably quite wide.

The final columns of the table give a convergence diagnostic *Rhat* ($\hat{R}$) and two measures of the effective number of independent samples, *ess_bulk* and *ess_tail*. These are all indicators of how well the chains converged and how reliable the estimates are. See McElreath (2020) for discussion of these measures. In general, Rhat should be 1 and definitely no greater than 1.05, and ess_bulk and ess_tail should not be too low relative to the number of non-warmup iterations, which here was 500. An Rhat larger than 1 and very low ess indicate poorly converged chains and probably a misspecified model.

We can show the shape of the posterior distribution of each parameter using the *mcmc_dens* command from the `bayesplot` package:

```
mcmc_dens(fit_RL_single$draws(variables = c("alpha","beta")))
```



Hopefully the actual values of 0.7 and 0.3 fall within these posterior distributions.

Although our model converged well and produced reasonably accurate estimates, later models will be more complex and not converge so well. One way to deal with this is to convert the parameters to log and logit scales. We will do this here so that later on it doesn't get confusing.

Converting parameters to log and logit scales is a way of constraining them to be positive and between 0 and 1 respectively. This is done in the Stan file `model17_single_log.stan`. In this file the `parameters` block of the Stan file is changed to:

```
parameters {
  real logit_alpha; // learning rate
  real log_beta; // inverse temperature
}
```

Rather than α and β, our parameters to estimate are *logit_alpha* and *log_beta*. These are declared as unconstrained `real` (continuous) numbers. Then in the `model` block we define the priors for these as normal distributions, declare α and β in the t-loop, and calculate them using **inv_logit** and **exp** on *logit_alpha* and *log_beta* respectively. Otherwise, everything is identical to before.

```
model {
  vector[4] Q_values; // Q values for each arm
  vector[4] probs; // probabilities for each arm

  // priors
  logit_alpha ~ normal(0,1);
  log_beta ~ normal(-1,1);

  // initial Q values all zero
  Q_values = rep_vector(0,4);

  // trial loop
  for (t in 1:100) {

    real alpha;
    real beta;

    // get softmax probabilities from Q_values
    beta = exp(log_beta);
    probs = softmax(beta * Q_values);

    // choose an arm based on probs
    choice[t] ~ categorical(probs);

    //update Q_values
    alpha = inv_logit(logit_alpha);
    Q_values[choice[t]] = Q_values[choice[t]] +
      alpha * (reward[t] - Q_values[choice[t]]);

  }

}
```

Finally, we add a new block called `generated quantities` to convert *logit_alpha* and *log_beta* back to α and β, rather than having to do this by hand afterwards:

```
generated quantities {
  real alpha;
  real beta;

  alpha = inv_logit(logit_alpha);
  beta = exp(log_beta);
}
```

Now we run our new model:

```
# load stan model from file
RL_single_log_stan <- cmdstan_model("model17_stanfiles/model17_single_log.stan")

# run model
fit_RL_single_log <- RL_single_log_stan$sample(
  data = dat,
  chains = 1,
  iter_warmup = 500,
  iter_sampling = 500)
```

```
## Running MCMC with 1 chain...
##
## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)
## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 0.4 seconds.
```

Again we use **print** to get the parameter estimates, specifying only α and β rather than *logit_alpha* and *log_beta*:

```
fit_RL_single_log$print(variables = c("alpha", "beta"))
```

```
##  variable mean median   sd  mad   q5  q95 rhat ess_bulk ess_tail
##     alpha 0.78   0.79 0.10 0.10 0.58 0.92 1.01      307      254
##     beta  0.24   0.24 0.04 0.04 0.18 0.32 1.00      400      257
```

They should be a good match with the estimates from *fit_RL_single*.

Next we do the same for **RL_multiple**, with individual heterogeneity in α and β. The simulated data looks like this:

```
data_model17b <- RL_multiple(N = 50,
                             alpha_mu = 0.7,
                             alpha_sd = 0.1,
                             beta_mu = 0.3,
                             beta_sd = 0.1)

head(data_model17b)
```

```
##   trial agent choice reward correct
## 1     1     1      4     14    TRUE
## 2     1     2      4     15    TRUE
## 3     1     3      1     12   FALSE
## 4     1     4      3     11   FALSE
## 5     1     5      2     10   FALSE
## 6     1     6      2     10   FALSE
```

We now have a trial column and an agent column given that there are $N = 50$ agents. As before, we need to convert these into a list. Instead of choice and reward being 100-length vectors, now we convert them to 100 x N matrices with each row a timestep as before but with each column representing an agent. These matrices are stored in *dat* along with N.

```
# convert choice and reward to matrices, ncol=N, nrow=t_max
choice_matrix <- matrix(data_model17b$choice, ncol = max(data_model17b$agent), byrow = T)
reward_matrix <- matrix(data_model17b$reward, ncol = max(data_model17b$agent), byrow = T)

# create data list
dat <- list(choice = choice_matrix,
            reward = reward_matrix,
            N = max(data_model17b$agent))
```

The Stan file `model17_multiple.stan` contains the stan code for this analysis. The `data` block imports the three elements of *dat*. N is an `int` (integer; we can't have half an agent). *reward* and *choice* are 100 x N arrays of integers (we could use matrices, but in Stan they can only hold `real` numbers, so arrays are preferable).

```
data {
  int N;  // number of agents
  array[100, N] int reward; // reward
  array[100, N] int<lower=1,upper=4> choice; // arm choice
}
```

Next the `parameters` block, along with a new `transformed parameters` block. As before we have *logit_alpha* and *log_beta*. But now there are several other parameters: z_i, σ_i, ρ_i and v_i. These are all used to implement heterogeneity across agents, sometimes called varying effects or random effects in this context. Consult McElreath (2020) for an explanation of these parameters. The key one is v_i which gives the individual-level variation in each parameter. Note that the `[2]` associated with these parameters refers to the number of parameters that have varying effects (α and β).

```
parameters {
  real logit_alpha; // learning rate grand mean
  real log_beta; // inverse temperature grand mean

  matrix[2, N] z_i;  // matrix of uncorrelated z-values
  vector<lower=0>[2] sigma_i; // sd of parameters across individuals
  cholesky_factor_corr[2] Rho_i; // cholesky factor
}

transformed parameters{
  matrix[2, N] v_i; // matrix of varying effects for each individual
  v_i = (diag_pre_multiply(sigma_i, Rho_i ) * z_i);
}
```

Finally we have the `model` block. There are a few changes to the previous single agent model. *Q_values* is now an N x 4 array in order to keep track of Q values for all N agents. There are priors for the new varying effects parameters, which again I leave you to consult McElreath (2020) for further explanation. Suffice to say that these are relatively flat priors with no strong expectations about the extent of the heterogeneity. Then we need to set up an i-loop to cycle through each agent within the t-loop. Vectorisation in Stan is possible but tricky, so I've kept it simple with an agent loop. The final change is that when calculating α and β for each agent, we add that agent's v_i term. This is indicated by the first index of v_i, so v_i[1,i] gives the varying effect for α for agent i and v_i[2,i] gives the varying effect for β for agent i. The rest is the same as before.

```
model {
  array[N] vector[4] Q_values; // Q values per agent
  vector[4] probs; // probabilities for each arm

  // priors
  logit_alpha ~ normal(0,1);
  log_beta ~ normal(-1,1);

  to_vector(z_i) ~ normal(0,1);
  sigma_i ~ exponential(1);
  Rho_i ~ lkj_corr_cholesky(4);

  // initial Q values all zero
  for (i in 1:N) Q_values[i] = rep_vector(0,4);

  // trial loop
  for (t in 1:100) {

    // agent loop
    for (i in 1:N) {

      real alpha;  // v_i parameter 1
      real beta;  // v_i parameter 2

      // get softmax probabilities from Q_values
      beta = exp(log_beta + v_i[2,i]);
      probs = softmax(beta * Q_values[i]);

      // choose an arm based on probs
      choice[t,i] ~ categorical(probs);

      //update Q_values
      alpha = inv_logit(logit_alpha + v_i[1,i]);
      Q_values[i,choice[t,i]] = Q_values[i,choice[t,i]] +
        alpha * (reward[t,i] - Q_values[i,choice[t,i]]);

    }

  }

}
```

The `generated quantities` block is identical to before.

```
generated quantities {
  real alpha;
  real beta;

  alpha = inv_logit(logit_alpha);
  beta = exp(log_beta);
}
```

Now we run the model:

```
# load stan model from file
fit_RL_multiple_stan <- cmdstan_model("model17_stanfiles/model17_multiple.stan")

# run model
```

```r
fit_RL_multiple <- fit_RL_multiple_stan$sample(
  data = dat,
  chains = 1,
  iter_warmup = 500,
  iter_sampling = 500)
```

```
## Running MCMC with 1 chain...

## Chain 1 Rejecting initial value:

## Chain 1   Gradient evaluated at the initial value is not finite.

## Chain 1   Stan can't start sampling from this initial value.

## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejec

## Chain 1 Exception: categorical_lpmf: Probabilities parameter is not a valid simplex

## Chain 1 If this warning occurs sporadically, such as for highly constrained variabl

## Chain 1 but if this warning occurs often then your model may be either severely ill

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejec

## Chain 1 Exception: categorical_lpmf: Probabilities parameter is not a valid simplex

## Chain 1 If this warning occurs sporadically, such as for highly constrained variabl

## Chain 1 but if this warning occurs often then your model may be either severely ill

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejec

## Chain 1 Exception: categorical_lpmf: Probabilities parameter is not a valid simplex
## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-conditione

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected because

## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be positive! (in

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-conditione

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected because

## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be positive! (in

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-conditione

## Chain 1

## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)
## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 162.9 seconds.
```

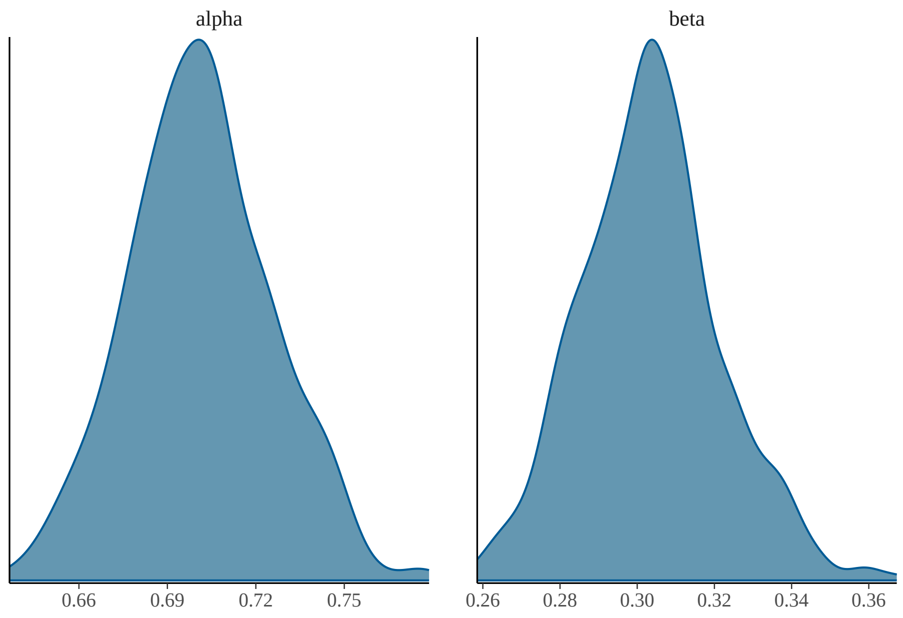
Here are the results:

```r
fit_RL_multiple$print(variables = c("alpha", "beta"))
```

```
##   variable mean median   sd  mad   q5  q95 rhat ess_bulk ess_tail
##      alpha 0.70   0.70 0.02 0.02 0.66 0.74 1.00      671      288
##      beta  0.30   0.30 0.02 0.02 0.28 0.34 1.00      246      333
```

Hopefully the model accurately estimated the original parameter values of 0.7 and 0.3. Indeed, given that we now have much more data (5000 data points from $N = 50$ rather than 100 data points from $N = 1$), these estimates should be more accurate than they were for **RL_single** as indicated by narrower confidence intervals. We can see the posteriors as before, which should also be more concentrated around the actual values:

```r
mcmc_dens(fit_RL_multiple$draws(variables = c("alpha","beta")))
```



And finally we run the whole analysis for **RL_social**:

```r
data_model17c <- RL_social(N = 50,
                           alpha_mu = 0.7,
                           alpha_sd = 0.1,
                           beta_mu = 0.3,
                           beta_sd = 0.1,
                           s_mu = 0.3,
                           s_sd = 0.1,
                           f_mu = 2,
                           f_sd = 0.1)

head(data_model17c)
```

```
##   trial agent choice reward correct
## 1     1     1      3     11   FALSE
## 2     1     2      4     15    TRUE
## 3     1     3      3      9   FALSE
## 4     1     4      4     14    TRUE
## 5     1     5      3     10   FALSE
## 6     1     6      1     11   FALSE
```

The output for **RL_social** is the same as for **RL_multiple**. As such the *dat* list is largely similar. However, to model conformity, we need for each timestep the number of agents on the previous timestep who chose each arm. The following code calculates the number of agents on each timestep who chose each arm and stores it in *n_matrix*, a 100 x 4 matrix with timesteps along the rows and arms along the columns.

```
# convert choice and reward to matrices, ncol=N, nrow=t_max
choice_matrix <- matrix(data_model17c$choice, ncol = max(data_model17c$agent), byrow = T)
reward_matrix <- matrix(data_model17c$reward, ncol = max(data_model17c$agent), byrow = T)

# create frequency matrix for conformity function
n_matrix <- matrix(nrow = 100, ncol = 4)
for (arm in 1:4) {
  for (t in 1:100) {
    n_matrix[t,arm] <- sum(data_model17c[data_model17c$trial == t,]$choice == arm)
  }
}

# create data list
dat <- list(choice = choice_matrix,
            reward = reward_matrix,
            N = max(data_model17c$agent),
            n = n_matrix)
```

Now we build our Stan file as before, which is saved as `model17_social.stan`. First, the `data` block contains N, *choice*s, *reward*s and the new *n_matrix* set of demonstrator numbers, entered as simply n:

```
data {
  int N;  // number of agents
  array[100, N] int reward; // reward
  array[100, N] int<lower=1,upper=4> choice; // arm choice
  array[100, 4] int<lower=0,upper=N> n; // number of agents picking each arm
}
```

The `parameters` and `transformed parameters` now include the social learning parameter. The frequency of social learning, s, is put on the logit scale as it is constrained to 0-1. The conformity parameter f is logged, as it is constrained to be positive. The varying effects parameters are now length 4 rather than 2, as there are now four parameters (α, β, s and f).

```
parameters {
  real logit_alpha; // learning rate grand mean
  real log_beta; // inverse temperature grand mean
  real logit_s; // social learning prob grand mean
  real log_f; // conformity parameter grand mean

  matrix[4, N] z_i;  // matrix of uncorrelated z-values
  vector<lower=0>[4] sigma_i; // sd of parameters across individuals
  cholesky_factor_corr[4] Rho_i; // cholesky factor
}

transformed parameters{
  matrix[4,N] v_i; // matrix of varying effects for each individual
  v_i = (diag_pre_multiply(sigma_i, Rho_i ) * z_i);
}
```

And the model:

```
model {
  array[N] vector[4] Q_values; // Q values per agent
  vector[4] probs; // probabilities for each arm
  vector[4] p_RL; // reinforcement learning probabilities
  vector[4] p_SL; // social learning probabilities

  // priors
  logit_alpha ~ normal(0,1);
  log_beta ~ normal(-1,1);
  logit_s ~ normal(0,1);
  log_f ~ normal(-1,1);

  to_vector(z_i) ~ normal(0,1);
  sigma_i ~ exponential(1);
```

```
Rho_i ~ lkj_corr_cholesky(4);

// initial Q values all zero
for ( i in 1:N ) Q_values[i] = rep_vector(0,4);

// trial loop
for (t in 1:100) {

  // agent loop
  for (i in 1:N) {

    real alpha;  // v_i parameter 1
    real beta;  // v_i parameter 2
    real s;  // v_i parameter 3
    real f;  // v_i parameter 4

    // get asocial softmax probabilities from Q_values
    beta = exp(log_beta + v_i[2,i]);
    p_RL = softmax(beta * Q_values[i]);

    if (t == 1) {

      // first generation has no social information
      probs = p_RL;

    } else {

      // from t=2 onwards, do conformity according to f

      f = exp(log_f + v_i[4,i]);
      for (arm in 1:4) p_SL[arm] = n[t-1,arm]^f;
      p_SL = p_SL / sum(p_SL);

      //update probs by combining p_RL and p_SL according to s
      s = inv_logit(logit_s + v_i[3,i]);
      probs = (1-s) * p_RL + s * p_SL;

    }

    // choose an arm based on probs
    choice[t,i] ~ categorical(probs);

    //update Q values
    alpha = inv_logit(logit_alpha + v_i[1,i]);
    Q_values[i,choice[t,i]] = Q_values[i,choice[t,i]] +
        alpha * (reward[t,i] - Q_values[i,choice[t,i]]);

    }

  }

}
```

Just as in the simulation, we define *p_RL* and *p_SL* as the probabilities for reinforcement and social learning respectively. From the second timestep onwards, we implement social learning. f is created from *log_f* and v_i[4,i] and s is created from *logit_s* and v_i[3,i], like we do for α and β. Now that we are not vectorising, the conformity equation is more straightforward, with each arm's frequency on timestep $t-1$ raised to the power of f and divided by the sum of all four arms.

And finally, `generated quantities` now includes s and f:

```
generated quantities {
  real alpha;
  real beta;
  real s;
  real f;

  alpha = inv_logit(logit_alpha);
  beta = exp(log_beta);
  s = inv_logit(logit_s);
  f = exp(log_f);
}
```

Let's run the model.

```
# load stan model from file
fit_RL_social_stan <- cmdstan_model("model17_stanfiles/model17_social.stan")

# run model
fit_RL_social <- fit_RL_social_stan$sample(
  data = dat,
  chains = 1,
  iter_warmup = 500,
  iter_sampling = 500)
```

```
## Running MCMC with 1 chain...

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejec
## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be positive! (in

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-condition

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected becaus

## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be positive! (in

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-condition

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected becaus

## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be positive! (in

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-condition

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected becaus

## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[3] is 0, but must be positive! (in

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable types lik

## Chain 1 but if this warning occurs often then your model may be either severely ill-condition

## Chain 1

## Chain 1 Iteration:   1 / 1000 [  0%]  (Warmup)

## Chain 1 Informational Message: The current Metropolis proposal is about to be rejected becaus
## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be posi

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable

## Chain 1 but if this warning occurs often then your model may be either severely ill-

## Chain 1

## Chain 1 Informational Message: The current Metropolis proposal is about to be reject

## Chain 1 Exception: lkj_corr_cholesky_lpdf: Random variable[2] is 0, but must be posi

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable

## Chain 1 but if this warning occurs often then your model may be either severely ill-

## Chain 1

## Chain 1 Iteration: 100 / 1000 [ 10%]  (Warmup)
## Chain 1 Iteration: 200 / 1000 [ 20%]  (Warmup)
## Chain 1 Iteration: 300 / 1000 [ 30%]  (Warmup)
## Chain 1 Iteration: 400 / 1000 [ 40%]  (Warmup)

## Chain 1 Informational Message: The current Metropolis proposal is about to be reject

## Chain 1 Exception: categorical_lpmf: Probabilities parameter is not a valid simplex

## Chain 1 If this warning occurs sporadically, such as for highly constrained variable

## Chain 1 but if this warning occurs often then your model may be either severely ill-

## Chain 1

## Chain 1 Iteration: 500 / 1000 [ 50%]  (Warmup)
## Chain 1 Iteration: 501 / 1000 [ 50%]  (Sampling)
## Chain 1 Iteration: 600 / 1000 [ 60%]  (Sampling)
## Chain 1 Iteration: 700 / 1000 [ 70%]  (Sampling)
## Chain 1 Iteration: 800 / 1000 [ 80%]  (Sampling)
## Chain 1 Iteration: 900 / 1000 [ 90%]  (Sampling)
## Chain 1 Iteration: 1000 / 1000 [100%]  (Sampling)
## Chain 1 finished in 348.2 seconds.
```

You will see some warning messages in the first few chains. This is fine, as long as it is only the first few chains, there are no warning messages at the end indicating divergent chains, and n_eff and Rhat look OK in the output.

