Simulation Models of Cultural Evolution

Alex Mesoudi

Model 2: Unbiased and biased mutation

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. 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 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)),
                        stringsAsFactors = FALSE)
```

```
    # 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.

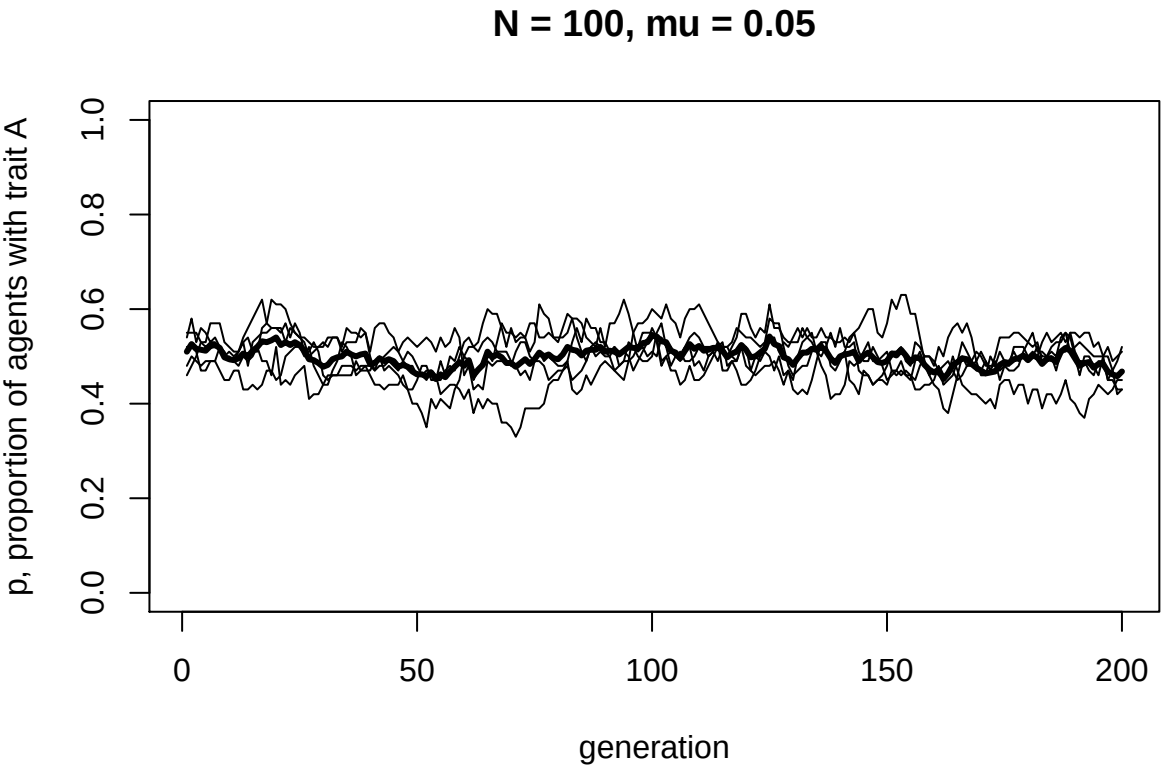
First we pre-specify the probability of mutating for each agent. This is done with the **runif** (**r**andom draws from a **unif**orm distribution) command which generates N random numbers each between 0 and 1, and puts

them into a variable called *mutate*. If the ith value in *mutate* is less than 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 $\mu = 1$. 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 $\mu = 0$. 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.

