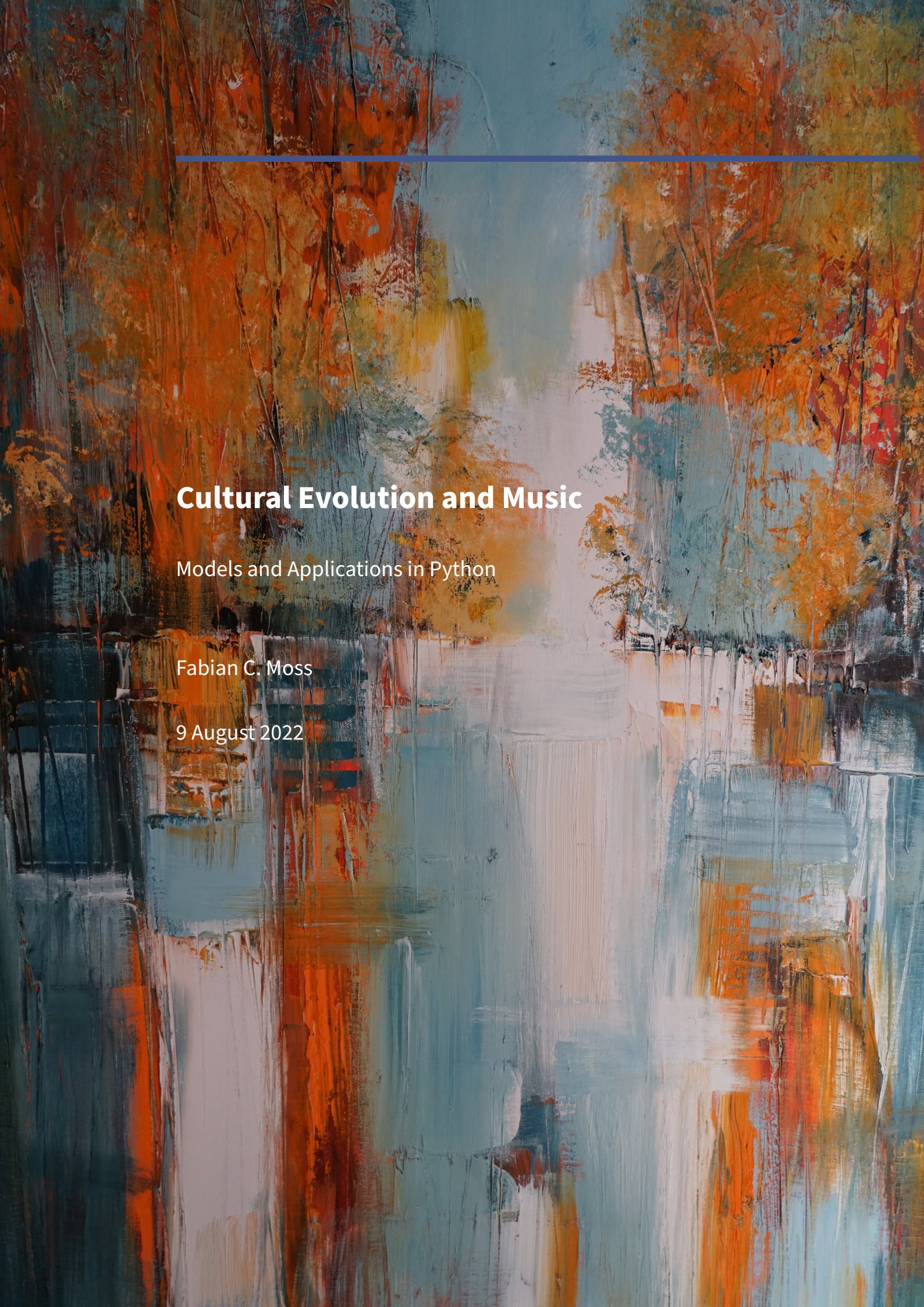


Cultural Evolution and Music

Models and Applications in Python

Fabian C. Moss

9 August 2022

Cover photo by Alan Scales on Unsplash.

Table of contents

Welcome

On these pages you will learn about cultural evolution and music. The overall aim is to attain a basic understanding of formal models in cultural evolution and learn about several recent approaches that apply them to the domain of music.

We start with a minimal introduction to the Python programming language that covers the necessary basic skills in order to follow the remainder of the book. Then, we summarize some general ideas about music and cultural evolution.

Subsequently, we follow the excellent learning path for computational models in cultural evolution provided by the book *Individual-based models of cultural evolution: A step-by-step guide using R* (Acerbi et al., 2022). These pages comprise a translation of this resource to Python. Finally, we will review and discuss a number of recent publications on music and cultural evolution in the advanced topics section at the end.

> **!** Important
>
> The material on this page has been adapted and designed for a research seminar in musicology at Würzburg University, Germany.
> Note that the materials here are still under development. Please inform me if you notice any errors or other issues.

> If you want to refer to this resource, please cite it as appropriately, e.g.: Moss, F. C. (2022). "Cultural Evolution and Music: Models and Applications in Python". https://fabianmoss.github.io/cultevopy

1. Introduction

The field of cultural evolution emerged in the 1980's, and has, in parallel with the advancement of computational facilities, gained momentum. In recent years, several approaches have attempted to apply formal models from cultural evolution to the domain of music (Youngblood et al., forthcoming).

In the present context, we first introduce some central ideas of cultural evolution and review a few major publications for the domain of music.

1.1. Music evolution

- (Wallin et al., 2001)
- (Morley, 2013)
- (Tomlinson, 2018)
- (Honing, 2018)

1.2. Music history and cultural evolution

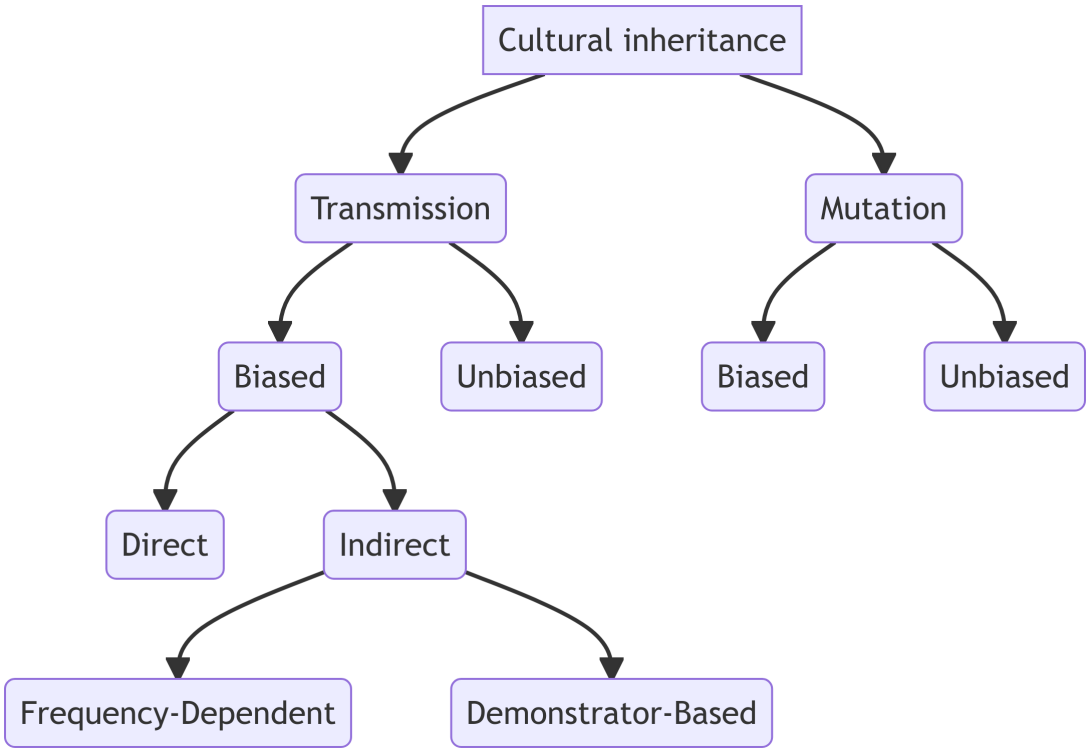
…

1.3. Central ideas and overview

Following this introduction, we introduce some minimal requirements to use Python for this course (Chapter 2). Subsequently, we introduce with six central mechanisms for cultural inheritance: unbiased transmission (Chapter 3), unbiased and biased mutation (Chapter 4), directly biased transmission (Chapter 5), frequency-dependent indirectly biased transmission (Chapter 6), and demonstrator-based indirectly biased transmission (Chapter 7).

We follow up with a chapter on vertical and horizontal transmission, and finally introduce the multiple traits model.

Schematically, this is shown in the following diagram:


Cultural inheritance
Transmission
Mutation
Biased
Unbiased
Biased
Unbiased
Direct
Indirect
Frequency-Dependent
Demonstrator-Based

2. A Python primer

> **Note**
>
> From now on, we will assume that you have a working Python installation running on your computer. You can check this by typing the following into a terminal/console/command line:
>
> ```
> python --version
> ```
>
> If the version number starts with a 3, you're all set. If not, please consider one of the many tutorials online on how to install Python.

2.1. Variables and types

Variable assignment in Python is straight-forward. You choose a name for the variable, and assign a value to it using the = operator, for example:

```
x = 100
```

assigns the value `100` to the variable `x`. If we call the variable now, we can see that it has the value we assigned to it:

```
x
```

```
100
```

Of course, we can also assign things other than numbers, for example:

```
name = "Fabian"
```

What we assigned to the variable `name` is called a *string*, it has the value `"Fabian"`. Strings are sequences of characters.

> **Tip**
>
> Note that `"Fabian"` is enclosed by double-quotes. Why is this the case? Why could we not just type `name = Fabian`?