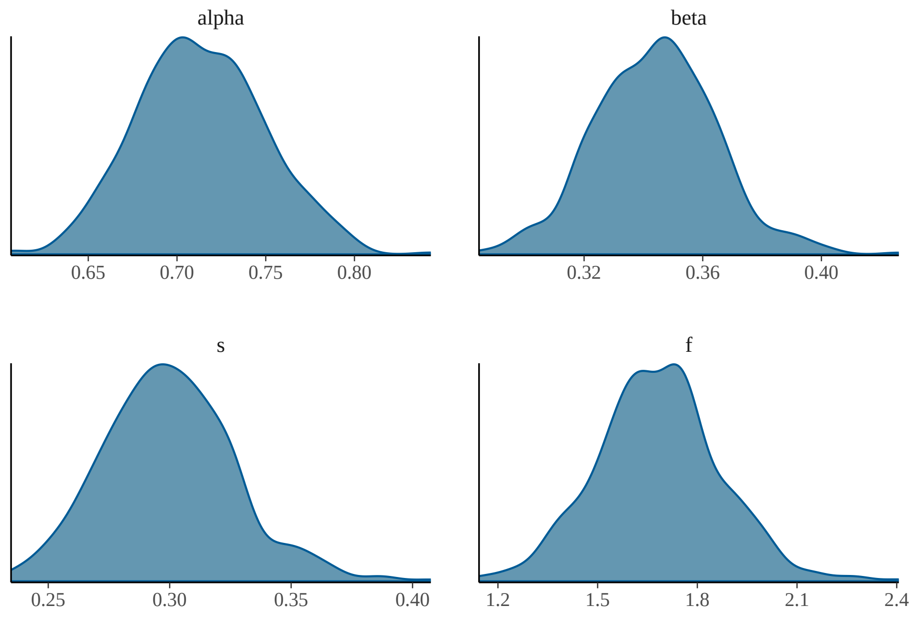
Here is the model output, checking that the n_eff are not too low and Rhat is 1:

```
fit_RL_social$print(variables = c("alpha", "beta", "s", "f"))
```

```
##  variable mean median   sd  mad   q5  q95 rhat ess_bulk ess_tail
##     alpha 0.71   0.71 0.04 0.04 0.66 0.77 1.01      321      375
##     beta  0.34   0.35 0.02 0.02 0.31 0.38 1.01      172      240
##     s     0.30   0.30 0.03 0.03 0.26 0.35 1.01      229      228
##     f     1.68   1.68 0.19 0.17 1.38 1.99 1.01      244      314
```

And the posterior distributions:

```
mcmc_dens(fit_RL_social$draws(variables = c("alpha", "beta", "s", "f")))
```



As before, our Stan model has estimated the parameter values reasonably well. If we were to run an experiment with this structure, and collect data from $N = 50$ participants, we can be reasonably confident that this Stan model will accurately estimate our participants' greediness and learning rate in reinforcement learning, and tendency to socially learn and conform.

References

Barrett, B. J., McElreath, R. L., & Perry, S. E. (2017). Pay-off-biased social learning underlies the diffusion of novel extractive foraging traditions in a wild primate. Proceedings of the Royal Society B, 284(1856), 20170358.

Deffner, D., Kleinow, V., & McElreath, R. (2020). Dynamic social learning in temporally and spatially variable environments. Royal Society Open Science, 7(12), 200734.

McElreath, R., Bell, A. V., Efferson, C., Lubell, M., Richerson, P. J., & Waring, T. (2008). Beyond existence and aiming outside the laboratory: estimating frequency-dependent and pay-off-biased social learning strategies. Philosophical Transactions of the Royal Society B, 363(1509), 3515-3528.

McElreath, R. (2020). Statistical rethinking: A Bayesian course with examples in R and Stan. Chapman and Hall/CRC.

McElreath R (2021). rethinking: Statistical Rethinking book package. R package version 2.21.

Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.

Toyokawa, W., Whalen, A., & Laland, K. N. (2019). Social learning strategies regulate the wisdom and madness of interactive crowds. Nature Human Behaviour, 3(2), 183-193.

Model 18: The evolution of social learning

Introduction

All the previous models in this series have involved purely cultural evolution. We have tracked the frequencies of different cultural traits and how they change in response to different learning biases (e.g. conformity) and demographic processes (e.g. migration). But we can also step back and ask why the capacities for cultural evolution, such as learning biases like conformity, evolved in the first place. These are models of gene-culture coevolution, assuming that there are genes for things like social learning biases, which emerge and spread as a consequence of the adaptive value of the cultural traits that they allow their bearers to acquire.

The most basic question we can ask is why social learning evolved from the presumably ancestral state of individual (asocial) learning. What are the adaptive costs and benefits of learning from others (social learning), rather than learning directly from the environment (individual learning)? A common answer to this question is that social learning evolves because individual learning is more costly than social learning. This premise makes sense, given that learning directly from the environment intuitively seems more costly than copying others. Trying out various mushrooms you find growing in the woods is surely more risky and less adaptive than eating the same mushrooms as you see your neighbour eating, assuming your neighbour isn't sick or dead.

Rogers (1988) presented a model addressing this question. This model showed that social learning does indeed evolve when it is less costly than individual learning, with social learners coexisting at equilibrium with individual learners. However, Rogers' model also showed that at this equilibrium, the mean fitness of the entire population is the same as a population composed entirely of individual learners. This contradicts the common claim that social learning (i.e. culture) is on average adaptive for individuals, populations and species. This contradiction has come to be known as "Rogers' paradox". Model 18a will recreate these results, before in Model 18b presenting a 'solution' to Rogers' paradox, i.e. a learning strategy that does increase the average fitness of the population relative to individual learning.

Model 18a: Rogers' model

Although our aim here is to recreate the findings of Rogers (1988), we will mostly follow the form of a more general version of Rogers' model presented by Enquist et al. (2007). I first describe the model verbally, then build it step by step, before putting it all inside a function.

We assume two types of agents: individual learners and social learners. Individual learners learn directly from the environment once per generation. The result of individual learning is behaviour that is either 'correct' or 'incorrect' in the current environment. With probability p_i, an individual learner identifies and adopts the correct behaviour. With probability $1 - p_i$ their behaviour is incorrect. Individual learning, whatever the outcome, entails a fitness cost of c_i.

Social learners pick an agent from the previous generation at random and copy their behaviour. If the random demonstrator had the correct behaviour in the current environment, the social learner adopts the correct behaviour. If the demonstrator had the incorrect behaviour in the current environment, the social learner adopts the incorrect behaviour. This random copying is the unbiased transmission from way back in Model 1. For simplicity, we assume that social learning is 100% accurate (i.e. $p_s = 1$ in Enquist et al. 2007). We also assume no cost to social learning (i.e. $c_s = 0$ in Enquist et al. 2007), so that whenever $c_i > 0$, individual learning is more costly than social learning consistent with our assumption from above.

To avoid negative fitnesses, we assume that all agents have a baseline fitness of 1. Agents with the correct behaviour after learning receive an additional b units of fitness. Agents with the incorrect behaviour receive no additional fitness benefit.

With probability v per generation, the environment changes such that all correct behaviours from the previous generation become incorrect. Environmental change won't affect individual learners, but it means that social learners can no longer copy the correct behaviour from the previous generation and therefore must be incorrect, whoever they copy.

After all agents have done their learning and fitnesses have been calculated, there is selection and reproduction. For simplicity we assume asexual reproduction. N new agents are created who inherit their learning strategy (individual or social) from the previous generation. Agents reproduce in proportion to their relative fitness. Agents with above average fitness are more likely to reproduce, while agents with below average fitness are less likely.

Finally there is random mutation of learning strategies in the newly created generation. With probability μ per agent, an individual learner flips to a social learner, or vice versa. We start in generation $t = 1$ with a population entirely composed of individual learners and allow mutation to introduce social learners, reflecting the assumption that individual learning is the ancestral state.

In this model we are interested in changes in the frequency of social learners over time, which we denote q_s. The frequency of individual learners is therefore $1 - q_s$. We can think of this as genetic change, with genes determining whether an agent learns individually or socially (although see Mesoudi et al. 2016 for an alternative perspective).

The first step, as always, is to define our N agents. They have a learning type, initially all 'IL' standing for 'Individual Learner', a behaviour, initially NA, and a fitness, also NA.

```r
N <- 100

# create agents, initially all individual learners
agent <- data.frame(learning = rep("IL", N),
                    behaviour = rep(NA, N),
                    fitness = rep(NA, N))

head(agent)
```

```
##   learning behaviour fitness
## 1       IL        NA      NA
## 2       IL        NA      NA
## 3       IL        NA      NA
## 4       IL        NA      NA
## 5       IL        NA      NA
## 6       IL        NA      NA
```

We also define our *output* dataframe which stores, for each timestep, the frequency of social learners, the mean fitness of social learners, and the mean fitness of individual learners, all initially NA. The frequency of individual learners and the population mean fitness can both be calculated from these values.

```r
t_max <- 500

# keep track of freq of social learners and fitnesses
output <- data.frame(SLfreq = rep(NA, t_max),
                     SLfitness = rep(NA, t_max),
                     ILfitness = rep(NA, t_max))

head(output)
```

```
##   SLfreq SLfitness ILfitness
## 1     NA        NA        NA
## 2     NA        NA        NA
## 3     NA        NA        NA
## 4     NA        NA        NA
## 5     NA        NA        NA
## 6     NA        NA        NA
```

We now start the cycle of events occurring in each generation. The first event is individual learning, with $p_i = 0.5$ such that agents have a 50% chance of discovering the correct behaviour:

```
p_i <- 0.5

# individual learning
agent$behaviour[agent$learning == "IL"] <- sample(c(1,0),
                                                  sum(agent$learning == "IL"),
                                                  replace = TRUE,
                                                  prob = c(p_i, 1 - p_i))

head(agent)
```

```
##   learning behaviour fitness
## 1       IL         1      NA
## 2       IL         1      NA
## 3       IL         0      NA
## 4       IL         1      NA
## 5       IL         0      NA
## 6       IL         0      NA
```

Here the 'correct' behaviour is denoted with a 1, and 'incorrect' behaviour with 0. The behaviour of each individual learner is set using the **sample** command, drawing a 1 with probability p_i and a 0 with probability $1-p_i$. Because we are using 1s and 0s, we can check that the proportion of correct behaviour amongst our N individual learners is roughly (but probably not exactly) p_i by taking its **mean**:

```
mean(agent$behaviour)
```

```
## [1] 0.51
```

Because there are no social learners, we will skip social learning until the next generation. Instead we get the agents' fitnesses. We use the **ifelse** command to give agents with the correct behaviour a fitness of $1+b$, i.e. the baseline 1 plus b

for being correct, and agents with the incorrect behaviour a fitness of 1, i.e. just the baseline (remember R treats a 1 as the same as TRUE and 0 as the same as FALSE). We then subtract the fitness cost of individual learning c_i from the individual learners, in this case all agents.

```
b <- 1
c_i <- 0.2

# get fitnesses
agent$fitness <- ifelse(agent$behaviour, 1 + b, 1)
agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i

head(agent)
```

```
##   learning behaviour fitness
## 1       IL         1     1.8
## 2       IL         1     1.8
## 3       IL         0     0.8
## 4       IL         1     1.8
## 5       IL         0     0.8
## 6       IL         0     0.8
```

Correct agents should have a fitness of $1 + b - c_i = 1.8$ while incorrect agents have fitness $1 - c_i = 0.8$.

Next we record the frequency of social learners, which right now will be zero; the mean fitness of social learners, which will be NA because there aren't any; and the mean fitness of individual learners. The latter will be approximately equal to the proportion of correct individual learners multiplied by the fitness of correct individual learners plus the proportion of incorrect individual learners multiplied by the fitness of incorrect individual learners, i.e. $p_i(1+b-c_i)+(1-p_i)(1-c_i) = 1.3$.

```
t <- 1

# record frequency and fitnesses
output$SLfreq[t] <- sum(agent$learning == "SL") / N
output$SLfitness[t] <- mean(agent$fitness[agent$learning == "SL"])
output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])

head(output)
```

```
##   SLfreq SLfitness ILfitness
## 1      0       NaN      1.31
## 2     NA        NA        NA
## 3      NA         NA         NA
## 4      NA         NA         NA
## 5      NA         NA         NA
## 6      NA         NA         NA
```

Next there is selection and reproduction. Similar to how we calculated relative payoffs in Model 4 (indirect bias), here we calculate relative fitness by dividing each agent's fitness by the sum of all agents' fitnesses so that they all add up to one. These are then used in a **sample** command to pick N new learning types from the current *agent* dataframe, weighted by the *relative_fitness* of each of these agents.

```
# selection and reproduction
relative_fitness <- agent$fitness / sum(agent$fitness)

agent$learning <- sample(agent$learning,
                         N,
                         prob = relative_fitness,
                         replace = TRUE)
```

Because we only have individual learners, nothing will change here. But now we introduce social learners via mutation. We set a relatively high mutation rate of $\mu = 0.1$ for demonstration purposes, meaning on average 10% of our individual learners should mutate into social learners. Similar to Model 2a (unbiased mutation), we store *agent* in *previous_agent* so that we are not mutating agents that have already mutated, then get N TRUE/FALSE values based on μ, then mutate ILs into SLs and SLs into ILs according to these TRUE/FALSE values.

```
mu <- 0.1

# mutation
previous_agent <- agent

mutate <- runif(N) < mu

agent$learning[previous_agent$learning == "IL" & mutate] <- "SL"
agent$learning[previous_agent$learning == "SL" & mutate] <- "IL"

sum(agent$learning == "SL")
```

```
## [1] 10
```

The number of social learners should be around $\mu N = 10$.

Now that we have some social learners we can implement social learning, imagining that we're now in the next generation. In the code below, first we store the current *agent* dataframe in *previous_agent* so that social learners can copy the previous generation. In the full code we would do this at the very start of the generation, before individual learning has changed behaviours. Then we check whether the environment has changed by comparing *v* against a single random number from 0 to 1 generated with **runif**. If the environment has changed, then all social learners must by assumption have the incorrect behaviour, irrespective of who they copy. If the environment hasn't changed, then we store a random selection of demonstrators from *previous_agent* in *dem*, and then give the correct behaviour to social learners who have copied a correct demonstrator, and incorrect behaviour to social learners who have copied an incorrect demonstrator (remembering again that because behaviour is stored as a 1 or 0 and R treats these as TRUE and FALSE, *dem* returns all the correct demonstrators as TRUE and incorrect demonstrators as FALSE, and !*dem* returns the reverse).

```
v <- 0.9

# store previous timestep agent
previous_agent <- agent

# social learning

# if environment has changed, all social learners have incorrect beh
if (v < runif(1)) {

  agent$behaviour[agent$learning == "SL"] <- 0 # incorrect

} else {

  # otherwise for each social learner, pick a random demonstrator from previous timestep
  # if demonstrator is correct, adopt correct beh

  dem <- sample(previous_agent$behaviour, N, replace = TRUE)

  agent$behaviour[agent$learning == "SL" & dem] <- 1 # correct
  agent$behaviour[agent$learning == "SL" & !dem] <- 0 # incorrect

}

agent[agent$learning == "SL",1:2]
```

```
##    learning behaviour
## 4        SL         1
## 7        SL         1
## 37       SL         1
## 46       SL         0
## 53       SL         0
## 72       SL         0
## 77       SL         0
## 89       SL         0
## 93       SL         1
## 96       SL         1
```

Here you should see some social learners have acquired the correct behaviour (1) and some incorrect (0), unless the environment changed in which case they're all incorrect. Rerun the code to see the different outcomes. Note that I omitted fitness from the displayed dataframe because it will be out of date, as we haven't re-calculated fitnesses based on these new behaviours.

The following function **RogersModel** puts all these bits together, with the within-generation events inside a t-loop and some default parameter values in the declaration. We use a larger $N = 10000$ to reduce stochasticity and better resemble the dynamics of Rogers' original analytic model, and a lower $\mu = 0.001$ to prevent mutation from affecting the frequencies too much. Note also that we add a column to the *output* dataframe to store the predicted fitness of individual learners given the parameter values; this will be used in the plots coming up next.

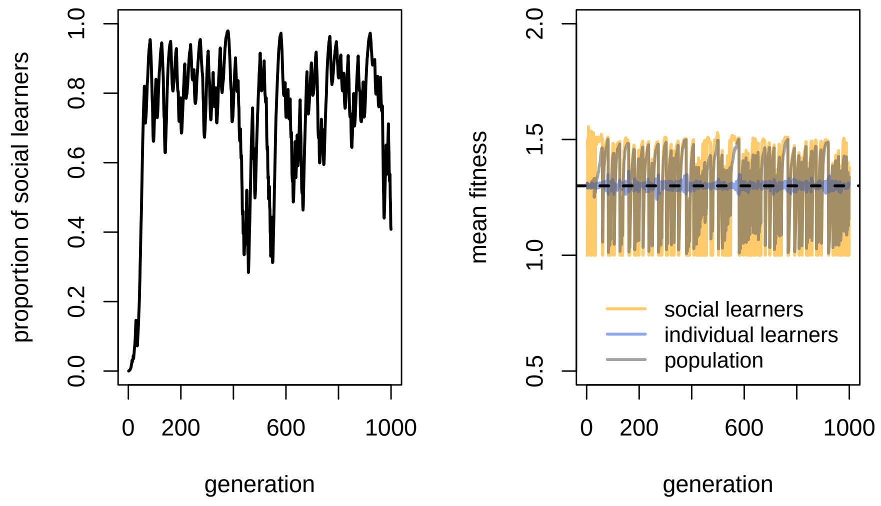
```
RogersModel <- function(N=10000, t_max=1000, p_i=0.5, b=1, c_i=0.2, mu=0.001, v=0.9) {

  # create agents, initially all individual learners
  agent <- data.frame(learning = rep("IL", N),
                      behaviour = rep(NA, N),
                      fitness = rep(NA, N))

  # keep track of freq of social learners and fitnesses
  output <- data.frame(SLfreq = rep(NA, t_max),
                       SLfitness = rep(NA, t_max),
                       ILfitness = rep(NA, t_max))

  # start timestep loop
  for (t in 1:t_max) {

    # store previous timestep agent
    previous_agent <- agent

    # individual learning:
    agent$behaviour[agent$learning == "IL"] <- sample(c(1,0),
                                                      sum(agent$learning == "IL"),
                                                      replace = TRUE,
                                                      prob = c(p_i, 1 - p_i))
    # social learning
    # if environment has changed, all social learners have incorrect beh
    if (v < runif(1)) {

      agent$behaviour[agent$learning == "SL"] <- 0 # incorrect

    } else {

      # otherwise for each social learner, pick a random demonstrator from previous timestep
      # if demonstrator is correct, adopt correct beh

      dem <- sample(previous_agent$behaviour, N, replace = TRUE)

      agent$behaviour[agent$learning == "SL" & dem] <- 1 # correct
      agent$behaviour[agent$learning == "SL" & !dem] <- 0 # incorrect

    }

    # get fitnesses
    agent$fitness <- ifelse(agent$behaviour, 1 + b, 1)
    agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i

    # record frequency and fitnesses
    output$SLfreq[t] <- sum(agent$learning == "SL") / N
    output$SLfitness[t] <- mean(agent$fitness[agent$learning == "SL"])
    output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])

    # selection and reproduction
    relative_fitness <- agent$fitness / sum(agent$fitness)

    agent$learning <- sample(agent$learning,
                             N,
                             prob = relative_fitness,
                             replace = TRUE)

    # mutation
    previous_agent <- agent
    mutate <- runif(N) < mu

    agent$learning[previous_agent$learning == "IL" & mutate] <- "SL"
    agent$learning[previous_agent$learning == "SL" & mutate] <- "IL"

  }

  # add predicted IL fitness to output and export
  output$predictedILfitness <- 1 + b*p_i - c_i
  output

}
```

We also define a plotting function **RogersPlots** to plot the proportion of social learners over time and the mean fitnesses of each type of learner, plus the mean population fitness, all obtained from the *output* generated by **RogersModel**. Note the use of the **rgb()** command with the argument *alpha* for the fitness colors. This generates transparent lines so that we can still see them all when they overlap, which they often will.

```
RogersPlots <- function(output) {

  par(mfrow=c(1,2)) # 1 row, 2 columns

  plot(output$SLfreq,
       type = 'l',
       lwd = 2,
       ylim = c(0,1),
       ylab = "proportion of social learners",
       xlab = "generation")

  # calculate population mean fitness
  POPfitness <- output$SLfreq * output$SLfitness + (1 - output$SLfreq) * output$ILfitne

  alpha=150
  plot(output$SLfitness,
       type = 'l',
       lwd = 2,
       col = rgb(255,165,0,max=255,alpha), # orange
       ylim = c(0.5,2),
       ylab = "mean fitness",
       xlab = "generation")
  lines(output$ILfitness,
        type = 'l',
        lwd = 2,
        col = rgb(65,105,225,max=255,alpha)) # royalblue
  lines(POPfitness,
        type = 'l',
        lwd = 2,
        col = rgb(100,100,100,max=255,alpha)) # grey
  abline(h = output$predictedILfitness[1],
         lty = 2,
         lwd = 2)
  legend("bottom",
       legend = c("social learners", "individual learners", "population"),
       col = c(rgb(255,165,0,max=255,alpha),
               rgb(65,105,225,max=255,alpha),
               rgb(100,100,100,max=255,alpha)),
       lty = 1,
       lwd = 2,
       bty = "n",
       cex = 0.9)

}
```

Here's one run of **RogersModel** with default parameter values:

```
data_model18a <- RogersModel()

RogersPlots(data_model18a)
```



We can see that there is a majority of social learners, fluctuating around a proportion of 0.8. There is quite a bit of fluctuation though. There is also fluctuation in the mean fitness of social learners, and consequently the mean population fitness, although the mean fitness of individual learners is fairly constant. The black dotted line shows the predicted mean fitness of individual learners, i.e. $1 + bp_i - c_i$. The blue individual learner line from the simulations matches this value quite well, barring small fluctuations because p_i is a probability rather than deterministic.