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)
```

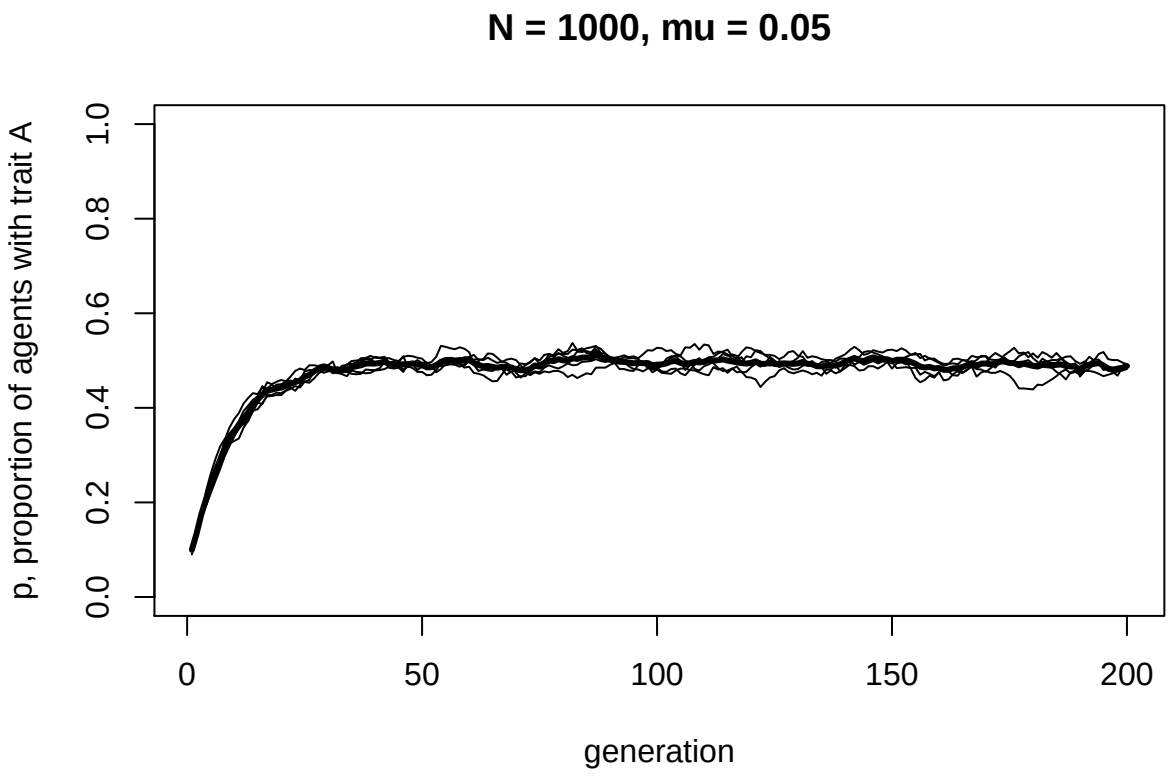


As one might expect, unbiased mutation produces random fluctuations over time, and does not alter the overall frequency of A which stays around $p = 0.5$. Because mutations from A to B are as equally likely as B to A, there is no overall directional trend.

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, 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 = 5)
```



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.

