Cultural Evolution Models

Alberto Acerbi / Fabian C. Moss

8/8/2022

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

1 Introduction

This is a book created from markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

2 Summary

In summary, this book has no content whatsoever.

3 Unbiased transmission

Alberto Acerbi / Fabian C. Moss

Import some modules.

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

import pandas as pd
```

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

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

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	trait
0	B
1	B
2	B
3	B
4	B

```
output = pd.DataFrame(
    {
        "generation": np.arange(t_max, dtype=int),
```

```
        "p": [np.nan] * t_max
    }
)
output
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	generation	p
0	0	NaN
1	1	NaN
2	2	NaN
3	3	NaN
4	4	NaN
5	5	NaN
6	6	NaN
7	7	NaN
8	8	NaN
9	9	NaN
10	10	NaN
11	11	NaN
12	12	NaN
13	13	NaN
14	14	NaN
15	15	NaN
16	16	NaN
17	17	NaN
18	18	NaN
19	19	NaN
20	20	NaN
21	21	NaN
22	22	NaN
23	23	NaN
24	24	NaN
25	25	NaN
26	26	NaN
27	27	NaN
28	28	NaN
29	29	NaN
30	30	NaN
31	31	NaN
32	32	NaN
33	33	NaN
34	34	NaN
35	35	NaN
36	36	NaN
37	37	NaN
38	38	NaN
39	39	NaN
40	40	NaN
41	41	NaN
42	42	NaN
43	43	NaN
44	44	NaN
45	45	NaN
46	46	NaN
47	47	NaN
48	48	NaN
49	49	NaN
50	50	NaN
51	51	NaN

```
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
    population = 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"] = population[ population["trait"] == "A"].shape[0] / N
```

```
output
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	generation	p
0	0	0.38
1	1	0.31
2	2	0.26
3	3	0.22
4	4	0.19
5	5	0.19
6	6	0.18
7	7	0.16
8	8	0.16
9	9	0.16
10	10	0.16
11	11	0.15
12	12	0.17
13	13	0.18
14	14	0.18
15	15	0.24
16	16	0.26
17	17	0.29
18	18	0.22
19	19	0.24
20	20	0.30
21	21	0.22
22	22	0.27
23	23	0.22
24	24	0.27
25	25	0.30
26	26	0.42
27	27	0.34
28	28	0.25
29	29	0.31
30	30	0.28
31	31	0.27
32	32	0.23
33	33	0.28
34	34	0.27
35	35	0.22
36	36	0.21
37	37	0.19
38	38	0.26
39	39	0.31
40	40	0.42
41	41	0.45
42	42	0.50
43	43	0.48
44	44	0.42
45	45	0.43
46	46	0.42
47	47	0.44
48	48	0.43
49	49	0.45
50	50	0.41
51	51	0.51

```
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_ma
    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
        population = 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"] = population[ population["trait"] == "A"].shape[0] / N

    return output
```

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

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

```
plot_single_run(data_model)
```

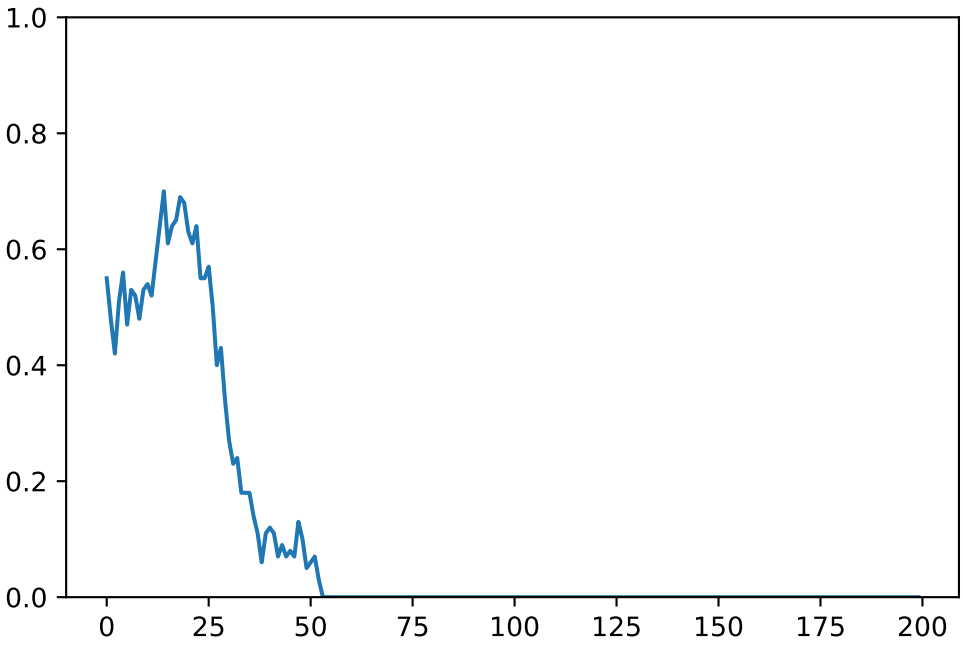


Figure 3.1: Single run of the unbiased transmission model for a population of $N = 100$ individuals and $t_{max} = 200$ generations.

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