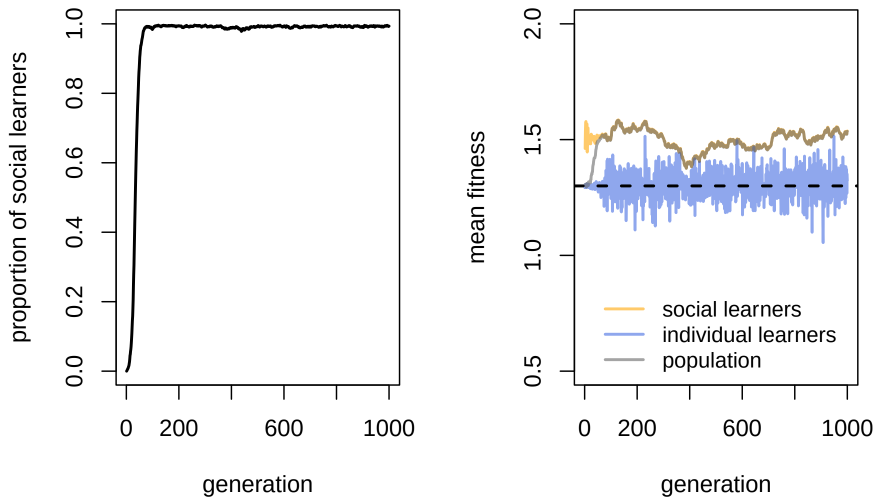
Note also that the mean fitness of social learners and the whole population also fluctuate around this value: sometimes higher, but often lower. This is Rogers' paradox. Even though we have a majority of social learners, the mean population fitness does not on average exceed the fitness of individual learners.

The reason for the fluctuations is environmental change. When the environment changes, all behaviour in the previous generation becomes incorrect. Social learners therefore copy incorrect behaviour, and are at a disadvantage compared to individual learners for whom environmental change is irrelevant. Individual learners always ignore the previous generation and sample directly from the current environment. However, once at least some individual learners have discovered the correct behaviour, social learners then have the upper hand due to the lower costs of social learning compared to individual learning. Social learners can copy the correct behaviour from the previous generation at no cost, whereas individual learners always pay a cost c_i for accuracy p_i. Social learners therefore increase in frequency, replacing individual learners. And the longer the environment remains constant, the more correct social learners there will be for new social learners to copy. That is, until the environment changes again, at which point the individual learners regain the advantage. And so on. The fluctuations in the left hand plot above are caused by this cycling back and forth between individual and social learners having the advantage.

At equilibrium, the fitness of individual and social learners must by definition be equal. Because individual learners' fitness is fixed at $1 + bp_i - c_i$, then the mean fitness of social learners must also be this value. Hence Rogers' paradox: the mean fitness of the population with social learners is no greater than that of a population entirely composed of individual learners. Culture makes no difference. The analytical appendix confirms this, and contains equations for calculating the equilibrium proportion of social learners for a given set of parameter values.

We can check this argument by manipulating our parameters. If we remove environmental change by setting $v = 1$, then according to the above logic, social learners should entirely replace individual learners. This is because social learners will acquire the correct behaviour from individual learners at the start, replace individual learners because they do not pay the cost of individual learning, and never suffer from their behaviour becoming out of date because the environment never changes.

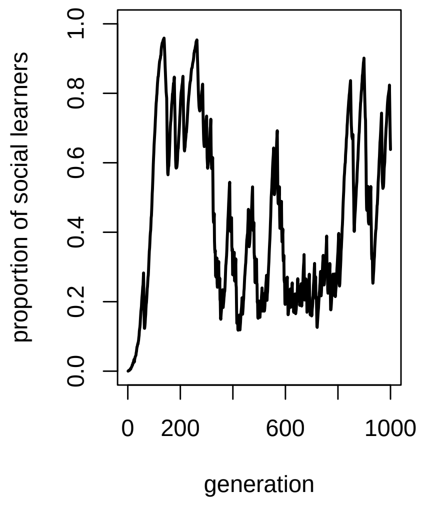
```r
data_model18a <- RogersModel(v = 1)

RogersPlots(data_model18a)
```

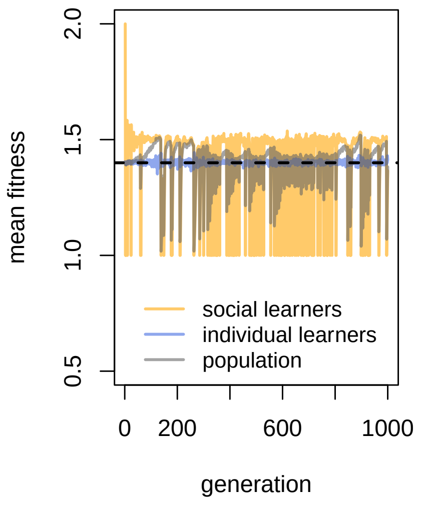


This is indeed what happens: the proportion of social learners goes almost to fixation, barring individual learners introduced each generation by mutation, and the mean fitness of social learners (and by extension the population, which is almost all social learners) clearly exceeds that of individual learners. The paradox is solved, in a way - if we are happy to assume that environments never change.

We can also check that the cost of individual learning is providing the advantage to social learners. Let's reduce this cost to $c_i = 0.1$, reintroducing the default value of environmental change:

```
data_model18a <- RogersModel(c_i = 0.1)

RogersPlots(data_model18a)
```





As predicted, the proportion of social learners has dropped to around 0.5. Because environmental change is back, Rogers' paradox is also restored.

Model 18b: Critical social learning

Rogers' model shows that social learning can evolve when it is less costly than individual learning, but that it does not increase the mean population fitness, contrary to claims that culture is adaptive. One solution to this 'paradox' is removing environmental change. However, this seems unrealistic as environments are always changing in some way.

Another solution to Rogers' paradox is to relax the unrealistic assumption that agents can either only learn individually or only learn socially. Obviously real organisms can do both. The evolution of social learning did not replace individual learning, it supplemented it. Indeed, social learning may rely on the same associative learning mechanisms as individual learning (Heyes 2012).

Enquist et al. (2007) therefore proposed a third, more plausible strategy: a critical social learner. These critical social learners learn socially first, and if the copied solution is incorrect, they then learn individually. Model 18b adds critical social learners to the simulation to see whether they do better than pure social and pure individual learners, and whether they overcome Rogers' paradox.

The following function **EnquistModel** adapts **RogersModel** to include critical learners. The parameters and their defaults are identical, as is the *agent* dataframe. The *output* dataframe now tracks the frequency and mean fitness of critical learners ("CL"s) in addition to social and individual learners. Individual learners individually learn as before, then both social and critical learners socially learn. The major new block of code implements individual learning for the incorrect critical learners. However, the actual code and fitness assignments are identical to those earlier in the function for individual learners, so there's nothing new here. The final difference is mutation, which now includes critical learners.

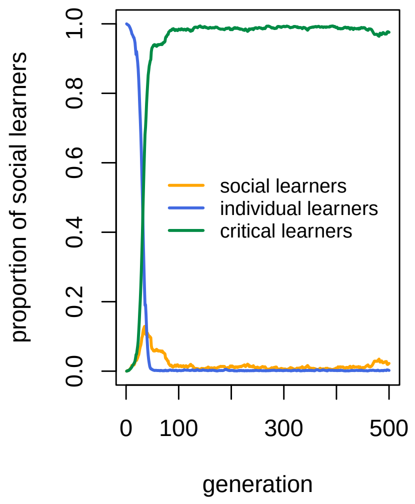
```
EnquistModel <- function(N=10000, t_max=500, p_i=0.5, b=1, c_i=0.2, mu=0.001, v=0.9) {

  # create agents, initially all individual learners
  agent <- data.frame(learning = rep("IL", N),
                      behaviour = rep(NA, N),
                      fitness = rep(NA, N))

  # track the proportion and mean fitness of each learning type
  output <- data.frame(SLfreq = rep(NA, t_max),
                       ILfreq = rep(NA, t_max),
                       CLfreq = rep(NA, t_max),
                       SLfitness = rep(NA, t_max),
                       ILfitness = rep(NA, t_max),
                       CLfitness = rep(NA, t_max))

  # start timestep loop
  for (t in 1:t_max) {

    # store previous timestep agent
    previous_agent <- agent

    # individual learning:
    agent$behaviour[agent$learning == "IL"] <- sample(c(1,0),
                                                      sum(agent$learning == "IL"),
                                                      replace = TRUE,
                                                      prob = c(p_i, 1 - p_i))

    # social learning for social and critical learners

    # if environment has changed, all social/critical learners have incorrect beh
    if (v < runif(1)) {

      agent$behaviour[agent$learning != "IL"] <- 0 # incorrect

    } else {
      # otherwise for each social learner, pick a random demonstrator from previous ti
      # if demonstrator is correct, adopt correct beh

      dem <- sample(previous_agent$behaviour, N, replace = TRUE)

      agent$behaviour[agent$learning != "IL" & dem] <- 1 # correct
      agent$behaviour[agent$learning != "IL" & !dem] <- 0 # incorrect

    }

    # get fitnesses
    agent$fitness <- ifelse(agent$behaviour, 1 + b, 1)
    agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c

    # incorrect critical learners engage in individual learning
    CL_0 <- agent$learning == "CL" & agent$behaviour == 0

    agent$behaviour[CL_0] <- sample(c(1,0),
                                    sum(CL_0),
                                    replace = TRUE,
                                    prob = c(p_i, 1-p_i))

    agent$fitness[CL_0 & agent$behaviour] <- agent$fitness[CL_0 & agent$behaviour] + b
    agent$fitness[CL_0] <- agent$fitness[CL_0] - c_i

    # record frequency and fitnesses before reproduction and mutation
    output$SLfreq[t] <- sum(agent$learning == "SL") / N
    output$ILfreq[t] <- sum(agent$learning == "IL") / N
    output$CLfreq[t] <- sum(agent$learning == "CL") / N
    output$SLfitness[t] <- mean(agent$fitness[agent$learning == "SL"])
    output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
    output$CLfitness[t] <- mean(agent$fitness[agent$learning == "CL"])

    # selection and reproduction
    relative_fitness <- agent$fitness / sum(agent$fitness)

    agent$learning <- sample(agent$learning,
                             N,
                             prob = relative_fitness,
                             replace = TRUE)

    # mutation
    mutate <- runif(N) < mu
    IL_mutate <- agent$learning == "IL" & mutate
    SL_mutate <- agent$learning == "SL" & mutate
    CL_mutate <- agent$learning == "CL" & mutate

    agent$learning[IL_mutate] <- sample(c("SL","CL"),
                                        sum(IL_mutate),
                                        replace = TRUE)

    agent$learning[SL_mutate] <- sample(c("IL","CL"),
                                        sum(SL_mutate),
                                        replace = TRUE)

    agent$learning[CL_mutate] <- sample(c("SL","IL"),
                                        sum(CL_mutate),
                                        replace = TRUE)

  }

  # add predicted IL fitness to output and export
  output$predictedILfitness <- 1 + b*p_i - c_i
  output

}
```

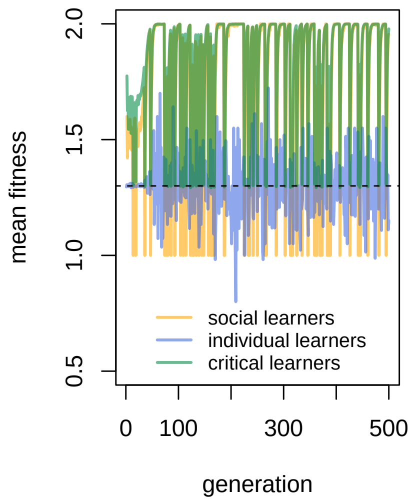
And here is a new plotting function for **EnquistModel**, plotting the frequency and mean fitness of CLs:

```
EnquistPlots <- function(output) {

  par(mfrow=c(1,2)) # 1 row, 2 columns

  # plot frequencies
  plot(output$SLfreq,
       type = 'l',
       lwd = 2,
       col = "orange",
       ylim = c(0,1),
       ylab = "proportion of social learners",
       xlab = "generation")
  lines(output$ILfreq,
        type = 'l',
        lwd = 2,
        col = "royalblue")
  lines(output$CLfreq,
        type = 'l',
        lwd = 2,
        col = "springgreen4")
  legend(x = nrow(output)/10, y = 0.6,
         legend = c("social learners", "individual learners", "critical learners"),
         col = c("orange", "royalblue", "springgreen4"),
         lty = 1,
         lwd = 2,
         bty = "n",
         cex = 0.8)

  alpha=150

  plot(output$SLfitness,
       type = 'l',
       lwd = 2,
       col = rgb(255,165,0,max=255,alpha), # orange
       ylim = c(0.5,2),
       ylab = "mean fitness",
       xlab = "generation")
  lines(output$ILfitness,
        type = 'l',
        lwd = 2,
        col = rgb(65,105,225,max=255,alpha)) # royalblue
  lines(output$CLfitness,
        type = 'l',
        lwd = 2,
        col = rgb(0,139,69,max=255,alpha)) # springgreen4
  abline(h = output$predictedILfitness[1],
         lty = 2)
  legend("bottom",
       legend = c("social learners", "individual learners", "critical learners"),
       col = c(rgb(255,165,0,max=255,alpha),
               rgb(65,105,225,max=255,alpha),
               rgb(0,139,69,max=255,alpha)),
       lty = 1,
       lwd = 2,
       bty = "n",
       cex = 0.8)

}
```

We can now run **EnquistModel** with default parameter values:

```
data_model18b <- EnquistModel()

EnquistPlots(data_model18b)
```





The left hand plot clearly shows that critical learners almost dominate the population, barring the occasional other types introduced each generation by mutation. The right hand plot shows why: the fitness of individual learners hovers around the expected value shown by the dotted line, the fitness of social learners fluctuates both above and below the dotted line depending on whether the environment has changed recently, while the fitness of critical learners also fluctuates but never drops below the dotted line. Critical learners have the best of both worlds: they can copy previous solutions at lower cost than individual learning when the environment is stable, but when the environment changes, they can detect that change and discover the new correct behaviour.

Summary

Rogers' model, and even Enquist et al's extension, are hugely unrealistic in many ways. Despite this, these models give valuable insights into the evolutionary advantages and disadvantages of social learning relative to individual learning. Social learning benefits from being less costly than individual learning. Yet social learning does not respond well to environmental change. When environments change, social learners are left copying each other's outdated incorrect behaviour. Individual learners suffer no such penalty when the environment changes. This trade off can lead to a mixed equilibrium where individual and social learners coexist.

Rogers' model resembles producer-scrounger models from ecology (Barnard & Sibly 1992). In a feeding context, 'producer' birds discover new sources of food, while 'scrounger' birds follow the producers and exploit the food sources they find. In Rogers' model, individual learners are 'information producers' discovering new cultural traits, while social learners are 'information scroungers' copying the producers' cultural traits without paying the cost.

In a feeding context it seems fairly obvious that scroungers do not contribute anything to the overall mean fitness of the group, population or species. Yet it wasn't until Rogers' model that the same was recognised for social learning. Mixed populations of social and individual learners have no greater mean fitness than populations entirely composed of individual learners. This came to be known as Rogers' Paradox, given that social learning and culture are often thought of as the secret of our species' evolutionary success. In Rogers' model, this is wrong.

However, models are only as good as their assumptions, and one major assumption of Rogers' model is that individuals can perform only one of the two kinds of learning. Real organisms do both. Hence Enquist et al. (2007), recreated here in Model 18b, showed that a 'critical learner' strategy of first learning socially and, if this results in incorrect behaviour, then engaging in individual learning, outperforms and replaces both pure individual learning and pure social learning. This might seem obvious in hindsight but was not fully recognised until Rogers' simple model and its extensions.

We should therefore expect cultural species to be composed of individuals who mostly learn socially early in life, then supplement this with individual learning later on. This seems to be the pattern in humans, where children are powerful, almost compulsive imitators (Lyons et al. 2007), while adolescents and adults do more experimentation and try to improve what they've learned during childhood.

Critical social learning can also be seen as supporting a form of cumulative culture. As critical learners increase in frequency, each new generation contains more and more correct individuals. New critical learners can copy these accumulated correct behaviours for free. That is, until the environment changes, at which point the accumulation begins again.

Enquist et al.'s model is still extremely simple. Real populations learn and accumulate multiple traits, not just one. Environmental change rarely causes all knowledge to be wiped out requiring cultural evolution to start from scratch. Organisms have more complex developmental 'learning schedules', undergoing multiple bouts of social and individual learning during their lifetimes. And there is no spatial or social structure in these models. Recent work has addressed some of these limitations (e.g. Rendell et al. 2010; Lehmann et al. 2013), but others remain to be modelled, and all require further empirical testing.

In terms of modelling techniques, Model 18 showed how to model the evolution of a capacity for cultural evolution, rather than just the cultural evolution of a socially learned trait. We specified the fitness costs and benefits of each learning strategy under different conditions, implemented selection and reproduction based on differential fitness, as well as the mutation of strategies. We then see which learning strategy, or mix of strategies, constitute an evolutionarily stable strategy after multiple generations for a given set of parameters. If we assume that the learning strategy is genetically specified, then this is a gene-culture coevolution model, where genetic strategies and cultural traits coevolve. The analytic appendix below shows how to derive the exact evolutionarily stable mix of strategies for specific parameter values, confirming and clarifying our simulation results.

Exercises

1. In **RogersModel**, what should happen to the proportion of social learners and the mean fitnesses when $v = 0$, i.e. the environment changes every generation? Run the simulation to check your predictions.
2. Add a cost to social learning, c_s, and an accuracy of social learning, p_s, to **RogersModel**. What effect do these variables have on the dynamics of the model?
3. Add multiple runs to both **RogersModel** and **EnquistModel**, and plot the mean frequencies and fitnesses across all r_{max} runs. Does this show Rogers' paradox as clearly as just plotting one run, as shown above?
4. Enquist et al. (2007) also modelled a 'conditional social learner' who learns individually first and if incorrect learns socially - the reverse of critical social learners. Add conditional social learners to **EnquistModel**. How do conditional learners fare against critical learners and individual learners?

