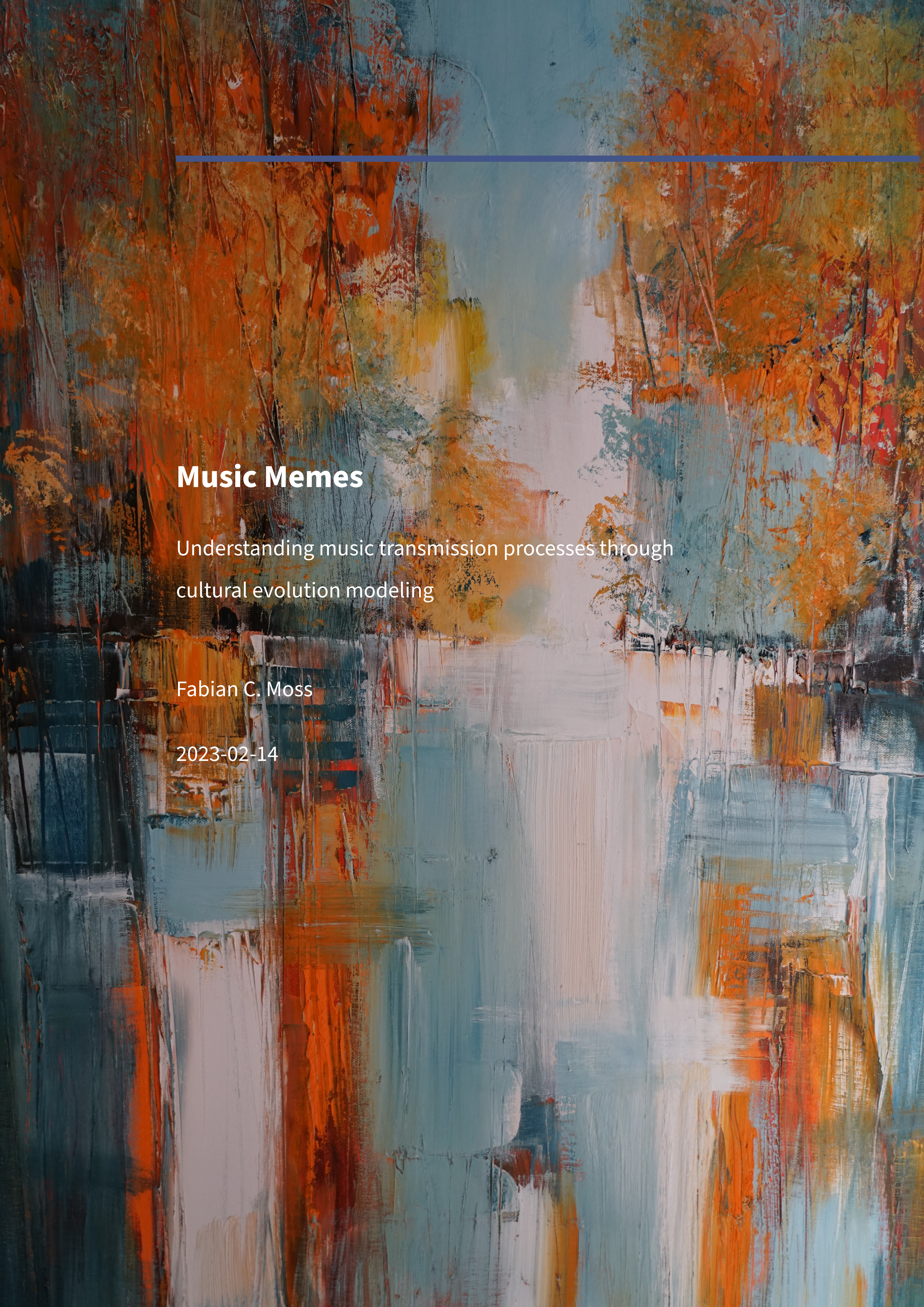


Music Memes

Understanding music transmission processes through cultural evolution modeling

Fabian C. Moss

2023-02-14

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.

> If you want to refer to this resource, please cite it as appropriately, e.g.:
> Moss, F. C. (2023). *Music memes: Understanding music transmission processes through cultural evolution modeling*. https://fabianmoss.github.io/musicmemes

> **!** Important
>
> The material on this page has been adapted and designed for my musicology research seminar "Music Memes: quantitative approaches and theories of cultural transmission of music" at Würzburg University, Germany.
> Note that the materials here are still under development. Please inform me if you notice any errors or other issues.

0.0.0.0.1. * Acknowledgements

I am grateful for encouragement from Alberto Acerbi to continue working on my Python translation of his book and many helpful comments.

Part I.

Prelude

1. Introduction

1.1. Musical Memes

The term ‘meme’ has been adopted by everyday language and is usually associated with a certain image, superimposed with text, that is shared on the internet and has a particular contextual meaning.

Figure 1.1.: A musical meme. This image itself has been ended up here after being passed along a chain of cultural transmission processes involving several social media: a Google search for "music meme" led to an image result pointing to Classic FM, which, in turn, had taken this image from the instagram page of user @musicmemes.for.supertonicteens.

Originally, however, the definition of 'meme' was much broader. Towards the end of *The Selfish Gene*, Richard Dawkins introduced the concept as follows:

> "We need […] a noun that conveys the idea of a unit of cultural transmission, or a unit of *imitation*. 'Mimeme' comes from a suitable Greek root,[a] but I want a monosyllable word that sounds a bit like 'gene'. I hope my classicist friends will forgive me if I abbreviate mimeme to *meme*." Dawkins

> (1976, p. 192)
>
> [a] https://en.wikipedia.org/wiki/Mimesis

Here, we adopt this original, more extensive concept of memes that, as we will discover, comprises the more or less funny internet pictures as a special case.

The field of cultural evolution emerged in the 1980's (e.g., Boyd & Richerson, 1985; Cavalli-Sforza & Feldman, 1981), and has, in parallel with the advancement of computational facilities, gained momentum. Theories on cultural evolution share many facets with approaches on memetics (Aunger, 2001; Blackmore, 2000; Dawkins, 1976; Howe & Windram, 2011), a field that has also been applied to the case of music (Jan, 2016).

In recent years, several approaches have attempted to apply formal models from cultural evolution to the domain of music.

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

A few selected important contributions are:

- "Cultural Transmission and Evolution: A Quantitative Approach" (Cavalli-Sforza & Feldman, 1981)
- "Culture and the Evolutionary Process" (Boyd & Richerson, 1985)
- "The Memetics of Music" (Jan, 2016)
- "Cultural Evolution and Music" (Youngblood et al., forthcoming)
- "Cultural Evolution of Music" (Savage, 2019)

1.2. Conditions for an evolutionary system

- variation
- selection
- retention

See Blackmore (2000), Chapter 2 "Universal Darwinism" for an excellent introduction.

These factors are, coincidentally, also central in the definition of folk music put forward by the *International Folk Music Council*:

> "Folk music is the product of a musical tradition that has been evolved through the process of oral transmission. The factors that shape the tradition are: (i) continuity which links the present with the past; (ii) variation which springs from the creative impulse of the individual or the group; and (iii) selection by the community, which determines the form or forms in which the music

> survives." (International Folk Music Council, 1955, p.23; see also Karpeles, 1955)

It is interesting that many contempory definitions of music, for example the aphoristic "humanly structured sound" one, do not refer explicitly to modes or conditions of transmission.

1.3. Music in human evolution

This book is about the cultural evolution of music. It has to be mentioned, however, that there is a large body of research on the biological evolution of music. This research asks questions about with which evolutionary advantages music endowed early humans, whether it is something that we share with other animals or whether it makes us unique. Some hold the view that music has no particularly relevant evolutionary function at all (Pinker, 1997), while others see in it a key to our success as a species (Cross, 2016).

Whatever the true role of music in the evolutionary history of humanity may have been, it is most certainly a fascinating topic to reflect upon. After all, human musical activity with dedicated instruments can be traced back at least 20,000 years, although it seems more than likely that human 'musicking' dates back much further, since our own bodies and voices already provided us with excellent musical instruments long before the first instruments have been devised.

If you are interested in learning more about biological-evolutionary aspects of music, I highly recommend to read, e.g. Wallin et al. (2001), Morley (2013), Tomlinson (2018), and Honing (2018).

1.4. History and cultural evolution

Documenting, describing, and interpreting changes in human culture is what historians do. Accordingly, changes in music belong to the field of music history, or historical musicology. However, most historians would probably be hesitant to employ models, or worse: formal models, in order to describe historical processes. The dictum "history doesn't repeat itself" seems to raise a fundamental argument against such endeavors that aim at explaining cultural or historical changes by means of underlying latent 'forces'. Modeling, in that view, seems to erroneously assume that history is teleological – directed towards a predetermined goal.

On the other hand, it is undeniable that there are many aspects of culture exhibit regularities and 'progresses' that are hard to explain if there are no guiding processes. Defining "culture" is, of course, yet another difficult enterprise. Here, I follow more or less the definition of Boyd and Richerson (1985, p. 33): "Culture is information capable of affecting individuals' phenotypes which they acquire from other conspecifics *by teaching or imitation* [emphasis mine]".[1] The part important to us here is "by

[1] I encourage you to read Chapter 3 of their seminal book in order to fully capture the meaning of this quote.

teaching or imitation", which is meant to imply: *not* by genetic inheritance. If humans can transmit information by other means than genetic inheritance, and if these transmission processes continue over many generations, then they are worth studying. Rest assured, the assumption of a hidden goal towards which all cultural processes are directed is not needed at all! I hope that you will share this conviction after working through this material.

1.5. Schema theory

What are the units (the basic memes) in music? Schema theory proposes a set of more or less fixed patterns, 'schemata', that underly a large body of music in the Baroque and Classical period (Gjerdingen, 2007).

Are schemata really memes? I would think not because they can not mutate freely without losing their meaning. I rather think schemata 'fall out' of several combinatorial possibilities to traverse the diatonic scale, and compositional mechanisms to elaborate and vary them. But this kind of variation is very different than the one we talk about in this book. Composed variation is very regular, not random, and relying both on the melodic and metric structures in which it unfolds. Those, however, could very well be understood as environments with very strong constraints (being 'out-of-scale' or 'off-beat').

1.6. Basic cultural inheritance mechanisms

Following this introduction, we introduce some minimal requirements to use Python for this course (Chapter 3).

Subsequently, we introduce with six central mechanisms for cultural inheritance: unbiased transmission (Chapter 6), unbiased and biased mutation (Chapter 7), directly biased transmission (Chapter 9), frequency-dependent indirectly biased transmission (Chapter 10), and demonstrator-based indirectly biased transmission (Chapter 11). We follow up with a chapter on vertical and horizontal transmission (Chapter 12), and finally introduce the multiple traits model (Chapter 13). The following diagram gives an overview of these processes:

After having a firm grasp on how these processes can be modeled in Python and how modeling results can be interpreted, we move to more advanced topics, and more specifically into a number of recent exciting results about cultural evolution and music. We conclude our journey (Chapter 18) with a more general discussion of the implications of cultural evolution for how we think about music, on the relevance of modeling in music research and the humanities more generally, as well as on the great potential of this approach for future research.

1.7. Progress?

Finally, at the end of this introduction, I want to circle back to an important issue that has been central to many concerns surrounding the application of evolutionary models to music, and to culture more generally. I am speaking about notions of progress and teleology in the sense that (cultural) evolution would be, in some sense, continuously improving. And not only that, it would be guided towards a certain goal, an optimal state of superiority. These concerns shouldn't be disregarded too lightly, especially in view of the misunderstandings and harmful consequences they have caused. On the other hand, we also need a differentiated perspective on evolutionary theory, its assumptions, and the extent to which one might apply it to cultural phenomena. I believe that the generally critical attitude in the humanities can be, in fact, a great catalyst to weed out inappropriate notions and interpretations. However, what is needed in addition is also a certain openness to engage in discussions with scientific approaches to culture and not an unreflected (and thus uncritical) dismissal *per se*. The general attitude towards evolutionary thinking in biology and culture is nicely summarized in a quote by Kwame Anthony Appiah:

> Nowadays, it is clear that one of the most distinctive marks of our species is that our inheritance is both biological *and* cultural. Each generation of human beings in a particular society can build on what was learned by the ones before […]. What makes us the wise species—*sapiens*, remember is the Latin for "wise"—is that our genes make brans that allow us to pick up things from one another that are *not* in our genes. (Appiah, 2018, p. 122)

This book wants to contribute to a fruitful discussion, provide entry points for humanists to better understand computational evolutionary models, and to show in which areas they have been and could be applied. After all, all of academia, the humanities and the sciences alike, are also subject of constant changes and adaptations to new environmental conditions. Music research is no exception.

2. Style

- What is meant by ‘style’?
- The role of learning: learning describes the acquisition of knowledge/information. Learning mechanisms have different names depending on the temporal scale: evolution, ontogenesis, development, learning

2.1. A model of style in music

A comprehensive model for theoretical treatment of style was proposed by Meyer (1989). In a nutshell, he proposes a model where certain constraints favor differnt kinds of patternings. He organises these constraints hierarchically so that this model is suitable to describe a variety of stylistic phenomena on different levels. This clear separation is more conceptual and practical only for theoretical discussions. In a real scenario, boundaries are often unclear or ambigous.

2.1.1. Constraints

1. Laws
2. Rules
3. Strategies

2.1.2. Patterns

1. Dialect
2. Idiom
3. Intraopus Style (here, we need a concept of “work”, but the extended “meme” concept or “memecomplex might also be useful”)

3. 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.

3.1. 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 items of the same type in lists—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'
```

3.2. 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:

```python
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, each one being printed on a new line. But what we wanted was all numbers 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 instead 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)`. Additionally, we can use the end keyword of the `print` function that allows us to print all numbers in one line, separated only by a single white space instead of each one on a new line.

```python
for i in range(1,11):
    print(i, end=" ")
```

```
1 2 3 4 5 6 7 8 9 10
```

Voilà!