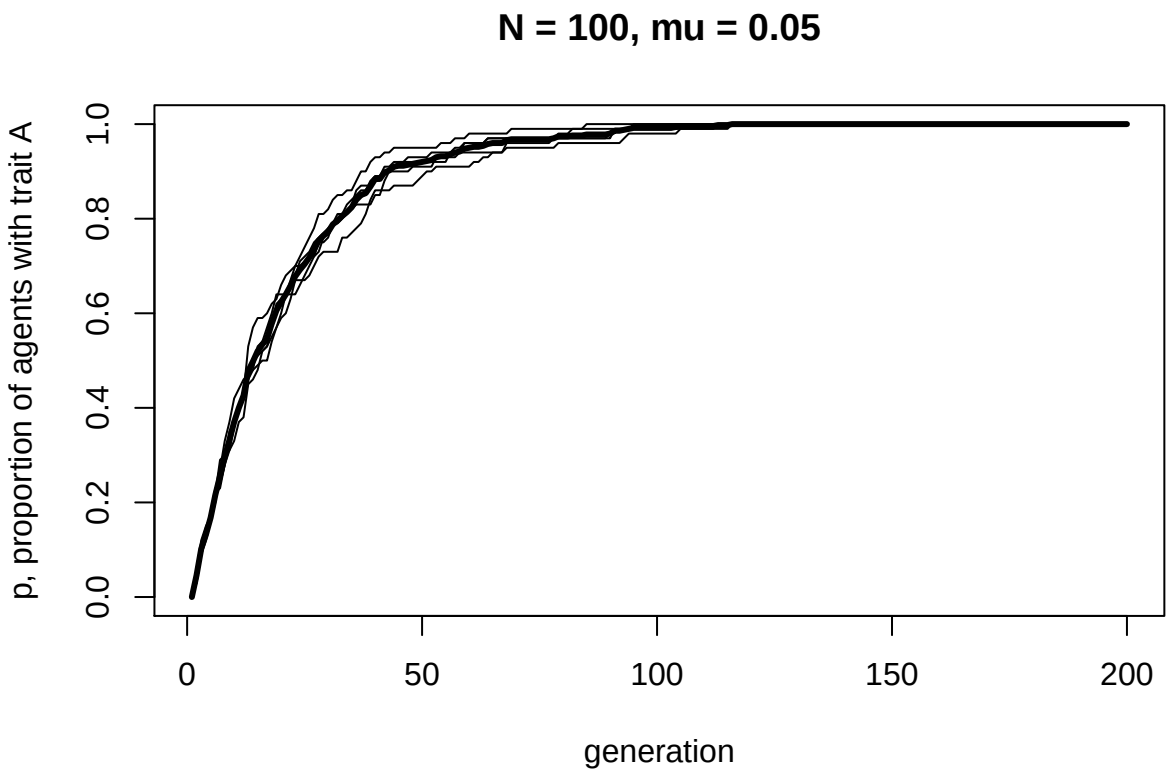
```
BiasedMutation <- function (N, mu_b, p_0, t_max, r_max) {

  # 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)),
                        stringsAsFactors = FALSE)

    # 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.

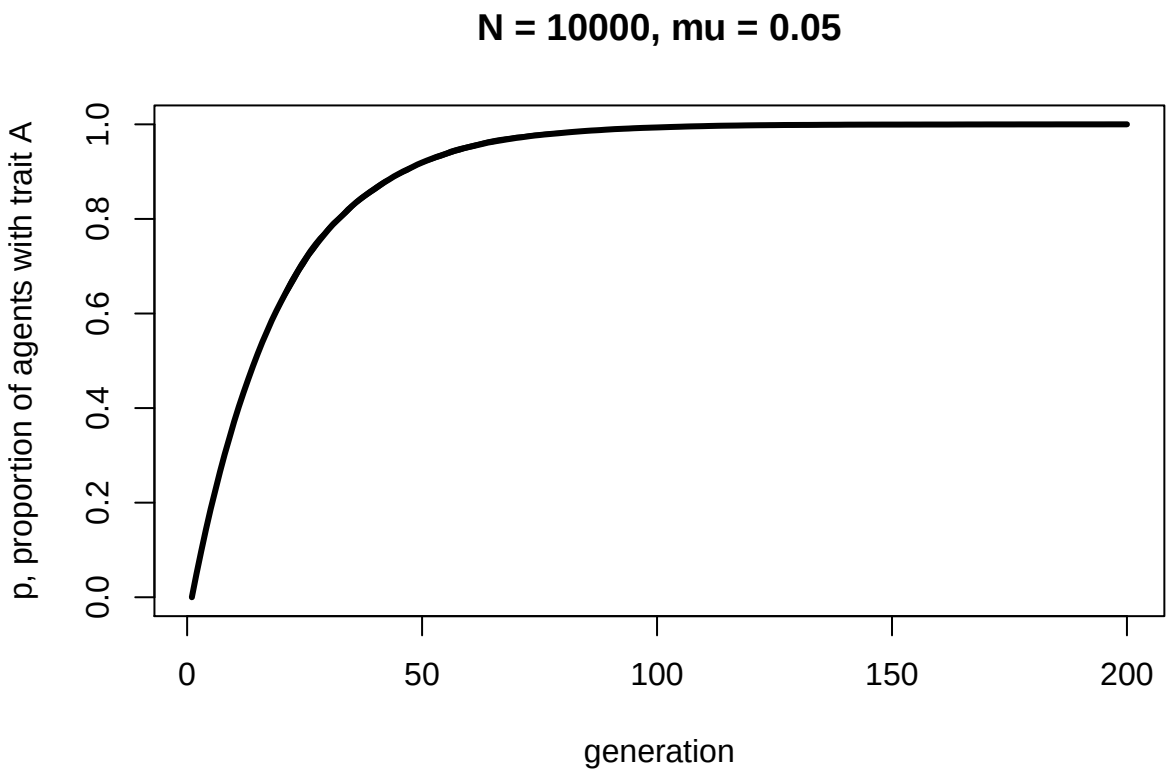
Let's see what effect this has by running **BiasedMutation**. We'll start with the population entirely