Analytical Appendix

We start with Model 18a, containing just individual and social learners. Following Enquist et al. (2007) we can derive expressions for the equilibrium frequency of social learning, and the fitnesses of social and individual learning. Note that we omit their c_s and p_s for simplicity, and to match the simulations above.

As already specified above, the fitness of individual learners, w_i, is given by:

$$w_i = 1 + bp_i - c_i \tag{18.1}$$

where the benefit b of correct behaviour is received with probability p_i, minus the cost of individual learning c_i.

For social learners, the benefit b is received when (i) a correct demonstrator from the previous generation is copied, with probability equal to the proportion of correct behaviour amongst all agents in the previous generation, denoted q_{t-1}, as long as (ii) the environment remains the same, which happens with probability v. The fitness of social learners, w_s, is therefore:

$$w_s = 1 + bq_{t-1}v \tag{18.2}$$

If q_i and q_s are the proportions of individual and social learners respectively in generation t, then the frequency of correct behaviour in generation t, q_t, is given by:

$$q_t = q_i p_i + q_s q_{t-1} v \tag{18.3}$$

At equilibrium, $q_t = q_{t-1} = q^*$, and we know that $q_i = 1 - q_s$, so:

$$q^* = (1 - q_s)p_i + q_s q^* v \tag{18.4}$$

Rearranging Equation 18.4 gives:

$$q^* = \frac{(1 - q_s)p_i}{1 - q_s v} \tag{18.5}$$

Replacing q_{t-1} in Eq 18.2 with q^* gives:

$$w_s = 1 + \frac{(1 - q_s)bp_i v}{1 - q_s v} \tag{18.6}$$

At equilibrium we also know that $w_i = w_s$. Setting 18.1 equal to 18.6 and rearranging for q_s gives the frequency of social learners at equilibrium:

$$q_s^* = \frac{c_i - bp_i(1 - v)}{c_i v} \tag{18.7}$$

For the default parameter values of **RogersModel**, q_s^* is

```
p_i <- 0.5
b <- 1
c_i <- 0.2
v <- 0.9

(c_i - b * p_i * (1 - v)) / (c_i * v)
```

```
## [1] 0.8333333
```

which is roughly the proportion found in the simulation (although given the fluctuations it's difficult to tell, which is one advantage of analytical models over simulations).

The next simulation used $v = 1$. From Equation 18.7 we can see that $v = 1$ always yields $q_s^* = c_i/c_i = 1$, i.e. 100% social learners, as found in the simulation (barring mutation, which is not included in the analytical model).

Finally we simulated $c_i = 0.1$, which gives:

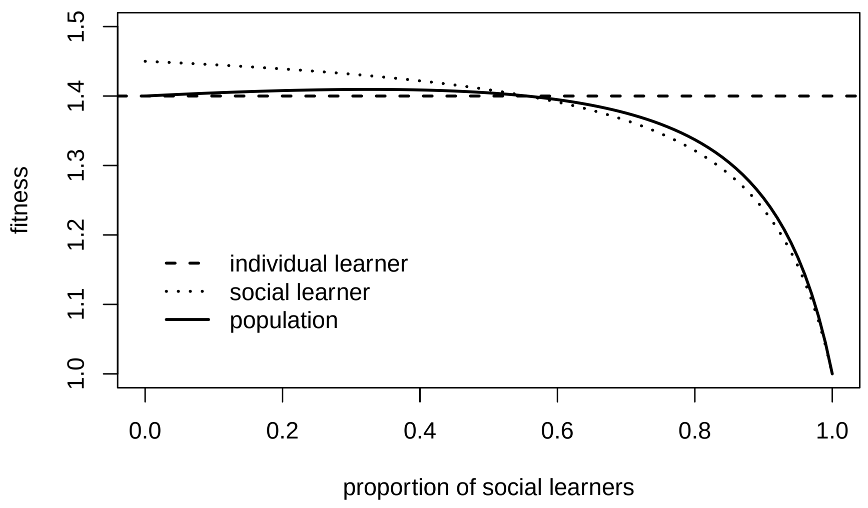
```
c_i <- 0.1

(c_i - b * p_i * (1 - v)) / (c_i * v)
```

```
## [1] 0.5555556
```

which again matches the final simulation plot showing that roughly half the agents are social learners (again, with lots of fluctuation).

We can use Equations 18.1 and 18.6 to create a plot that encapsulates Rogers' paradox, showing the mean fitness of individual learners, social learners, and the entire population as a function of the proportion of social learners, q_s.

```
# parameter values
p_i <- 0.5
b <- 1
c_i <- 0.1
v <- 0.9

# x-axis, frequency of social learners, q_s
q_s <- seq(0, 1, 0.01)

# fitness of ILers, w_i (Eq 18.1)
w_i <- 1 + b*p_i - c_i

# fitness of SLers, w_s (Eq 18.6)
w_s <- 1 + ((1 - q_s)*b*p_i*v)/ (1 - q_s*v)

# mean population fitness
w <- w_s*q_s + w_i*(1 - q_s)

plot(x = q_s,
     y = w_s,
     type = 'l',
     lwd = 2,
     xlab = "proportion of social learners",
     ylab = "fitness",
     ylim = c(1,1.5),
     lty = 3)
abline(h = w_i,
       lty = 2,
       lwd = 2)
lines(x = q_s,
      y = w,
      lty = 1,
      lwd = 2)
legend(x = 0,
       y = 1.2,
       legend = c("individual learner", "social learner", "population"),
       lty = c(2, 3, 1),
       bty = "n",
       lwd = 2)
```



This qualitatively recreates Fig 1 in Rogers (1988) and Fig 1b in Enquist et al. (2007). The point where the three lines cross is the equilibrium frequency of social learners, q^*, from above, where the fitness of individual learners and social learners is equal.

The figure above shows that the fitness of individual learners does not change with the proportion of social learners. It depends on the fixed parameters b, p_i and c_i. The fitness of social learners, however, does change, with social learners doing well when relatively uncommon ($q_s < q^*$), and poorly when common ($q_s > q^*$). We can say that the fitness of social learners is *frequency-dependent*, i.e. its fitness depends at least partly on its own frequency in the population.

As noted above, the underlying reason for this frequency dependence is the balance between environmental change favouring individual learning and costly individual learning favouring social learning. At the extreme left of the graph, when $q_s = 0$, a single social learner will have fitness $w_s = 1 + bp_i v$, compared to $w_i = 1 + bp_i - c_i$. Hence the social learner will get the benefit b of copying the correct behaviour from the previous generation of individual learners with probability p_i, assuming the environment has stayed the same (with probability v) and without paying the cost c_i. Hence the advantage of social learning is greater when v and c_i are both larger. As we move to the right of the plot, increasing numbers of social learners means that social learners will be increasingly copying other social learners. When the environment changes, there will be fewer individual learners to discover the correct behaviour, and social learners will be increasingly disadvantaged. When $q_s > q^*$, this disadvantage outweighs the benefits of not paying the costs of individual learning, and individual learners do better.

We now add critical social learners. Their fitness, w_c, is given by the fitness of social learners w_s plus the fitness of individual learners w_i weighted by the probability that social learners are incorrect, $(1 - v)$:

$$w_c = w_s + w_i(1 - v) = 1 + bq_{t-1}v + (bp_i - c_i)(1 - v) \tag{18.8}$$

The proportion of correct behaviour in the current generation q_t, previously given in Eq 18.3, must now contain an additional term for the frequency of correct behaviour amongst critical learners, q_c:

$$q_t = q_i p_i + q_s q_{t-1} v + q_c(q_{t-1}v + (1 - q_{t-1}v)p_i) \tag{18.9}$$

Setting $q_t = q_{t-1} = q^*$ and $q_i = 1 - q_s - q_c$, after rearranging we get:

$$q^* = \frac{p_i(1 - q_s)}{(1 - v(q_s + q_c(1 - p_i))} \tag{18.10}$$

At equilibrium, critical learners must have higher fitness than social learners if individual learning is adaptive (i.e. from Eq 18.8, when $w_i > 0$, assuming $v < 1$). Hence at equilibrium $q_s = 0$, such that Eq 18.10 simplifies to:

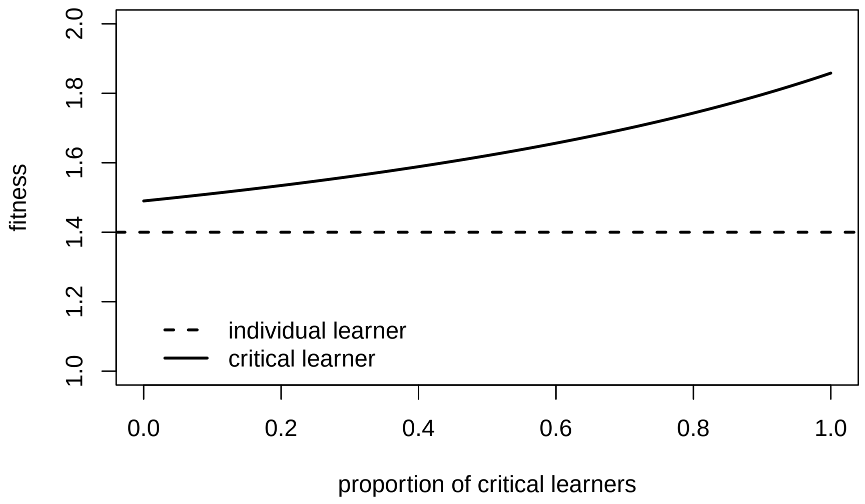
$$q^* = \frac{p_i}{1 - vq_c(1 - p_i)} \tag{18.11}$$

Substituting Eq 18.11 into Eq 18.8 with $q_{t-1} = q^*$ gives:

$$w_c = 1 + \frac{bvp_i}{1 - vq_c(1 - p_i)} + (bp_i - c_i)(1 - v) \tag{18.12}$$

This allows us to plot the fitness of critical learners w_c against the proportion of critical learners q_c, alongside the fixed fitness of individual learners, similar to what we did with social learners in the previous plot:

```r
# parameter values
p_i <- 0.5
b <- 1
c_i <- 0.1
v <- 0.9

# x-axis, frequency of critical learners, q_c
q_c <- seq(0, 1, 0.01)

# fitness of ILers, w_i (Eq 18.1)
w_i <- 1 + b*p_i - c_i

# fitness of CLers, w_s (Eq 18.12)
w_c <- 1 + (b*v*p_i) / (1 - v*q_c*(1 - p_i)) + (b*p_i - c_i)*(1 - v)

plot(x = q_c,
     y = w_c,
     type = 'l',
     lwd = 2,
     xlab = "proportion of critical learners",
     ylab = "fitness",
     ylim = c(1,2),
     lty = 1)
abline(h = w_i,
       lty = 2,
       lwd = 2)
legend(x = 0,
       y = 1.2,
       legend = c("individual learner", "critical learner"),
       lty = c(2, 1),
       bty = "n",
       lwd = 2)
```



This illustrates the evolutionary advantage of critical learners. Whereas the fitness of social learners decreased with their frequency, the fitness of critical learners increases with their frequency, and is always greater than that of individual learners.

This is again due to the combined action of environmental change and costly individual learning. At the extreme left, when $q_c = 0$, Eq 18.12 tells us that a single critical learner will have fitness $w_c = 1 + bp_i - c_i(1 - v)$. This is identical to the fitness of individual learners except the final term c_i is multiplied by $(1 - v)$, the probability of the environment changing. This reflects the fact that critical learners only have to pay the cost of individual learning when the environment has changed, otherwise they can copy the correct solution from the previous generation. At $q_c = 0$, the previous generation will be composed of individual learners, p_i of whom will have the correct solution, hence the bp_i term. As q_c increases, critical learners will be increasingly likely to copy other critical learners. The longer the environment remains constant, the more likely those critical learners will be correct. Hence as q_c increases, critical learners are increasingly able to copy the correct behaviour for free, compared to individual learners who must always pay a cost c_i for accuracy p_i.

References

Barnard, C. J., & Sibly, R. M. (1981). Producers and scroungers: a general model and its application to captive flocks of house sparrows. Animal Behaviour, 29(2), 543-550.

Enquist, M., Eriksson, K., & Ghirlanda, S. (2007). Critical social learning: a solution to Rogers's paradox of nonadaptive culture. American Anthropologist, 109(4), 727-734.

Heyes, C. (2012). What's social about social learning?. Journal of Comparative Psychology, 126(2), 193-202.

Lehmann, L., Wakano, J. Y., & Aoki, K. (2013). On optimal learning schedules and the marginal value of cumulative cultural evolution. Evolution, 67(5), 1435-1445.

Lyons, D. E., Young, A. G., & Keil, F. C. (2007). The hidden structure of overimitation. Proceedings of the National Academy of Sciences, 104(50), 19751-19756.

Mesoudi, A., Chang, L., Dall, S. R., & Thornton, A. (2016). The evolution of individual and cultural variation in social learning. Trends in Ecology & Evolution, 31(3), 215-225.

Rendell, L., Fogarty, L., & Laland, K. N. (2009). Rogers' paradox recast and resolved: population structure and the evolution of social learning strategies. Evolution, 64(2), 534-548.

Rogers, A. R. (1988). Does biology constrain culture?. American Anthropologist, 90(4), 819-831.

Model 19: The evolution of social learning strategies

Introduction

Model 18b showed that social learning can readily evolve when combined with individual learning. So-called 'critical learners' who first try to socially learn behaviour from another agent, and if that behaviour is incorrect then try individual learning, out-compete both pure social learners who only copy others and pure individual learners who never copy (Enquist et al. 2007). But in Model 18, social learning took the form of unbiased transmission. Social learners picked a random agent from the previous generation and copied their behaviour. This is, of course, quite unrealistic.

In Model 19 we will therefore look at the evolution of the more sophisticated social learning strategies (Laland 2004) that we have covered previously: *directly biased transmission* from Model 3, where agents preferentially copy traits based on their characteristics, in this case their payoffs (hence 'payoff bias' for short), and *conformist biased transmission* from Model 5, where agents preferentially copy the most common trait amongst a set of demonstrators irrespective of trait characteristics. While those previous models examined the cultural dynamics that each of these transmission biases or strategies generate, here we are interested in when and why a capacity or tendency to employ such strategies evolve in the first place.

Previous modelling work (e.g. Nakahashi et al. 2012) has suggested that the answer to this question partly depends on whether environments vary *temporally*, as in Model 18, where the optimal behaviour occasionally changes over time, or *spatially*, as in the migration model of Model 7, where different groups inhabit different environments in which different behaviours are optimal. Hence Model 19a assumes temporally varying environments and Model 19b assumes spatially varying environments. The driving question in both is: when and why do payoff and conformist biased social learning strategies out-compete unbiased transmission?

Model 19a: Temporally varying environments

Here we add social learning strategies to the critical learning strategy of Model 18b, assuming that critical learning has already evolved. As in Model 18b, there is one behaviour that is 'correct' and gives higher fitness of $1 + b$ compared to an 'incorrect' behaviour that just gives the baseline fitness of 1. We keep pure individual learners (IL) for reference and to populate the first generation. As before, individual learners discover the correct behaviour in each generation with probability p_i at cost c_i. We remove pure social learners because we know from Model 18 that critical learning is always superior.

Now critical learners employ one of three strategies: unbiased transmission (UB), conformist bias (CB) or payoff bias (PB).

Critical learners employing unbiased learning (UB) copy a member of the previous generation at random, as per Model 18b. There is no cost to UB.

Critical learners employing conformist bias (CB) are disproportionately more likely to copy the majority trait amongst n randomly chosen demonstrators as in Model 5. However unlike Model 5, we use the formulation of conformity introduced in Model 17c (and also used by Nakahashi et al. 2012). This formulation is more flexible than the one in Model 5 as it can handle more than two traits, which will be needed in Model 19b. As a reminder, Equation 17.2 specified the probability of adopting trait i, p_i, as

$$p_i = \frac{n_i^f}{\sum_{j=1}^{k} n_j^f}$$

where n_i is the number of the n demonstrators who have trait i, the exponent f controls the strength of conformity, and the denominator gives the sum of the number of agents who have each of k traits, each raised to the power f. When $f = 1$ transmission is unbiased, as the equation gives the exact proportion of agents with trait i. As f increases, CB agents are increasingly likely to pick the most-common trait. Whether the behaviour is correct or incorrect does not matter for conformity. It works purely on behaviour frequency. CB agents bear a fixed cost c_c representing the time and energy required to identify the majority behaviour amongst n demonstrators.

Critical learners employing payoff bias (PB) copy the correct behaviour if it is exhibited by any of n randomly chosen demonstrators, otherwise they copy the incorrect behaviour. This version of directly biased transmission resembles horizontal transmission from Model 6c (with $s_h = 1$, i.e. the correct behaviour is always copied successfully; this reduces the number of parameters in this more-complex model). As noted in Model 6, there are many ways of implementing direct bias, but this one nicely mirrors conformist bias in that it involves picking from n demonstrators. Note that this payoff-based direct bias is different to the payoff-based indirect bias from Model 4a: the former involves copying traits based on payoffs, while the latter involves copying individuals based on payoffs. Here, PB agents are copying traits/behaviours. PB agents bear a fixed cost c_p representing the time and energy required to identify the correct behaviour amongst n demonstrators.