```
plot_single_run(data_model)
```

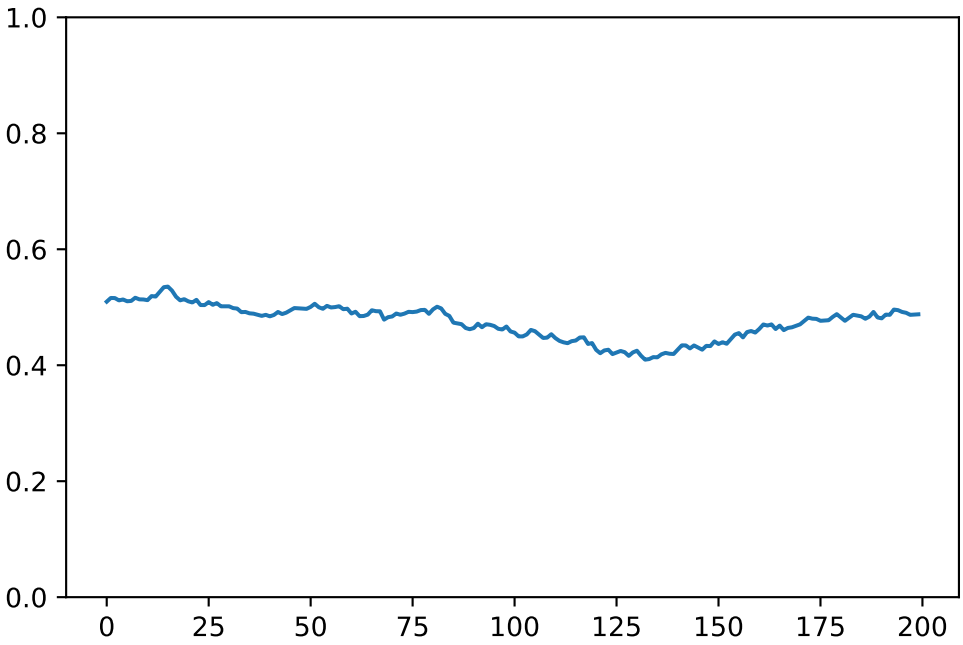


Figure 3.2: Single run of the unbiased transmission model for a population of $N = 10,000$ individuals and $t_{max} = 200$ generations.

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

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

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

        # 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" ].shap

    return output
```

```
unbiased_transmission_2(100, 100, 3)
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	generation	p	run
0	0	0.49	0
1	1	0.48	0
2	2	0.47	0
3	3	0.50	0
4	4	0.49	0
5	5	0.47	0
6	6	0.43	0
7	7	0.35	0
8	8	0.36	0
9	9	0.33	0
10	10	0.36	0
11	11	0.35	0
12	12	0.36	0
13	13	0.38	0
14	14	0.34	0
15	15	0.41	0
16	16	0.39	0
17	17	0.29	0
18	18	0.34	0
19	19	0.39	0
20	20	0.34	0
21	21	0.38	0
22	22	0.39	0
23	23	0.44	0
24	24	0.42	0
25	25	0.45	0
26	26	0.45	0
27	27	0.41	0
28	28	0.46	0
29	29	0.54	0
30	30	0.60	0
31	31	0.64	0
32	32	0.58	0
33	33	0.55	0
34	34	0.55	0
35	35	0.73	0
36	36	0.68	0
37	37	0.71	0
38	38	0.74	0
39	39	0.82	0
40	40	0.76	0
41	41	0.75	0
42	42	0.73	0
43	43	0.75	0
44	44	0.73	0
45	45	0.77	0
46	46	0.75	0
47	47	0.73	0
48	48	0.71	0
49	49	0.70	0
50	50	0.65	0
51	51	0.58	0

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

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

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

```
plot_multiple_runs(data_model)
```

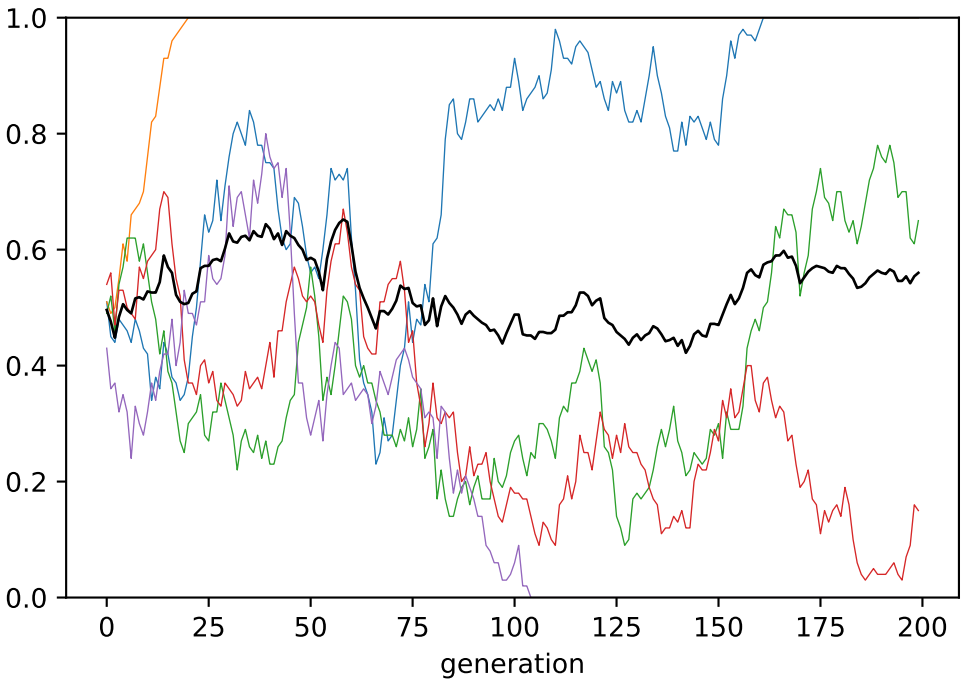


Figure 3.3: Multiple runs of the unbiased transmission model for a population of $N = 100$ individuals, with average (black line).

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