We can also assign a list of things to a variable:

```
mylist = [1, 2, 3, "A", "B", "C"]
```

Lists are enclosed by square brackets. As you can see, Python allows you to store any kind of data in lists (here: integer numbers and character strings). However, it is good practice to include only—you'll understand later why.

Another structured data type in python are dictionaries. Dictionaries are collections of key-value pairs. For example, a dictionary `addresses` could store the email addresses of certain people:

```
addresses = {
    "Andrew" : "andrew@example.com",
    "Zoe" : "zoe@example.com"
}
```

Now, if we wanted to look up Zoe's email address, we could to so with:

```
addresses["Zoe"]
```

```
'zoe@example.com'
```

2.2. On repeat

Coding something is only useful if you can't do the job as fast or as efficient by yourself. Especially when it comes to repeating the same little thing many, many times, knowing how to code comes in handy.

As a simple example, imagine you want to write down all numbers from 1 to 10, or from 1 to 100, or… you get the idea. In Python, you would do it as follows:

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

You see that this is not exactly what we wanted. We're seeing numbers from 0 to 9, but we wanted everything from 1 to 10. Before we fix the code to produce the desired result, let's explain the bits and pieces of the code above. What we just did was to use a so-called *for-loop*, probably the most common way to repeat things in Python. First we create an *iterator variable* `i` (we could have named any other variable name as well), which takes its value from the list of numbers specified by `range(10)`. If only one number `n` is provided to `range(n)`, it will enumerate all numbers from 0 to `n-1`. If two arguments are provided, the first one determines the starting number, and the second one stands for the terminating number minus one—confusing, right? So, in order to enumerate all numbers from 1 to 10, we have to write `range(1,11)`. Finally, the `print` function outputs the value of `i` for each iteration.

```
for i in range(1,11):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

```
10
```

Voilà!

2.3. Functions

With more and more experience in programming, it is likely that your code will become more and more complex. That means that it will become harder to keep track of what every piece of it is supposed to do. A good strategy to deal with this is to aim for writing code that is *modular*: it can be broken down into smaller units (modules) that are easier to understand. Moreover, it is sometimes necessary to reuse the same code several times. It would, however, be inefficient to write the same lines over and over again. With your code being modular you can wrap the pieces that you need in several places into a *function*.

Let's look at an example! Assume, your (fairly) complex code involves calculating the sum of the products of two numbers. In Python, we use the + operator to calculate sums and the ** operator to raise a number to a certain power (**2 for the square of a number).

```
x = 3
y = 5

sum_of_squares = x**2 + y**2
```

The variable `sum_of_squares` now contains the sum of squares of `x=3` and `y=5`. We can inspect the result by calling the variable:

```
sum_of_squares
```

```
34
```

Now, imagine that you would have to do the same calculation several times for different combinations of values for x and y (and always keeping in mind that this stands in for much more complex examples with several lines of code). We can this code in a function:

```
def func_sum_of_squares(x, y):
    return x**2 + y**2
```

Now, each time we want to calculate a sum of squares, we can do so by simply invoking

```
func_sum_of_squares(5,4)
```

```
41
```

And, of course, we could chose a shorter name for the function as well:

```
f = func_sum_of_squares

f(5,4)
```

```
41
```

2.4. Libraries you'll love

Luckily, you don't have to programm all functions by yourself. There is a huge community of Python programmers out there that works and collaborates on several *libraries*. A library is (more or less) simply a collection of certain functions (and some more, but we don't get into this here). This means, instead of writing a function yourself, you can rely on functions that someone else has programmed.

> **Danger**
>
> Whether a Python library or function does actually do what it promises is another story. Popular libraries with tens of thousands of users are very trust-worthy because you can be almost sure that someone would have noticed erroneous behavior. But it is certainly possible that badly-maintained libraries contain errors. So be prudent when using the code of others.

One of the most popular Python libaries is NumPy for numerical computations. We will rely a lot on the functions in this library, especially in order to draw random samples—more on this later! To use the functions or variables from this library, they have to be *imported* so that you can use them. There are several ways to do this. For example, you can import the libary entirely:

```
import numpy
```

Now, you can use the (approximated) value of π stored in this library by typing

```
numpy.pi
```

```
3.141592653589793
```

A different way is to just import everything from the library by writing

```
from numpy import *
```

Here, the * stands for 'everything'. Now, to use the value of π we could simply type

```
pi
```

```
3.141592653589793
```

This is, however discouraged for the following reason: imagine we had another library, `numpy2` that also stores the value of π, but less precisely (e.g. only as `3.14`). If we write

```
from numpy import *
from numpy2 import *
```

We would have imported the variables holding the value of π from both libraries. But, because they have the same name `pi`. In this case, `pi` would equal `3.14` because we imported `numpy2` last. This is confusing and shouldn't be so! To avoid this, it is better to keep references to imported libraries explicit. In order not to have to type too much (we're all lazy, after all), we can define an alias for the library.

```
import numpy as np
np.pi
```

```
3.141592653589793
```

All functions of NumPy are now accessible with the prefix `np.`. You can choose any alias when importing a library (it can even by longer than the library name) but certain conventions have emerged that you're encouraged to follow. Importing the most commonly used Python libraries for data-science tasks ("The data science triad"), use the following:

```
import numpy as np # for numerical computations
import pandas as pd # for tabular data
import matplotlib.pyplot as plt # for data visualization
```

We will use all three of them in the following chapters and you'll learn to love them.

Part I.

Basic models

3. Unbiased transmission

What happens if people just randomly copy?

> **Note**
>
> This chapter is based on “Chapter 1: Unbiased transmission” in Acerbi et al. (2022).

In this chapter, we introduce the most basic model for cultural inheritance: unbiased transmission. This process quite literally corresponds to randomly copying traits from previous generations, without any further distinctions and constraints. While this is obviously a too reductive model for how cultural transmission works, it is ideally suited to get us started with the enterprise of modeling evolutionary processes involving random variation.

First we import some modules.

```
import numpy as np
import pandas as pd
```

Because we will model evolutionary processes that are not strictly deterministic, we need to simulate variations due to random change. For this, we can use the *default random number generator* from the NumPy library and store it in the variable `rng`.

```
rng = np.random.default_rng()
```

Next, we define some basic variables that we take into account for our first model. We consider a population of $N = 100$ individuals as well as a time-frame of $t_{max} = 100$ generations.

3.1. Simulating a population

```
N = 100
t_max = 100
```

> **Note**
>
> In general, we use the variable `t` to designate generation counts.

Now we create a variable `population` that will store the data about our simulated population. This population has either of two traits `"A"` and `"B"`, with a certain probability. We store all of this in a so-called 'data frame', which is a somewhat fancy, Pandas-specific term for a table.

```
population = pd.DataFrame(
    {"trait": rng.choice(["A", "B"], size=N, replace=True)}
)
```

Let's take this code apart to understand it better. From the Pandas library, which we imported as the alias `pd`, we create a `DataFrame` object. The data contained in this the data frame `population` is specified via a dictionary that has `"trait"` as its key and a fairly complex expression starting with the random number generator `rng` as its value. What this value says is, from the list `["A", "B"]` choose randomly `N` instances with replacement (if `replace` were set to `False`, we could at most sample 2 values from the list). So, the data frame `population` should contain 100 randomly sampled values of A's and B's. Let's confirm this:

```
population.head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	trait
0	B
1	A
2	B
3	A
4	A

As you can see, `population` stores a table (many of the 100 rows are omitted here for display reasons) and a single column called 'trait'. The `.head()` method appended to the `population` data frame shows restricts the output to only the first 5 rows (0 through 4). Each row in the 'trait' column contains either the value A or B. To the left of the data frame you can see the numbers of rows explicitly spelled out. This is called the data frame's *index*.

> **Note**
>
> A and B are just placeholder names for any of two mutually exclusive cultural traits. These could be, for example, preference for red over white whine (ignoring people who like rosé as well as people who have no preference). You see already here that this is a massive oversimplification of actual taste preferences. The point here is not to construct a plausible model but rather to gradually build up a simple one in order to understand well its inner workings.
> It will help to pause for a moment and to think of other examples that "A" and "B" could stand for. Can you come up with a music-related one?

We can count the number of A's and B's as follows:

```
population["trait"].value_counts()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	trait
B	55
A	45

You can read the above code as “From the population table, select the ‘trait’ colum and count its values.”. Since there were only two values to sample from and they were randomly (uniformly) sampled, the number of A’s and the number of B’s should be approximately equal. We can obtain their relative frequencies by adding setting the `normalize` keyword to `True`:

```
population["trait"].value_counts(normalize=True)
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	trait
B	0.55
A	0.45

3.2. Tracing cultural change

We now create a second data frame `output` in which we will store the output of our model. This data frame has two columns: `generation`, which is the number of the simulated generation, and p which stands for “the probability of an individual of having trait A”.

```
output = pd.DataFrame(
    {
        "generation": np.arange(t_max, dtype=int),
        "p": [np.nan] * t_max
    }
)
```

The `generation` column contains all numbers from `0` to `t_max - 1`. Because we count the numbers of generations (rather than assuming a time-continuous process), we specified that numbers in this column have to be intergers (`dtype=int`). The values for the p column must look cryptic. It literally says: put the `np.nan` value `t_max` times into the p colum. `np.nan` stands for “not a number” (from the NumPy library), since we haven’t assigned any values to this probability yet.

```
output.head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	generation	p
0	0	NaN
1	1	NaN
2	2	NaN
3	3	NaN
4	4	NaN

Don't worry that both the index and the 'generation' column contain all numbers from 0 to 99. We need this later when things become more involved.

As the saying goes, from nothing comes nothing, so we have to start somewhere, meaning that we need to assume that the initial probability of having trait A in our population is an actual number. The most sensible thing is to start with the proportions of A and B in our sampled population as a starting value.

So, we approximate the probability of an individual having trait A with the relative frequency of trait A in the population:

```
population["trait"].value_counts(normalize=True)["A"]
```

```
0.45
```

You already know this code from above, we just added the `["A"]` part at the end to select only the relative frequencies of trait A. We want to set this as the value of p of the first generation. This can be achieved with the `.loc` (location) method:

```
output.loc[0, "p"] =
 ↪  population["trait"].value_counts(normalize=True)["A"]
```

In words, this reads: "Set location 0 (first row) in the p column of the `output` data frame to the relative frequency of the trait 'A' in the population."

3.3. Iterating over generations

Recall that we are trying to observe cultural change over the course of `t_max = 100` generations. We thus simply repeat what we just did for the first generation: based on the relative frequencies of A's and B's in the previous generation, we sample the traits of 100 new individuals for the next generation.