Now for the model. First we create our N agents, the same as in Model 18b, all initially individual learners (IL):

```
N <- 1000

# create agents, initially all individual learners
agent <- data.frame(learning = rep("IL", N),
                    behaviour = rep(NA, N),
                    fitness = rep(NA, N))

head(agent)
```

```
##   learning behaviour fitness
## 1       IL        NA      NA
## 2       IL        NA      NA
## 3       IL        NA      NA
## 4       IL        NA      NA
## 5       IL        NA      NA
## 6       IL        NA      NA
```

Next we create the *output* dataframe, this time holding the frequency and mean fitness of all four types of agents:

```
t_max <- 500

# keep track of freq and fitnesses of each type
output <- data.frame(ILfreq = rep(NA, t_max),
                     UBfreq = rep(NA, t_max),
                     PBfreq = rep(NA, t_max),
                     CBfreq = rep(NA, t_max),
                     ILfitness = rep(NA, t_max),
                     UBfitness = rep(NA, t_max),
                     PBfitness = rep(NA, t_max),
                     CBfitness = rep(NA, t_max))

head(output)
```

```
##   ILfreq UBfreq PBfreq CBfreq ILfitness UBfitness PBfitness CBfitness
## 1     NA     NA     NA     NA        NA        NA        NA        NA
## 2     NA     NA     NA     NA        NA        NA        NA        NA
## 3     NA     NA     NA     NA        NA        NA        NA        NA
## 4     NA     NA     NA     NA        NA        NA        NA        NA
## 5     NA     NA     NA     NA        NA        NA        NA        NA
## 6     NA     NA     NA     NA        NA        NA        NA        NA
```

Before getting to the main code, let's simulate the first generation of individual learning to provide the critical learners with demonstrators in generation two onwards. The following code, all familiar from Model 18, runs individual learning and calculates fitnesses:

```
t <- 1
p_i <- 0.5
b <- 1
c_i <- 0.2

#individual learning for ILs
agent$behaviour[agent$learning == "IL"] <- sample(c(1,0),
                                                  sum(agent$learning == "IL"),
                                                  replace = TRUE,
                                                  prob = c(p_i, 1 - p_i))

# get fitnesses
agent$fitness <- ifelse(agent$behaviour, 1 + b, 1)
agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i

head(agent)
```

```
##   learning behaviour fitness
## 1       IL         1     1.8
## 2       IL         1     1.8
## 3       IL         1     1.8
## 4       IL         0     0.8
## 5       IL         1     1.8
## 6       IL         0     0.8
```

And now we record frequencies and mean fitnesses of each learning type, now including the (currently absent) UB, PB and CB agents:

```
# record frequencies and fitnesses
output$ILfreq[t] <- sum(agent$learning == "IL") / N
output$UBfreq[t] <- sum(agent$learning == "UB") / N
output$PBfreq[t] <- sum(agent$learning == "PB") / N
output$CBfreq[t] <- sum(agent$learning == "CB") / N
output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
```

```
output$UBfitness[t] <- mean(agent$fitness[agent$learning == "UB"])
output$PBfitness[t] <- mean(agent$fitness[agent$learning == "PB"])
output$CBfitness[t] <- mean(agent$fitness[agent$learning == "CB"])

head(output)
```

```
##   ILfreq UBfreq PBfreq CBfreq ILfitness UBfitness PBfitness CBfitness
## 1      1      0      0      0     1.305       NaN       NaN       NaN
## 2     NA     NA     NA     NA        NA        NA        NA        NA
## 3     NA     NA     NA     NA        NA        NA        NA        NA
## 4     NA     NA     NA     NA        NA        NA        NA        NA
## 5     NA     NA     NA     NA        NA        NA        NA        NA
## 6     NA     NA     NA     NA        NA        NA        NA        NA
```

And finally mutation, with an unrealistically high mutation rate for demonstration purposes:

```
mu <- 0.5

# mutation
mutate <- runif(N) < mu
IL_mutate <- agent$learning == "IL" & mutate
UB_mutate <- agent$learning == "UB" & mutate
PB_mutate <- agent$learning == "PB" & mutate
CB_mutate <- agent$learning == "CB" & mutate

agent$learning[IL_mutate] <- sample(c("UB","PB","CB"), sum(IL_mutate), replace = TRUE)
agent$learning[UB_mutate] <- sample(c("IL","PB","CB"), sum(UB_mutate), replace = TRUE)
agent$learning[PB_mutate] <- sample(c("IL","UB","CB"), sum(PB_mutate), replace = TRUE)
agent$learning[CB_mutate] <- sample(c("IL","UB","PB"), sum(CB_mutate), replace = TRUE)

head(agent[,1:2], 10)
```

```
##    learning behaviour
## 1        IL         1
## 2        CB         1
## 3        PB         1
## 4        UB         0
## 5        PB         1
## 6        CB         0
## 7        IL         1
## 8        UB         1
## 9        IL         0
## 10       IL         1
```

Now we can start the full sequence of events that occur in each generation at $t = 2$. First we put *agent* into *previous_agent* so that critical learners can copy the previous unaltered generation. Then individual learners engage in individual learning:

```
t <- 2

# 1. store previous timestep agent
previous_agent <- agent

# 2. individual learning for ILs
agent$behaviour[agent$learning == "IL"] <- sample(c(1,0),
                                                  sum(agent$learning == "IL"),
                                                  replace = TRUE,
                                                  prob = c(p_i, 1 - p_i))
```

Next is social learning. As in Model 18, with probability v the environment changes meaning that all critical learners must copy the incorrect behaviour. Otherwise, agents engage in social learning, as explored next.

```
v <- 0.9

# 3. social learning for critical learners

# if environment has changed, all social learners have incorrect beh
if (v < runif(1)) {

  agent$behaviour[agent$learning != "IL"] <- 0 # incorrect

} else {

  # social learning code

}
```

```
## NULL
```

Now to fill in the social learning code. First we pick n random demonstrators from the previous generation for each agent and store them in a matrix called *dems*:

```
n <- 3

# create matrix for holding n demonstrators for N agents
# fill with randomly selected agents from previous gen
dems <- matrix(data = sample(previous_agent$behaviour, N*n, replace = TRUE),
               nrow = N, ncol = n)

head(dems)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    1
## [2,]    0    0    1
## [3,]    1    0    0
## [4,]    0    0    0
## [5,]    0    0    1
## [6,]    0    1    1
```

Each row of *dems* contains the behaviours (1 indicating correct and 0 indicating incorrect) of n randomly chosen demonstrators for one agent.

UB agents then copy the behaviour of the first of these demonstrators (in column 1 of *dems*, i.e. *dems[,1]*), ignoring the rest. For illustration, we calculate the proportion correct behaviours before and after UB.

```
# proportion correct before social learning
mean(agent$behaviour[agent$learning == "UB"])
```

```
## [1] 0.4611111
```

```
# for UBs, copy the behaviour of the 1st dem in dems
agent$behaviour[agent$learning == "UB" & dems[,1]] <- 1 # correct
agent$behaviour[agent$learning == "UB" & !dems[,1]] <- 0 # incorrect

# proportion correct after social learning
mean(agent$behaviour[agent$learning == "UB"])
```

```
## [1] 0.5055556
```

There is probably not much change here, given that unbiased transmission is random.

Next, PB agents copy the correct behaviour if at least one demonstrator is correct, as calculated using **rowSums()** > 0. Otherwise they adopt the incorrect behaviour.

```
# proportion correct before social learning
mean(agent$behaviour[agent$learning == "PB"])
```

```
## [1] 0.487013
```

```
# for PB, copy correct if at least one of dems is correct
agent$behaviour[agent$learning == "PB" & rowSums(dems) > 0] <- 1 # correct
agent$behaviour[agent$learning == "PB" & rowSums(dems) == 0] <- 0 # incorrect

# proportion correct after social learning
mean(agent$behaviour[agent$learning == "PB"])
```

```
## [1] 0.8571429
```

There are many more correct PB agents after payoff bias than before, given that payoff bias favours correct behaviour.

Finally, CB agents get the probability of copying the correct behaviour from the n demonstrators according to Equation 17.2 above, then copy the correct behaviour with this probability, otherwise adopting the incorrect behaviour. Note that because there are only two behaviours (correct and incorrect), the denominator of Equation 17.2 is the sum of the number of correct demonstrators *CB_rowSums* raised to the power f, plus the sum of the number incorrect, which will be n - *CB_rowSums*, also raised to the power f. And because there are only two behaviours, we only need to calculate the probability of one of them.

```
f <- 3

# proportion correct before social learning
mean(agent$behaviour[agent$learning == "CB"])
```

```
## [1] 0.5185185
```

```
# for CB, copy majority behaviour according to parameter f
copy_probs <- rowSums(dems)^f / (rowSums(dems)^f + (n - rowSums(dems))^f)
probs <- runif(N)
agent$behaviour[agent$learning == "CB" & probs < copy_probs] <- 1 # correct
agent$behaviour[agent$learning == "CB" & probs >= copy_probs] <- 0 # incorrect

# proportion correct after social learning
mean(agent$behaviour[agent$learning == "CB"])
```

```
## [1] 0.462963
```

Because conformity is blind to payoffs, and in this second generation there won't be a majority of correct behaviours, the frequency of correct behaviour amongst CB agents is unlikely to have increased much, if at all.

Now that social learning is complete, all incorrect social learners engage in individual learning as per the critical learner strategy (see Model 18b):

```
# 4. incorrect social learners engage in individual learning
incorrect_SL <- agent$learning != "IL" & agent$behaviour == 0

agent$behaviour[incorrect_SL] <- sample(c(1,0),
                                        sum(incorrect_SL),
                                        replace = TRUE,
                                        prob = c(p_i, 1-p_i))
```

Next we calculate fitnesses, this time including the fitness costs to PB and CB, as well as the cost of individual learning for incorrect social learners:

```
c_p <- 0.02
c_c <- 0.02

# 5. calculate fitnesses

# all correct agents get bonus b
agent$fitness <- ifelse(agent$behaviour, 1 + b, 1)

# costs of learning
agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i
agent$fitness[agent$learning == "PB"] <- agent$fitness[agent$learning == "PB"] - c_p
agent$fitness[agent$learning == "CB"] <- agent$fitness[agent$learning == "CB"] - c_c
agent$fitness[incorrect_SL] <- agent$fitness[incorrect_SL] - c_i

head(agent, 10)
```

```
##    learning behaviour fitness
## 1        IL         1    1.80
## 2        CB         0    0.78
## 3        PB         1    1.98
## 4        UB         1    1.80
## 5        PB         1    1.98
## 6        CB         1    1.98
## 7        IL         0    0.80
## 8        UB         0    0.80
## 9        IL         0    0.80
## 10       IL         0    0.80
```

There should be a range of fitnesses here, depending on the agent learning type, whether they are correct or incorrect, and whether critical learners engaged in individual learning.

Step 6 involves recording frequencies and mean fitnesses of each agent type, as before:

```
# 6. record frequencies and fitnesses of each type
output$ILfreq[t] <- sum(agent$learning == "IL") / N
output$UBfreq[t] <- sum(agent$learning == "UB") / N
output$PBfreq[t] <- sum(agent$learning == "PB") / N
output$CBfreq[t] <- sum(agent$learning == "CB") / N
output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
output$UBfitness[t] <- mean(agent$fitness[agent$learning == "UB"])
output$PBfitness[t] <- mean(agent$fitness[agent$learning == "PB"])
output$CBfitness[t] <- mean(agent$fitness[agent$learning == "CB"])

head(output)
```

```
##   ILfreq UBfreq PBfreq CBfreq ILfitness UBfitness PBfitness CBfitness
## 1  1.000   0.00  0.000  0.000  1.305000       NaN       NaN       NaN
## 2  0.504   0.18  0.154  0.162  1.315873  1.684444  1.886494  1.644198
## 3     NA     NA     NA     NA        NA        NA        NA        NA
## 4     NA     NA     NA     NA        NA        NA        NA        NA
## 5     NA     NA     NA     NA        NA        NA        NA        NA
## 6     NA     NA     NA     NA        NA        NA        NA        NA
```

The *output* dataframe shows that IL agents drop sharply in frequency, replaced with roughly equal proportions of the other agent types. Consistent with this, all critical learners have higher fitness than IL agents. And as hinted at above, PB agents have slightly higher mean fitness than UB and CB agents.

Step 7 is selection and reproduction. Agents reproduce their learning type with probability equal to their relative fitness. We can use **table()** to show the proportions of each type before and after selection.

```
# before selection
table(agent$learning)/N
```

```
## 
##    CB    IL    PB    UB 
## 0.162 0.504 0.154 0.180
```

```
 # 7. selection and reproduction
relative_fitness <- agent$fitness / sum(agent$fitness)

agent$learning <- sample(agent$learning,
                         N,
                         prob = relative_fitness,
                         replace = TRUE)

# after selection
table(agent$learning)/N
```

```
##
##    CB    IL    PB    UB
## 0.182 0.421 0.193 0.204
```

Reflecting their fitness differences, IL agents probably dropped in frequency while PB agents increased.

The final within-generation step is mutation, which we have already seen:

```
# 8. mutation
mutate <- runif(N) < mu
IL_mutate <- agent$learning == "IL" & mutate
UB_mutate <- agent$learning == "UB" & mutate
PB_mutate <- agent$learning == "PB" & mutate
CB_mutate <- agent$learning == "CB" & mutate

agent$learning[IL_mutate] <- sample(c("UB","PB","CB"), sum(IL_mutate), replace = TRUE)
agent$learning[UB_mutate] <- sample(c("IL","PB","CB"), sum(UB_mutate), replace = TRUE)
agent$learning[PB_mutate] <- sample(c("IL","UB","CB"), sum(PB_mutate), replace = TRUE)
agent$learning[CB_mutate] <- sample(c("IL","UB","PB"), sum(CB_mutate), replace = TRUE)
```

The following function **SLStemporal** combines all these chunks, with a larger N and lower μ as defaults.

```
SLStemporal <- function(N=5000, t_max=500, p_i=0.5, b=1, c_i=0.2, mu=0.001, v=0.9,
                        n=3, f=3, c_c=0.02, c_p=0.02) {

  # create agents, initially all individual learners
  agent <- data.frame(learning = rep("IL", N),
                      behaviour = rep(NA, N),
                      fitness = rep(NA, N))

  # keep track of freq and fitnesses of each type
```

```r
output <- data.frame(ILfreq = rep(NA, t_max),
                     UBfreq = rep(NA, t_max),
                     PBfreq = rep(NA, t_max),
                     CBfreq = rep(NA, t_max),
                     ILfitness = rep(NA, t_max),
                     UBfitness = rep(NA, t_max),
                     PBfitness = rep(NA, t_max),
                     CBfitness = rep(NA, t_max))

# start timestep loop
for (t in 1:t_max) {

  # 1. store previous timestep agent
  previous_agent <- agent

  # 2. individual learning for ILs
  agent$behaviour[agent$learning == "IL"] <- sample(c(1,0),
                                                    sum(agent$learning == "IL"),
                                                    replace = TRUE,
                                                    prob = c(p_i, 1 - p_i))

  # 3. social learning for critical learners

  # if environment has changed, all social learners have incorrect beh
  if (v < runif(1)) {

    agent$behaviour[agent$learning != "IL"] <- 0 # incorrect

  } else {

    # otherwise create matrix for holding n demonstrators for N agents
    # fill with randomly selected agents from previous gen
    dems <- matrix(data = sample(previous_agent$behaviour, N*n, replace = TRUE),
                   nrow = N, ncol = n)

    # for UBs, copy the behaviour of the 1st dem in dems
    agent$behaviour[agent$learning == "UB" & dems[,1]] <- 1 # correct
    agent$behaviour[agent$learning == "UB" & !dems[,1]] <- 0 # incorrect

    # for PB, copy correct if at least one of dems is correct
    agent$behaviour[agent$learning == "PB" & rowSums(dems) > 0] <- 1 # correct
    agent$behaviour[agent$learning == "PB" & rowSums(dems) == 0] <- 0 # incorrect

    # for CB, copy majority behaviour according to parameter f
    copy_probs <- rowSums(dems)^f / (rowSums(dems)^f + (n - rowSums(dems))^f)
```

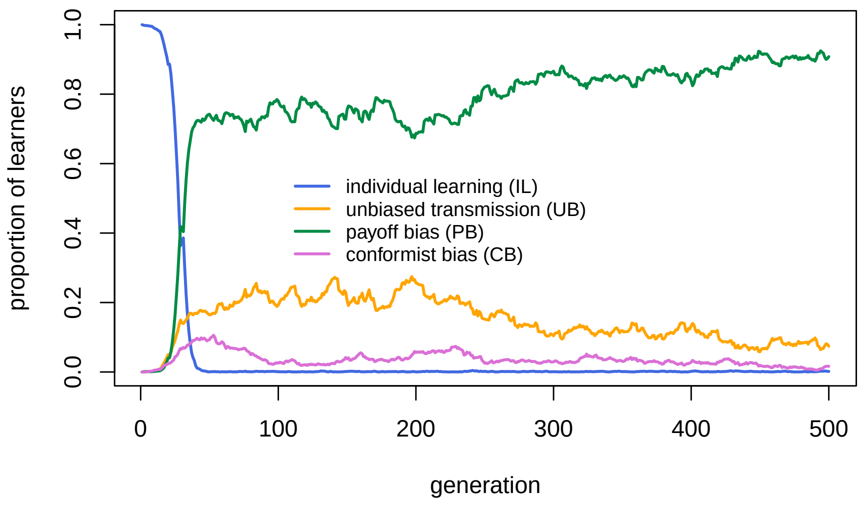
```
      probs <- runif(N)
      agent$behaviour[agent$learning == "CB" & probs < copy_probs] <- 1 # correct
      agent$behaviour[agent$learning == "CB" & probs >= copy_probs] <- 0 # incorrect

    }

    # 4. incorrect social learners engage in individual learning
    incorrect_SL <- agent$learning != "IL" & agent$behaviour == 0

    agent$behaviour[incorrect_SL] <- sample(c(1,0),
                                            sum(incorrect_SL),
                                            replace = TRUE,
                                            prob = c(p_i, 1-p_i))

    # 5. calculate fitnesses

    # all correct agents get bonus b
    agent$fitness <- ifelse(agent$behaviour, 1 + b, 1)

    # costs of learning
    agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i
    agent$fitness[agent$learning == "PB"] <- agent$fitness[agent$learning == "PB"] - c_p
    agent$fitness[agent$learning == "CB"] <- agent$fitness[agent$learning == "CB"] - c_c
    agent$fitness[incorrect_SL] <- agent$fitness[incorrect_SL] - c_i

    # 6. record frequencies and fitnesses of each type
    output$ILfreq[t] <- sum(agent$learning == "IL") / N
    output$UBfreq[t] <- sum(agent$learning == "UB") / N
    output$PBfreq[t] <- sum(agent$learning == "PB") / N
    output$CBfreq[t] <- sum(agent$learning == "CB") / N
    output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
    output$UBfitness[t] <- mean(agent$fitness[agent$learning == "UB"])
    output$PBfitness[t] <- mean(agent$fitness[agent$learning == "PB"])
    output$CBfitness[t] <- mean(agent$fitness[agent$learning == "CB"])

    # 7. selection and reproduction
    relative_fitness <- agent$fitness / sum(agent$fitness)

    agent$learning <- sample(agent$learning,
                             N,
                             prob = relative_fitness,
                             replace = TRUE)

    # 8. mutation
    mutate <- runif(N) < mu
    IL_mutate <- agent$learning == "IL" & mutate
    UB_mutate <- agent$learning == "UB" & mutate
    PB_mutate <- agent$learning == "PB" & mutate
    CB_mutate <- agent$learning == "CB" & mutate

    agent$learning[IL_mutate] <- sample(c("UB","PB","CB"), sum(IL_mutate), replace = T
    agent$learning[UB_mutate] <- sample(c("IL","PB","CB"), sum(UB_mutate), replace = T
    agent$learning[PB_mutate] <- sample(c("IL","UB","CB"), sum(PB_mutate), replace = T
    agent$learning[CB_mutate] <- sample(c("IL","UB","PB"), sum(CB_mutate), replace = T

  }

  # add predicted IL fitness to output and export
  output$predictedILfitness <- 1 + b*p_i - c_i
  output

}
```

The following two plotting functions plot the frequencies and fitnesses of the four strategies:

```
SLSfreqplot <- function(output) {

  plot(output$ILfreq,
       type = 'l',
       lwd = 2,
       col = "royalblue",
       ylim = c(0,1),
       ylab = "proportion of learners",
       xlab = "generation")

  lines(output$UBfreq,
        type = 'l',
        lwd = 2,
        col = "orange")

  lines(output$PBfreq,
        type = 'l',
        lwd = 2,
        col = "springgreen4")

  lines(output$CBfreq,
        type = 'l',
        lwd = 2,
        col = "orchid")
  legend(x = nrow(output)/5, y = 0.6,
         legend = c("individual learning (IL)",
                    "unbiased transmission (UB)",
                    "payoff bias (PB)",
                    "conformist bias (CB)"),
         col = c("royalblue", "orange", "springgreen4","orchid"),
         lty = 1,
         lwd = 2,
         bty = "n",
         cex = 0.8)

}

SLSfitnessplot <- function(output) {

  alpha = 150

  plot(output$ILfitness,
       type = 'l',
       lwd = 2,
       col = rgb(65,105,225,max=255,alpha), # royalblue
       ylim = c(0.5,2),
       ylab = "mean fitness",
       xlab = "generation")

  lines(output$UBfitness,
        type = 'l',
        lwd = 2,
        col = rgb(255,165,0,max=255,alpha)) # orange

  lines(output$PBfitness,
        type = 'l',
        lwd = 2,
        col = rgb(0,139,69,max=255,alpha)) # springgreen4

  lines(output$CBfitness,
        type = 'l',
        lwd = 2,
        col = rgb(218,112,214,max=255,alpha)) # orchid

  legend(x = nrow(output)/5, y = 0.9,
         legend = c("individual learning (IL)",
                    "unbiased transmission (UB)",
                    "payoff bias (PB)",
                    "conformist bias (CB)"),
         col = c("royalblue", "orange", "springgreen4","orchid"),
         lty = 1,
         lwd = 2,
         bty = "n",
         cex = 0.8)

  abline(h = output$predictedILfitness[1],
         lty = 2)

}
```

We can now run the model with default parameter values:

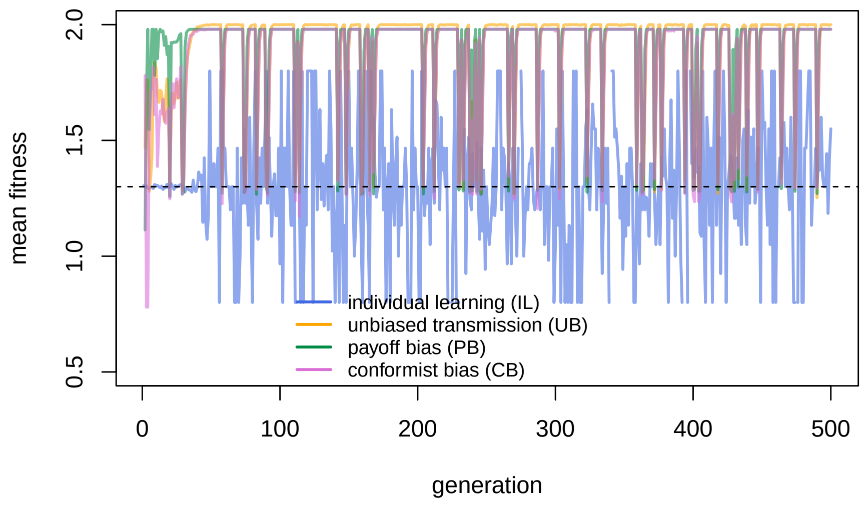
```
data_model19a <- SLStemporal()

SLSfreqplot(data_model19a)
```



Payoff bias clearly outperforms the other strategies. Individual learning is virtually absent, as expected given that the critical learners all incorporate individual learning. Unbiased and conformist learners each have negligible frequencies.