```
plot_multiple_runs(data_model)
```

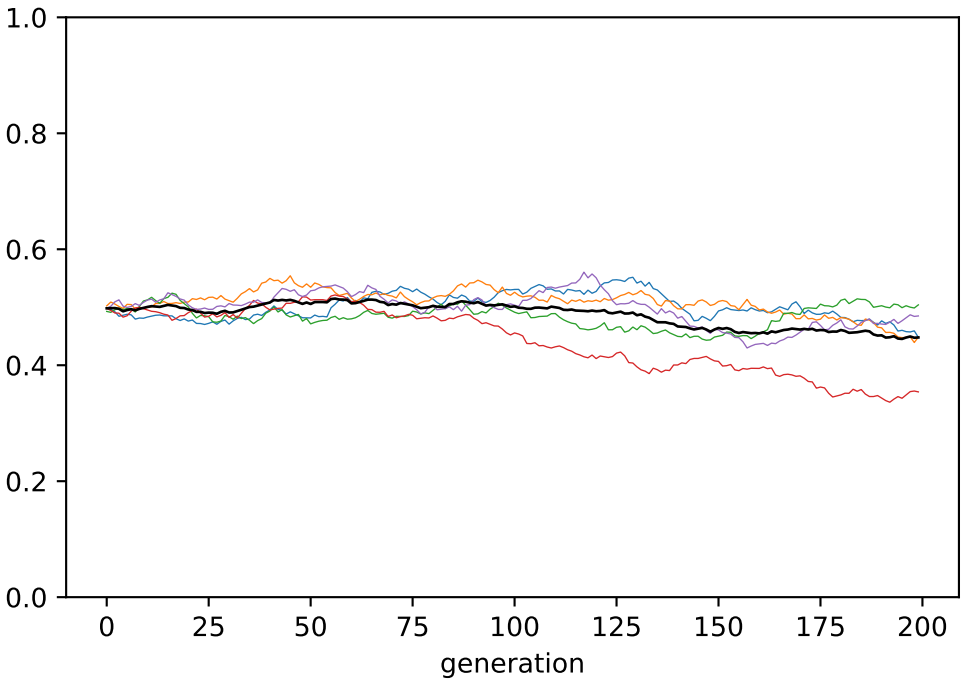
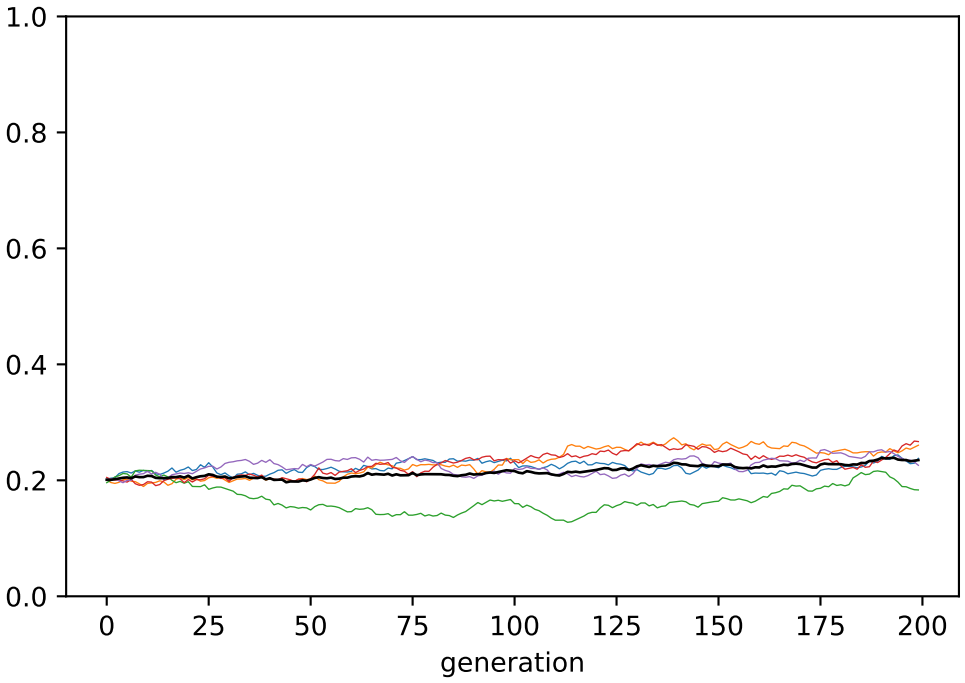


Figure 3.4: Multiple runs of the unbiased transmission model for a population of $N = 10,000$ individuals, with average (black line).

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

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

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

        # 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" ].shap

    return output
```