```
for t in range(1, t_max):
    # Copy the population data frame to `previous_population`
    previous_population = population.copy()

    # Randomly copy from previous generation's individuals
    new_population = previous_population["trait"].sample(N,
↪   replace=True).to_frame()

    # Get p and put it into the output slot for this generation t
    output.loc[t, "p"] = new_population[ new_population["trait"] ==
↪   "A"].shape[0] / N
```

This procedure assignes a probability of having trait "A" for each generation (each row of the p colum is filled now):

```
output.head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	generation	p
0	0	0.45
1	1	0.46
2	2	0.35
3	3	0.47
4	4	0.34

To make things easier, we wrap the above code in a function that we'll call `unbiased_transmission` that can take different values for the population size `N` and number of generations `t_max` as parameters. The code below is exactly the same as above.

```
def unbiased_transmission_1(N, t_max):
    population = pd.DataFrame({"trait": rng.choice(["A", "B"], size=N,
↪   replace=True)})

    output = pd.DataFrame({"generation": np.arange(t_max, dtype=int),
↪   "p": [np.nan] * t_max })

    output.loc[0, "p"] = population[ population["trait"] ==
↪   "A"].shape[0] / N

    for t in range(1, t_max):
        # Copy the population tibble to previous_population tibble
        previous_population = population.copy()

        # Randomly copy from previous generation's individuals
        new_population = previous_population["trait"].sample(N,
↪   replace=True).to_frame()

        # Get p and put it into the output slot for this generation t
        output.loc[t, "p"] = new_population[ new_population["trait"] ==
↪   "A"].shape[0] / N

    return output
```

```
data_model = unbiased_transmission_1(N=100, t_max=200)
```

```
def plot_single_run(data_model):
    data_model["p"].plot(ylim=(0,1))
```

```
plot_single_run(data_model)
```

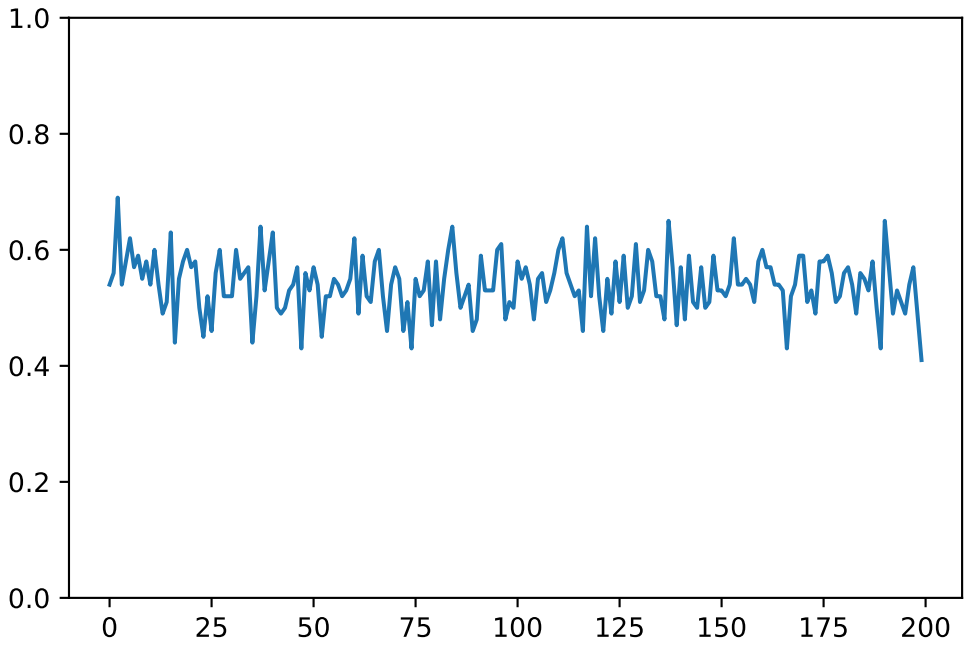


Figure 3.1.: Single run of the unbiased transmission model for a population of $N = 100$ individuals and $t_{max} = 200$ generations.

```
data_model = unbiased_transmission_1(N=10_000, t_max=200)
```

```
plot_single_run(data_model)
```

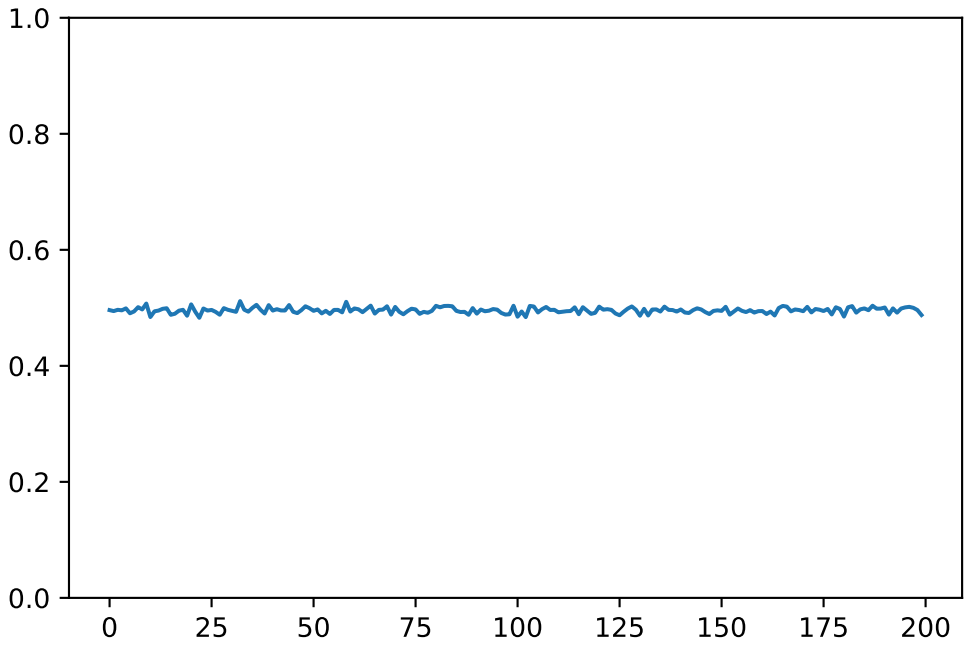


Figure 3.2.: Single run of the unbiased transmission model for a population of $N = 10,000$ individuals and $t_{max} = 200$ generations.

```
def unbiased_transmission_2(N, t_max, r_max):
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
        # Create first generation
        population = pd.DataFrame({"trait": rng.choice(["A", "B"],
↪   size=N, replace=True)})

        # Add first generation's p for run r
        output.loc[ r * t_max, "p"] = population[ population["trait"] ==
↪   "A" ].shape[0] / N

        # For each generation
        for t in range(1,t_max):
            # Copy individuals to previous_population DataFrame
```

```
            previous_population = population.copy()

            # Randomly compy from previous generation
            population = population["trait"].sample(N,
    ↪  replace=True).to_frame()

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[r * t_max + t, "p"] = population[
    ↪  population["trait"] == "A" ].shape[0] / N

    return output
```

```
unbiased_transmission_2(100, 100, 3).head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	generation	p	run
0	0	0.46	0
1	1	0.48	0
2	2	0.53	0
3	3	0.58	0
4	4	0.49	0

> **Tip**
>
> Why could we append `.head()` to the `unbiased_transmission_2` function?

```
data_model = unbiased_transmission_2(N=100, t_max=200, r_max=5)
```

```
def plot_multiple_runs(data_model):
    groups = data_model.groupby("run")
    for _, g in groups:
```

```
        g.index = g["generation"]
        g["p"].plot(lw=.5, ylim=(0,1))

    data_model.groupby("generation")["p"].mean().plot(c="k", lw="1")
```

```
plot_multiple_runs(data_model)
```

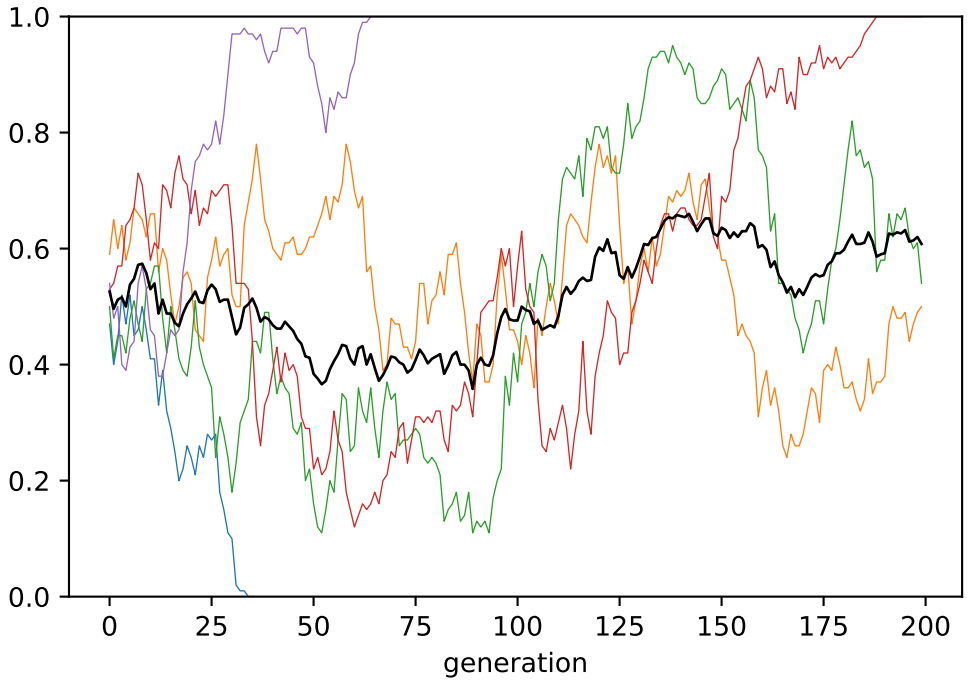


Figure 3.3.: Multiple runs of the unbiased transmission model for a population of $N = 100$ individuals, with average (black line).