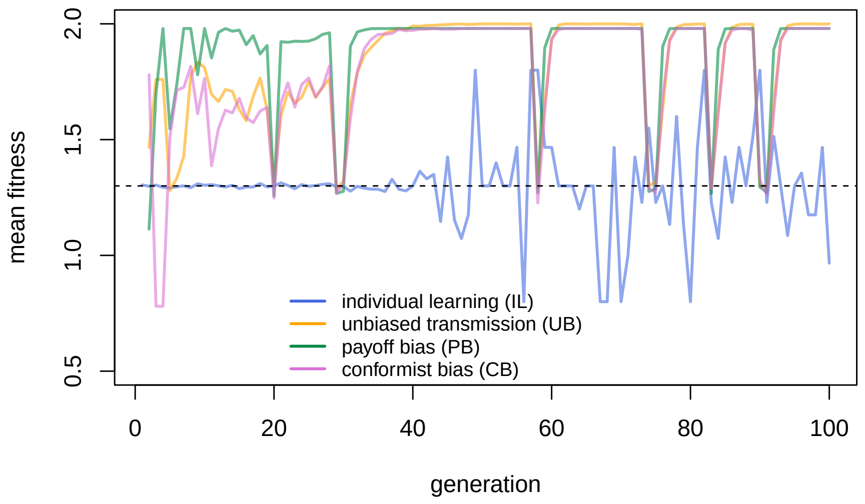
We can plot the fitnesses to explore this further:

```
SLSfitnessplot(data_model19a)
```



Individual learners fluctuate around their expected mean fitness indicated with the horizontal dotted line. They fluctuate quite wildly given that there are so few of them. The critical learners peak much higher, with drops when the environment changes. Let's zoom in to the first few generations for a closer look:

```
SLSfitnessplot(data_model19a[1:100, ])
```



Here the advantage of payoff bias becomes apparent. While UB agents peak slightly higher due to their lower costs of learning, PB agents acquire the correct behaviour quicker after an environmental change than UB and CB agents.

This makes sense. Immediately after an environmental change, all critical learners will be incorrect, and therefore fall back on individual learning. Some will discover the new correct behaviour, others won't, as determined by p_i. The following generation, critical learners can potentially copy the correct behaviour. PB agents have more chance of copying the correct behaviour amongst their n demonstrators than UB agents who only copy a single demonstrator, assuming $n > 1$. CB agents copy the majority of n demonstrators, but immediately after an environmental change the majority is likely to still be the old, now-incorrect behaviour.

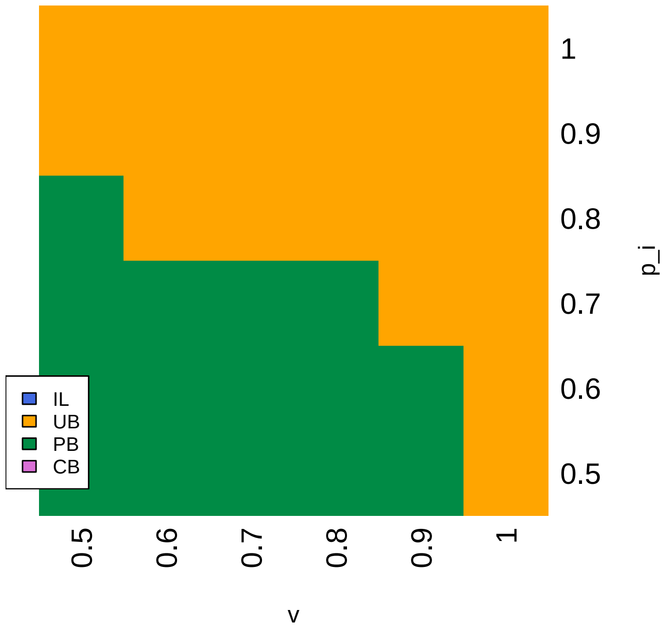
To get an idea of which strategy wins over a range of parameter values and avoid the temptation to cherry-pick particular values, or accidentally miss interesting parts of the parameter space, we can run the simulation over multiple values of two parameters and plot the results in a heat map. The function **SLStemporalheatmap** takes as its first argument a list of 2 parameters and their values, e.g. `list(n = 1:5, v = seq(0.5,1,0.1))`, and runs **SLStemporal** for each parameter value combination. The more values you enter, the higher the resolution of the heatmap (and the longer it takes to run). The winning strategy for each run is the one that exceeds the frequency *cutoff* after *t_max* generations. By default *cutoff* is 0.5. The winning strategy is stored in a matrix called *strategy* which is then used to create a heatmap with cells coloured according to the winning strategy. If no strategy exceeds *cutoff* then the cell is coloured grey.

To speed things up when running multiple simulations, we reduce N without much loss of inference.

```
SLStemporalheatmap <- function(parameters, cutoff = 0.5,
                              N = 1000, t_max = 1000, p_i=0.5, b=1, c_i=0.2, mu=0.001,
                              v=0.9, n=3, f=3, c_c=0.02, c_p=0.02) {

  # for brevity
  p <- parameters

  # list of default arguments
  arguments <- data.frame(N, t_max, p_i, b, c_i, mu, v,
                          n, f, c_c, c_p)

  # for modifying later on
  p1_pos <- which(names(arguments) == names(p[1])) # position of p1 in arguments
  p2_pos <- which(names(arguments) == names(p[2])) # position of p2 in arguments

  # matrix to hold winning strategies
  # 0=none,1=IL,2=UB,3=PB,4=CB
  strategy <- matrix(nrow = length(p[[1]]),
                     ncol = length(p[[2]]))

  for (p1 in p[[1]]) {

    for (p2 in p[[2]]) {

      # set this run's parameter values
      arguments[p1_pos] <- p1
      arguments[p2_pos] <- p2

      # run the model
      output <- SLStemporal(as.numeric(arguments[1]),
                            as.numeric(arguments[2]),
                            as.numeric(arguments[3]),
                            as.numeric(arguments[4]),
                            as.numeric(arguments[5]),
                            as.numeric(arguments[6]),
                            as.numeric(arguments[7]),
                            as.numeric(arguments[8]),
                            as.numeric(arguments[9]),
                            as.numeric(arguments[10]),
                            as.numeric(arguments[11]))

      # record which strategy has frequency above cutoff; if none, set to zero
      if (length(which(output[nrow(output),1:4] > cutoff) > 0)) {
        strategy[which(p[[1]] == p1), which(p[[2]] == p2)] <-
          which(output[nrow(output),1:4] > cutoff)
      } else {
        strategy[which(p[[1]] == p1), which(p[[2]] == p2)] <- 0
      }

    }

  }

  # draw heatmap from strategy
  heatmap(strategy,
          Rowv = NA,
          Colv = NA,
          labRow = p[[1]],
          labCol = p[[2]],
          col = c("grey","royalblue","orange","springgreen4","orchid"),
          breaks = c(-.5,0.5,1.5,2.5,3.5,4.5),
          scale = "none",
          ylab = names(p[1]),
          xlab = names(p[2]))
  legend("bottomleft",
         legend = c("IL", "UB", "PB", "CB"),
         fill = c("royalblue", "orange", "springgreen4","orchid"),
         cex = 0.8,
         bg = "white")

  # export strategy
  strategy

}
```

Here we create a heatmap across values of v (rate of environmental change) and p_i (accuracy of individual learning):

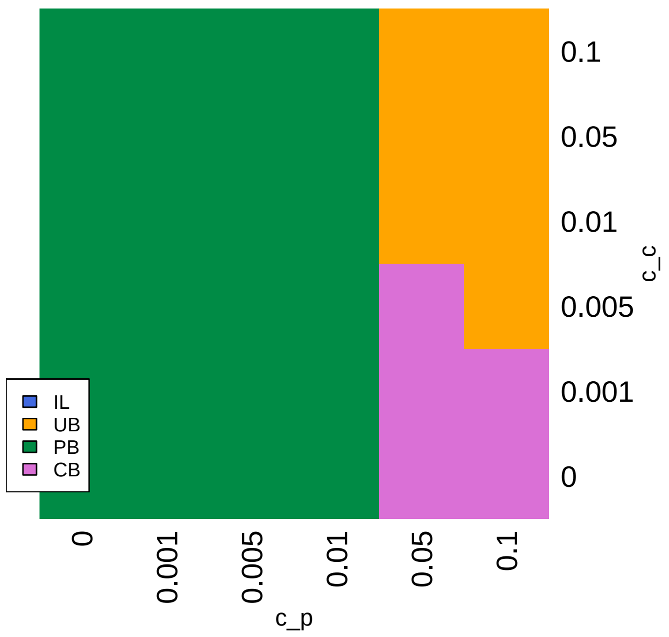
```
parameters <- list(p_i = seq(0.5,1,0.1), v = seq(0.5,1,0.1))

strategy <- SLStemporalheatmap(parameters)
```



The heatmap above shows that PB is favoured at lower values of v and p_i, otherwise UB is favoured. This makes sense. When v is high, environments are relatively stable, and the advantage to PB of more rapidly identifying the new correct behaviour following environmental change is not so useful. UB agents with their lower costs do better once enough critical learners with the correct behaviour have accumulated in the population. Similarly, when p_i is high, IL agents or critical learners using individual learning can rapidly identify the correct behaviour following an environmental change. UB agents can then copy one of these correct agents at random at a lower cost than PB agents.

Are there any parameter values where CB outperforms PB? This should depend first on the relative cost to CB vs PB:

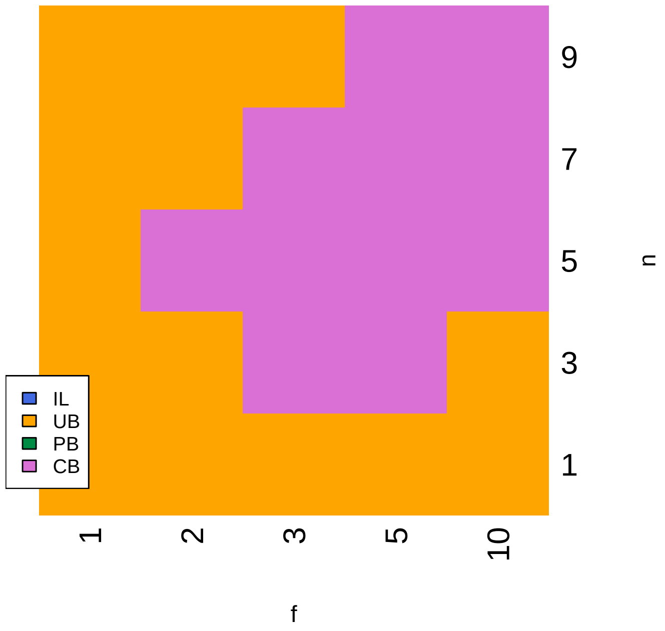
```
parameters <- list(c_c = c(0,0.001,0.005,0.01,0.05,0.1),
                   c_p = c(0,0.001,0.005,0.01,0.05,0.1))

strategy <- SLStemporalheatmap(parameters)
```



This heatmap shows that CB is favoured when the cost to CB c_c is low, and the cost to PB c_p is high (the bottom right of the heatmap). When the cost to both is high then the costless UB is favoured (top right). Otherwise PB is favoured.

If we set the relative cost of PB to be very high such that PB is effectively removed from the simulation, and reduce the cost of CB so that it has a bigger advantage over UB, we can explore the effect of n and f on the success of CB:

```
parameters <- list(n = c(1,3,5,7,9), f = c(1,2,3,5,10))

strategy <- SLStemporalheatmap(parameters, c_p = 0.1, c_c = 0.01)
```



CB is favoured at higher values of f and n. Given that f is the strength of conformity, this makes sense. Conformity will also be more effective with more demonstrators (larger n), as it is more likely that the majority correct behaviour in the entire sub-population is the majority in the sample of n demonstrators. However, larger n will also favour payoff bias, which as we have already seen out-performs CB unless c_p is very large.

Model 19b: Spatially varying environments

Now let's make environments vary spatially rather than temporally and see if the results change. This situation is similar to Model 7 (migration) where we simulated different groups or sub-populations within the larger population with occasional migration between those groups. We assume here that each group has a different optimal behaviour. The rate of spatial environmental variation is therefore the rate of migration, i.e. the probability that an agent finds themselves in a new group/environment. We remove temporal variation entirely, so optimal behaviours within groups stay the same throughout the simulation.

More specifically, and following Nakahashi et al. (2012), we assume G groups each containing N agents, such that there are now NG agents in total. There are also G different behaviours labelled 1, 2, 3…G. In each group, one behaviour is correct and all the others are (equally) incorrect. Hence in group 1, behaviour 1 is correct and gives a fitness of $1+b$ while behaviours 2 to G are all incorrect and give a fitness of 1. In group 2, behaviour 2 gives fitness $1+b$ and the others give fitness of 1, and so on. There are no behaviours that are incorrect in all groups.

As in Model 7, the rate of migration is m and migration follows Wright's island model. Each generation, each agent has a probability m of being taken out of its group and into a pool of migrants. These migrants are then randomly put back into the now-empty slots ignoring group structure.

Social learning, whether UB, CB or PB, occurs within groups. UB agents pick a single random member of their group to copy, CB agents adopt the majority behaviour amongst n demonstrators from their group, and PB agents copy the correct behaviour if it is present in at least one of n demonstrators from their group.

Individual learners are unaffected by spatial environmental variation just as they were unaffected by temporal environmental variation. The critical learners are affected, but differently to temporal variation. Whereas temporal environmental change renders all previous agents' behaviour incorrect, migrants to a new group can learn from demonstrators who have been in that group previously and potentially already acquired the correct behaviour.

The first step is to create our agent dataframe, this time with a variable *group* denoting the group id of each agent. Although currently unspecified, the *behaviour* variable of *agent* will now hold the numeric behaviour value from 1 to G, rather than 0 or 1. Agents are correct and get a fitness bonus when their *behaviour* value matches their *group* id. The *output* dataframe is identical to Model 19a.

```r
N <- 100
G <- 5
t_max <- 100

# create agents, initially all individual learners
agent <- data.frame(group = rep(1:G, each = N),
                    learning = rep("IL", N*G),
                    behaviour = rep(NA, N*G), # this is now 1,2...G
                    fitness = rep(NA, N*G))

# keep track of freq and fitnesses of each type
output <- data.frame(ILfreq = rep(NA, t_max),
                     UBfreq = rep(NA, t_max),
                     PBfreq = rep(NA, t_max),
                     CBfreq = rep(NA, t_max),
                     ILfitness = rep(NA, t_max),
                     UBfitness = rep(NA, t_max),
                     PBfitness = rep(NA, t_max),
                     CBfitness = rep(NA, t_max))
```

As before, we first run one generation of individual learning before composing the full set of timestep events. Note the new code for individual learning which is now group-specific. We cycle through each group and set the behaviour to match the group id g with probability p_i, otherwise another random id from 1 to G excluding g (denoted by `(1:G)[-g]`) with equal probability. Fitness is also group specific, with a bonus only if the agent's behaviour matches their group. The code for recording type frequency and fitness, and for mutation, are identical to Model 19a except now there are NG agents rather than N.

```
t <- 1
p_i <- 0.5
b <- 1
c_i <- 0.2
mu <- 0.5

# individual learning for ILs

# for each group, acquire correct group beh with prob p_i, otherwise random other
for (g in 1:G) {

  ingroup <- agent$learning == "IL" & agent$group == g

  agent$behaviour[ingroup] <- sample(c(g,(1:G)[-g]),
                                     sum(ingroup),
                                     replace = TRUE,
                                     prob = c(p_i, rep((1-p_i)/(G-1), G-1)))

}

# get fitnesses
agent$fitness <- ifelse(agent$behaviour == agent$group, 1 + b, 1)
agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i

head(agent)
```

```
##   group learning behaviour fitness
## 1     1       IL         4     0.8
## 2     1       IL         2     0.8
## 3     1       IL         3     0.8
## 4     1       IL         1     1.8
## 5     1       IL         3     0.8
## 6     1       IL         5     0.8
```

```r
# record frequencies and fitnesses of each type
output$ILfreq[t] <- sum(agent$learning == "IL") / (N*G)
output$UBfreq[t] <- sum(agent$learning == "UB") / (N*G)
output$PBfreq[t] <- sum(agent$learning == "PB") / (N*G)
output$CBfreq[t] <- sum(agent$learning == "CB") / (N*G)
output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
output$UBfitness[t] <- mean(agent$fitness[agent$learning == "UB"])
output$PBfitness[t] <- mean(agent$fitness[agent$learning == "PB"])
output$CBfitness[t] <- mean(agent$fitness[agent$learning == "CB"])

head(output)
```

```
##   ILfreq UBfreq PBfreq CBfreq ILfitness UBfitness PBfitness CBfitness
## 1      1      0      0      0     1.304       NaN       NaN       NaN
## 2     NA     NA     NA     NA        NA        NA        NA        NA
## 3     NA     NA     NA     NA        NA        NA        NA        NA
## 4     NA     NA     NA     NA        NA        NA        NA        NA
## 5     NA     NA     NA     NA        NA        NA        NA        NA
## 6     NA     NA     NA     NA        NA        NA        NA        NA
```

```r
# mutation
mutate <- runif(N) < mu
IL_mutate <- agent$learning == "IL" & mutate
UB_mutate <- agent$learning == "UB" & mutate
PB_mutate <- agent$learning == "PB" & mutate
CB_mutate <- agent$learning == "CB" & mutate

agent$learning[IL_mutate] <- sample(c("UB","PB","CB"), sum(IL_mutate), replace = TRUE)
agent$learning[UB_mutate] <- sample(c("IL","PB","CB"), sum(UB_mutate), replace = TRUE)
agent$learning[PB_mutate] <- sample(c("IL","UB","CB"), sum(PB_mutate), replace = TRUE)
agent$learning[CB_mutate] <- sample(c("IL","UB","PB"), sum(CB_mutate), replace = TRUE)
```

Now we can start the sequence of within-generation events. First we record the previous agent dataframe and do individual learning, as above.

```r
t <- 2

# 1. store previous timestep agent
previous_agent <- agent

# 2. individual learning for ILs

# for each group, acquire correct group beh with prob p_i, otherwise random other
for (g in 1:G) {
  ingroup <- agent$learning == "IL" & agent$group == g

  agent$behaviour[ingroup] <- sample(c(g,(1:G)[-g]),
                                     sum(ingroup),
                                     replace = TRUE,
                                     prob = c(p_i, rep((1-p_i)/(G-1), G-1)))

}
```

Next is social learning. Now that demonstrators must come from the agent's own group, we need slightly more complex code to create the *dems* dataframe. The following code cycles through each group, samples n demonstrators for each agent within that group, and adds them to the *dems* matrix using the **rbind** command.

```
n <- 3

# 3. social learning for critical learners

# select group-specific demonstrators
dems <- NULL

# cycle through each group
for (g in 1:G) {

  ingroup <- agent$group == g

  dems <- rbind(dems,
                matrix(data = sample(previous_agent$behaviour[ingroup], N*n, replace = TRUE),
                       nrow = N, ncol = n))

}

head(dems)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    2
## [2,]    1    1    1
## [3,]    5    1    1
## [4,]    4    1    1
## [5,]    1    2    4
## [6,]    3    1    1
```

UB agents copy the first demonstrator stored in the first column of *dems*. As before, there is no directional change in the frequency of correct behaviour due to UB.

```
# proportion correct before social learning
mean(agent$behaviour[agent$learning == "UB"] == agent$group[agent$learning == "UB"])
```

```
## [1] 0.4810127
```

```
# for UBs, copy the behaviour of the 1st dem in dems
agent$behaviour[agent$learning == "UB"] <- dems[agent$learning == "UB", 1]

# proportion correct after social learning
mean(agent$behaviour[agent$learning == "UB"] == agent$group[agent$learning == "UB"])
```

```
## [1] 0.6329114
```

PB agents copy the correct behaviour in their group if at least one demonstrator has the correct behaviour, otherwise they copy their first (incorrect) random demonstrator. Here *PB_correct* identifies PB agents who observe at least one correct demonstrator, and *PB_incorrect* identifies PB agents who observe all incorrect demonstrators.

```
# proportion correct before social learning
mean(agent$behaviour[agent$learning == "PB"] == agent$group[agent$learning == "PB"])
```

```
## [1] 0.5833333
```

```
# for PB, copy correct if at least one of dems is correct

PB_correct <- agent$learning == "PB" & rowSums(dems == agent$group) > 0
PB_incorrect <- agent$learning == "PB" & rowSums(dems == agent$group) == 0

agent$behaviour[PB_correct] <- agent$group[PB_correct]
agent$behaviour[PB_incorrect] <- dems[PB_incorrect, 1]


# proportion correct after social learning
mean(agent$behaviour[agent$learning == "PB"] == agent$group[agent$learning == "PB"])
```

```
## [1] 0.7916667
```

As in Model 19a, you should see the frequency of correct behaviours increase after payoff biased copying.