3.3. Just in case

Often we encounter a situation where we would execute some code **only if** certain conditions are met. In python, this is done with the `if` statement. For example, if we want to only print the even numbers in the range from 1 to 10, we could adapt the code from above as follows:

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
```

```
2 is even
4 is even
6 is even
8 is even
10 is even
```

You can read this as "if the remainder of dividing i by 2 is zero, then print 'i is even' ".

Now, we could also want to print a similar statement in the case that `i` is odd:

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

And, finally, we could also have more than just one condition. An if-statement allows for arbitrary many if-else clauses, with which we can formulate several different conditions by writing `elif` (shorthand for 'or else if'):

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
    elif i % 3 == 0:
        print(i, "is divisible by 3")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is divisible by 3
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is divisible by 3
10 is even
```

We now know when a number is even and when it is divisible by 3. But what about numbers that are *both* even *and* divisible by 3? We just add another condition to the `elif` statement and enclose each condition in parentheses, so that Python knows how things group together:

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
    elif (i % 2 == 0) and (i % 3 == 0):
        print(i, "is even and divisible by 3")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is odd
4 is even
```

```
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

Why did this not work? The number 6 is even *and* divisible by 3! The reason is that the three statements (`if`, `elif`, and `else`) are being executed in the order that we wrote them down. That means, that Python will first check for each number whether it is even (and nothing more), and if it is, it will follow the instruction to print it and go on to the next number. So, once we arrived at 6, the programm will only check if the number is even. That is not the desired result and we have to make a little change to it. We will switch the conditions in the `if` and `elif` statements:

```
for i in range(1,11):
    if (i % 2 == 0) and (i % 3 == 0):
        print(i, "is even and divisible by 3")
    elif i % 2 == 0:
        print(i, "is even")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even and divisible by 3
7 is odd
8 is even
9 is odd
10 is even
```

Now it works! Note that new never specified any conditions for the `else` statement. This is because whatever follows it will be executed in case none of the conditions in `if` or `elif` are met.

3.4. 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 squares 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 code this 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, although I would recommend to always use function names that make clear what the function does:

```
f = func_sum_of_squares

f(5,4)
```

```
41
```

3.5. 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.

> Caution
>
> 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.

3.5.1. NumPy

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`.`.

3.5.2. Pandas

TODO

3.5.3. Matplotlib

TODO

3.5.4. Summary

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.

Concepts covered

- variables
- types (integers, strings, lists, dictionaries)
- functions
- libraries, importing and aliases

4. Randomness and order

This course is about music and evolution, specifically about using computational models to better understand the mechanisms underlying certain evolutionary processes. In order to do so, we will often run simulations to observe and interpret the resulting scenarios against the backdrop of our modeling assumptions and our domain knowledge.

Simulations are useful because for many situations we are not able to provide deterministic mathematical formulae to obtain outcomes from inputs. Rather, we incorporate our knowledge about the world in a computational model and use randomness to include the inherent uncertainty about the exact outcomes of the simulations.

> Add section on probability.

But randomness is a difficult concept to capture. For the purpose of this book, we will simulate a speficic kind of randomness by sampling from a certain set of items. Less technically, we can imagine a bag with a certain number of balls in it, each having a certain color (multiple balls can have the same color). A *random uniform sample with replacement* then corresponds to picking a ball from the bag without looking in it and putting it back in the bag.

4.1. Random draws from a bag

Let's try this in Python. We will use the `random` modul of the *NumPy* library:

```
import numpy as np
rng = np.random.default_rng() # initialize a default random generator

bag = range(4)
ball = rng.choice(bag)

print(ball)
```

2

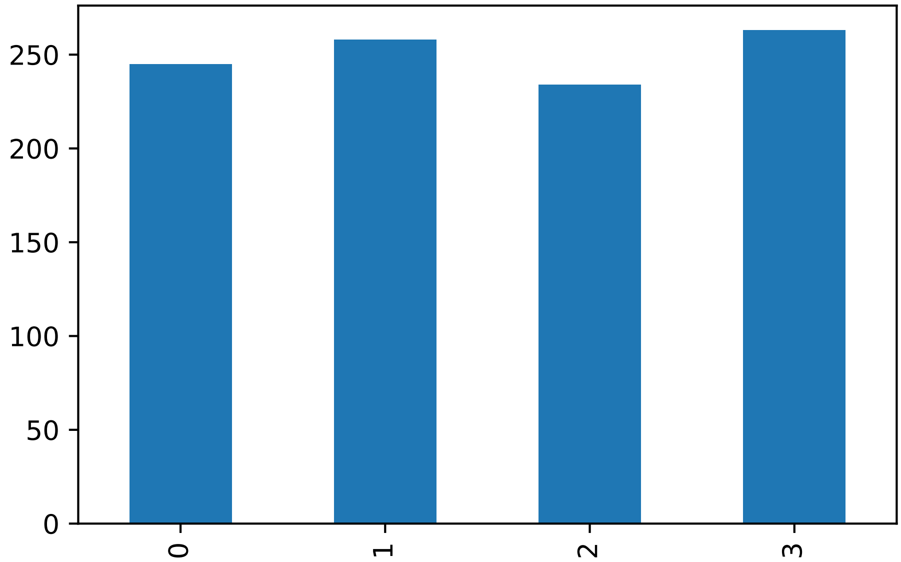
If this draw were really random, we would expect that each number is equally likely. We can test this by repeating this procedure again and again, tallying the result each time.

```
import pandas as pd
import matplotlib.pyplot as plt

samples = []

for i in range(1000):
    ball = rng.choice(bag)
    samples.append(ball)

s = pd.Series(samples)

print(s.head(10))
print(s.value_counts().sort_index())
```

```
0    3
1    0
2    0
3    2
4    0
5    3
6    3
7    1
8    2
9    2
dtype: int64
0    245
1    258
2    234
3    263
Name: count, dtype: int64
```

The number are not exactly the same but pretty close. If we would continue sampling from our bag, they would get more and more similar to one another. It is easier to understand this by visualizing it.

```
s.value_counts().sort_index().plot(kind="bar")
plt.show()
```

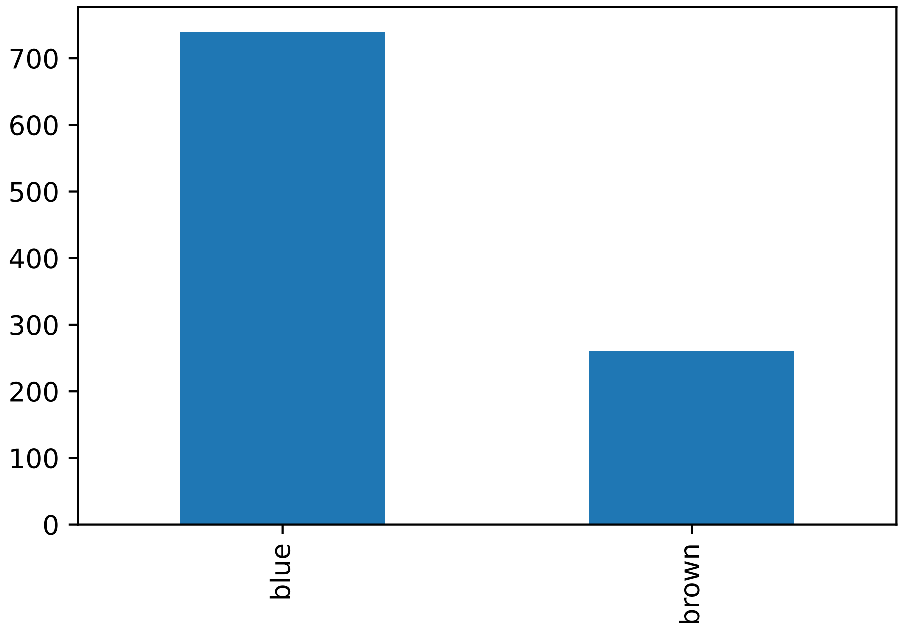


Now, what if the numbers in the bag were not just numbered, but had different colors? Let's assume we have another bag, `bag2`, with 4 balls, three brown, one blue:

```
bag2 = [ "blue", "blue", "blue", "brown" ]

samples2 = []

for i in range(1000):
    ball = np.random.choice(bag2)
    samples2.append(ball)

s2 = pd.Series(samples2)

print(s2.head(10))
print(s2.value_counts().sort_index())

s2.value_counts().sort_index().plot(kind="bar")
plt.show()
```

```
0      blue
1      blue
```

```
2      blue
3      blue
4      blue
5      blue
6      blue
7     brown
8      blue
9     brown
dtype: object
blue     740
brown    260
Name: count, dtype: int64
```



This is remarkable: by randomly (uniformly) drawing from the second bag, the frequencies of all samples approach the ratio of brown to blue balls (3:1)!

4.2. Composing random melodies

Since this book is about music, let's see how we can use randomness to create (a resemblance of) music. For instance, we can 'compose' a random melody by using only the white keys on a piano within some octave:

```
notes = list("CDEFGAB")
melody = rng.choice(notes, size=10)
print(melody, end=" ")
```

```
['F' 'B' 'D' 'D' 'G' 'E' 'D' 'B' 'B' 'C']
```

We composed a little melody by randomly drawing a note from the list of notes. This is also called *sampling*. Note that some notes repeat, showing that we sample with replacement: after each draw, the note is put back in the bag, so to speak. Of course, there are many things that we would have to generate, too, to make this a real melody. For instance, we do not know the duration of any of these notes, we don't know the meter nor the key, we don't know the tempo or volume, and so on. But our goal here is not to create a beautiful piece of music, but rather to show how we can use randomness to generate something.

As you might remember from the previous chapter, we can also write a function to do this, so that we can perform this operation (composition of a random melody) more easily, while at the same time having more control over it through its parameters. The following function does exactly this, having only one parameter that controlls the length of the melody (the number of notes to be sampled).

```
def melody(n):
    notes = list("CDEFGAB")
    return rng.choice(notes, size=n)
```

We can now use this function to easily create random melodies of different lengths:

```
print(melody(7))
```

```
['D' 'A' 'E' 'D' 'F' 'A' 'D']
```

```
print(melody(12))
```

```
['D' 'G' 'B' 'G' 'C' 'F' 'G' 'C' 'F' 'G' 'C' 'A']
```

4.3. Synthesizing a corpus

The functionalities introduced above allow us to synthesize an artificial corpus of melodies, here simplified as lists of pitch classes and containing varying numbers of notes.

```
N = 4 # number of pieces in the corpus
corpus = [ melody(12) for _ in range(N)]
```

The first three melodies of our corpus are:

```
for mel in corpus:
    print(mel)
```

```
['G' 'G' 'G' 'D' 'G' 'E' 'C' 'B' 'D' 'C' 'A' 'A']
['F' 'A' 'D' 'G' 'F' 'D' 'B' 'D' 'B' 'G' 'C' 'F']
['G' 'E' 'B' 'A' 'G' 'E' 'C' 'A' 'E' 'E' 'F' 'A']
['C' 'F' 'C' 'A' 'E' 'D' 'G' 'F' 'A' 'A' 'E' 'G']
```

Of course, melodies are not always of the same length. We could vary the lenght of the melodies by creating a hand-crafted list specifying the number of notes for each melody in the corpus.

```
corpus = [ melody(n) for n in [10, 5, 7, 13] ]

for mel in corpus:
    print(mel)
```

```
['A' 'E' 'G' 'C' 'D' 'D' 'D' 'G' 'G' 'D']
['F' 'B' 'C' 'G' 'E']
['E' 'B' 'C' 'A' 'C' 'A' 'F']
['C' 'G' 'E' 'F' 'D' 'C' 'F' 'E' 'B' 'A' 'D' 'F' 'B']
```

However, specifying the lenghts of the melodies for a large corpus would be a very time-consuming task. In order to model the variability in length of melodies in a musical corpus, we will *randomly sample* them from a specified probability distribution. A good candidate for such a distribution is the Poisson distribution, that we can access from our random number generator `rng`.

```
lam = 25 # average number of notes in melody
N = 1000 # number of pieces in the corpus

corpus = [ melody(rng.poisson(lam=lam)) for _ in range(N) ]

lengths = pd.Series([ len(m) for m in corpus ]).value_counts()
```

```
idx = range(0, max(lengths))
lengths = lengths.sort_index().reindex(idx).fillna(0)
plt.bar(idx, lengths)
plt.axvline(lam, c="red")
plt.show()
```

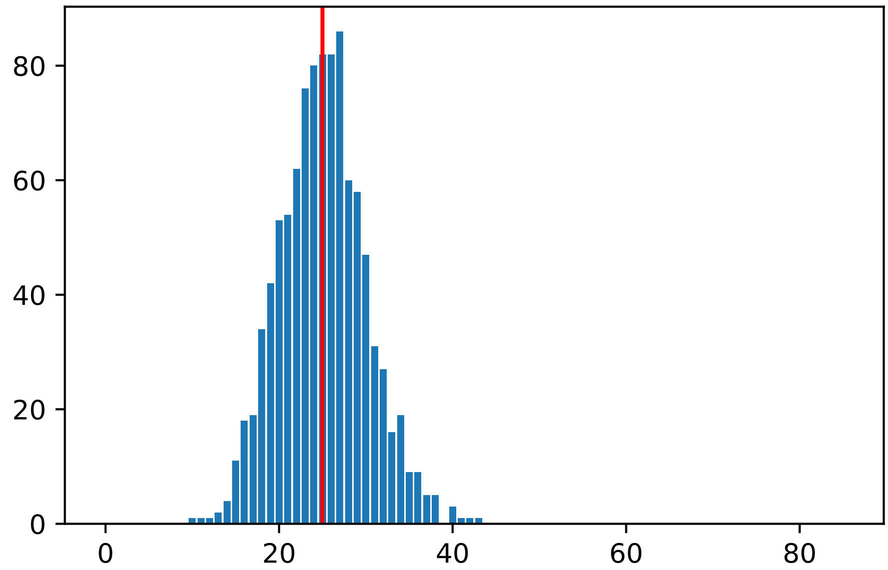


Figure 4.1.: Distribution of melody lengths in the corpus."