```
data_model = unbiased_transmission_2(N=10_000, t_max=200, r_max=5)
```

```
plot_multiple_runs(data_model)
```

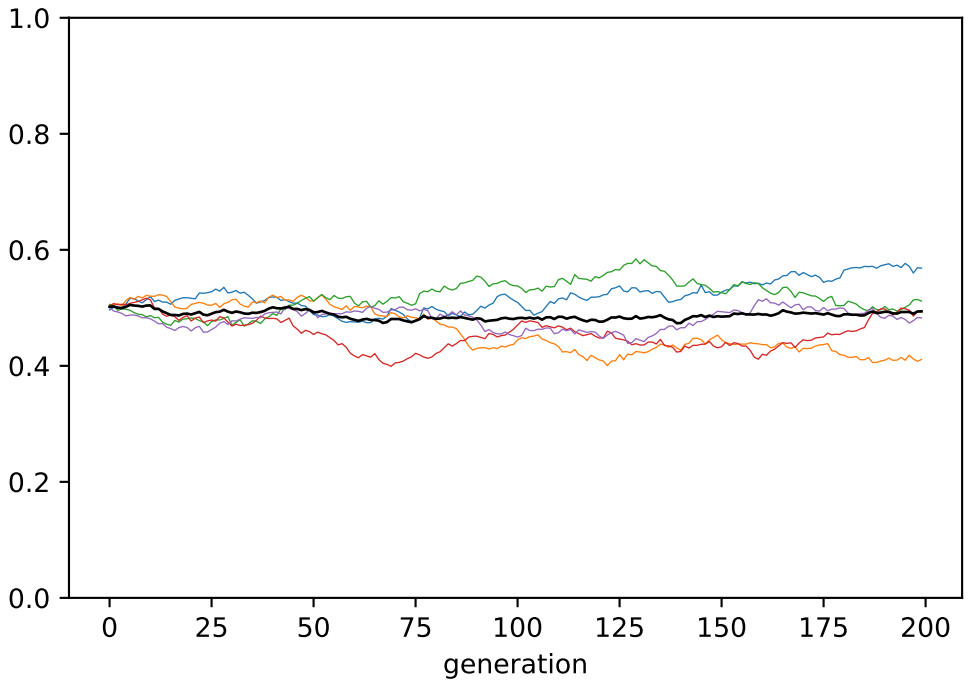
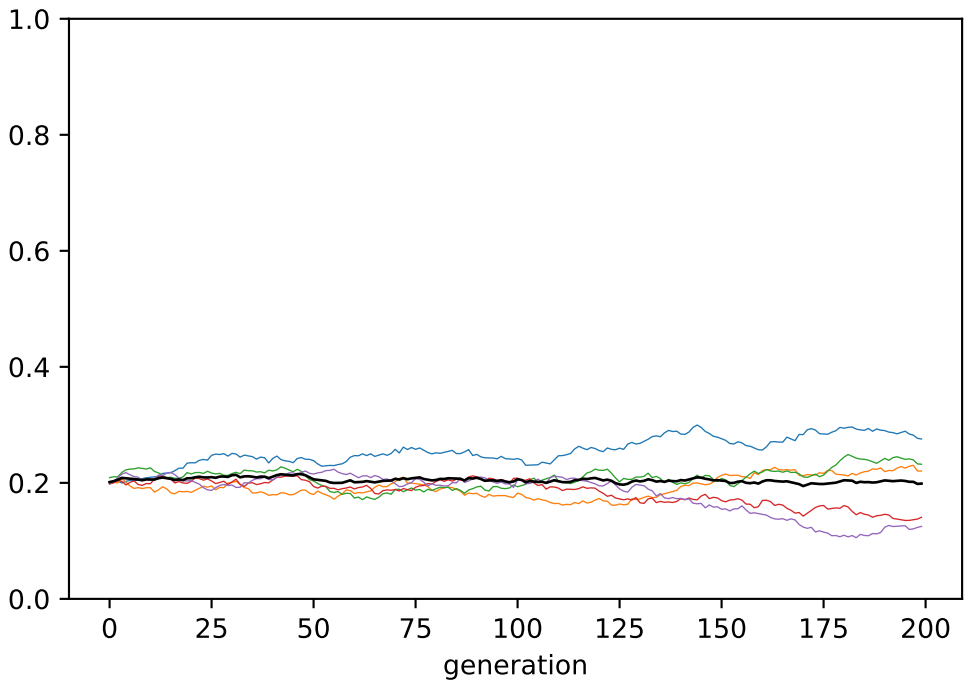


Figure 3.4.: Multiple runs of the unbiased transmission model for a population of $N = 10,000$ individuals, with average (black line).

```
def unbiased_transmission_3(N, p_0, t_max, r_max):
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
        # Create first generation
        population = pd.DataFrame({"trait": rng.choice(["A", "B"],
↪   size=N, replace=True, p=[p_0, 1 - p_0])})

        # Add first generation's p for run r
        output.loc[ r * t_max, "p"] = population[ population["trait"] ==
↪   "A" ].shape[0] / N

        # For each generation
        for t in range(1,t_max):
            # Copy individuals to previous_population DataFrame
```

```
            previous_population = population

            # Randomly compy from previous generation
            population = population["trait"].sample(N,
 ↪  replace=True).to_frame()

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[r * t_max + t, "p"] = population[
 ↪  population["trait"] == "A" ].shape[0] / N

    return output
```

```
data_model = unbiased_transmission_3(10_000, p_0=.2, t_max=200,
 ↪  r_max=5)
plot_multiple_runs(data_model)
```



4. Unbiased and biased mutation

> **Note**
>
> This chapter is based on “Chapter 2: Unbiased and biased mutation” in Acerbi et al. (2022).

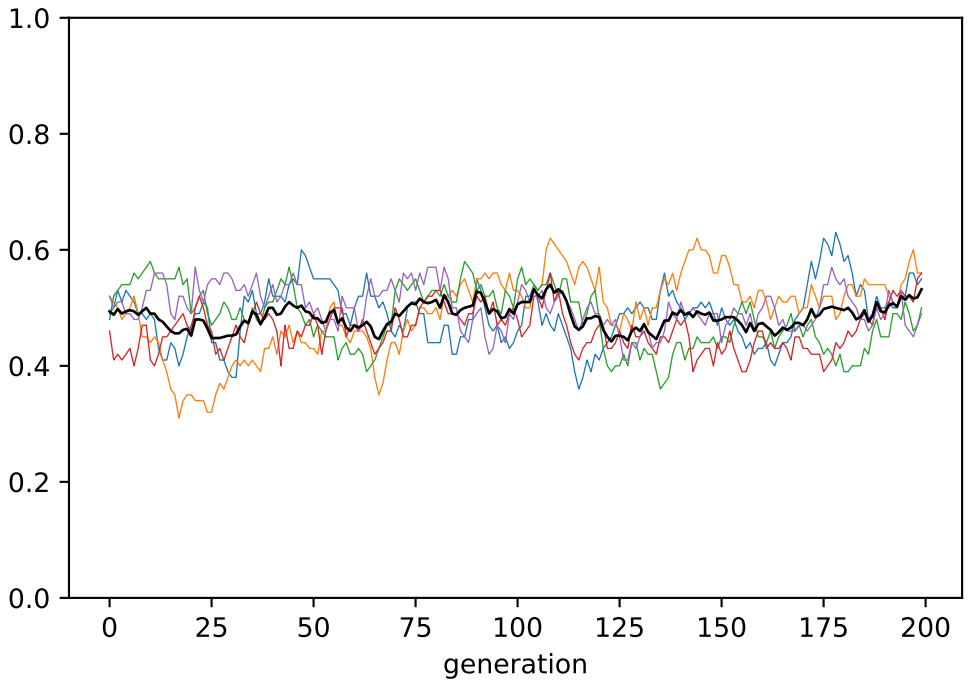
```
import numpy as np
rng = np.random.default_rng()

import pandas as pd
```

```
def unbiased_mutation(N, mu, p_0, t_max, r_max):
    # Create an output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
        # Create first generation
        population = pd.DataFrame({"trait": rng.choice(["A", "B"],
↪   size=N, replace=True, p=[p_0, 1 - p_0])})

        # Add first generation's p for run r
        output.loc[ r * t_max, "p"] = population[ population["trait"] ==
↪   "A" ].shape[0] / N

        # For each generation
        for t in range(1,t_max):
            # Copy individuals to previous_population DataFrame
            previous_population = population.copy()