composed of agents with B, i.e. $p_0 = 0$, to see how quickly and in what manner A spreads via biased mutation.

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



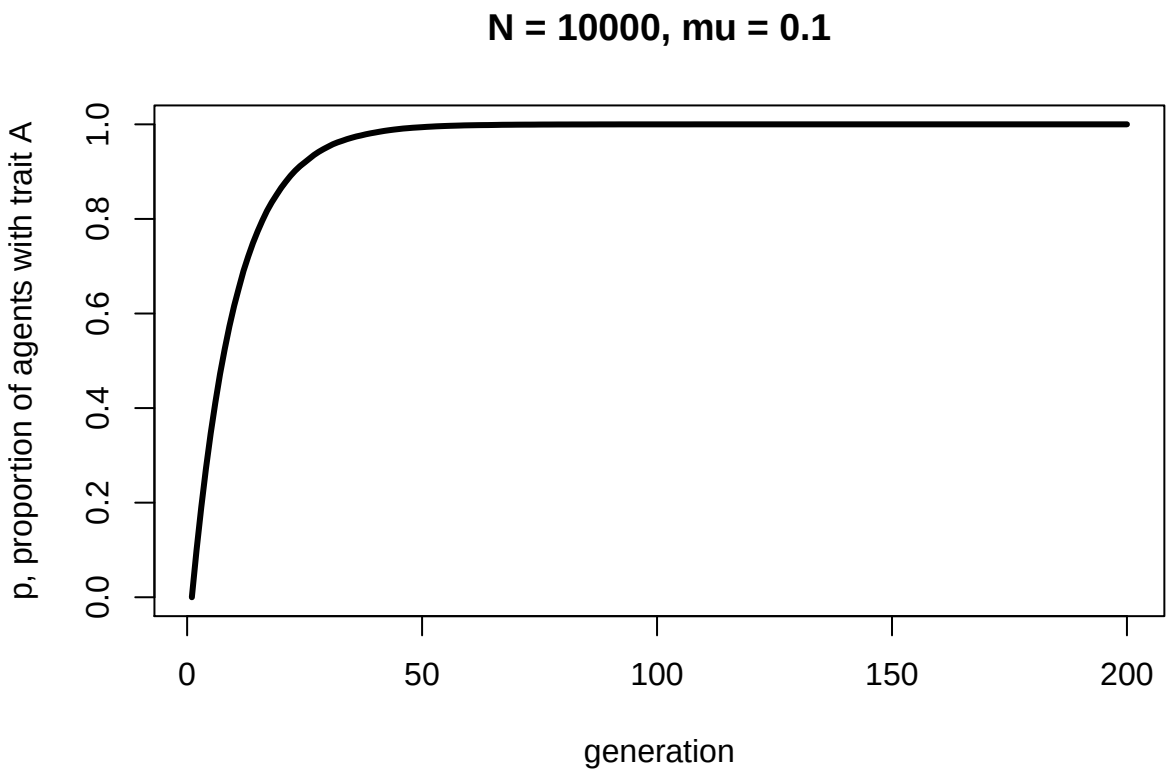
The plot should show a steep increase that slows and plateaus at $p = 1$ by around generation $t = 100$. There should be a bit of fluctuation in the different runs, but not much. Now let's try a larger sample size.