Now the variable `corpus` contains lists of pitch classes (*aka* melodies) of different lengths, most of them around the preset average value `lam`,[1] also indicated by the vertical red line. It is moreover evident that the corpus contains rather few very short or long melodies.

We can, of course, not only observe the distribution of melody lengths, but also look at the overall distribution of note occurrence in the corpus:

```
counts = []

for m in corpus:
    c = pd.Series(m)
    counts.append(c)

pd.concat(counts).value_counts().plot(kind="bar")
```

[1] Short for the Greek letter λ ("lambda").

```
plt.show()
```

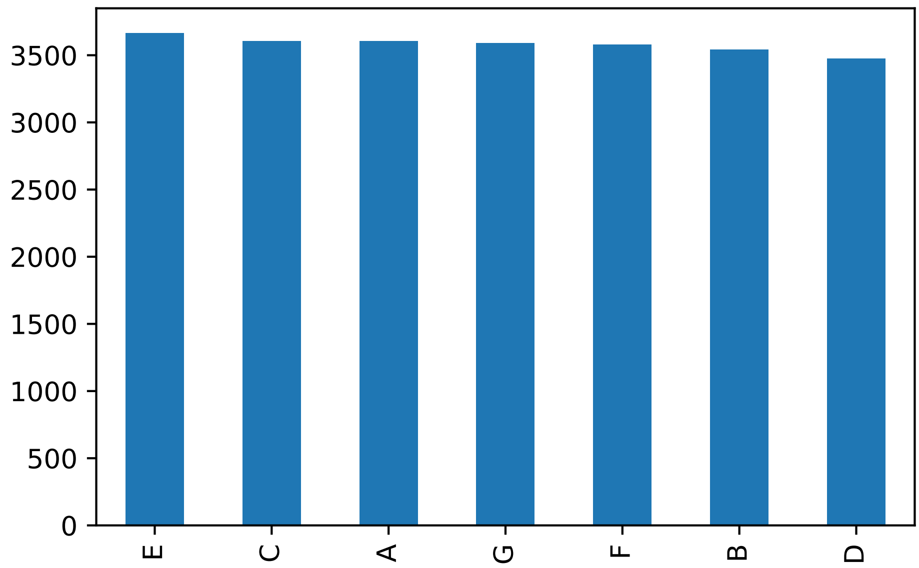


Figure 4.2.: Overall note frequencies in the artificial corpus.

At this point, we should stop and celebrate. We have just written our first *probabilistic model* to generate melodies. Admittedly, it is not a very good model for actual melodies, for example because notes are drawn *uniformly at random* from the set of diatonic pitch classes using the `.choice()` method, which leads to the somewhat unrealistic picture in Figure 4.2. One would expect that in real melodies some notes occur more often than others and that the occurrence of a note does, for instance, also depend on the notes that come before and after it. But, in principle, these other constraints could be added to our model to make it more realistic. The point here was mainly to illustrate how artificial corpora can be generated probabilistically. This will prove useful later on because it allows us to compare real-world corpora of music against synthetic ones generated by our models.

> **Exercise**
>
> Expand our melody model so that it also includes octave information for each pitch class in order to make it a bit more musical.

4.4. Pattern search

4.4.1. Incipits

Now that we have a corpus that we understand very well because we specified how it has been created, we can apply some simple analytical questions in order to warm up for later. For instance, we could want to have a function that allows us to search for *incipits*. Incipits are the beginnings of musical melodies that already to characterize themes and motives because incipits are often characteristic. For example, we would want to look for all melodies that begin with "C", "D", "E" and, for simplicity, we might want to pass a string like "CDE" to the function to facilitate the input.

```
import re

def find_incipit(incip="", mel=None):
    melody = "".join(mel)
    if re.search("^" + incip, melody):
        return True
    else:
        return False

for m in corpus[:10]:
    if find_incipit(incip="CDE", mel=m):
        print("".join(m))
```

4.4.2. Finals

We can apply a similar logic to find finals, the last notes of a melody. Instead of only allowing to search for a single note as a final, we will allow more generally to allow for a pattern that concludes a melody:

```
def find_finals(end="", mel=None):
    melody = "".join(mel)
    if re.search(end + "$", melody):
        return True

for m in corpus[:100]:
    if find_finals(end="GC", mel=m):
        print(m)
```

```
['D' 'E' 'A' 'G' 'C' 'F' 'A' 'A' 'F' 'D' 'G' 'E' 'B' 'F' 'D' 'G' 'C' 'A'
 'D' 'F' 'B' 'B' 'F' 'E' 'A' 'G' 'C' 'E' 'F' 'G' 'C']
['C' 'B' 'E' 'B' 'C' 'C' 'A' 'G' 'C' 'A' 'B' 'G' 'B' 'F' 'D' 'C' 'G' 'A'
 'C' 'D' 'A' 'A' 'C' 'G' 'C']
```

As you can see, all found melodies end with a falling perfect fifth form "G" to "C".

The last function, `find_finals()`, introduced the "^" (caret) character. In the context of regular expressions, this character signifies "at the end of a string", exactly what we needed to find finals.

4.4.3. Patterns more generally

> Todo
>
> Introduce regexes more flexibly and write a general pattern matcher.

4.5. Horrible Homophony

Four-part writing is a core part of Western composition history. Here, we will build a mock version of a four-part chorale by randomly generating each voice and putting them together in a table. Doing so will show you how you can create tables, which we will need later on. The most popular way to work with tables in Python is by using the `pandas` library. In `pandas`, tables are called 'data frames', and there is a `DataFrame` object to represent tables. Let's see how we could create a four-part homophonic chorale with eight 'chords':

```
import pandas as pd

n = 8

chorale = pd.DataFrame({
    "S" : melody(n),
    "A" : melody(n),
    "T" : melody(n),
    "B" : melody(n)
})
```

The variable `chorale` now stores our little composition and we can inspect it:

```
chorale
```

	S	A	T	B
0	D	E	A	B
1	B	G	F	B
2	D	D	F	G
3	E	F	F	F
4	B	A	A	C
5	F	D	A	B
6	F	E	E	E
7	C	C	E	A

Here we have generated each voice using the `melody` function. We can use it to create a new function that will directly give us a new chorale of a certain length:

```
def chorale(n):
    df = pd.DataFrame({
        "S" : melody(n=n),
        "A" : melody(n=n),
        "T" : melody(n=n),
        "B" : melody(n=n)
    })

    return df
```

```
my_chorale = chorale(n=12)
my_chorale
```

	S	A	T	B
0	F	B	D	B
1	C	F	B	F

	S	A	T	B
2	F	C	C	D
3	E	D	B	D
4	A	C	G	C
5	D	E	B	A
6	C	A	B	B
7	B	C	B	D
8	E	G	C	A
9	B	A	E	A
10	G	F	B	E
11	F	E	A	C

It will look a bit closer to musical notation if we transpose the data frame by using the `.transpose()` method:

```
my_chorale.transpose()
```

	0	1	2	3	4	5	6	7	8	9	10	11
S	F	C	F	E	A	D	C	B	E	B	G	F
A	B	F	C	D	C	E	A	C	G	A	F	E
T	D	B	C	B	G	B	B	B	C	E	B	A
B	B	F	D	D	C	A	B	D	A	A	E	C

4.6. Accessing data

Having the variable `my_chorale` store our data frame, this is how we can access individual voices:

```
my_chorale["T"]
```

```
0     D
1     B
2     C
3     B
4     G
5     B
6     B
7     B
8     C
9     E
10    B
11    A
Name: T, dtype: object
```

You can verify that it is the same 'melody' as above in the chorale. If we want a specific note from this voice, say the fifth one, we can access is this way:

```
my_chorale["T"][4]
```

```
'G'
```

We first select the "T" column, and then select the fifth element (remember that we start counting at 0, so we need to insert 4 to get the fifth). We can also get entire ranges of a voice:

```
my_chorale["A"][4:8]
```

```
4    C
5    E
6    A
7    C
Name: A, dtype: object
```

This gives us the fifths to ninth note in the Alto voice. If we want to apply the same logic also to column ranges, we have to write it a bit differently using the `.loc()` method for localising data:

```
my_chorale.loc[1:3, "S":"A"]
```

	S	A
1	C	F
2	F	C
3	E	D

`.loc()` takes two arguments: the rows (or row range), and the columns (or column range). We can use it to 'slice' our data frame in order to get specific portions of it.

5. Models

Throughout this book we will use on quantitative models to describe evolutionary processes. A model is a simplyfied description that captures what we know about a phenomenon of interest in terms of variables and their relations to one another. While models in this sense are always reductions of the real-world process being modeled, they have the benefit that all of their parts can be understood. Moreover, formal models usually incorporate one or several *parameters*, that is, variables the value of which is mutable. Changing parameter values thus renders different instances of a model that, in turn, allow one to ask the question “What if?” and to abserve the behavior of simulated processes in the long run. Answering this question under different conditions (different parameter settings) then will allow us to draw inferences from the modeling outcomes to interpret our phenomenon of interest.

It is precisely the ability of formal models to be adapted at will why they form the core part of this book. Starting with simple models with only very few variables, we will gradually make them more complex by including additional variables. However, it will remain our attempt to keep our models as simple as possible. While it is theoretically possible to include any number of variables in modeling, more complex models become harder to understand, and the interactions between the numerous variables more difficult to interpret.

As mentioned above, the goal of modeling is often to approximate rather than to recreate a process in the world. Sometimes, the goal is rather to show that a real process does *not* conform to the outcomes of certain modeling assumptions, from which can be deduced that other factors—possibly yet unknown ones—must be taken into account. One of the most common baseline comparison models is the so-called ’neutral model (see, e.g., Bentley et al., 2004), where cultural traits are randomly copied from previous generations. We will see in Chapter 6 what the consequences of this assumption are.

Questions concerning model comparison—chosing the ‘best’ model for a problem at hand—or dealing with the trade-off between model complexity and interpretability are central issues in many areas of empirical science, such as machine learning, computational sociology, or cognitive psychology, and many textbooks dedicate some discussion to the topic (e.g. Bishop, 2012; Farrell & Lewandowsky, 2018; McElreath, 2020). For excellent discussions and examples of modeling in music research, see Honing (2006) and Finkensiep et al. (forthcoming).

5.1. A simple example

Models are meant to be abstract representations of (some part of) reality. They necessarily need to be simpler than reality, simple enough so that we can understand them, but close enough to reality so that they are actually useful to us.

Why not having more complex models that accurately represent reality? Well, first of all that is a very difficult endeavour. But, more importantly:

> 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. (Acerbi et al., 2022, Introduction)

So what does that mean in practice? I will try to demonstrate this with an admittedly boring but hopefully illustrative example. Assume we want to model the entire area of this (fictional) country:

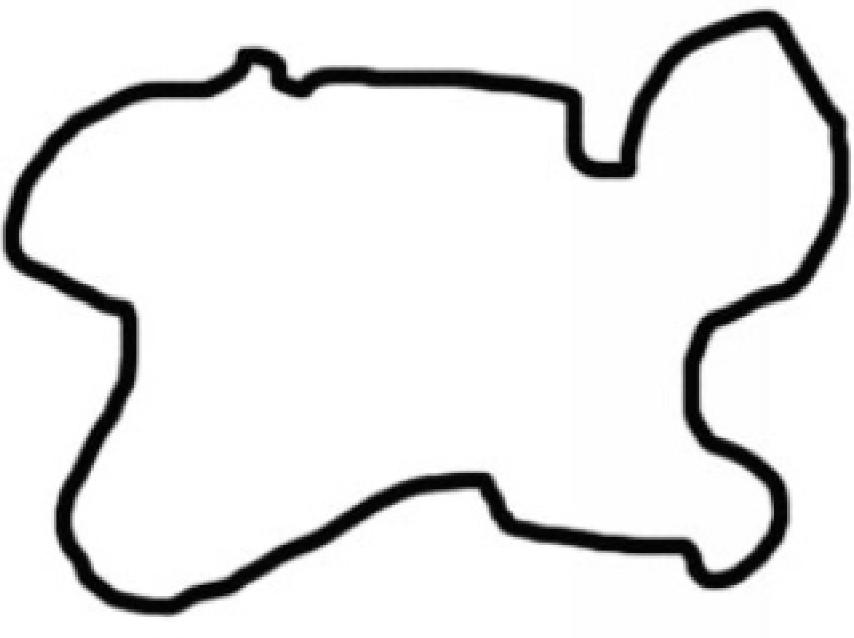


Figure 5.1.: The outline of a fictional country.

Maybe the most simple, albeit naive approach would be to say that the total area of this country can be modeled with a square. A square is a very parsimonious model: in order to describe its area, only one parameter is needed, its side length.