            # Determine "mutant" individuals
```

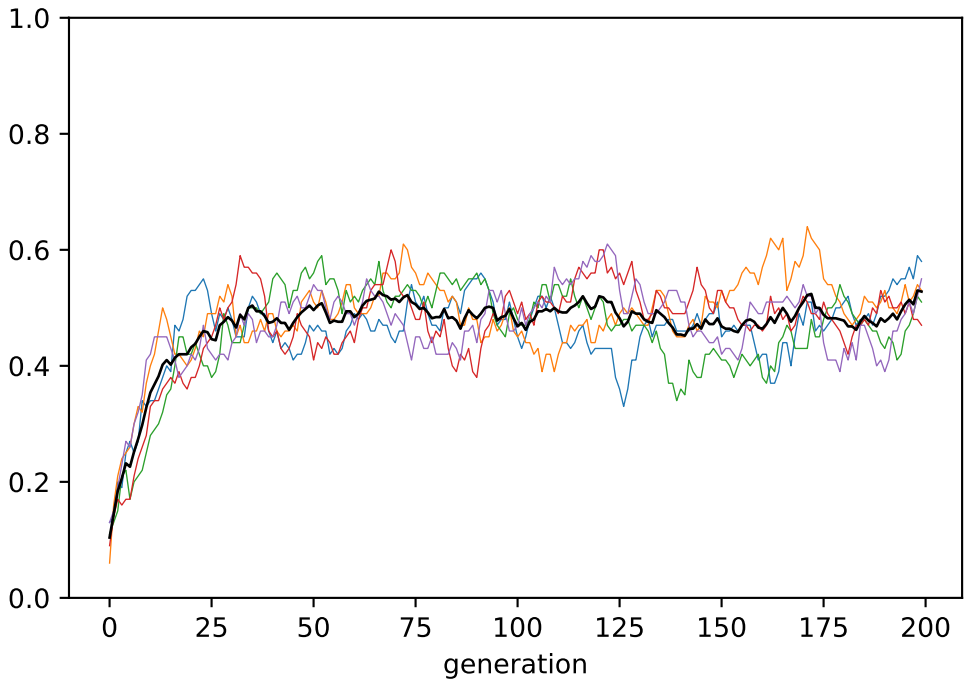
```
            mutate = rng.choice([True, False], size=N, p=[mu, 1-mu],
↪   replace=True)

            # TODO: Something is off here! Changing the order of the
             ↪  conditions affects
            # the result. Should be constant with random noise but
             ↪  converges to either A or B

            # If there are "mutants" from A to B
            conditionA = mutate & (previous_population["trait"] == "A")
            if conditionA.sum() > 0:
                population.loc[conditionA, "trait"] = "B"

            # If there are "mutants" from B to A
            conditionB = mutate & (previous_population["trait"] == "B")
            if conditionB.sum() > 0:
                population.loc[conditionB, "trait"] = "A"

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[r * t_max + t, "p"] = population[
↪   population["trait"] == "A" ].shape[0] / N

    return output
```

```
def plot_multiple_runs(data_model):
    groups = data_model.groupby("run")
    for _, g in groups:
        g.index = g["generation"]
        g["p"].plot(lw=.5, ylim=(0,1))

    data_model.groupby("generation")["p"].mean().plot(c="k", lw="1")
```

```
data_model = unbiased_mutation(N=100, mu=.05, p_0=0.5, t_max=200,
↪   r_max=5)
plot_multiple_runs(data_model)
```



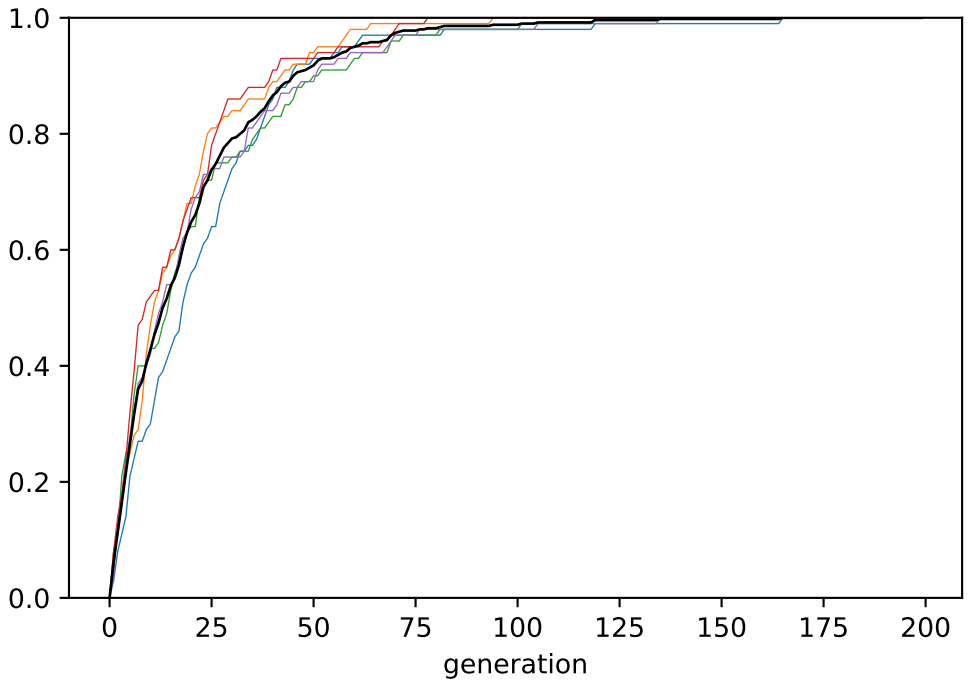

```
data_model = unbiased_mutation(N=100, mu=.05, p_0=0.1, t_max=200,
 ↪  r_max=5)
plot_multiple_runs(data_model)
```



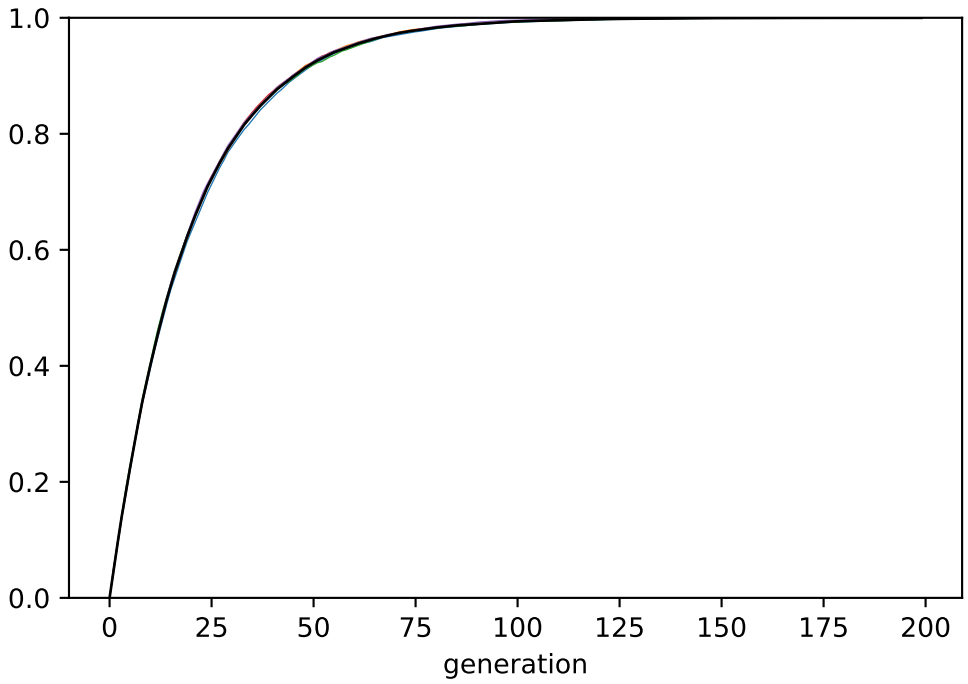

```
def biased_mutation(N, mu_b, p_0, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
        # Create first generation
        population = pd.DataFrame({"trait": rng.choice(["A", "B"],
↪   size=N, replace=True, p=[p_0, 1 - p_0])})

        # Add first generation's p for run r
        output.loc[ r * t_max, "p"] = population[ population["trait"] ==
↪   "A" ].shape[0] / N

        # For each generation
        for t in range(1,t_max):
            # Copy individuals to previous_population DataFrame
            previous_population = population.copy()
```

```
            # Determine "mutant" individuals
            mutate = rng.choice([True, False], size=N, p=[mu_b, 1-mu_b],
↪   replace=True)

            # TODO: Something is off here! Changing the order of the
             ↪  conditions affects
            # the result. Should be constant with random noise but
             ↪  converges to either A or B

            # If there are "mutants" from B to A
            conditionB = mutate & (previous_population["trait"] == "B")
            if conditionB.sum() > 0:
                population.loc[conditionB, "trait"] = "A"