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



4

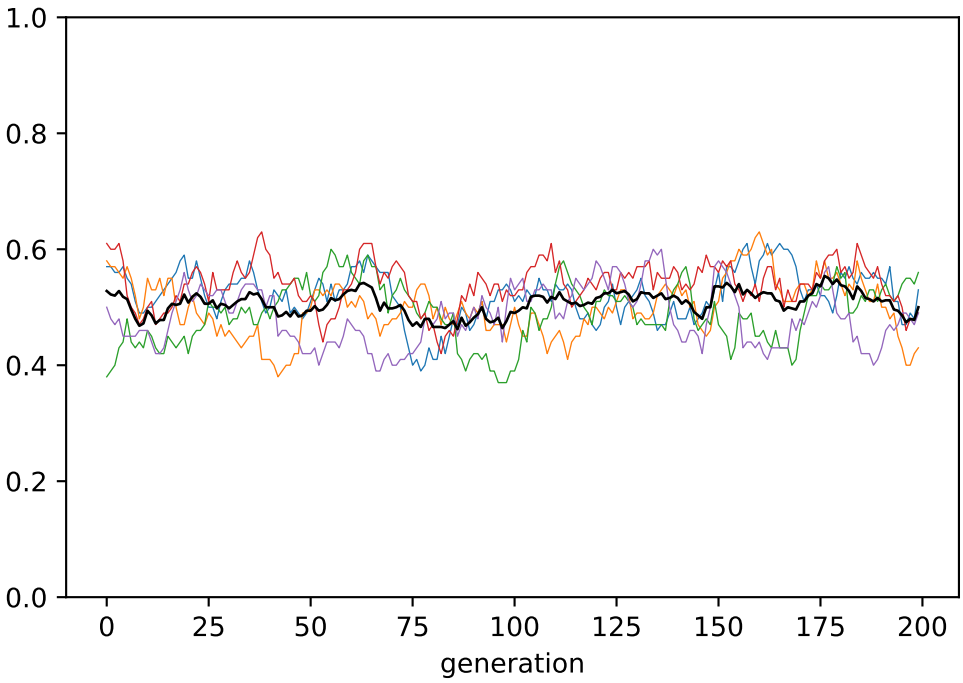
```
import numpy as np
rng = np.random.default_rng()

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

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

        # 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 o

            # If there are "mutants" from A to B
            conditionA = mutate & (previous_population["trait"] == "A")
```

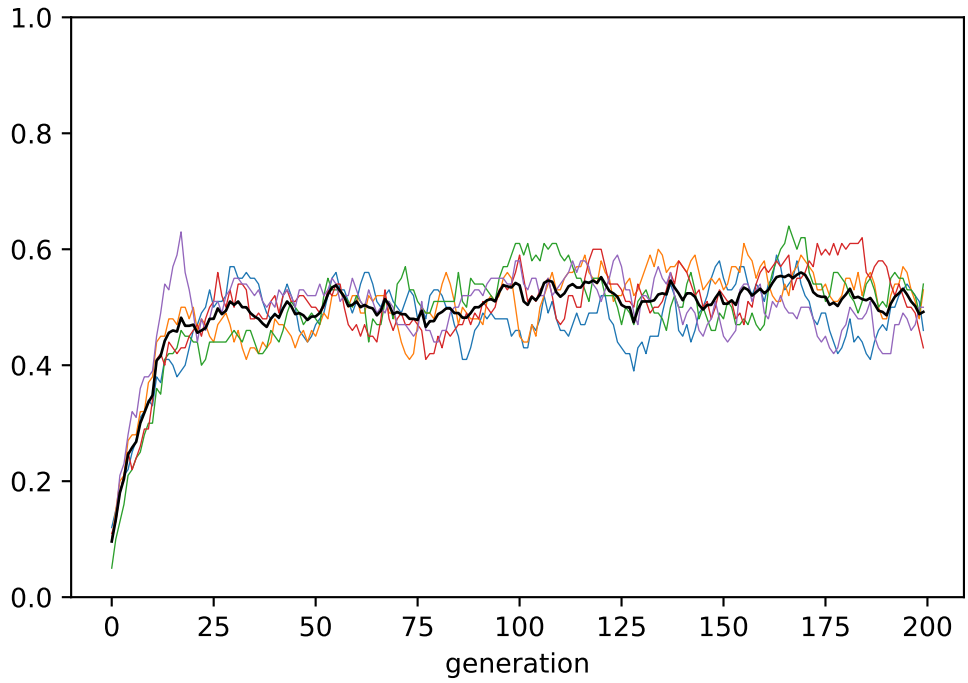
```
            if conditionA.sum() > 0:
                population.loc[conditionA, "trait"] = "B"

            # 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" ].shap

    return output
```