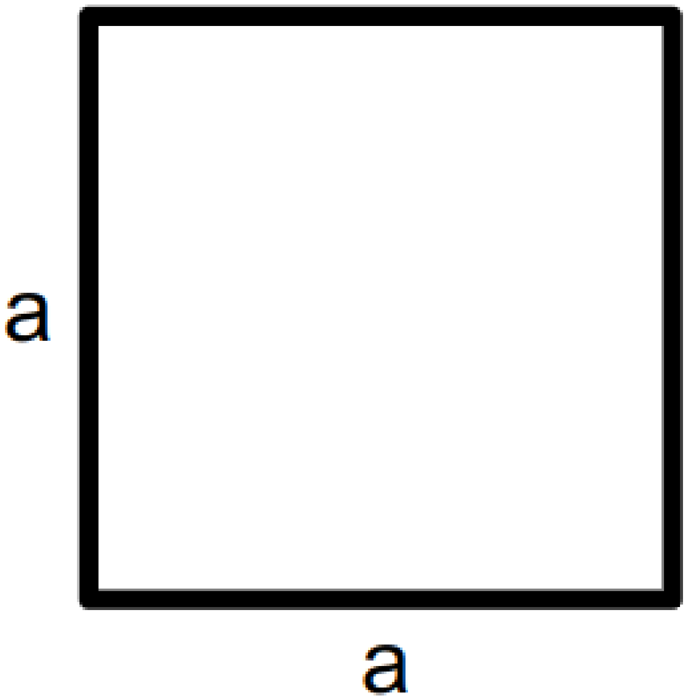


Figure 5.2.: A square.

What's more, we can even give a precise mathematical formula for the area of our fictional country under the square model:

$$M_1(a) = a^2$$

In Python code, we could express this model as the following function:

```
def square_model(a):
    return a**2
```

Of course, this is not a good model of the area of the country. But one of the strengths of formal models is that they are unquivocal. There might be situations, in which the rough estimate of the square model is actually sufficient for our purpose. So why use a more complex model if the simplest one does the job?

Most of the time, however, such a simplistic model will not suffice and we need to invest brain power to come up with a better one.[1] The following model is one way to improve upon our first mode:

[1]'Better' means here that it is closer to the reality we actually want to describe, while at the same time being as simple as

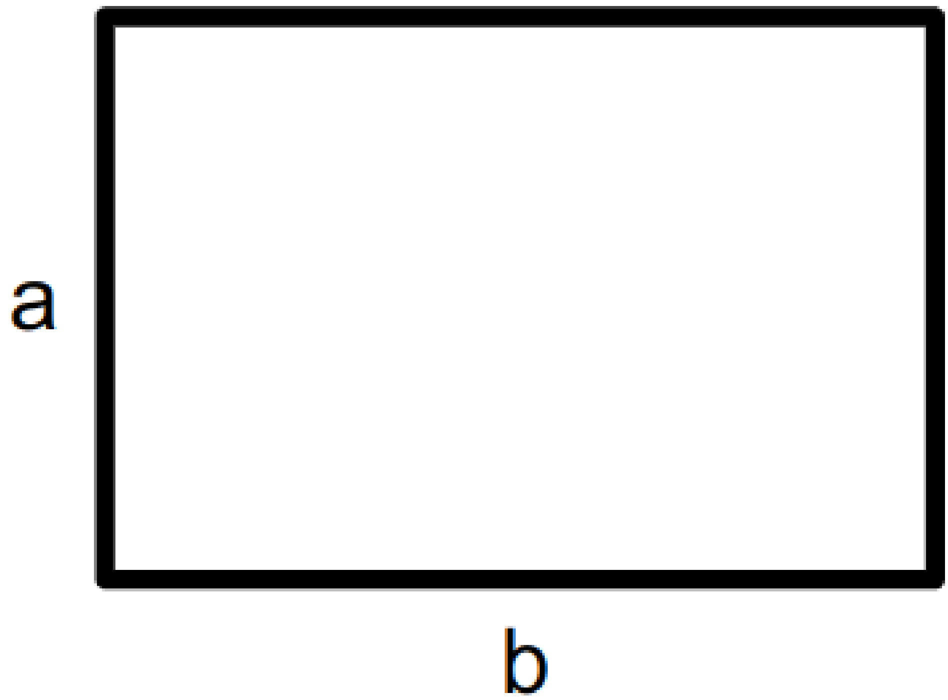


Figure 5.3.: A rectangle.

The shape of the rectangle seems to fit our country's outline better. Hooray! As for the square model, we do have a mathematical formuly to describe our rectangle model:

$$M_2(a, b) = a \cdot b$$

In Python code:

```
def rectangle(a,b):
    return a * b
```

But this improvement comes at a price: instead of having only one parameter, a, we now have two, a and b. Our model has instantly become twice as complex!

Modeling is at the core of social science and there are many good text about this topic (McElreath, 2020; Smaldino, 2017, 2023). In the humanities, there are fewer approaches taking modeling seriously, but they are growing in number (Finkensiep et al., forthcoming; Piotrowski, 2019).

possible. This trade-off is usually called "Occam's razor". Google it!

Part II.

Foundations

6. Unbiased transmission

What happens if people just blindly 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(seed=42)
```

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.

6.1. Creating a conceptual understanding

It is a useful exercise to imagine first what we want to implement. This way we can check whether we have a good conceptual understanding, which will help us to write the code more clearly and make fewer mistakes. The following figure shows in the upper panel a scenario, in which there is a population of `N` individuals present in each generation. Those individuals are uniquely characterized by one of two traits: whether they like opera (salmon color) or not (grey). This visualization captures all assumptions we make for our first model:

1. The number of individuals per generation does not change.
2. Generations are disjunct, there is no overlap.
3. In the first generation, opera preference is randomly assigned to individuals.
4. In each subsequent generation, each individual randomly picks an 'ancestor' from the previous generation and blindly copies that individual's preference for opera.

The models throughout this book assume that invidivuals in one generation learn from individuals of (only) the previous generation by "picking" an older individual and adopting its traits according to some probabilistic rules built into the model. This means an arrow from individual `n` in generation `t` to an individual `m` in generation `t + 1` should be interpreted as: "Individual `m` learns from individual `m`" and not the other way around (because information is flowing from generation `t` to `t+1`).

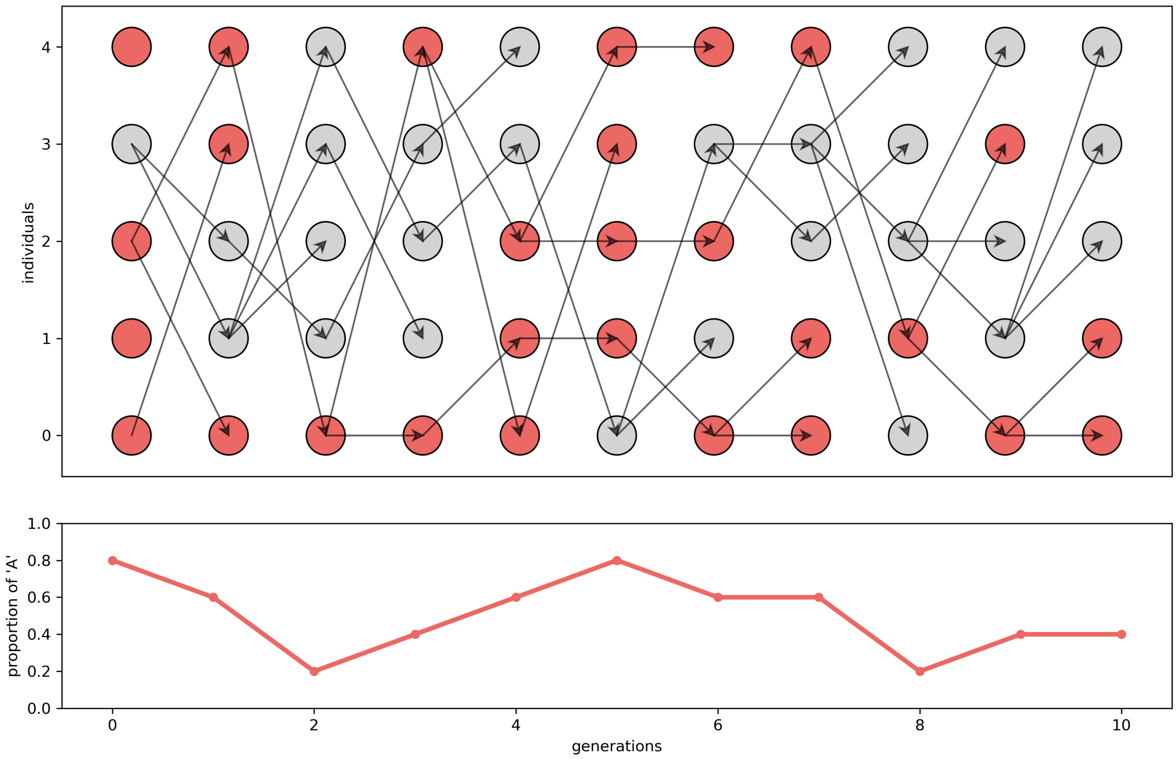


Figure 6.1.: A conceptual depiction of unbiased transmission.

The lower plot shows, at each time, point the percentage of individuals having a preference for opera. A logical consequence of this process is that each transmission chain (each sequence of directly connected arrows) will pass through one and only one trait (color).

We can make further observations from this initial example. In the first generation, there are two individuals who do like opera and three who don't. In subsequent generations, the proporation of these two traits changes. It is not *monotonic* (it goes both up and down), as the red line shows. But in the sixth generation, everything changes. Individuals in the sixth generation *could* inherit their trait from the one individual in generation 5 that likes opera. But because they randomly pick their ancestors, that individual is not among the ancestors. Consequently, no individual in generation 6 likes opera. This also means that, from now on, now one will ever like opera again. Opera fans have become extinct.

We can see that transmission of information is still going on: there are arrows between generations, so individuals still receive information from the previous generation. But nothing changes. That means that we *do have* transmission of information, but we would not anymore speak of it as *cultural evolution*, since the first fundamental criterion (see Chapter X), variation, is not fulfilled.

Note also, that some 'traditions' are likewise not continued. For example, following the transmission chain of individual 0 in the first generation to individual 2 in the second generation to individual 3 in the third generation, we see that this latter individual is not picked as an ancestor by anyone from the next generation. This individual (and all of its ancestors) have no more impact on future generations. Importantly, however, we are not interested in individual fates, but rather in population-level statistics. That is why the lower plot only traces the proportion of individuals having a preference for opera.

6.2. Simulating a population

With this conceptual understanding in mind, we will now look at how we can reproduce this model of unbiased transmission. Since we are assuming that new individuals randomly pick an ancestor, we should not assume that our results will be exactly the same as above, but they should nonetheless be qualitatively similar.

```
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 by looking at the first 10 individuals of the population:

```
population.head(10)
```

	trait
0	A
1	B
2	B
3	A
4	A
5	B
6	A
7	B
8	A
9	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?

For instance, we could say that the mutually exclusive traits "A" and "B" correspond to "Individual likes opera" and "Individual doesn't like opera". People are often opinionated about opera, so we will stick to this example for the remainder of this part of the book. But be encouraged to try to transfer the following to different hypothetical scenarios.

Figure 6.2.: Scene from "What's opera, doc?" (1957).

Back to our artificial population. We can count the number of A's and B's amongst the individuals as follows:

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

```
trait
B    52
A    48
Name: count, dtype: int64
```

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)
```

```
trait
B    0.52
```

```
A    0.48
Name: proportion, dtype: float64
```

6.3. 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()
```

	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"]
```

```
0.48
```

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."

6.4. 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()
```

	generation	p
0	0	0.48
1	1	0.39
2	2	0.37
3	3	0.53
4	4	0.45

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)
```

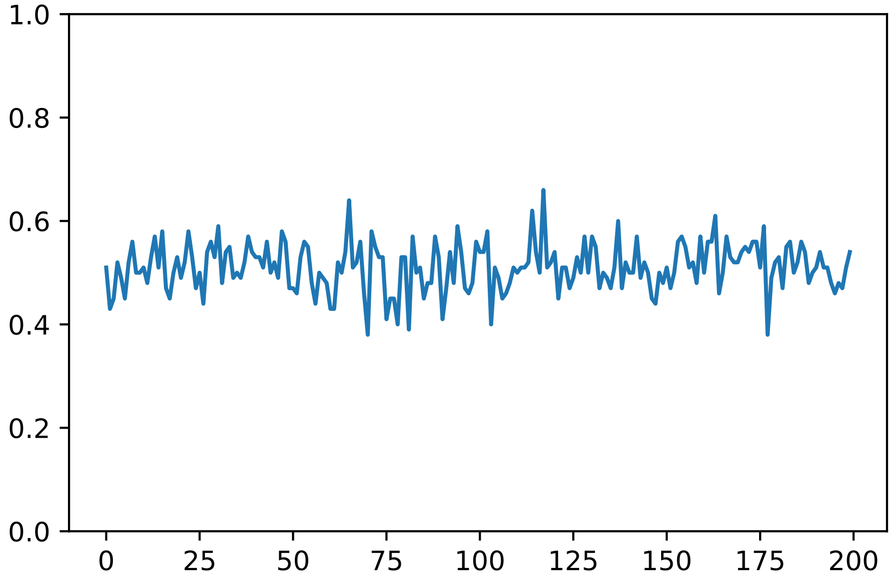


Figure 6.3.: Single run of the unbiased transmission model for a population of $N = 100$ individuals and $t_{max} = 200$ generations.

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

```
plot_single_run(data_model)
```

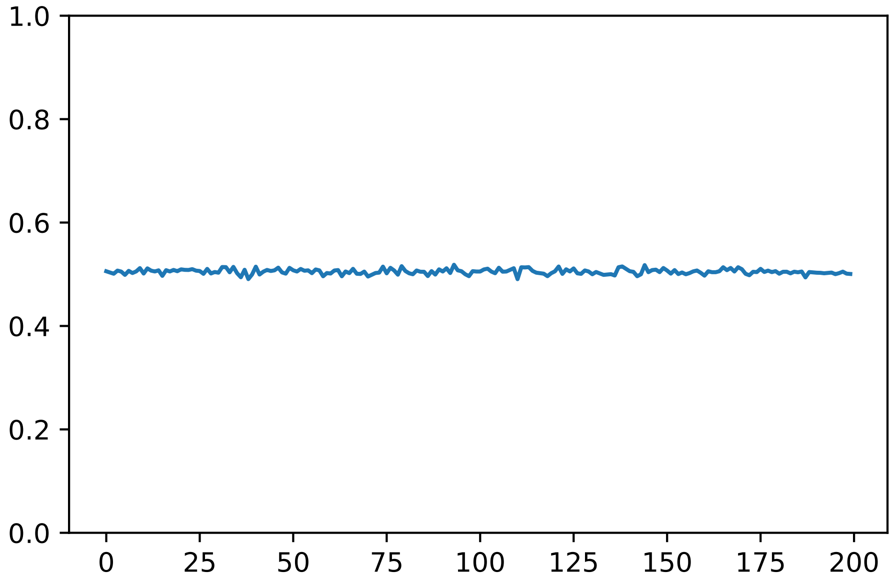


Figure 6.4.: Single run of the unbiased transmission model for a population of $N = 10,000$ individuals and $t_{max} = 200$ generations.

Now, let's adapt the code somewhat.