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[r * t_max + t, "p"] = population[
↪   population["trait"] == "A" ].shape[0] / N

    return output
```

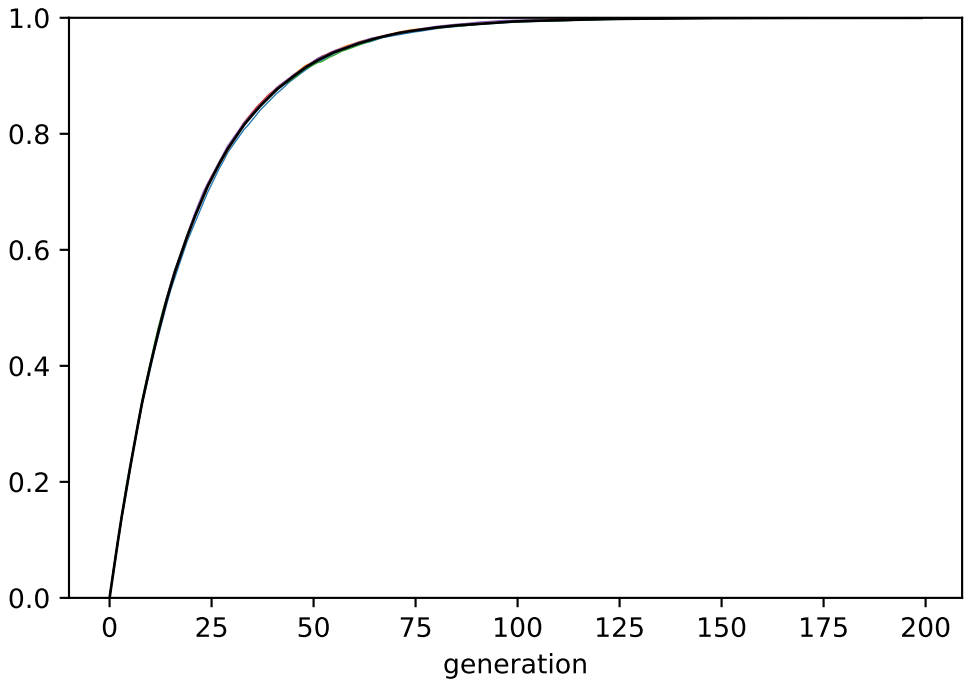
```
data_model = biased_mutation(N = 100, mu_b = 0.05, p_0 = 0, t_max =
 ↪  200, r_max = 5)
plot_multiple_runs(data_model)
```




```
data_model = biased_mutation(N = 10000, mu_b = 0.05, p_0 = 0, t_max =
 ↪  200, r_max = 5)
plot_multiple_runs(data_model)
```




```
data_model <- biased_mutation(N = 10000, mu_b = 0.1, p_0 = 0, t_max =
 ↪  200, r_max = 5)
plot_multiple_runs(data_model)
```


1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
200
generation

5. Biased transmission: direct bias

> **Note**
>
> This chapter is based on “Chapter 3: Biased transmission: direct bias” in Acerbi et al. (2022).

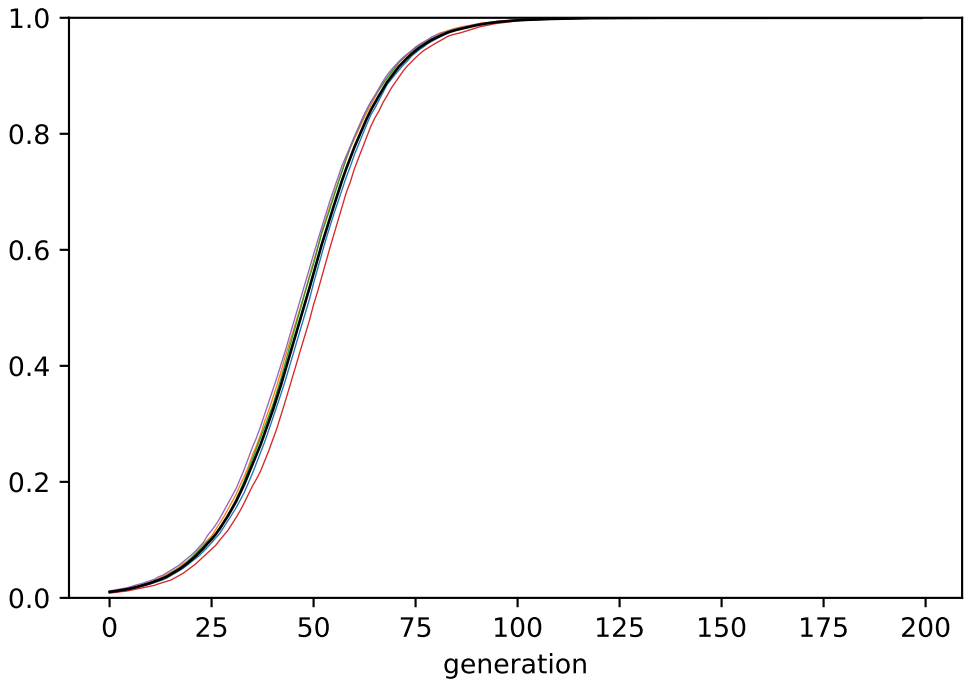
```
import numpy as np
rng = np.random.default_rng()
import pandas as pd
```

```
def plot_multiple_runs(data_model):
    groups = data_model.groupby("run")
    for _, g in groups:
        g.index = g["generation"]
        g["p"].plot(lw=.5, ylim=(0,1))

    data_model.groupby("generation")["p"].mean().plot(c="k", lw="1")
```

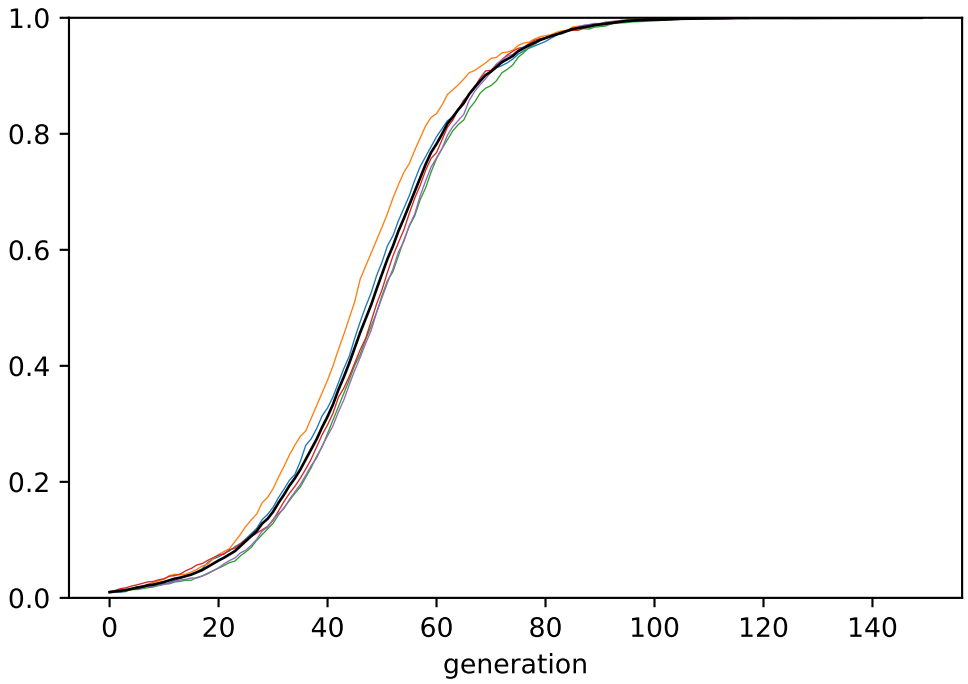
```
def biased_transmission_direct(N, s_a, s_b, p_0, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
        # Create first generation
        population = pd.DataFrame({"trait": rng.choice(["A", "B"],
↪   size=N, replace=True, p=[p_0, 1 - p_0])})

        # Add first generation's p for run r
        output.loc[ r * t_max, "p"] = population[ population["trait"] ==
↪   "A" ].shape[0] / N
```

```
        # For each generation
        for t in range(1,t_max):
            # Copy individuals to previous_population DataFrame
            previous_population = population.copy()

            # For each individual, pick a random individual from the
             ↪  previous generation
            demonstrator_trait = previous_population["trait"].sample(N,
↪   replace=True).reset_index()

            # Biased probabilities to copy
            copy_a = rng.choice([True, False], size=N, replace=True,
↪   p=[s_a, 1 - s_a])
            copy_b = rng.choice([True, False], size=N, replace=True,
↪   p=[s_b, 1 - s_b])

            # If the demonstrator has trait A and the individual wants
             ↪  to copy A, then copy A
            condition = copy_a & (demonstrator_trait["trait"] == "A")
            if condition.sum() > 0:
                population.loc[condition, "trait"] = "A"

            # If the demonstrator has trait B and the individual wants
             ↪  to copy B, then copy B
            condition = copy_b & (demonstrator_trait["trait"] == "B")
            if condition.sum() > 0:
                population.loc[condition, "trait"] = "B"

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[r * t_max + t, "p"] = population[
↪   population["trait"] == "A" ].shape[0] / N

    return output
```

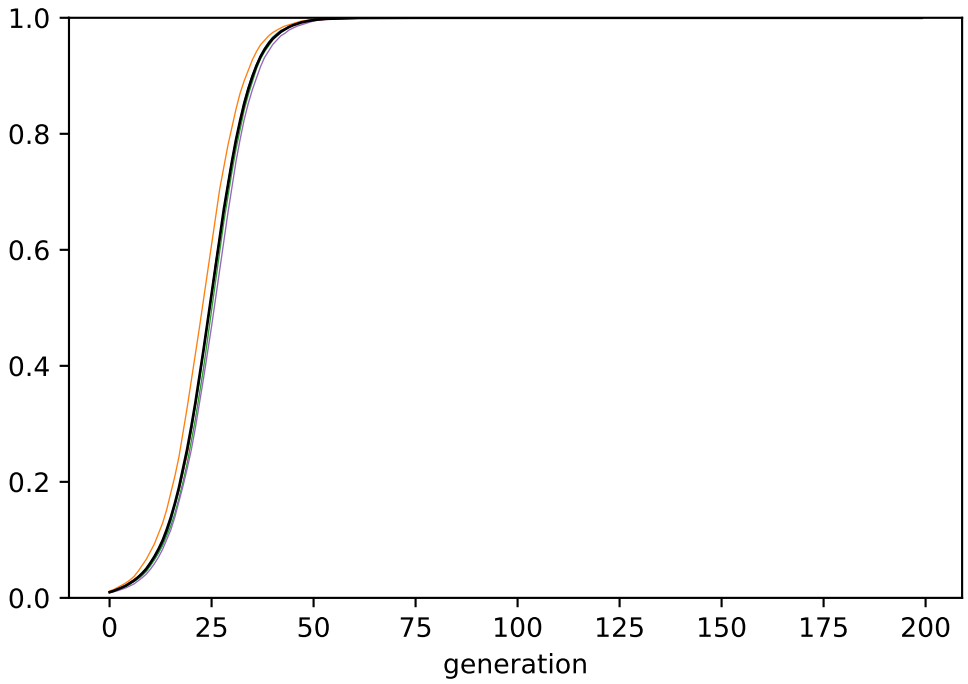
```
data_model = biased_transmission_direct(N=10_000, s_a=.1, s_b=0,
                                        p_0=.01, t_max=200, r_max=5)
plot_multiple_runs(data_model)
```