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

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

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



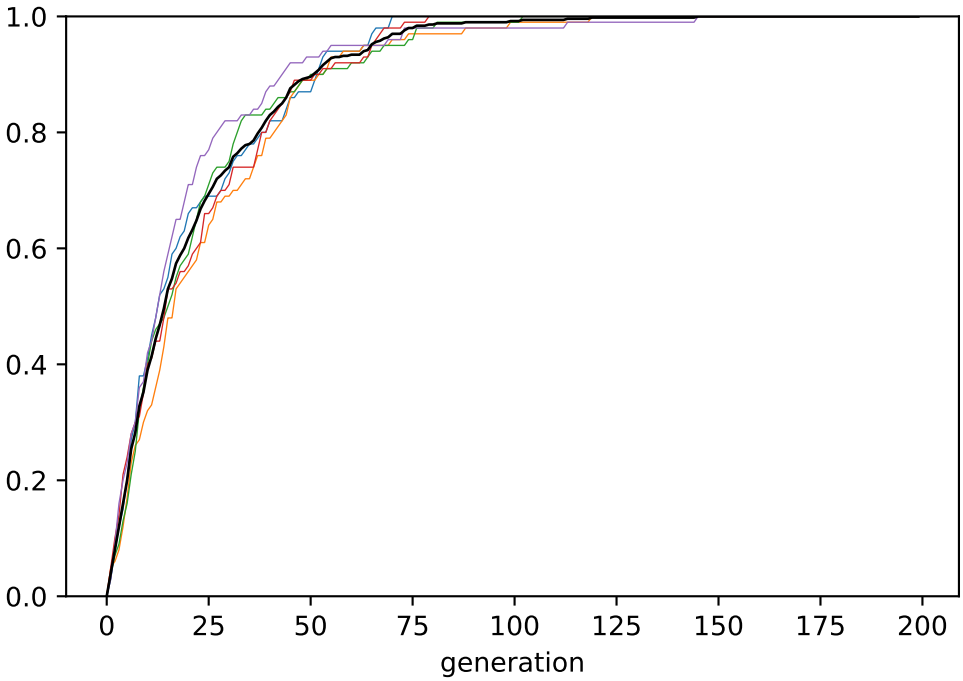

```
data_model = unbiased_mutation(N=100, mu=.05, p_0=0.1, t_max=200, r_max=5)
plot_multiple_runs(data_model)
```



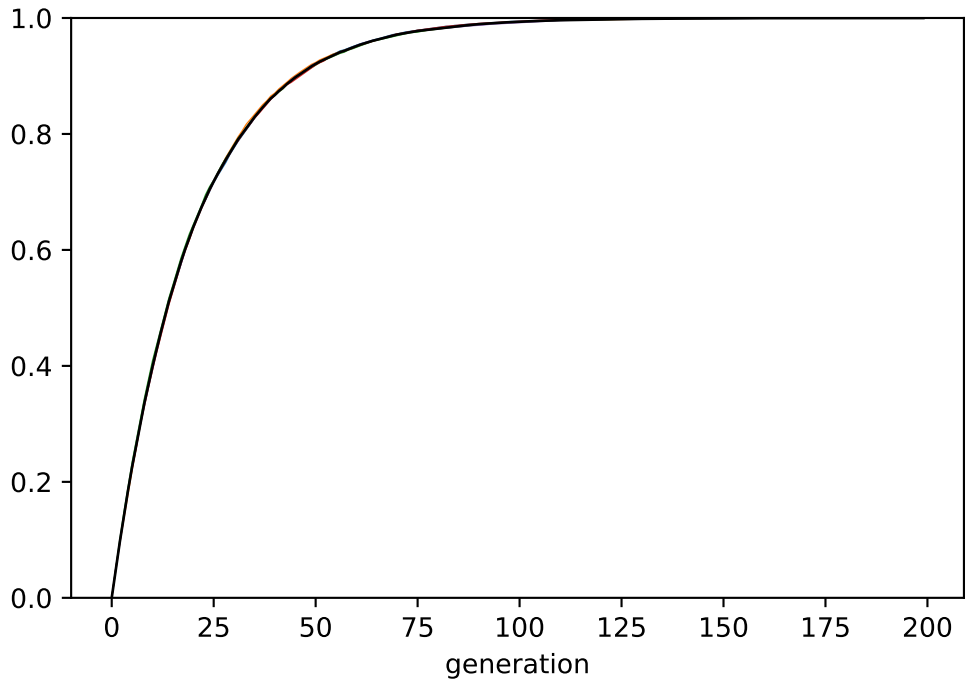

```
def biased_mutation(N, mu_b, p_0, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

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

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

        # 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 o

            # 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" ].shap

    return output
```