```
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()
```

	generation	p	run
0	0	0.45	0
1	1	0.42	0
2	2	0.35	0
3	3	0.38	0
4	4	0.39	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)
```

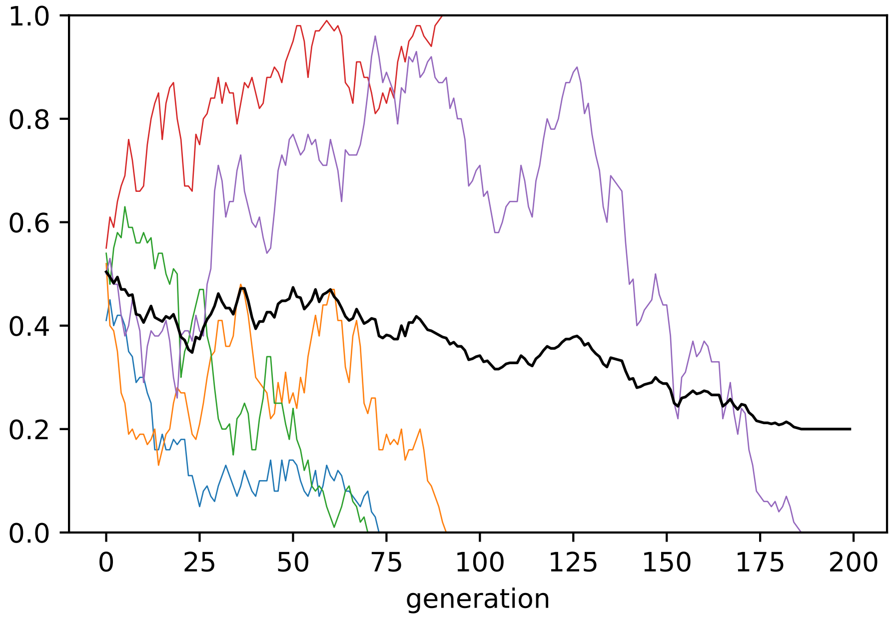


Figure 6.5.: Multiple runs of the unbiased transmission model for a population of $N = 100$ individuals, with average (black line).

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

```
plot_multiple_runs(data_model)
```

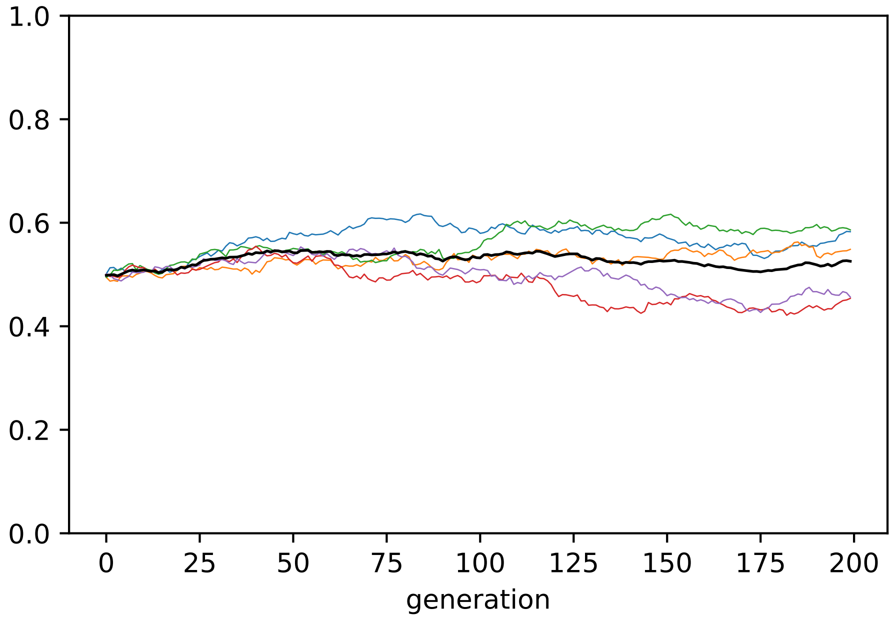
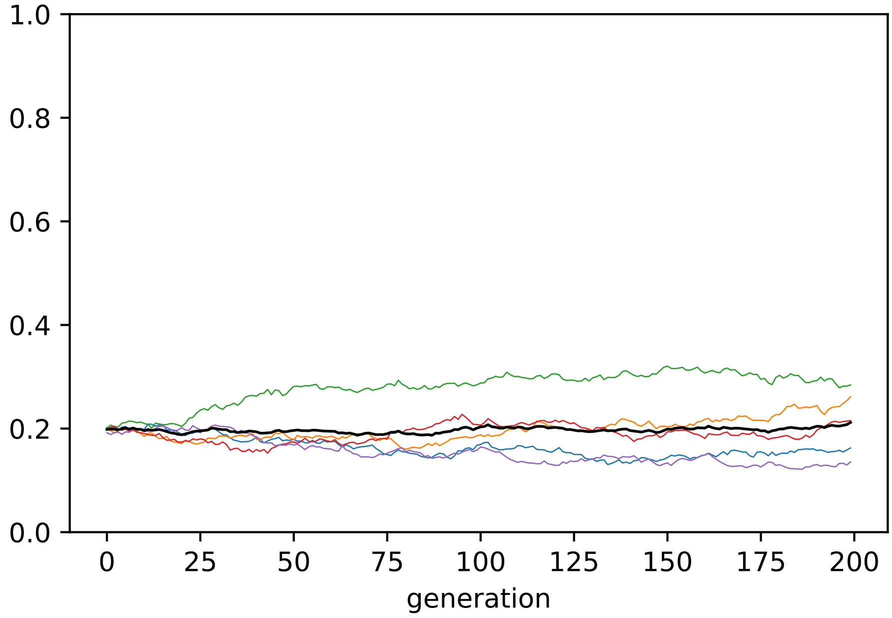


Figure 6.6.: Multiple runs of the unbiased transmission model for a population of $N = 10,000$ individuals, with average (black line).

```
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)
```



The lesson here is that, if the population is large enough and individuals randomly copy from the previous generation, the average frequency of traits in the population will not significantly change.

7. Unbiased mutation

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

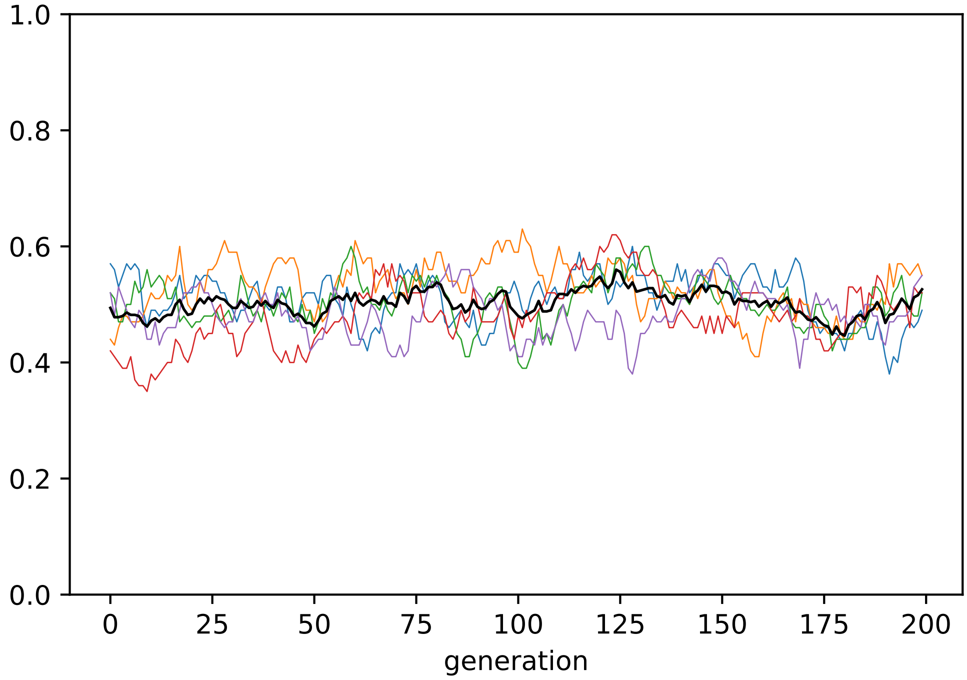
```
import numpy as np
rng = np.random.default_rng()

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
```

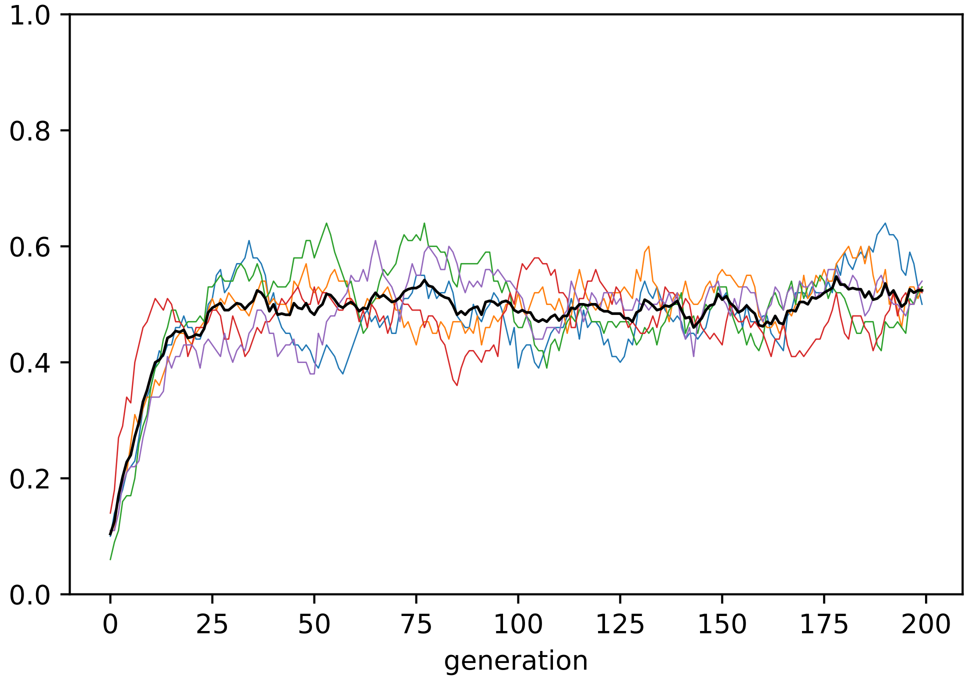
```
            mutate = rng.choice([True, False], size=N, p=[mu, 1-mu],
    ↪   replace=True)

            # 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)
```



8. Biased mutation

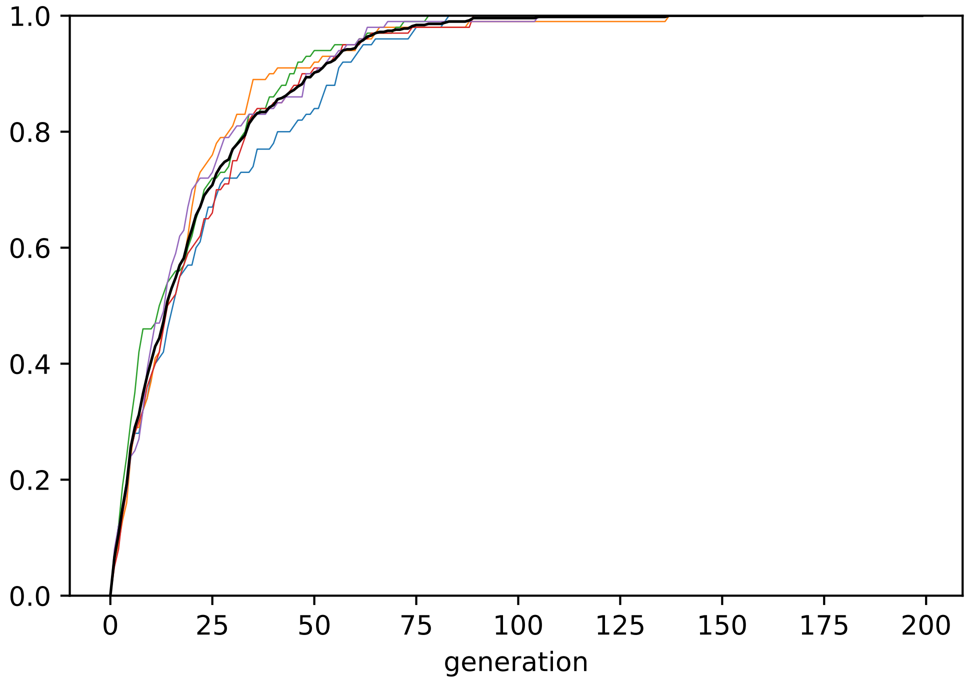
```
import numpy as np
rng = np.random.default_rng()

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")
```