```
data_model = biased_transmission_direct(N=10_000, s_a=.6, s_b=.5,
                                        p_0=.01, t_max=150, r_max=5)
plot_multiple_runs(data_model)
```




```
data_model = biased_transmission_direct(N=10_000, s_a=.2, s_b=0,
                                        p_0=.01, t_max=200, r_max=5)
plot_multiple_runs(data_model)
```


1.0
0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
200
generation

6. Biased transmission: frequency-dependent indirect bias

> **Note**
>
> This chapter is based on "Chapter 4: Biased transmission: frequency-dependent indirect bias" in Acerbi et al. (2022).

```
import numpy as np
rng = np.random.default_rng()

import pandas as pd
```

```
def plot_multiple_runs(data_model):
    groups = data_model.groupby("run")
    for _, g in groups:
        g.index = g["generation"]
        g["p"].plot(lw=.5, ylim=(0,1))

    data_model.groupby("generation")["p"].mean().plot(c="k", lw="1")
```

```
N = 100
p_0 = .5
D = 1.
```

```
# Create first generation
population = pd.DataFrame({"trait": rng.choice(["A", "B"], size=N,
 ↪  replace=True, p=[p_0, 1-p_0])})
```

```
# Create a DataFrame with a set of 3 randomly-picked demonstrators for
 ↪  each agent
```

```
demonstrators = pd.DataFrame({
    "dem1" : population["trait"].sample(N, replace=True).values,
    "dem2" : population["trait"].sample(N, replace=True).values,
    "dem3" : population["trait"].sample(N, replace=True).values
})
```

```
# Visualize the DataFrame
demonstrators.head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	dem1	dem2	dem3
0	A	A	B
1	B	B	B
2	B	B	A
3	A	B	A
4	A	B	B

```
# Get the number of A's in each 3-demonstrator combination
num_As = (demonstrators == "A").apply(sum, axis=1)
num_As.head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	0
0	2
1	0
2	1
3	2
4	1

```
# For 3-demonstrator combinations with all A's, set to A
population[ num_As == 3 ] = "A"
# For 3-demonstrator combinations with all B's, set to B
population[ num_As == 0 ] = "B"
```

```
prob_majority = rng.choice([True, False], p=[(2/3 + D/3), 1-(2/3 +
 ↪  D/3)], size=N, replace=True)
prob_minority = rng.choice([True, False], p=[(1/3 + D/3), 1-(1/3 +
 ↪  D/3)], size=N, replace=True)
```

```
# 3-demonstrator combinations with two As and one B
condition = prob_majority & (num_As == 2)
if condition.sum() > 0:
    population.loc[condition, "trait"] = "A"
condition = ~prob_majority & (num_As == 2)
if condition.sum() > 0:
    population.loc[condition, "trait"] = "B"

# 3-demonstrator combinations with two B's and one A
condition = ~prob_minority & (num_As == 1)
if condition.sum() > 0:
    population.loc[condition, "trait"] = "A"
condition = prob_minority & (num_As == 1)
if condition.sum() > 0:
    population.loc[condition, "trait"] = "B"
```

```
demonstrators["new_trait"] = population["trait"]
demonstrators.head()
```

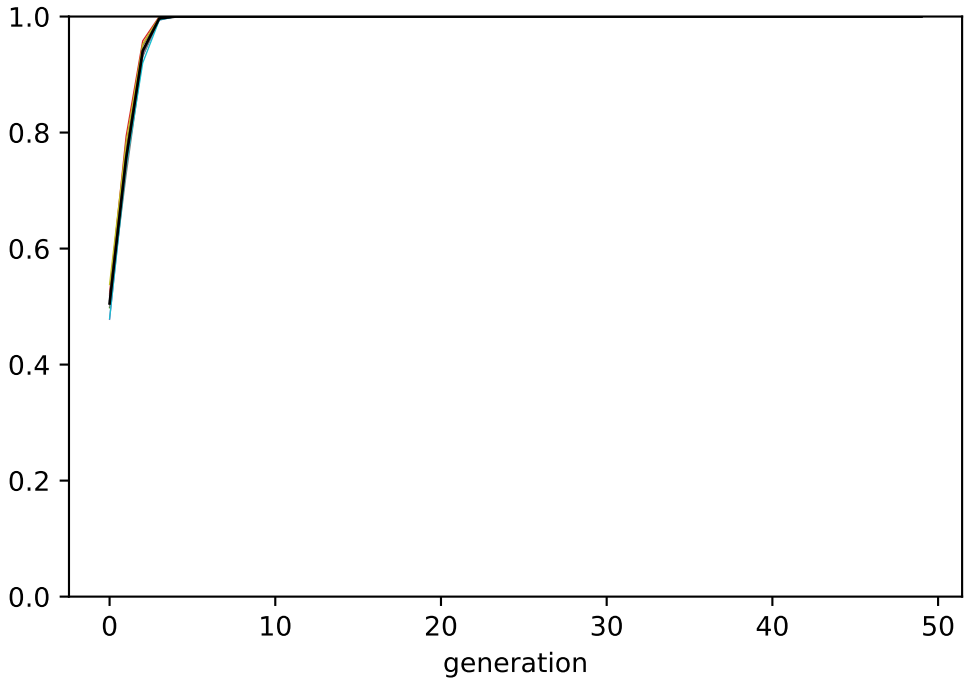
```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34

In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	dem1	dem2	dem3	new_trait
0	A	A	B	A
1	B	B	B	B
2	B	B	A	A
3	A	B	A	A
4	A	B	B	A

```
def conformist_transmission(N, p_0, D, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
        # Create first generation
        population = pd.DataFrame({"trait": rng.choice(["A", "B"],
↪   size=N, replace=True, p=[p_0, 1 - p_0])})

        # Add first generation's p for run r
       output.loc[ r * t_max, "p"] = population[ population["trait"] ==
↪   "A" ].shape[0] / N

        # For each generation
        for t in range(1,t_max):
            demonstrators = pd.DataFrame({
                "dem1" : population["trait"].sample(N,
                 ↪  replace=True).values,
                "dem2" : population["trait"].sample(N,
                 ↪  replace=True).values,
                "dem3" : population["trait"].sample(N,
                 ↪  replace=True).values
            })
```

```
            # Get the number of A's in each 3-demonstrator combination
            num_As = (demonstrators == "A").apply(sum, axis=1)

            # For 3-demonstrator combinations with all A's, set to A
            population[ num_As == 3 ] = "A"
            # For 3-demonstrator combinations with all A's, set to A
            population[ num_As == 3 ] = "A"
            # For 3-demonstrator combinations with all B's, set to B
            population[ num_As == 0 ] = "B"

            prob_majority = rng.choice([True, False], p=[(2/3 + D/3),
↪  1-(2/3 + D/3)], size=N, replace=True)
            prob_minority = rng.choice([True, False], p=[(1/3 + D/3),
↪  1-(1/3 + D/3)], size=N, replace=True)

            # 3-demonstrator combinations with two As and one B
            condition = prob_majority & (num_As == 2)
            if condition.sum() > 0:
                population.loc[condition, "trait"] = "A"
            condition = ~prob_majority & (num_As == 2)
            if condition.sum() > 0:
                population.loc[condition, "trait"] = "B"

            # 3-demonstrator combinations with two B's and one A
            condition = prob_minority & (num_As == 1)
            if condition.sum() > 0:
                population.loc[condition, "trait"] = "A"
            condition = ~prob_minority & (num_As == 1)
            if condition.sum() > 0:
                population.loc[condition, "trait"] = "B"

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[r * t_max + t, "p"] = population[
↪  population["trait"] == "A" ].shape[0] / N

    return output
```

```
data_model = conformist_transmission(N=1_000, p_0 = 0.5, D = 1, t_max =
 ↪  50, r_max = 10)
plot_multiple_runs(data_model)
```



7. Biased transmission: demonstrator-based indirect bias

> **Note**
>
> This chapter is based on “Chapter 5: Biased transmission: demonstrator-based indirect bias” in Acerbi et al. (2022).

```
import numpy as np
rng = np.random.default_rng()

import pandas as pd
```

```
def plot_multiple_runs(data_model):
    groups = data_model.groupby("run")
    for _, g in groups:
        g.index = g["generation"]
        g["p"].plot(lw=.5, ylim=(0,1))

    data_model.groupby("generation")["p"].mean().plot(c="k", lw="1")
```

```
N = 100
p_0 = 0.5
p_s = 0.05
```

```
population = pd.DataFrame({
    "trait": rng.choice(["A", "B"], size=N, replace=True, p=[p_0,
     ↪  1-p_0]),
    "status": rng.choice(["high", "low"], size=N, replace=True, p=[p_s,
     ↪  1-p_s])
})
```

```
population.head()
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:34
```

```
In future versions `DataFrame.to_latex` is expected to utilise the base implementat
```

	trait	status
0	A	low
1	A	low
2	A	low
3	A	low
4	A	low

```
p_low = 0.01
p_demonstrator = np.ones(N)
p_demonstrator[ population["status"] == "low" ] = p_low
```