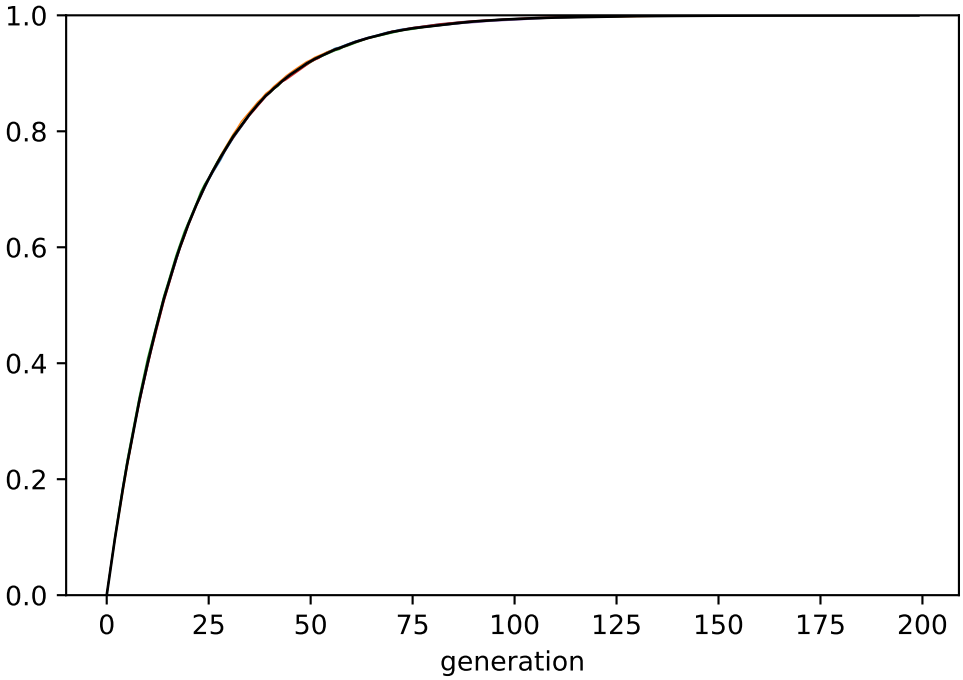
```
data_model = biased_mutation(N = 100, mu_b = 0.05, p_0 = 0, t_max = 200, r_max = 5)
plot_multiple_runs(data_model)
```



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




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


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

5

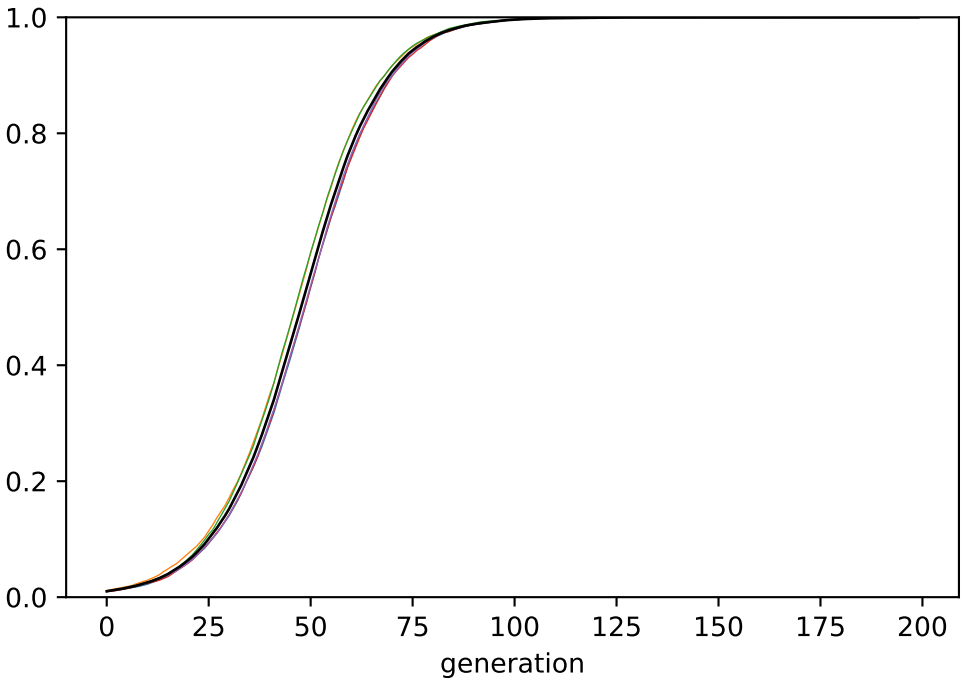
```
import numpy as np
rng = np.random.default_rng()
import pandas as pd
```

```
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")
```