```
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 of Model 2

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 $N = 10000$. Whenever biased mutation is present ($\mu_b > 0$), the favoured trait goes to fixation, even if it is not initially present.

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.

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:

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

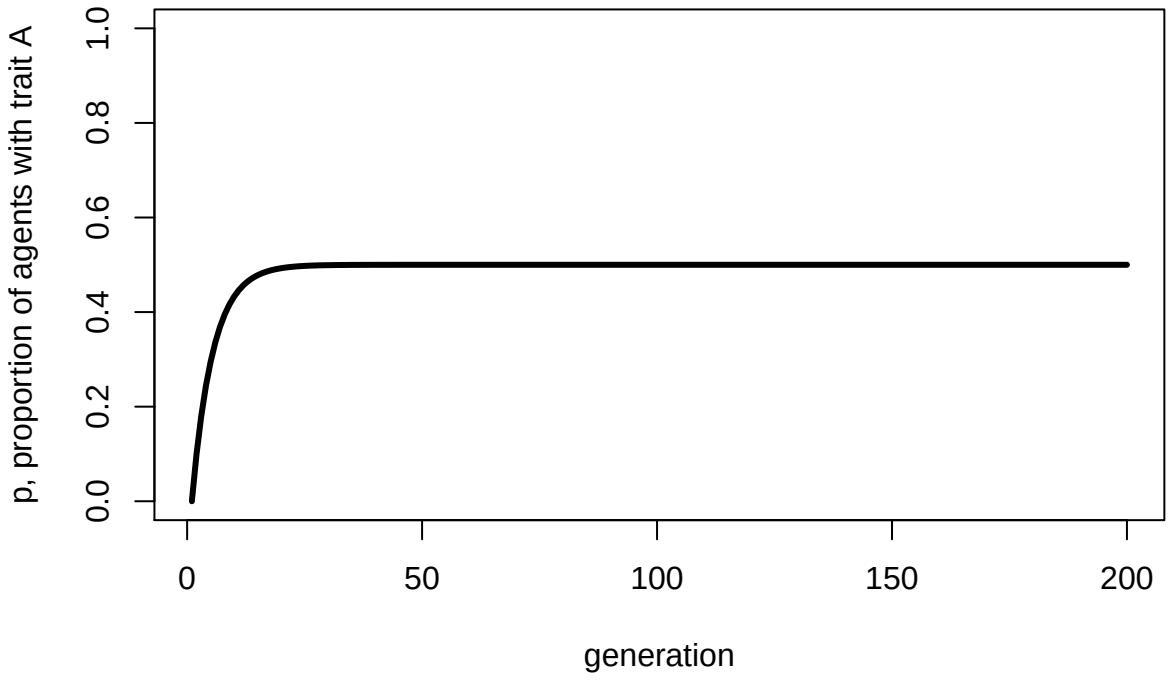
Rearranging Equation 2.2 gives:

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

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)
```



Again, 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,

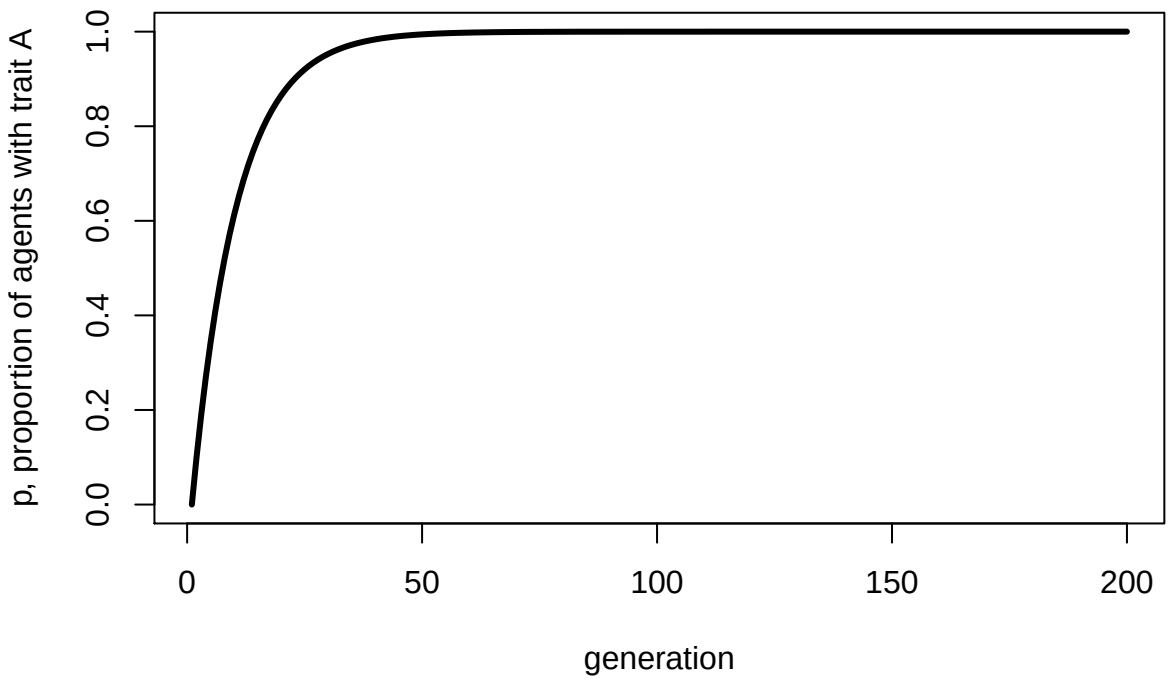
$$p' = p + (1 - p)\mu_b \tag{2.4}$$

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:

```
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.