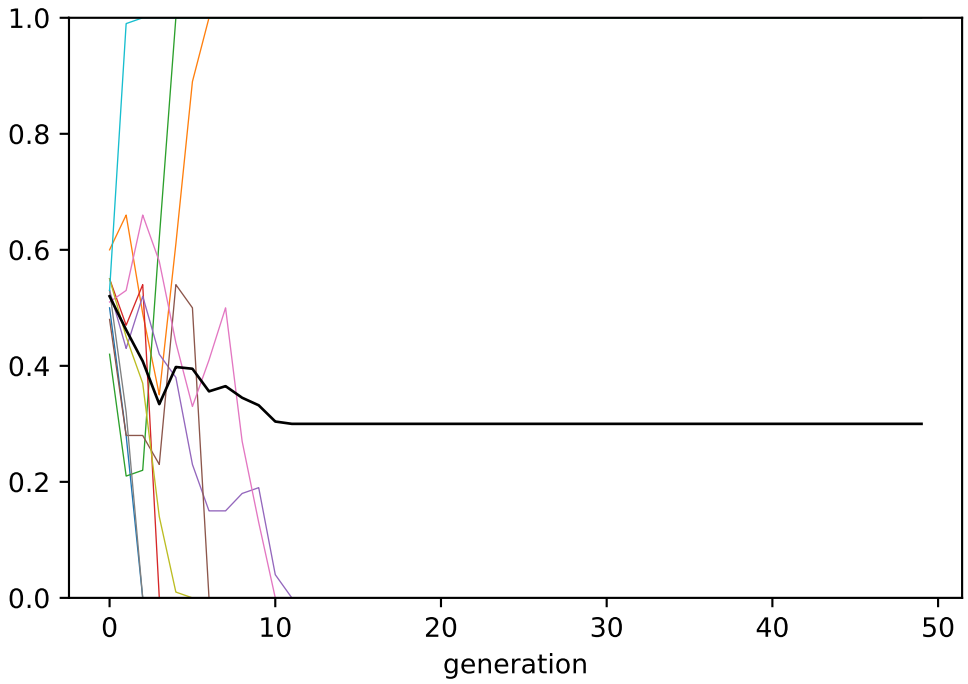
```
if sum(p_demonstrator) > 0:
    ps = p_demonstrator / p_demonstrator.sum()
    demonstrator_index = rng.choice(np.arange(N), size=N, p=ps,
    ↪  replace=True)
    population["trait"] = population.loc[demonstrator_index,
    ↪  "trait"].values
```

```
def biased_transmission_demonstrator(N, p_0, p_s, p_low, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    for r in range(r_max):
            # Create first generation
            population = pd.DataFrame({
```

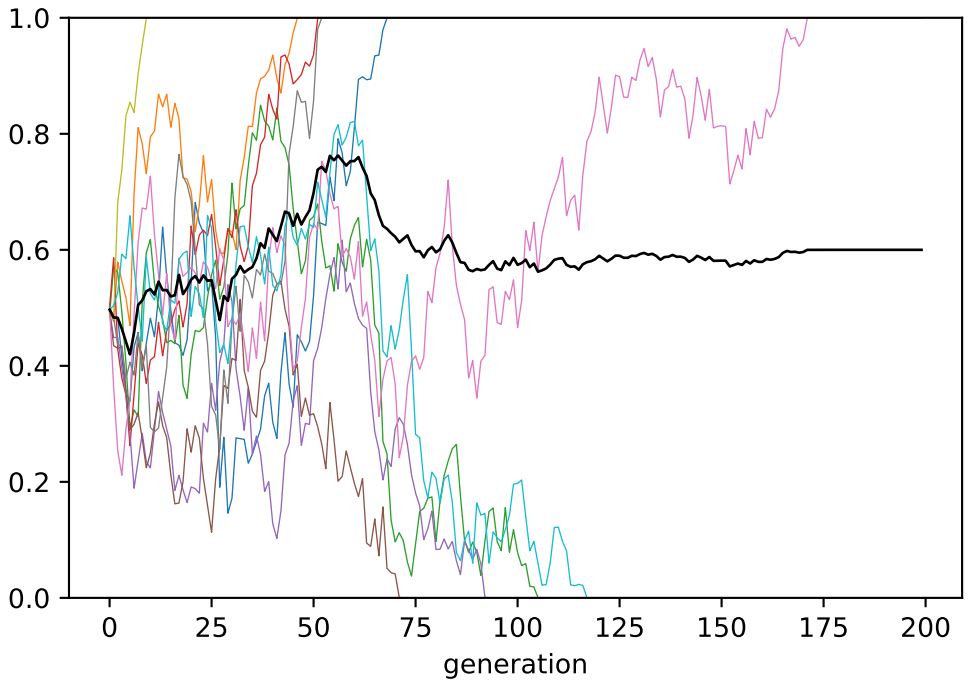
```
                "trait": rng.choice(["A", "B"], size=N, replace=True,
                 ↪  p=[p_0, 1-p_0]),
                "status": rng.choice(["high", "low"], size=N,
                 ↪  replace=True, p=[p_s, 1-p_s])
            })

            # Assign copying probabilities based on individuals' status
            p_demonstrator = np.ones(N)
            p_demonstrator[population["status"] == "low"] = p_low

            # Add first generation's p for run r
            output.loc[ r * t_max, "p"] = population[
↪   population["trait"] == "A" ].shape[0] / N

            for t in range(1, t_max):
                # Copy individuals to previous_population DataFrame
                previous_population = population.copy()

                # Copy traits based on status
                if sum(p_demonstrator) > 0:
                    ps = p_demonstrator / p_demonstrator.sum()
                    demonstrator_index = rng.choice(np.arange(N),
↪   size=N, p=ps, replace=True)
                    population["trait"] =
↪   population.loc[demonstrator_index, "trait"].values

                # Get p and put it into output slot for this generation
                 ↪  t and run r
                output.loc[r * t_max + t, "p"] = population[
↪   population["trait"] == "A" ].shape[0] / N

    return output
```

```
data_model = biased_transmission_demonstrator(N=100, p_s=0.05,
↪   p_low=0.0001, p_0=0.5, t_max=50, r_max=10)
```

```
plot_multiple_runs(data_model)
```




```
data_model = biased_transmission_demonstrator(N=10_000, p_s=0.005,
 ↪  p_low=0.0001, p_0=0.5, t_max=200, r_max=10)
plot_multiple_runs(data_model)
```




```
def biased_transmission_demonstrator_2(N, p_0, p_s, p_low, t_max,
 ↪  r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    ...

    return output
```

```
data_model = biased_transmission_demonstrator_2(N=100, p_s=0.1,
 ↪  p_low=0.0001, p_0=0.5, t_max=50, r_max=50)
```

Part II.

Advanced topics

8. Advanced topics

- “Culture and the Evolutionary Process” (Boyd & Richerson, 1985)
- “The Memetics of Music” (Jan, 2016)
- “Cultural Evolution and Music” (Youngblood et al., forthcoming)

8.1. More specific work

- “Cultural Evolution of Music” (Savage, 2019)
- “The role of population size in folk tune complexity” (Street et al., 2022)
- “Statistical Evolutionary Laws in Music Styles” (Nakamura & Kaneko, 2019)
- “The line of fifths and the co-evolution of tonal pitch-classes” (Moss et al., 2022)

References

Acerbi, A., Mesoudi, A., & Smolla, M. (2022). *Individual-based models of cultural evolution: A step-by-step guide using R*. Routledge. https://acerbialberto.com/IBM-cultevo/

Boyd, R., & Richerson, P. J. (1985). *Culture and the Evolutionary Process*. The University of Chicago Press.

Honing, H. (2018). On the biological basis of musicality. *Annals of the New York Academy of Sciences*. https://doi.org/10.1111/nyas.13638

Jan, S. (2016). *The Memetics of Music: A Neo-Darwinian View of Musical Structure and Culture*. Routledge.

Morley, I. (2013). *The Prehistory of Music. Human Evolution, Archaeology, and the Origins of Musicality*. Oxford University Press.

Moss, F. C., Neuwirth, M., & Rohrmeier, M. (2022). The line of fifths and the co-evolution of tonal pitch-classes. *Journal of Mathematics and Music*, 1–25. https://doi.org/10.1080/17459737.2022.2044927

Nakamura, E., & Kaneko, K. (2019). Statistical evolutionary laws in music styles. *Nature Scientific Reports*, *9*(1), 15993. https://doi.org/10.1038/s41598-019-52380-6

Savage, P. E. (2019). Cultural evolution of music. *Palgrave Communications*, *5*(1), 1–16. https://doi.org/10.1057/s41599-019-0221-1

Street, S., Eerola, T., & Kendal, J. R. (2022). The role of population size in folk tune complexity. *Humanities and Social Sciences Communications*, *9*(1), 1–12. https://doi.org/10.1057/s41599-022-01139-y

Tomlinson, G. (2018). *A Million Years of Music*. Princeton University Press. https://press.princeton.edu/books/paperback/9781890951528/a-million-years-of-music

Wallin, N. L., Merker, B., & Brown, S. (Eds.). (2001). *The Origins of Music*. MIT Press.

Youngblood, M., Ozaki, Y., & Savage, P. E. (forthcoming). Cultural evolution and music. In J. Tehrani, J. R. Kendal, & R. L. Kendal (Eds.), *Oxford Handbook of Cultural Evolution*. Oxford University Press. https://psyarxiv.com/xsb7v