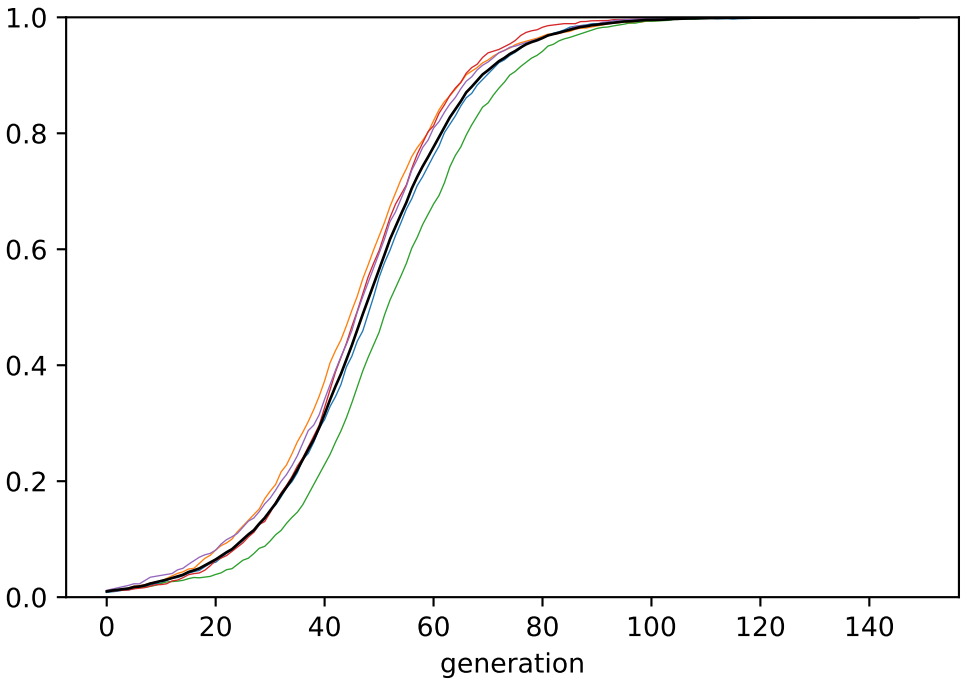
```
def biased_transmission_direct(N, s_a, s_b, p_0, t_max, r_max):
    # Create the output DataFrame
    output = pd.DataFrame({
        "generation" : np.tile(np.arange(t_max), r_max),
        "p" : [ np.nan ] * t_max * r_max,
        "run" : np.repeat(np.arange(r_max), t_max)
    })

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

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

        # 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).rese

            # 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 cop
            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 cop
            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" ].shap

    return output
```

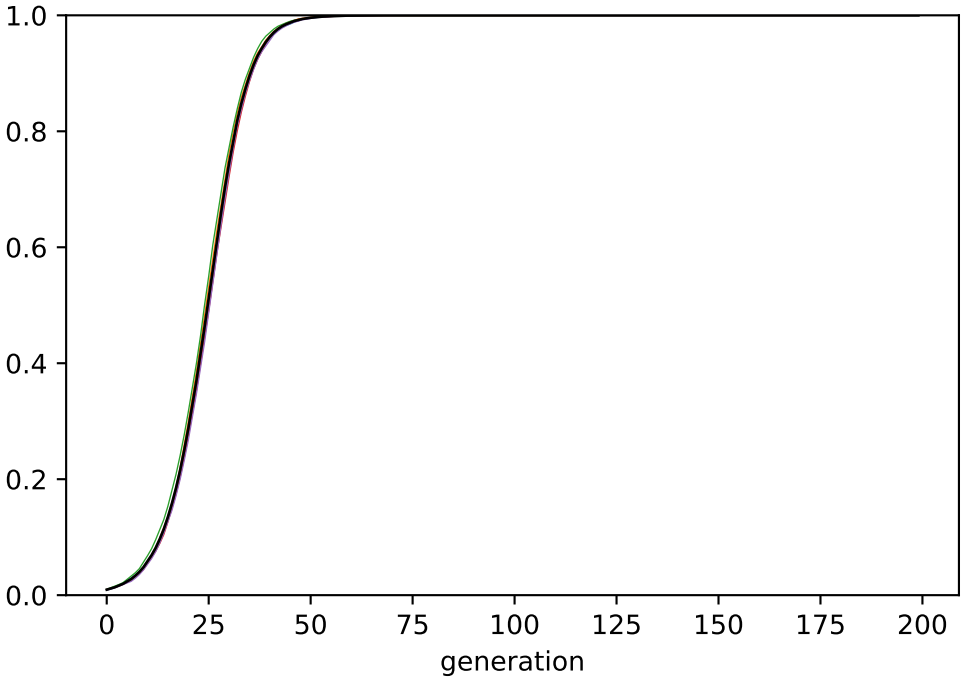
```
data_model = biased_transmission_direct(N=10_000, s_a=.1, s_b=0,
                                        p_0=.01, t_max=200, r_max=5)
plot_multiple_runs(data_model)
```




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




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


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

6 Create first generation

```
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.
```

```
population = pd.DataFrame({"trait": rng.choice(["A", "B"], size=N, replace=True, p=[p_0, 1
```

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

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

```
# Visualize the DataFrame
demonstrators
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn
In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	dem1	dem2	dem3
0	A	A	A
1	A	B	B
2	A	B	A
3	A	B	A
4	A	B	B
5	A	A	A
6	B	A	B
7	B	B	B
8	A	B	A
9	B	B	B
10	A	A	B
11	A	B	A
12	A	B	B
13	A	A	A
14	B	A	A
15	B	A	A
16	A	B	B
17	A	A	B
18	B	A	B
19	A	A	B
20	B	A	A
21	B	A	B
22	A	A	B
23	B	B	A
24	A	B	B
25	B	A	A
26	A	B	A
27	A	A	B
28	B	B	A
29	A	B	B
30	B	B	A
31	A	A	B
32	A	B	A
33	A	A	A
34	B	A	A
35	A	B	B
36	B	A	B
37	B	B	B
38	A	A	B
39	A	B	A
40	B	B	B
41	B	A	A
42	B	A	A
43	A	B	A
44	A	A	B
45	B	A	B
46	A	B	B
47	A	A	B
48	B	B	A
49	B	B	A
50	B	B	B
51	A	A	A

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

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	0
0	3
1	1
2	2
3	2
4	1
5	3
6	1
7	0
8	2
9	0
10	2
11	2
12	1
13	3
14	2
15	2
16	1
17	2
18	1
19	2
20	2
21	1
22	2
23	1
24	1
25	2
26	2
27	2
28	1
29	1
30	1
31	2
32	2
33	3
34	2
35	1
36	1
37	0
38	2
39	2
40	0
41	2
42	2
43	2
44	2
45	1
46	1
47	2
48	1
49	1
50	0
51	3

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