Finally CB agents are disproportionately more likely to copy the majority trait amongst the demonstrators. Because there are now potentially more than two behaviours when $G > 2$, we need to calculate the number of times each trait appears amongst the n demonstrators for each agent (stored in *counts*), get copy probabilities for each trait based on the counts and Equation 17.2 (stored in *copy_probs*), then actually pick a trait based on these probabilities using **apply** on *copy_probs*.

```
f <- 3

# proportion correct before social learning
mean(agent$behaviour[agent$learning == "CB"] == agent$group[agent$learning == "CB"])
```

```
## [1] 0.53125
```

```
# for CB, copy majority behaviour according to parameter f

# get number of times each trait appears in n dems for each agent
counts <- matrix(nrow = N*G, ncol = G)
for (g in 1:G) counts[,g] <- rowSums(dems == g)

# get copy probs based on counts and f
copy_probs <- matrix(nrow = N*G, ncol = G)
for (g in 1:G) copy_probs[,g] <- rowSums(dems == g)^f / rowSums(counts^f)

# pick a trait based on copy_probs
CB_beh <- apply(copy_probs, 1, function(x) sample(1:G, 1, prob = x))

# CB agents copy the trait
agent$behaviour[agent$learning == "CB"] <- CB_beh[agent$learning == "CB"]


# proportion correct after social learning
mean(agent$behaviour[agent$learning == "CB"] == agent$group[agent$learning == "CB"])
```

```
## [1] 0.5625
```

As in Model 19a, CB is unlikely to have increased the frequency of correct behaviours by much, if at all.

Steps 4, 5 and 6 are similar to before: incorrect critical learners engage in individual learning, fitnesses are calculated, and frequencies and mean fitness of each type are recorded in *output*.

```
c_c <- 0.02
c_p <- 0.02

# 4. incorrect social learners engage in individual learning
incorrect_SL <- agent$learning != "IL" & agent$behaviour != agent$group

for (g in 1:G) {

  ingroup <- incorrect_SL & agent$group == g

  agent$behaviour[ingroup] <- sample(c(g,(1:G)[-g]),
                                     sum(ingroup),
                                     replace = TRUE,
                                     prob = c(p_i, rep((1-p_i)/(G-1), G-1)))

}

# 5. calculate fitnesses

# all correct agents get bonus b
agent$fitness <- ifelse(agent$behaviour == agent$group, 1 + b, 1)

# costs of learning
agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c_i
agent$fitness[agent$learning == "PB"] <- agent$fitness[agent$learning == "PB"] - c_p
agent$fitness[agent$learning == "CB"] <- agent$fitness[agent$learning == "CB"] - c_c
agent$fitness[incorrect_SL] <- agent$fitness[incorrect_SL] - c_i

# 6. record frequencies and fitnesses of each type
output$ILfreq[t] <- sum(agent$learning == "IL") / (N*G)
output$UBfreq[t] <- sum(agent$learning == "UB") / (N*G)
output$PBfreq[t] <- sum(agent$learning == "PB") / (N*G)
output$CBfreq[t] <- sum(agent$learning == "CB") / (N*G)
output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
output$UBfitness[t] <- mean(agent$fitness[agent$learning == "UB"])
output$PBfitness[t] <- mean(agent$fitness[agent$learning == "PB"])
output$CBfitness[t] <- mean(agent$fitness[agent$learning == "CB"])

head(output)
```

```
##   ILfreq UBfreq PBfreq CBfreq ILfitness UBfitness PBfitness CBfitness
## 1   1.00  0.000  0.000  0.000  1.304000       NaN       NaN       NaN
## 2   0.57  0.158  0.144  0.128  1.294737       1.8  1.813333  1.626875
## 3     NA     NA     NA     NA        NA        NA        NA        NA
## 4     NA     NA     NA     NA        NA        NA        NA        NA
## 5     NA     NA     NA     NA         NA         NA         NA         NA
## 6     NA     NA     NA     NA         NA         NA         NA         NA
```

Step 7 is selection and reproduction. We make this occur within groups; if social learning occurs within groups, it makes sense that mating and reproduction also happens within groups.

```
# before selection
table(agent$learning)/(N*G)
```

```
##
##    CB    IL    PB    UB
## 0.128 0.570 0.144 0.158
```

```
# 7. selection and reproduction
# (now within-group)

for (g in 1:G) {

  ingroup <- agent$group == g

  relative_fitness <- agent$fitness[ingroup] / sum(agent$fitness[ingroup])

  agent$learning[ingroup] <- sample(agent$learning[ingroup],
                                    N,
                                    prob = relative_fitness,
                                    replace = TRUE)

}

# after selection
table(agent$learning)/(N*G)
```

```
##
##    CB    IL    PB    UB
## 0.128 0.514 0.184 0.174
```

Next is mutation, identical to above, and finally some new code implementing migration. Just as in Model 4, we select migrants with probability m per agent, add these migrants' learning type and behaviour to a *migrants* dataframe, then put these randomly back into the migrant slots ignoring group id.

```
m <- 0.1

# 8. mutation
mutate <- runif(N*G) < mu
IL_mutate <- agent$learning == "IL" & mutate
UB_mutate <- agent$learning == "UB" & mutate
PB_mutate <- agent$learning == "PB" & mutate
CB_mutate <- agent$learning == "CB" & mutate

agent$learning[IL_mutate] <- sample(c("UB","PB","CB"), sum(IL_mutate), replace = TRUE)
agent$learning[UB_mutate] <- sample(c("IL","PB","CB"), sum(UB_mutate), replace = TRUE)
agent$learning[PB_mutate] <- sample(c("IL","UB","CB"), sum(PB_mutate), replace = TRUE)
agent$learning[CB_mutate] <- sample(c("IL","UB","PB"), sum(CB_mutate), replace = TRUE)

# 9. migration

# NG probabilities, one for each agent, to compare against m
probs <- runif(1:(N*G))

# with prob m, add an agent's learning and behaviour to list of migrants
migrants <- agent[probs < m, c("learning", "behaviour")]

# put migrants randomly into empty slots
agent[probs < m, c("learning", "behaviour")] <- migrants[sample(rownames(migrants)),]
```

The following function **SLSspatial** combines all these code chunks:

```
SLSspatial <- function(N=1000, t_max=500, p_i=0.5, b=1, c_i=0.2, mu=0.001,
                       G=5, n=3, f=3, c_c=0.02, c_p=0.02, m=0.1) {

  # create agents, initially all individual learners
  agent <- data.frame(group = rep(1:G, each = N),
                      learning = rep("IL", N*G),
                      behaviour = rep(NA, N*G), # this is now 1,2...G
                      fitness = rep(NA, N*G))

  # keep track of freq and fitnesses of each type
  output <- data.frame(ILfreq = rep(NA, t_max),
                       UBfreq = rep(NA, t_max),
                       PBfreq = rep(NA, t_max),
                       CBfreq = rep(NA, t_max),
                       ILfitness = rep(NA, t_max),
                       UBfitness = rep(NA, t_max),
                       PBfitness = rep(NA, t_max),
                       CBfitness = rep(NA, t_max))
```

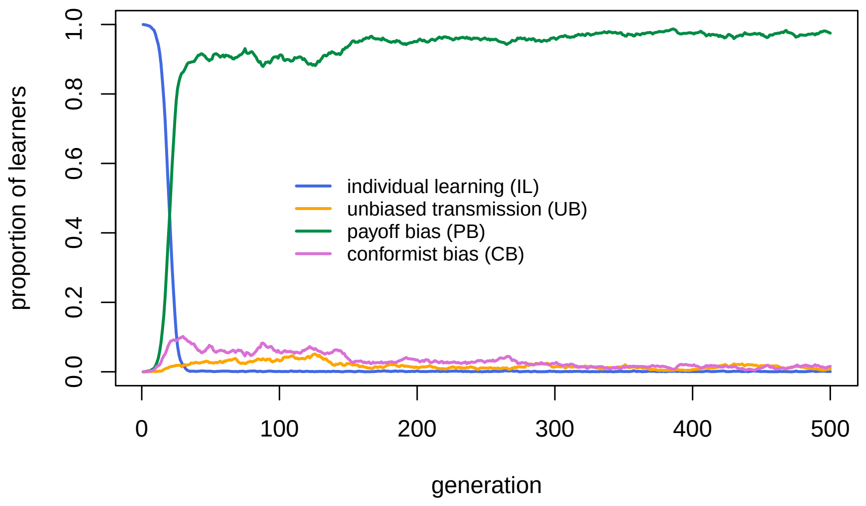
```
# start timestep loop
for (t in 1:t_max) {

  # 1. store previous timestep agent
  previous_agent <- agent

  # 2. individual learning for ILs

  # for each group, acquire correct group beh with prob p_i, otherwise random other
  for (g in 1:G) {

    ingroup <- agent$learning == "IL" & agent$group == g

    agent$behaviour[ingroup] <- sample(c(g,(1:G)[-g]),
                                       sum(ingroup),
                                       replace = TRUE,
                                       prob = c(p_i, rep((1-p_i)/(G-1), G-1)))

  }

  # 3. social learning for social learners

  # skip if no social learners
  if (sum(agent$learning == "IL") < N*G) {

    # select group-specific demonstrators
    dems <- NULL

    # cycle through each group
    for (g in 1:G) {

      ingroup <- agent$group == g

      dems <- rbind(dems,
                    matrix(data = sample(previous_agent$behaviour[ingroup], N*n, replace = TRUE
                           nrow = N, ncol = n))

    }

    # for UBs, copy the behaviour of the 1st dem in dems
    agent$behaviour[agent$learning == "UB"] <- dems[agent$learning == "UB", 1]

    # for PB, copy correct group beh if at least one of dems is correct
    PB_correct <- agent$learning == "PB" & rowSums(dems == agent$group) > 0
    PB_incorrect <- agent$learning == "PB" & rowSums(dems == agent$group) == 0
    agent$behaviour[PB_correct] <- agent$group[PB_correct]
    agent$behaviour[PB_incorrect] <- dems[PB_incorrect, 1]

    # for CB, copy majority behaviour according to parameter f

    # get number of times each trait appears in n dems for each agent
    counts <- matrix(nrow = N*G, ncol = G)
    for (g in 1:G) counts[,g] <- rowSums(dems == g)

    # get copy probs based on counts and f
    copy_probs <- matrix(nrow = N*G, ncol = G)
    for (g in 1:G) copy_probs[,g] <- rowSums(dems == g)^f / rowSums(counts^f)

    # pick a trait based on copy_probs
    CB_beh <- apply(copy_probs, 1, function(x) sample(1:G, 1, prob = x))

    # CB agents copy the trait
    agent$behaviour[agent$learning == "CB"] <- CB_beh[agent$learning == "CB"]

  }

  # 4. incorrect social learners engage in individual learning
  incorrect_SL <- agent$learning != "IL" & agent$behaviour != agent$group

  for (g in 1:G) {

    ingroup <- incorrect_SL & agent$group == g

    agent$behaviour[ingroup] <- sample(c(g,(1:G)[-g]),
                                       sum(ingroup),
                                       replace = TRUE,
                                       prob = c(p_i, rep((1-p_i)/(G-1), G-1)))

  }

  # 5. calculate fitnesses

  # all correct agents get bonus b
  agent$fitness <- ifelse(agent$behaviour == agent$group, 1 + b, 1)

  # costs of learning
  agent$fitness[agent$learning == "IL"] <- agent$fitness[agent$learning == "IL"] - c
  agent$fitness[agent$learning == "PB"] <- agent$fitness[agent$learning == "PB"] - c
  agent$fitness[agent$learning == "CB"] <- agent$fitness[agent$learning == "CB"] - c
  agent$fitness[incorrect_SL] <- agent$fitness[incorrect_SL] - c_i
```

```
# 6. record frequencies and fitnesses of each type
output$ILfreq[t] <- sum(agent$learning == "IL") / (N*G)
output$UBfreq[t] <- sum(agent$learning == "UB") / (N*G)
output$PBfreq[t] <- sum(agent$learning == "PB") / (N*G)
output$CBfreq[t] <- sum(agent$learning == "CB") / (N*G)
output$ILfitness[t] <- mean(agent$fitness[agent$learning == "IL"])
output$UBfitness[t] <- mean(agent$fitness[agent$learning == "UB"])
output$PBfitness[t] <- mean(agent$fitness[agent$learning == "PB"])
output$CBfitness[t] <- mean(agent$fitness[agent$learning == "CB"])

# 7. selection and reproduction
# (now within-group)

for (g in 1:G) {

  ingroup <- agent$group == g

  relative_fitness <- agent$fitness[ingroup] / sum(agent$fitness[ingroup])

  agent$learning[ingroup] <- sample(agent$learning[ingroup],
                                    N,
                                    prob = relative_fitness,
                                    replace = TRUE)

}

# 8. mutation
mutate <- runif(N*G) < mu
IL_mutate <- agent$learning == "IL" & mutate
UB_mutate <- agent$learning == "UB" & mutate
PB_mutate <- agent$learning == "PB" & mutate
CB_mutate <- agent$learning == "CB" & mutate

agent$learning[IL_mutate] <- sample(c("UB","PB","CB"), sum(IL_mutate), replace = TRUE)
agent$learning[UB_mutate] <- sample(c("IL","PB","CB"), sum(UB_mutate), replace = TRUE)
agent$learning[PB_mutate] <- sample(c("IL","UB","CB"), sum(PB_mutate), replace = TRUE)
agent$learning[CB_mutate] <- sample(c("IL","UB","PB"), sum(CB_mutate), replace = TRUE)

# 9. migration

# NG probabilities, one for each agent, to compare against m
probs <- runif(1:(N*G))

# with prob m, add an agent's learning and behaviour to list of migrants
migrants <- agent[probs < m, c("learning", "behaviour")]
```

```
    # put migrants randomly into empty slots
    agent[probs < m, c("learning", "behaviour")] <- migrants[sample(rownames(migrants)

  }

  # add predicted IL fitness to output and export
  output$predictedILfitness <- 1 + b*p_i - c_i
  output

}
```

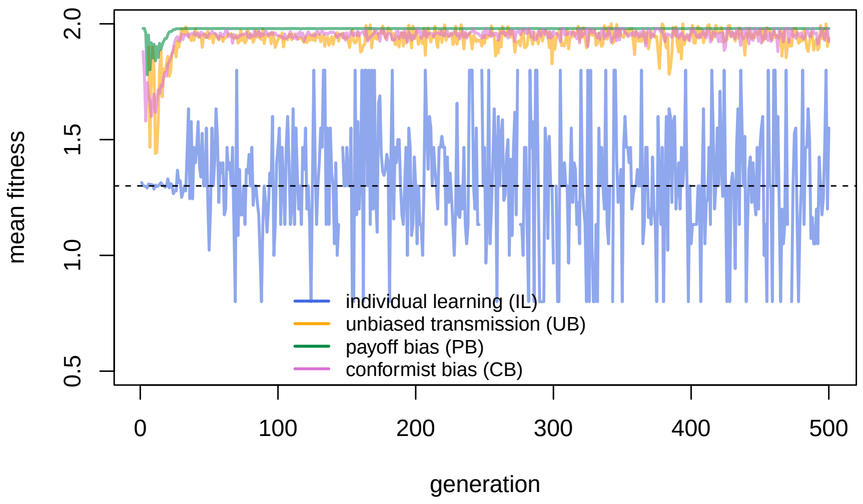
The plotting functions **SLSfreqplot** and **SLSfitnessplot** work fine for **SLSspatial**. Here is a default run:

```
data_model19b <- SLSspatial()

SLSfreqplot(data_model19b)
```



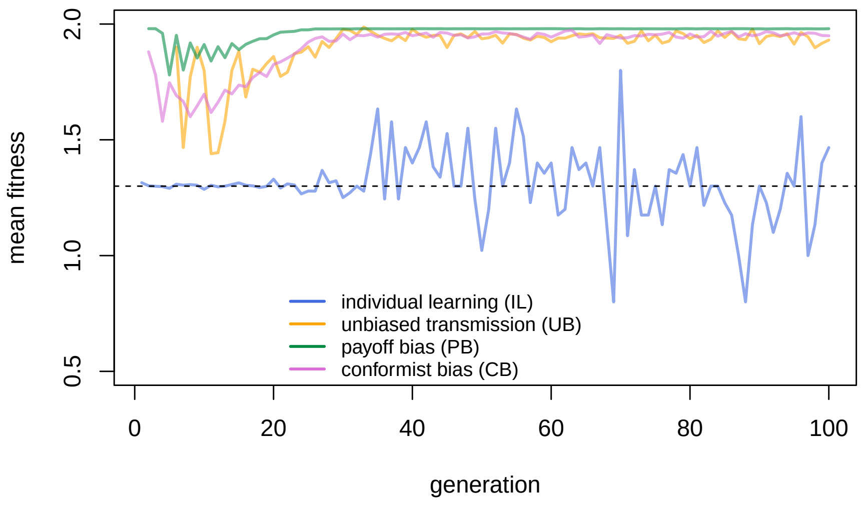
As before, PB is favoured over all other strategies. The fitness plot, however, looks different:

```
SLSfitnessplot(data_model19b)
```



And zooming in on the first few generations:

```r
SLSfitnessplot(data_model19b[1:100, ])
```

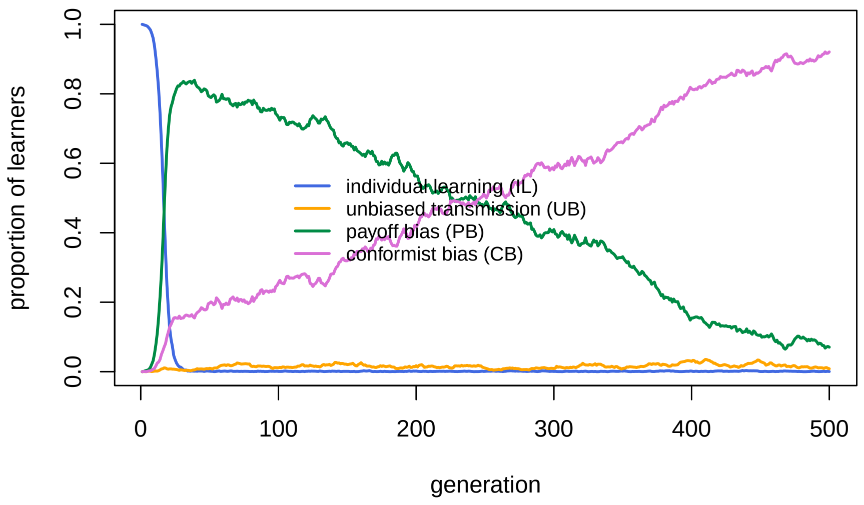


Whereas in the temporal model we could see PB agents adjusting more rapidly to periodic environmental change events, here PB agents gain an advantage at the start and keep that advantage throughout. This is because migration occurs at a constant rate. Every generation a fraction m of agents move to a random group. These migrants will be incorrect in their new environment. PB agents have a better chance that one of their n demonstrators will be correct than UB agents who only copy a single demonstrator, who might be an incorrect migrant.