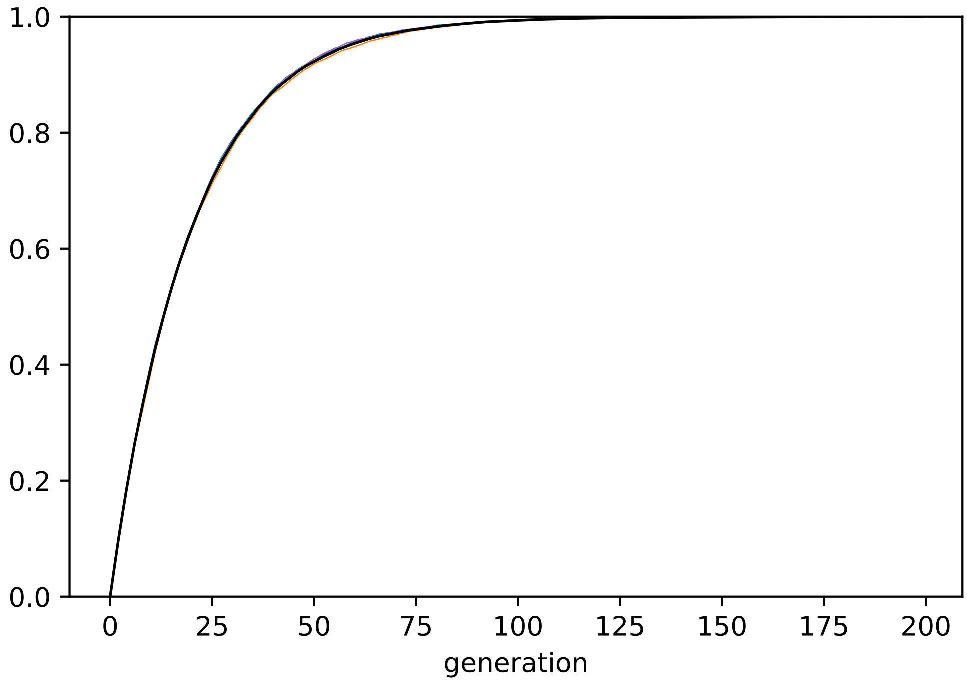
```
def biased_mutation(N, mu_b, p_0, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    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
```

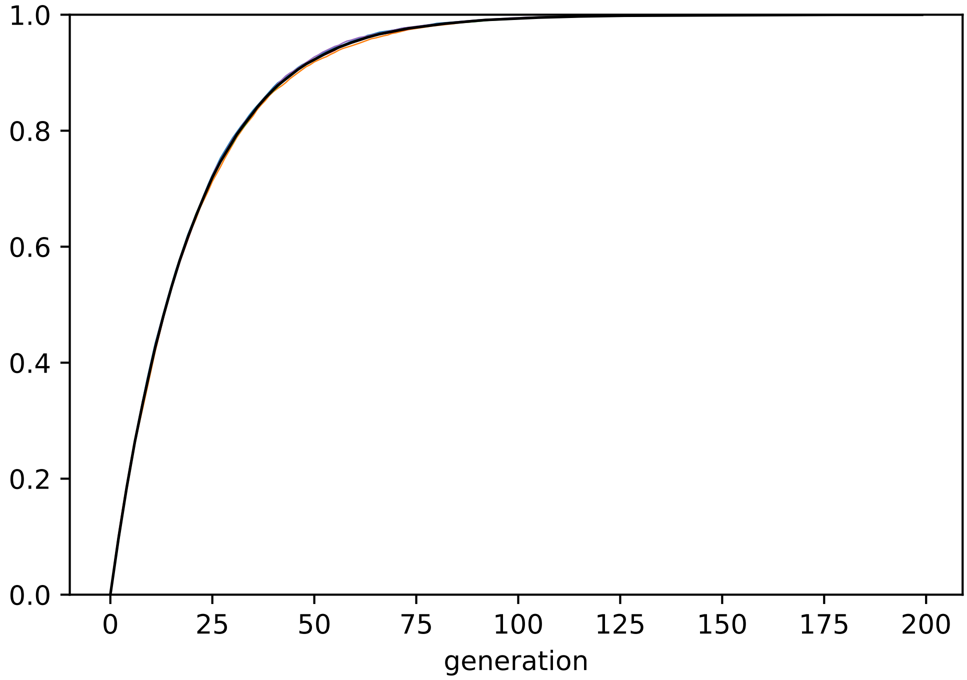
```
data_model = biased_mutation(N = 100, mu_b = 0.05, p_0 = 0, t_max =
↪  200, r_max = 5)
plot_multiple_runs(data_model)
```




```
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)
```



Part III.

Modeling biases

9. Biased transmission: direct bias

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

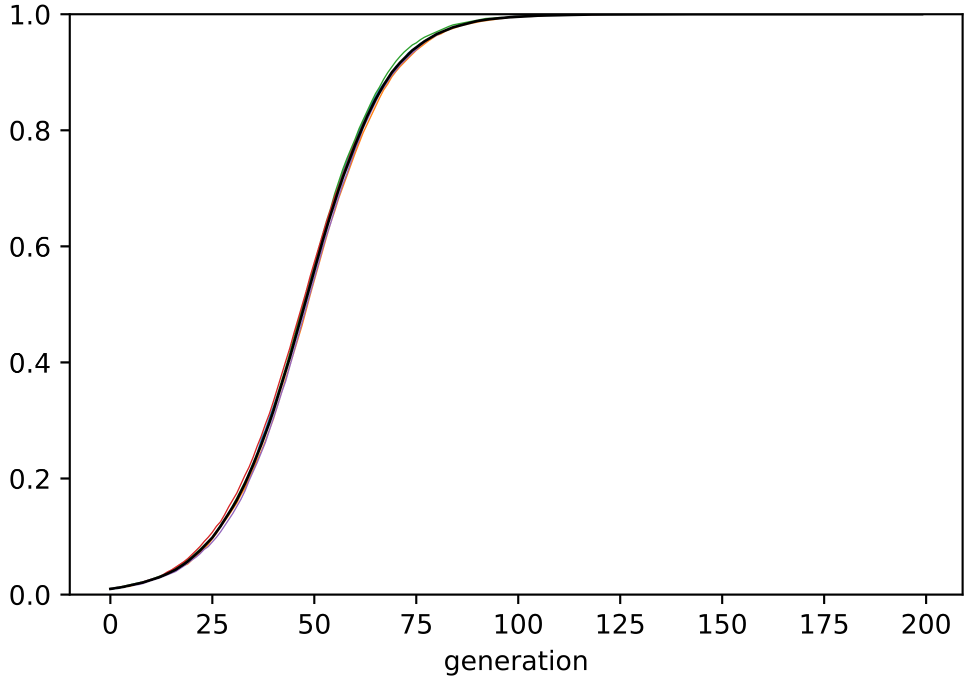
```
import numpy as np
rng = np.random.default_rng()
import pandas as pd
```

```
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")
```

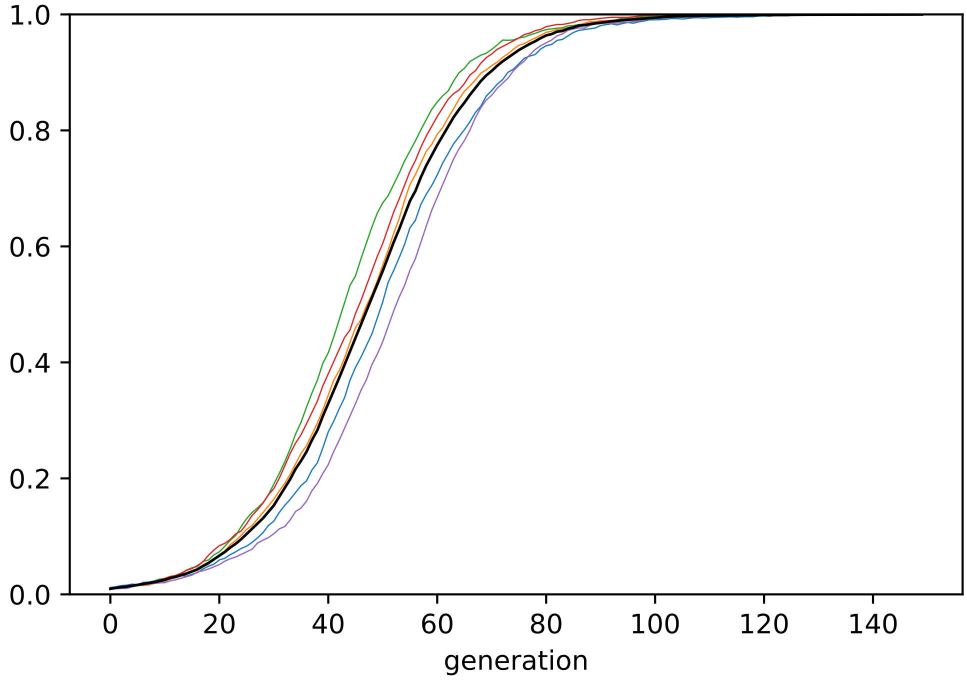
```
def biased_transmission_direct(N, s_a, s_b, p_0, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

    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
```

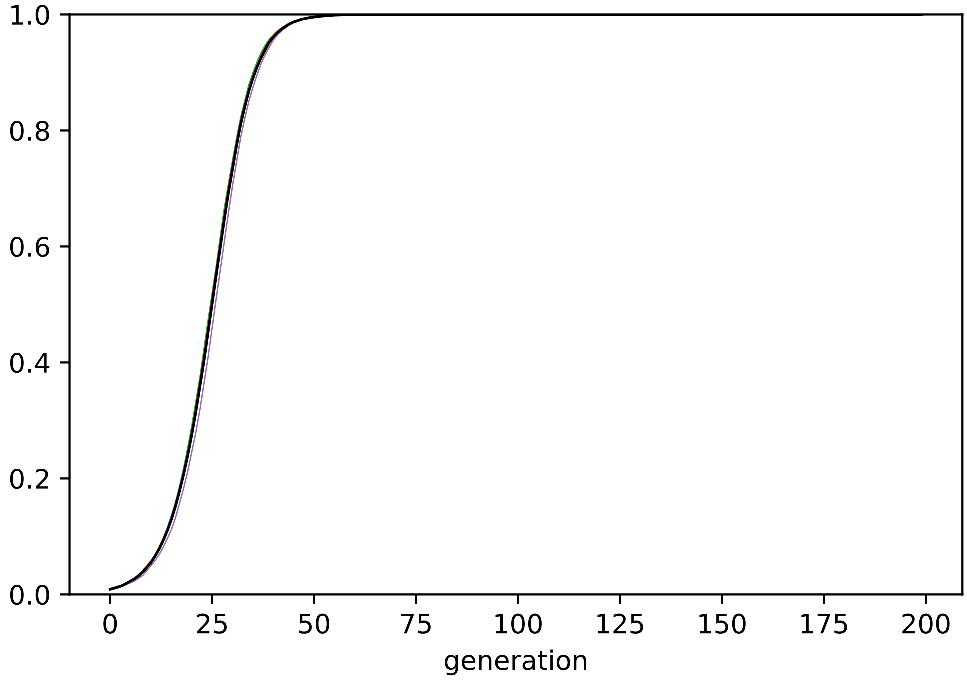
```
data_model = biased_transmission_direct(N=10_000, s_a=.1, s_b=0,
                                         p_0=.01, t_max=200, r_max=5)
plot_multiple_runs(data_model)
```




```
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)
```



10. 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
```

```python
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
})
```

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

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

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

	0
0	2
1	1
2	1
3	3
4	2

```python
# 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()
```

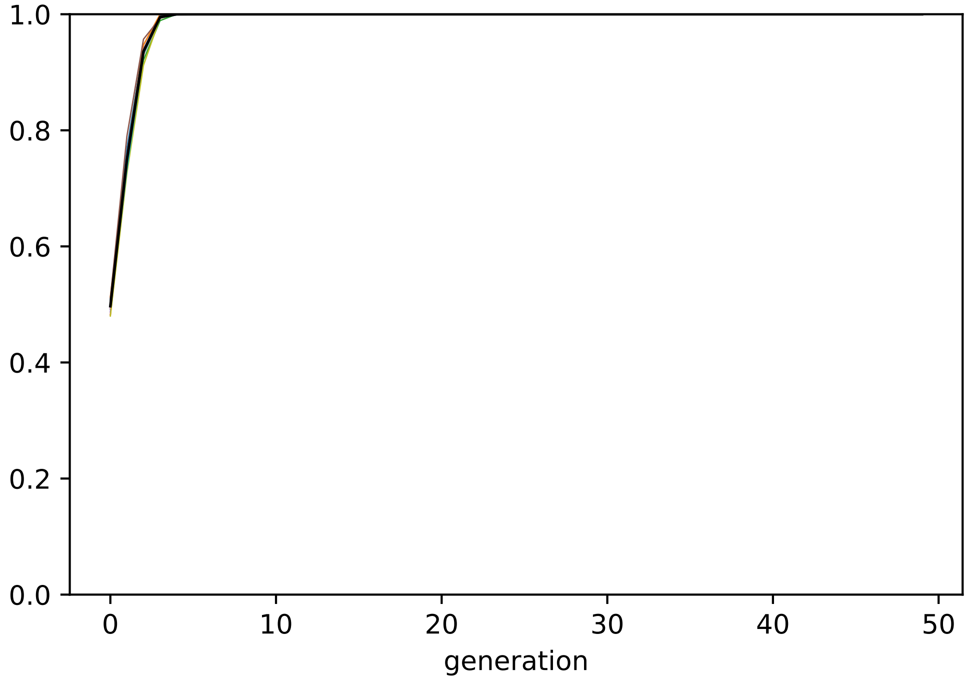
	dem1	dem2	dem3	new_trait
0	B	A	A	A
1	B	B	A	B
2	B	A	B	B
3	A	A	A	A
4	A	B	A	A

```
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)
```


1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50
generation

11. Biased transmission: influencer-based indirect bias

ℹ Note

This chapter is based on "Chapter 5: Biased transmission: demonstrator-based indirect bias" in Acerbi et al. (2022).

Instead of calling them demonstrators, we will call them influencers.

Figure 11.1.: An influencer on a popular video platform.