CB agents, perhaps surprisingly, do no better than UB agents. As long as m is not too high, the majority of agents in each group will remain correct, and CB migrants can adopt this correct majority behaviour. But even with $f > 1$, conformity is still probabilistic, and with only $n = 3$ demonstrators there is a chance that the majority amongst N agents is not the majority amongst n.

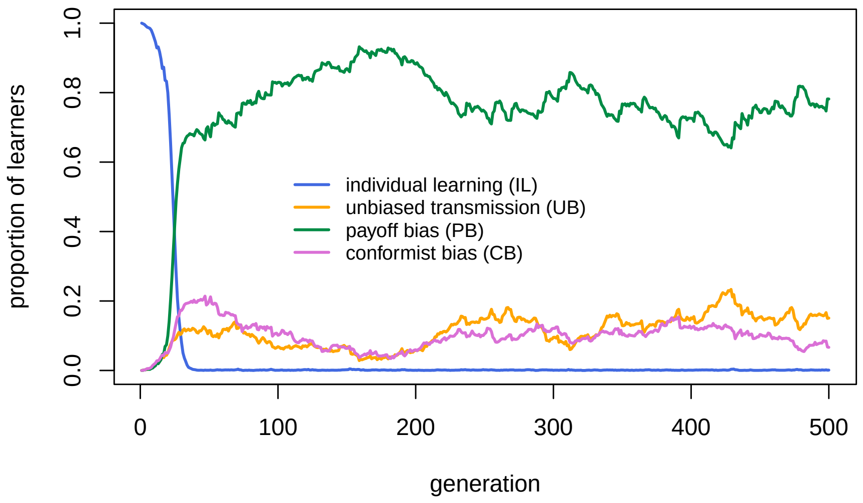
So let's increase f to increase our chances of picking the majority trait, and increase n to increase the chances that the correct behaviour is the majority. We also make the cost to CB slightly less than that to PB.

```r
data_model19b <- SLSspatial(f = 5, n = 5, c_c = 0.01, c_p = 0.03)

SLSfreqplot(data_model19b)
```



Now CB is favoured over PB. Whereas in temporally varying environments CB was only favoured when the costs to PB were so high as to remove PB from the competition, here CB is favoured when its costs are only slightly lower than PB. We can confirm this by running the temporal model with the same parameter values:

```
data_model19a <- SLStemporal(f = 5, n = 5, c_c = 0.01, c_p = 0.03)

SLSfreqplot(data_model19a)
```



For the same parameter values, CB agents do much better relative to PB agents in spatially varying environments than in temporally varying environments.

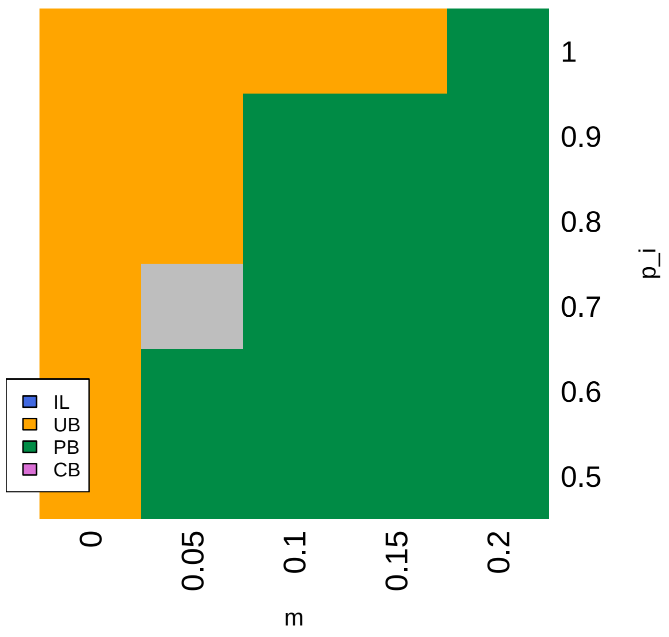
As in Model 19a, heatmaps across several parameter values can give a broader understanding of the model dynamics. The heatmap function needs to be modified given that **SLSspatial** has different parameters to **SLStemporal**, like so:

```
SLSspatialheatmap <- function(parameters, cutoff = 0.5,
                              N = 500, t_max = 1000, p_i=0.5, b=1, c_i=0.2, mu=0.001,
                              G=5, n=3, f=3, c_c=0.02, c_p=0.02, m=0.1) {

  # for brevity
  p <- parameters

  # list of default arguments
  arguments <- data.frame(N, t_max, p_i, b, c_i, mu,
                          G, n, f, c_c, c_p, m)

  # for modifying later on
  p1_pos <- which(names(arguments) == names(p[1])) # position of p1 in arguments
```

```r
p2_pos <- which(names(arguments) == names(p[2])) # position of p2 in arguments

# matrix to hold winning strategies
# 0=none,1=IL,2=UB,3=PB,4=CB
strategy <- matrix(nrow = length(p[[1]]),
                   ncol = length(p[[2]]))

for (p1 in p[[1]]) {

  for (p2 in p[[2]]) {

    # set this run's parameter values
    arguments[p1_pos] <- p1
    arguments[p2_pos] <- p2

    # run the model
    output <- SLSspatial(as.numeric(arguments[1]),
                         as.numeric(arguments[2]),
                         as.numeric(arguments[3]),
                         as.numeric(arguments[4]),
                         as.numeric(arguments[5]),
                         as.numeric(arguments[6]),
                         as.numeric(arguments[7]),
                         as.numeric(arguments[8]),
                         as.numeric(arguments[9]),
                         as.numeric(arguments[10]),
                         as.numeric(arguments[11]),
                         as.numeric(arguments[12]))

    # record which strategy has frequency above cutoff; if none, set to zero
    if (length(which(output[nrow(output),1:4] > cutoff) > 0)) {
      strategy[which(p[[1]] == p1), which(p[[2]] == p2)] <-
        which(output[nrow(output),1:4] > cutoff)
    } else {
      strategy[which(p[[1]] == p1), which(p[[2]] == p2)] <- 0
    }

  }

}

# draw heatmap from strategy
heatmap(strategy,
        Rowv = NA,
        Colv = NA,
          labRow = p[[1]],
          labCol = p[[2]],
          col = c("grey","royalblue","orange","springgreen4","orchid"),
          breaks = c(-.5,0.5,1.5,2.5,3.5,4.5),
          scale = "none",
          ylab = names(p[1]),
          xlab = names(p[2]))
  legend("bottomleft",
         legend = c("IL", "UB", "PB", "CB"),
         fill = c("royalblue", "orange", "springgreen4","orchid"),
         cex = 0.8,
         bg = "white")

  # export strategy
  strategy

}
```

Whereas in Model 19a we varied v, the rate of environmental change, we now vary m, the equivalent parameter controlling the rate of spatial environmental change experienced by each agent (i.e. their probability of migrating to a new environment), along with the accuracy of individual learning p_i:

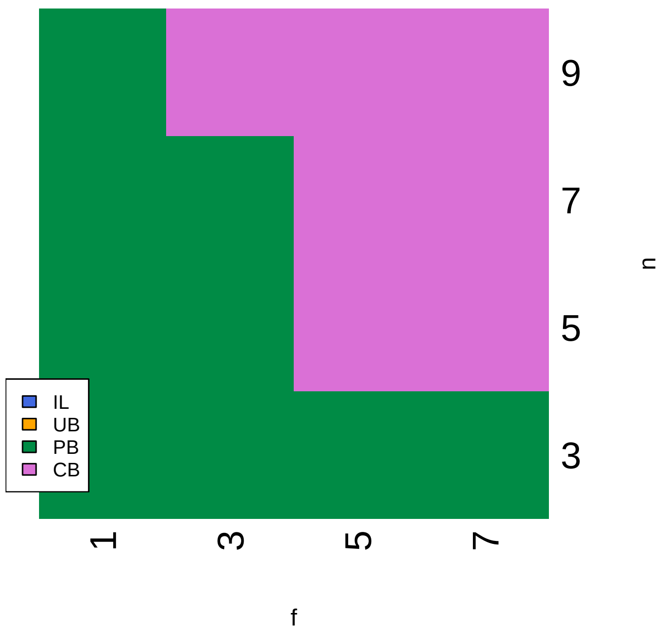
```
parameters <- list(p_i = seq(0.5,1,0.1),
                   m = c(0,0.05,0.1,0.15,0.2))

strategy <- SLSspatialheatmap(parameters)
```



PB is favoured when m is relatively large and p_i is relatively low. When $m = 0$ there is no migration and hence no environmental change. Just as for stable environments in Model 19a, UB agents are favoured because the correct behaviour quickly spreads in each sub-population and UB agents can copy this correct behaviour at lower cost than the other strategies. As migration increases, PB agents do better than UB agents when they migrate because of their larger pool of demonstrators when $n > 1$. Similarly, as p_i increases then the correct behaviour becomes more common even amongst migrants, such that UB agents can easily copy the correct behaviour at lower cost than PB and CB.

Above we determined that CB agents do well when f and n are both large, and the cost of CB is slightly lower than those of PB. Here we can test this more thoroughly:

```
parameters <- list(n = c(3,5,7,9), f = c(1,3,5,7))

strategy <- SLSspatialheatmap(parameters, c_c = 0.01)
```



When CB is only slightly less costly than PB, and f and n are sufficiently large, then CB is favoured. For the same parameter values in the temporal model, CB is never favoured.

Summary

Model 19 examined the evolution of two commonly studied social learning strategies, payoff bias (PB) and conformist bias (CB), relative to unbiased transmission (UB) and individual learning (IL). Model 19a found that in temporally varying environments PB is widely favoured, unless environments are very stable or individual learning is very accurate in which case UB is favoured. UB is favoured in these cases because correct behaviour becomes very common, allowing UB agents to copy the correct behaviour at lower cost than PB. When correct behaviour is less common then PB agents have the advantage of being able to identify that correct behaviour more effectively by learning from multiple demonstrators. CB is only favoured in temporally varying environments when the costs to PB are so high as to effectively remove them from the population. This recreates the findings of a similar mathematical model by Nakahashi et al. (2012) who concluded that “[CB is] not very important in a temporally varying environment, especially when [PB] are in the mix” (p.406).

In spatially varying environments, PB is also widely favoured unless environments are stable (i.e. there is no migration) or individual learning is very accurate, in which case UB again is favoured. However, CB is much more widely favoured in spatially varying environments than temporally varying environments. CB is favoured when its costs are only slightly less than those of PB, and when the strength of conformity (f) and number of demonstrators (n) are reasonably large. Under these conditions migrating CB agents can take advantage of the majority correct behaviour that is found in each group. This again recreates the analytical findings of Nakahashi et al. (2012) who state that "If the cost to [PB] is larger than that to [CB], [PB] never evolve" (p.406).

The recreation of findings from Nakahashi et al. (2012) is reassuring given that Model 19 is an agent-based simulation and theirs is an analytical model. There are also several other differences: they modelled pure social learning strategies whereas we modelled critical learners who combine social and individual learning; their conformity and payoff bias involved learning from every member of the population rather than a sample (i.e. in their model $n = N$), and they set the strength of conformity to infinity so agents always picked the majority. Nevertheless, the major findings are the same. If the same findings come from different modelling approaches and assumptions, we can be more confident in their reliability. Other similar models include Henrich & Boyd (1998) and Kendal et al. (2009); for a review see Aoki & Feldman (2014).

The finding that conformity should be used when environments vary spatially rather than temporally has been supported experimentally by Deffner et al. (2020), who found that participants used conformist social learning more often after migrating to a new group than when environments changed globally. Deffner et al. (2020) used the same formulation of conformity as used here and provide a mean estimate from their participants of $f = 3.30$, suggesting reasonably strong but not excessively strong conformity. They also found that participants often targeted fellow participants who had been in their groups for the longest. This was not possible in our model given that the n demonstrators were picked at random. However, non-random demonstrator selection (effectively, the indirectly biased transmission of Model 4) would be an interesting extension. Conformity following migration is a form of acculturation, which has major consequences for the maintenance of between-group cultural variation (see Henrich & Boyd 1998; Mesoudi 2018).

The success of CB in spatially varying environments hinges on whether it is less costly than PB. Is this a plausible assumption? Nakahashi et al. (2012) think so, arguing that "[t]his assumption seems plausible, given that [PB] have a more complicated task than [CB], which involves assessing payoffs or at least relative payoff differences for the cultural traits present." (p.405). While that may be the case, tallying the frequencies of multiple traits across multiple demonstrators also seems quite costly. Empirically quantifying the costs of social learning strategies such as CB and PB would be a worthwhile endeavour.

In terms of programming, much of the code here is recycled from previous models that have examined unbiased, conformist and payoff/directly biased cultural transmission, as well as Model 18 which simulated the basic UB critical learners. The major addition is the use of a heatmap to explore a wide parameter space.

This can be more effective than cherry-picking particular parameter values, and provide a good visualisation of how two parameters interact. The limitation is that you still have to pick upper and lower parameter bounds, and you can only examine two parameters at a time (3D parameter space plots are possible but hard to interpret).

Exercises

1. Rather than fixed costs to CB and PB, it seems more plausible to assume that the costs of conformity and payoff bias increase with n. Calculating the majority or identifying the correct behaviour amongst 100 demonstrators would take more time and effort than calculating the majority or identifying the correct behaviour amongst 3 demonstrators. To be comparable with UB agents, who pay no cost for locating a single random agent as a demonstrator, rewrite Model 19 to give CB and PB agents the first demonstrator for free, then penalise the remaining $n-1$ demonstrators at c_c and c_p per demonstrator, i.e. $-c_c(n-1)$ and $-c_p(n-1)$ respectively. These increasing costs should introduce an interesting trade-off, given that CB and PB both work better with more demonstrators, but will also incur greater costs of learning.

2. Nakahashi et al. (2012) found that conformity is more likely to evolve in spatially varying environments as the number of traits increases. In Model 19b a single parameter G controls both the number of groups and the number of traits/behaviours. First, vary G under conditions that generally favour conformity to see whether Nakahashi et al.'s conclusion also holds here. Second, separate the number of groups and the number of traits into two independent parameters. Does increasing the number of traits increase the likelihood of conformity evolving independently of the number of groups?

3. In Model 19 f and n are fixed at the same value throughout the simulation. Modify Model 19 to make them evolve alongside the learning strategies. Each agent should have a different f and n initially drawn from a uniform or normal distribution. Then just as agents currently reproduce their learning strategy based on their relative fitness, make them also reproduce their value of f (for CB agents) and n (for PB and CB agents). Add mutation to f and n as well. In line with our observations that the strength of CB and PB increase with increasing n and f, do these parameters evolve upwards throughout the simulation? (NB you might want to set an upper limit to avoid excessive run times) What if you make the cost of PB and CB depend on n, as per Exercise 1 above?

References

Aoki, K., & Feldman, M. W. (2014). Evolution of learning strategies in temporally and spatially variable environments: a review of theory. Theoretical Population Biology, 91, 3-19.

Deffner, D., Kleinow, V., & McElreath, R. (2020). Dynamic social learning in temporally and spatially variable environments. Royal Society Open Science, 7(12), 200734.

Enquist, M., Eriksson, K., & Ghirlanda, S. (2007). Critical social learning: a solution to Rogers's paradox of nonadaptive culture. American Anthropologist, 109(4), 727-734.

Henrich, J., & Boyd, R. (1998). The evolution of conformist transmission and the emergence of between-group differences. Evolution and Human Behavior, 19(4), 215-241.

Kendal, J., Giraldeau, L. A., & Laland, K. (2009). The evolution of social learning rules: payoff-biased and frequency-dependent biased transmission. Journal of Theoretical Biology, 260(2), 210-219.

Laland, K. N. (2004). Social learning strategies. Animal Learning & Behavior, 32(1), 4-14.

Mesoudi, A. (2018). Migration, acculturation, and the maintenance of between-group cultural variation. PloS one, 13(10), e0205573.

Nakahashi, W., Wakano, J. Y., & Henrich, J. (2012). Adaptive social learning strategies in temporally and spatially varying environments: How temporal vs. spatial variation, number of cultural traits, and costs of learning influence the evolution of conformist-biased transmission, payoff-biased transmission, and individual learning. Human Nature, 23, 386-418.

Table of parameters for Models 1-10

Parameter	Definition	Model first introduced
N	Number of agents in the population.	1
t_{max}	Maximum number of timesteps or generations.	1
r_{max}	Maximum number of independent simulation runs.	1
p	Frequency of trait A.	1
p_0	Starting value of p.	1
μ	Probability of unbiased cultural mutation. Specifically, the probability of trait A mutating into trait B, or trait B mutating into trait A.	2
μ_b	Probability of biased cultural mutation. Specifically, the probability of trait B mutating into trait A.	2
s	Strength of biased transmission / cultural selection. Specifically, in Model 3 (direct bias) the probability of switching to a more favourable trait upon encountering another agent with that trait, or in Model 4 (indirect bias) the payoff advantage to trait A relative to trait B.	3
q	Frequency of a second trait in a two-trait model (trait X in Model 4), or the frequency of trait A in a second sub-population or group (Model 7).	4
q_0	Starting value of q.	4
L	Probability in two-trait models that the two traits are linked. Specifically, the probability that, if trait 1 is A, then trait 2 is X.	4
D	Strength of conformity. Specifically, the increased probability of adopting a majority trait, relative to unbiased transmission.	5

Parameter	Definition	Model first introduced
s_v	Strength of biased transmission / cultural selection under vertical cultural transmission. Specifically, the increased probability of adopting a favoured trait, relative to unbiased transmission, when only one parent holds that favoured trait.	6
s_h	Strength of biased transmission / cultural selection under horizontal cultural transmission. Specifically, the probability of switching to a more favourable trait upon encountering at least one of n demonstrators with that trait.	6
a	Probability of assortative mating under vertical cultural transmission, such that both parents have identical cultural traits.	6
n	Number of demonstrators from whom an agent learns under horizontal transmission (Model 6) or blending inheritance (Model 8).	6
m	Strength of migration. Specifically, the probability that each agent migrates to a randomly chosen sub-population.	7
e	Error in copying the traits of n demonstrators under blending inheritance. Specifically, the variance of the normal distribution with mean of the demonstrator trait value, from which the copied trait value is drawn.	8
α	Copying error in the 'Tasmanian' model of cultural gain/loss. Specifically, the amount by which the mode of a gumbel distribution is reduced relative to the highest skill level in the previous generation.	9
β	Inferential guesses or experimentation in the 'Tasmanian' model of cultural gain/loss. Specifically, the dispersion of the gumbel distribution from which the new skill level is drawn.	9
z_i	Culturally transmitted skill level of the ith agent in the 'Tasmanian' model of cultural gain/loss.	9
$\bar{z}$	Mean culturally transmitted skill level across all agents of one generation in the 'Tasmanian' model of cultural gain/loss.	9
g	The number of cultural features in Axelrod's model of polarization, with each feature taking one of ten possible trait values.	10
N_{side}	The number of agents along one side of a square grid in Axelrod's model of polarization, giving ${N_{side}}^2$ agents in total.	10