```
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()
```

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

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

```
if sum(p_influencer) > 0:
    ps = p_influencer / p_influencer.sum()
```

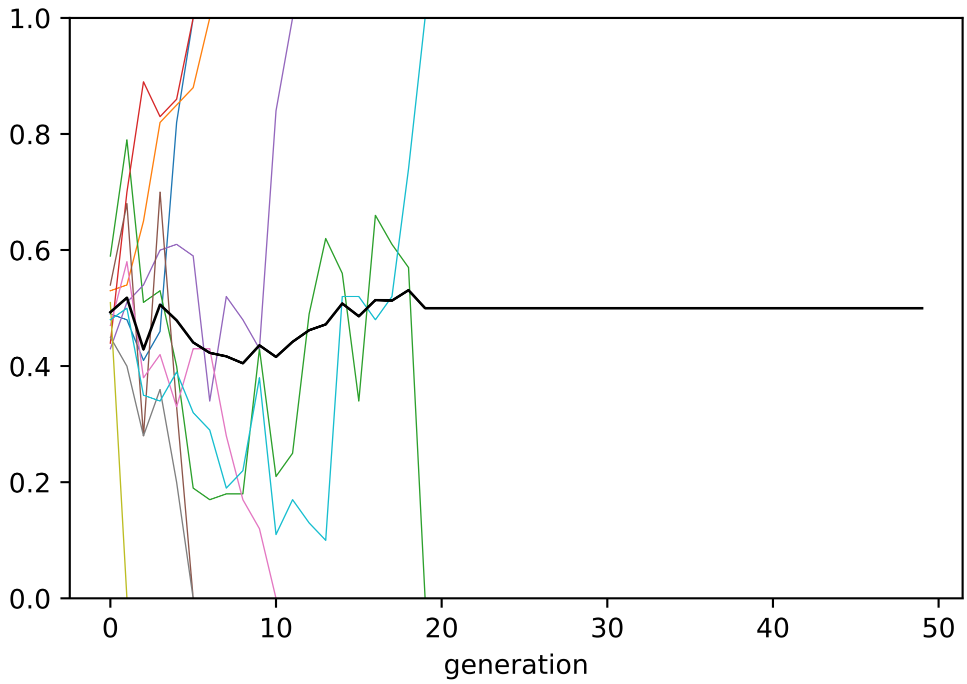
```
    influencer_index = rng.choice(np.arange(N), size=N, p=ps,
↪   replace=True)
    population["trait"] = population.loc[influencer_index,
↪   "trait"].values
```

```
def biased_transmission_influencer(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_influencer = np.ones(N)
            p_influencer[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_influencer) > 0:
                    ps = p_influencer / p_influencer.sum()
                    influencer_index = rng.choice(np.arange(N), size=N,
↪   p=ps, replace=True)
```

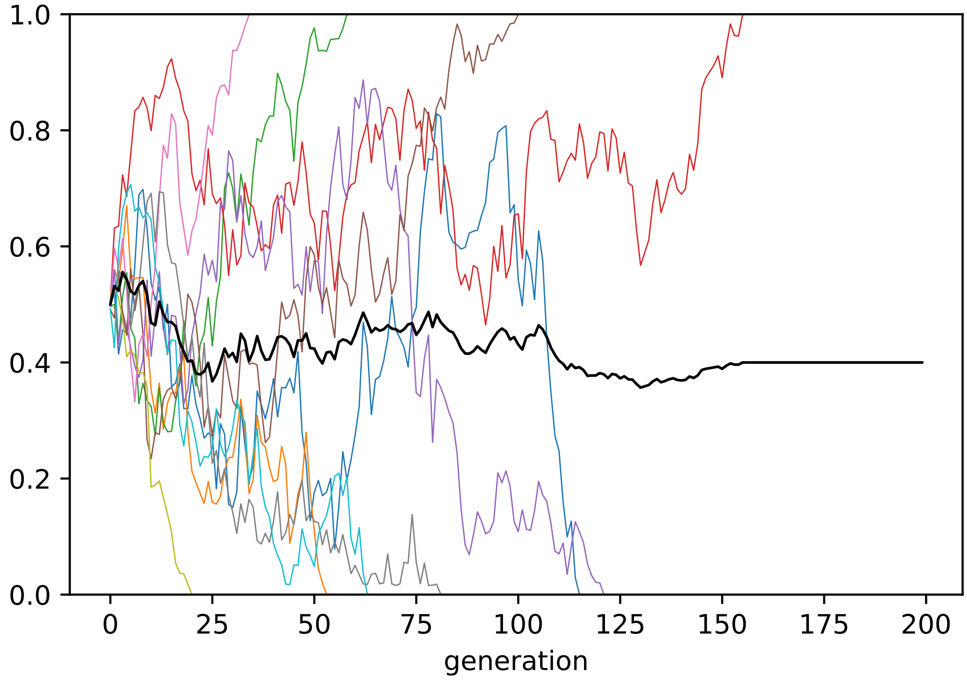
```
                population["trait"] =
↪   population.loc[influencer_index, "trait"].values

            # 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_influencer(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_influencer(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_influencer_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)
    })

    ... # TODO

    return output
```

```
data_model = biased_transmission_influencer_2(N=100, p_s=0.1,
↪   p_low=0.0001, p_0=0.5, t_max=50, r_max=50)
```

12. Vertical and horizontal cultural transmission

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

import pandas as pd
from tqdm import tqdm
```

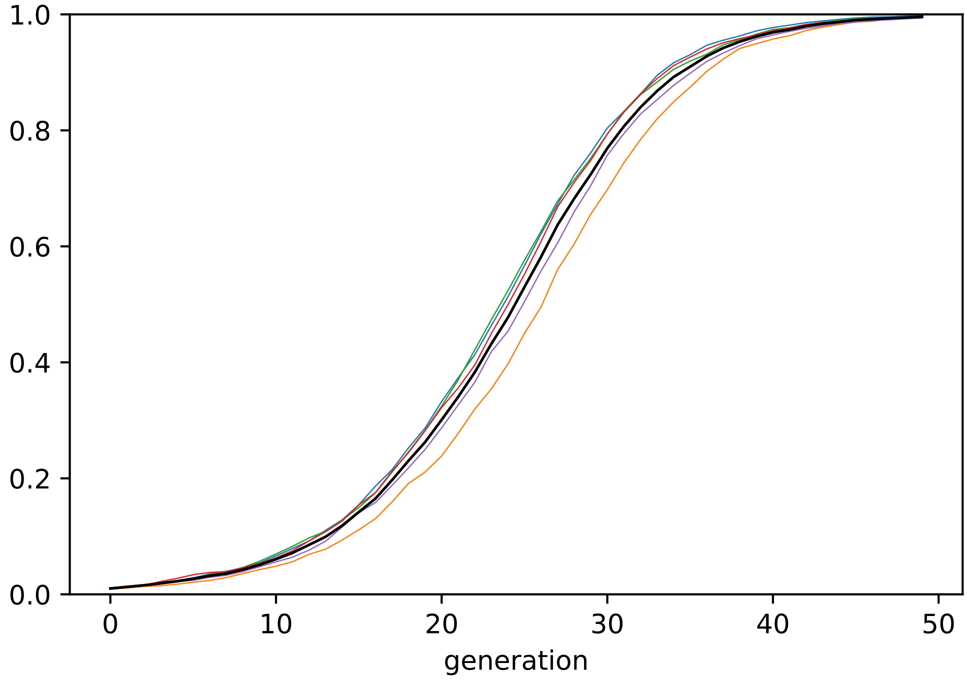
```
def plot_multiple_runs(data_model):
    groups = data_model.groupby("run")
    for _, g in groups:
        g.index = g["generation"]
        g["p"].plot(lw=.5, ylim=(0,1))

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

```
def vertical_transmission(N, p_0, b, 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
```

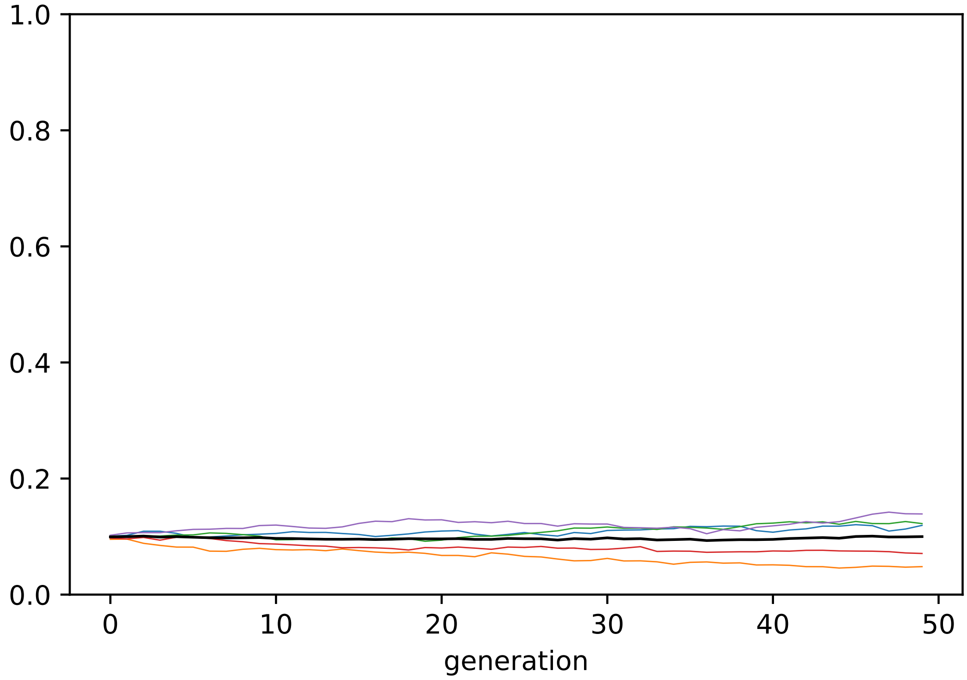
```
        for t in range(1, t_max):
            # Copy individuals to previous_population DataFrame
            previous_population = population.copy()

            # randomly pick mothers and fathers
            mother = previous_population["trait"].sample(N,
↪   replace=True).reset_index(drop=True)
            father = previous_population["trait"].sample(N,
↪   replace=True).reset_index(drop=True)

            # prepare next generation
            population = pd.DataFrame({"trait": [np.nan] * N })

            # Both parents are A, thus child adopts A
            both_A = (mother == "A") & (father == "A")
            # if sum(both_A) > 0:
            population.loc[both_A,"trait"] = "A"

            # Both parents are A, thus child adopts A
            both_B = (mother == "B") & (father == "B")
            # if sum(both_B) > 0:
            population.loc[both_B,"trait"] = "B"

            # If any empty NA slots are present (i.e. one A and one B
             ↪  parent) they adopt A with probability b
            remaining = rng.choice(["A", "B"],
↪   size=population["trait"].isna().sum(), replace=True, p=[b, 1 - b])
            population.loc[population["trait"].isna(),"trait"] =
↪   remaining

            # 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 = vertical_transmission(N=10_000, p_0=0.01, b=0.6,t_max=50,
↪   r_max=5)
```

```
plot_multiple_runs(data_model)
```




```
data_model = vertical_transmission(N=10_000, p_0=0.1, b=0.5,t_max=50,
 ↪  r_max=5)
plot_multiple_runs(data_model)
```




```
def vertical_horizontal_transmission(N, p_0, b, n, g, t_max, r_max):
    # Create an empty dataframe for the output
    output = pd.DataFrame({
        'generation': np.tile(np.arange(1, t_max + 1), r_max),
        'p': np.full(t_max * r_max, np.nan),
        'run': np.repeat(np.arange(1, r_max + 1), t_max)
    })

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

        # Add first generation's p for run r
        output.loc[(output['generation'] == 1) & (output['run'] == r +
    ↪  1), 'p'] = (population['trait'] == 'A').sum() / N

        for t in range(1, t_max):
            # Vertical transmission
            ↪  --------------------------------------------------
```

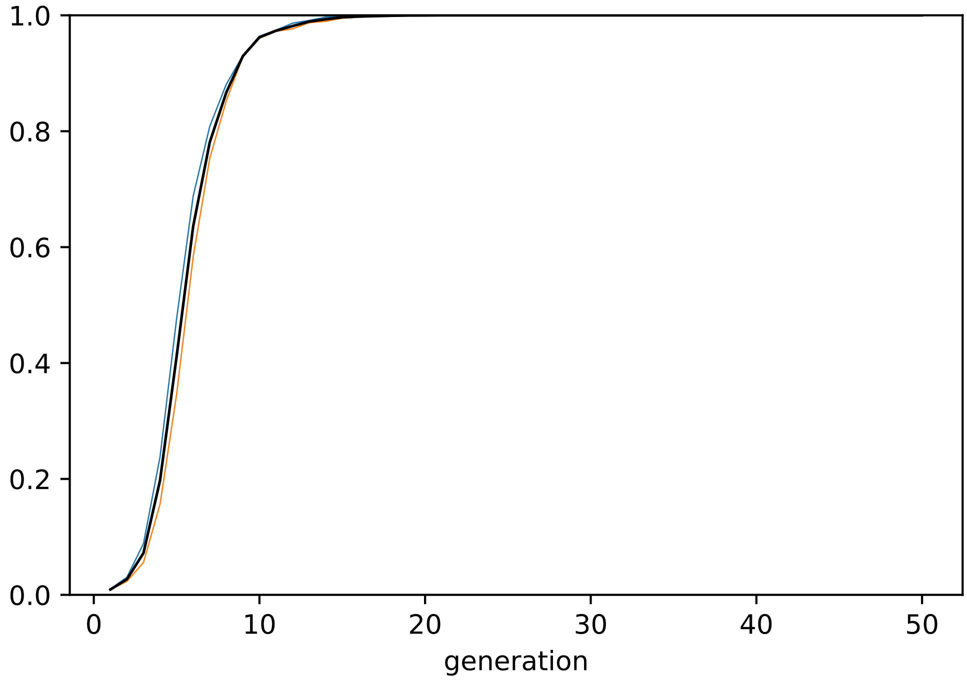
```
        # Copy individuals to previous_population dataframe
        previous_population = population.copy()

        # Randomly pick mothers and fathers
        mother = np.random.choice(previous_population['trait'], N,
↪   replace=True)
        father = np.random.choice(previous_population['trait'], N,
↪   replace=True)

        # Prepare next generation
        population = pd.DataFrame({'trait': [np.nan] * N})

        # Both parents are A, thus child adopts A
        both_A = (mother == 'A') & (father == 'A')
        population.loc[both_A, 'trait'] = 'A'

        # Both parents are B, thus child adopts B
        both_B = (mother == 'B') & (father == 'B')
        population.loc[both_B, 'trait'] = 'B'

        # If any empty NA slots (i.e. one A and one B parent) are
        ↪  present
        if population['trait'].isna().any():
            # They adopt A with probability b
            population.loc[population['trait'].isna(), 'trait'] =
↪   np.random.choice(
                ['A', 'B'],
                size=population['trait'].isna().sum(),
                p=[b, 1 - b]
            )

        # Horizontal transmission
        ↪  ------------------------------------------------

        # Previous_population are children before horizontal
        ↪  transmission
        previous_population = population.copy()

        # N_B = number of Bs
        N_B = (previous_population['trait'] == 'B').sum()
```

```
            # If there are B individuals to switch, and n is not zero
            if N_B > 0 and n > 0:
                # For each B individual...
                for i in range(N_B):
                    # Pick n demonstrators
                    demonstrator =
 ↪  np.random.choice(previous_population['trait'], n, replace=True)
                    # Get probability g
                   copy = np.random.choice([True, False], size=n, p=[g,
 ↪  1 - g])

                    # If any demonstrators with A are to be copied
                    if (demonstrator == 'A').any() & copy.any():
                        # The B individual switches to A
                        b_indices =
 ↪  previous_population[previous_population['trait'] == 'B'].index
                        population.loc[b_indices[i], 'trait'] = 'A'

            # Get p and put it into output slot for this generation t
             ↪  and run r
            output.loc[(output['generation'] == t + 1) & (output['run']
 ↪  == r + 1), 'p'] = (population['trait'] == 'A').sum() / N

    return output
```

```
vertical_horizontal_transmission(N=1000, p_0=0.01, b=0.5, n=5, g=0.1,
 ↪  t_max=10, r_max=1)
```

	generation	p	run
0	1	0.007	1
1	2	0.011	1
2	3	0.023	1
3	4	0.065	1
4	5	0.169	1
5	6	0.411	1
6	7	0.618	1
7	8	0.758	1
8	9	0.851	1
9	10	0.908	1

```
data_model = vertical_horizontal_transmission(N=5_000, p_0=0.01, b=0.5,
    n=5, g=0.1, t_max=50, r_max=2)
plot_multiple_runs(data_model)
```



13. The multiple traits model

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

import matplotlib.pyplot as plt

N = 100
population = pd.DataFrame(
    {"trait" : rng.integers(N, size=N)}
)
```

```
population.head()
```

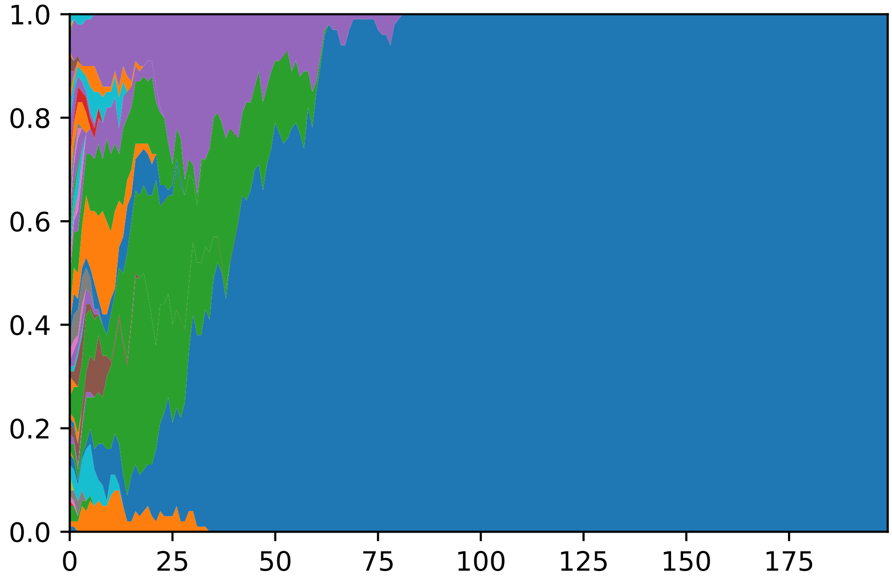
	trait
0	35
1	88
2	7
3	4
4	75

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

    # Create first generation
```

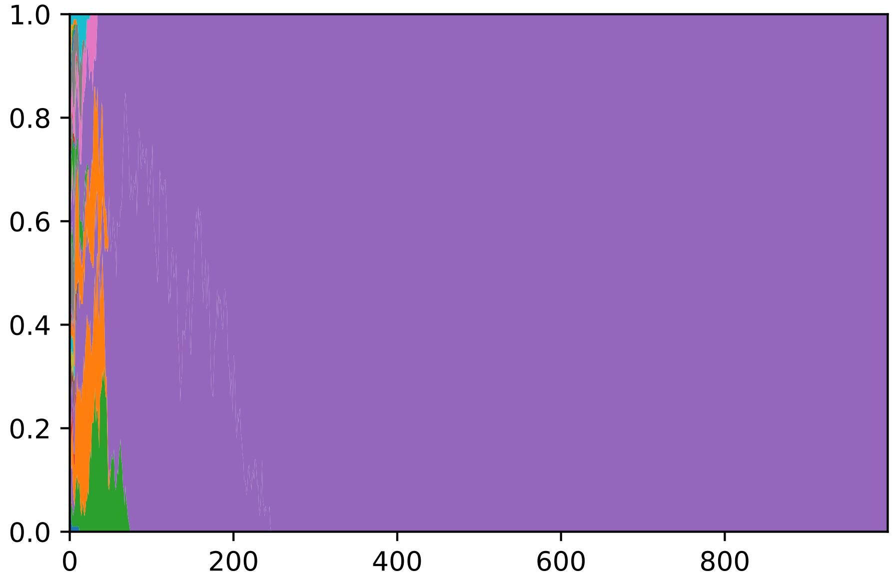
```
    population = pd.DataFrame({"trait" : rng.integers(N, size=N)})

    # Add first generation's p for all traits
    output.loc[output["generation"] == 0, "p"] = popula-
 ↪  tion["trait"].value_counts(normalize=True).reindex(range(N)).fillna(0.).values

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

        # Randomly copy from previous generation
        population = pd.DataFrame({"trait" :
 ↪  previous_population["trait"].sample(N, replace=True)})

        # Get p for all traits and put it into output slot for this
         ↪  generation t
        output.loc[output["generation"] == t, "p"] = popula-
 ↪  tion["trait"].value_counts(normalize=True).reindex(range(N)).fillna(0.).values

    return output
```

```
def plot_multiple_traits(data_model):
    ps = []
    for _, g in data_model.groupby("trait"):
        x = g["generation"]
        ps.append(g["p"])

    plt.stackplot(x, *ps, cmap="tab20")
    plt.margins(0.)
    plt.show()
```

```
data_model = multiple_traits(N=100, t_max=200)
plot_multiple_traits(data_model)
```




```
data_model = multiple_traits(N=100, t_max=1000)
plot_multiple_traits(data_model)
```



13.1. Introducing innovation

Part IV.

Advanced topics

14. The usage of pitches and intervals

- "Cross-cultural data shows musical scales evolved to maximise imperfect fifths" McBride and Tlusty (2020)
- "The line of fifths and the co-evolution of tonal pitch-classes" (Moss et al., 2022)

15. Folk tune complexity

“The role of population size in folk tune complexity” (Street et al., 2022)

16. Music communities

studying the cultural evolution of electronic music

“Phylogenetic reconstruction of the cultural evolution of electronic music via dynamic community detection (1975–1999)” (Youngblood et al., 2021)

17. Style evolution

- "Statistical Evolutionary Laws in Music Styles" (Nakamura & Kaneko, 2019)
- "Investigating style evolution of Western classical music: A computational approach" (Weiß et al., 2019)

18. Conclusion

18.1. What can cultural evolution tell us about music

18.2. What is the role of models for musicology

18.3. Avenues for future research

19. Appendix

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/

Appiah, K. A. (2018). *The Lies that Bind: Rethinking Identity*. W. W. Norton & Company.

Aunger, R. (2001). *Darwinizing Culture: The Status of Memetics as a Science*. Oxford University Press, USA. Retrieved December 17, 2021, from http://gen.lib.rus.ec/book/index.php?md5=7329e2aa9adcddfed967088219426193

Bentley, R. A., Hahn, M. W., & Shennan, S. J. (2004). Random drift and culture change. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, *271*(1547), 1443–1450. https://doi.org/10.1098/rspb.2004.2746

Bishop, C. M. (2012). Model-based machine learning. *Philosophical Transactions of The Royal Society A*, *0222*(371). https://doi.org/10.1098/rsta.2012.0222

Blackmore, S. (2000, March 16). *The Meme Machine*. Oxford University Press.

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

Cavalli-Sforza, L. L., & Feldman, M. W. (1981). *Cultural Transmission and Evolution*. Princeton University Press.

Cross, I. (2016). The nature of music and its evolution. In S. Hallam, I. Cross, & M. Thaut (Eds.), *The Oxford Handbook of Music Psychology* (2nd ed., pp. 1–20). Oxford University Press. https://doi.org/10.1093/oxfordhb/9780199298457.013.0001

Dawkins, R. (1976). *The Selfish Gene*. Oxford University Press.
OCLC: 2681149.

Farrell, S., & Lewandowsky, S. (2018). *Computational Modeling of Cognition and Behavior*. Cambridge University Press. https://doi.org/10.1017/CBO9781316272503

Finkensiep, C., Neuwirth, M., & Rohrmeier, M. (forthcoming). Music Theory and Model-driven Corpus Research. In D. Shanahan, J. A. Burgoyne, & I. Quinn (Eds.), *Oxford Handbook of Music and Corpus Studies*. Oxford University Press.

Gjerdingen, R. O. (2007). *Music in the Galant Style*. Oxford University Press.

Honing, H. (2006). Computational Modeling of Music Cognition: A Case Study on Model Selection. *Music Perception*, *23*(5), 365–376. https://doi.org/10.1525/mp.2006.23.5.365

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

Howe, C. J., & Windram, H. F. (2011). Phylomemetics—Evolutionary Analysis beyond the Gene. *PLOS Biology*, *9*(5), e1001069. https://doi.org/10.1371/journal.pbio.1001069

International Folk Music Council. (1955). Resolutions. *Journal of the International Folk Music Council*, *7*, 23–23. Retrieved March 21, 2023, from https://www.jstor.org/stable/834530

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

Karpeles, M. (1955). Definition of Folk Music. *Journal of the International Folk Music Council*, *7*, 6–7. https://doi.org/10.2307/834518

McBride, J. M., & Tlusty, T. (2020, June 1). *Cross-cultural data shows musical scales evolved to maximise imperfect fifths*. Retrieved September 18, 2021, from http://arxiv.org/abs/1906.06171

McElreath, R. (2020). *Statistical Rethinking: A Bayesian Course with Examples in R and STAN* (Second). Chapman and Hall/CRC.

Meyer, L. B. (1989). *Style and Music. Theory, History, and Ideology*. University of Chicago Press.

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*. 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

Pinker, S. (1997). *How the mind works*. Norton.

Piotrowski, M. (2019). Historical Models and Serial Sources. *Journal of European Periodical Studies*, *4*(1), 8–18. https://doi.org/10.21825/jeps.v4i1.10226

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

Smaldino, P. E. (2017, May). Models Are Stupid, and We Need More of Them. In R. R. Vallacher, S. J. Read, & A. Nowak (Eds.), *Computational Social Psychology* (First, pp. 311–331). Routledge. https://doi.org/10.4324/9781315173726-14

Smaldino, P. E. (2023). *Modeling Social Behavior: Mathematical and Agent-Based Models of Social Dynamics and Cultural Evolution*. Princeton University Press.

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

Tomlinson, G. (2018). *A Million Years of Music*. Princeton University Press. Retrieved January 27, 2022, from 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.

Weiß, C., Mauch, M., Dixon, S., & Müller, M. (2019). Investigating style evolution of Western classical music: A computational approach. *Musicae Scientiae*, *23*(4), 486–507. https://doi.org/10.1177/1029864918757595

Youngblood, M., Baraghith, K., & Savage, P. E. (2021). Phylogenetic reconstruction of the cultural evolution of electronic music via dynamic community detection (1975–1999). *Evolution and Human Behavior*. https://doi.org/10.1016/j.evolhumbehav.2021.06.002

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