```
# 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
```

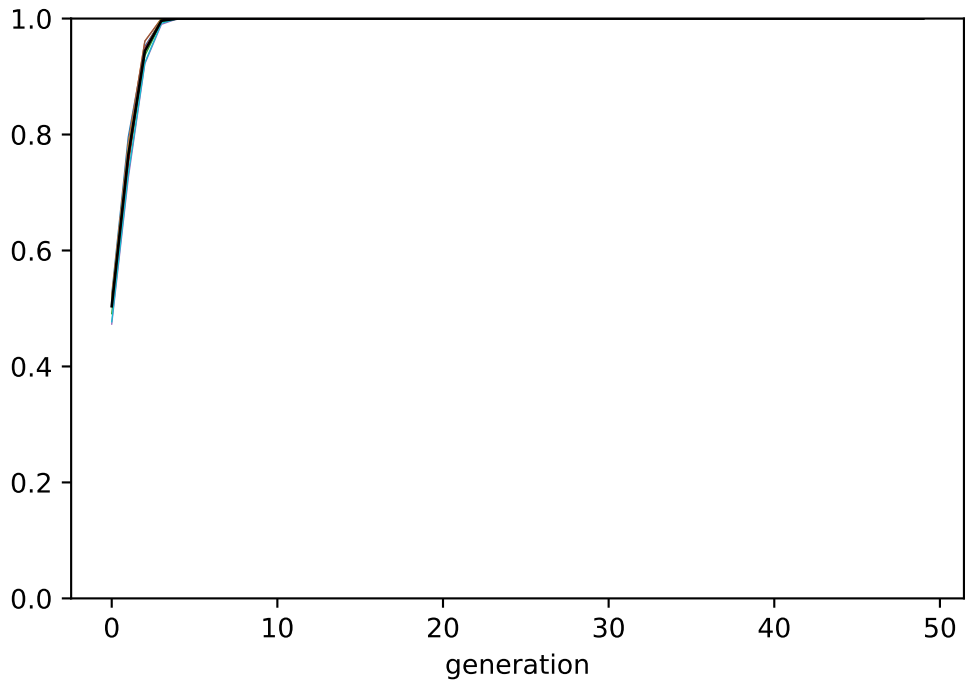
```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	dem1	dem2	dem3	new_trait
0	A	A	A	A
1	A	B	B	B
2	A	B	A	A
3	A	B	A	A
4	A	B	B	B
5	A	A	A	A
6	B	A	B	A
7	B	B	B	B
8	A	B	A	A
9	B	B	B	B
10	A	A	B	A
11	A	B	A	A
12	A	B	B	B
13	A	A	A	A
14	B	A	A	A
15	B	A	A	A
16	A	B	B	B
17	A	A	B	A
18	B	A	B	B
19	A	A	B	A
20	B	A	A	A
21	B	A	B	B
22	A	A	B	A
23	B	B	A	B
24	A	B	B	A
25	B	A	A	A
26	A	B	A	A
27	A	A	B	A
28	B	B	A	B
29	A	B	B	B
30	B	B	A	A
31	A	A	B	A
32	A	B	A	A
33	A	A	A	A
34	B	A	A	A
35	A	B	B	B
36	B	A	B	A
37	B	B	B	B
38	A	A	B	A
39	A	B	A	A
40	B	B	B	B
41	B	A	A	A
42	B	A	A	A
43	A	B	A	A
44	A	A	B	A
45	B	A	B	B
46	A	B	B	B
47	A	A	B	A
48	B	B	A	B
49	B	B	A	B
50	B	B	B	B
51	A	A	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

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

        # 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
            prob_minority = rng.choice([True, False], p=[(1/3 + D/3), 1-(1/3 + D/3)], size

            # 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" ].shap

    return output
```

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



7 Chapter 5 - Biased transmission: demonstrator-based indirect bias

```
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
```

```
/home/fmoss/miniconda3/lib/python3.9/site-packages/IPython/core/formatters.py:343: FutureWarn

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	trait	status
0	B	low
1	B	low
2	A	low
3	B	low
4	A	low
5	A	low
6	B	low
7	B	low
8	A	high
9	B	low
10	B	low
11	B	low
12	A	low
13	B	low
14	A	low
15	B	low
16	B	low
17	A	low
18	A	low
19	A	low
20	B	low
21	B	low
22	B	low
23	B	low
24	B	low
25	A	low
26	B	low
27	B	low
28	B	low
29	B	low
30	A	low
31	A	low
32	B	low
33	A	low
34	B	low
35	B	low
36	B	low
37	A	low
38	B	low
39	B	low
40	B	low
41	A	low
42	B	low
43	B	low
44	A	low
45	A	low
46	B	low
47	B	low
48	B	low
49	B	low
50	B	low
51	A	low

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

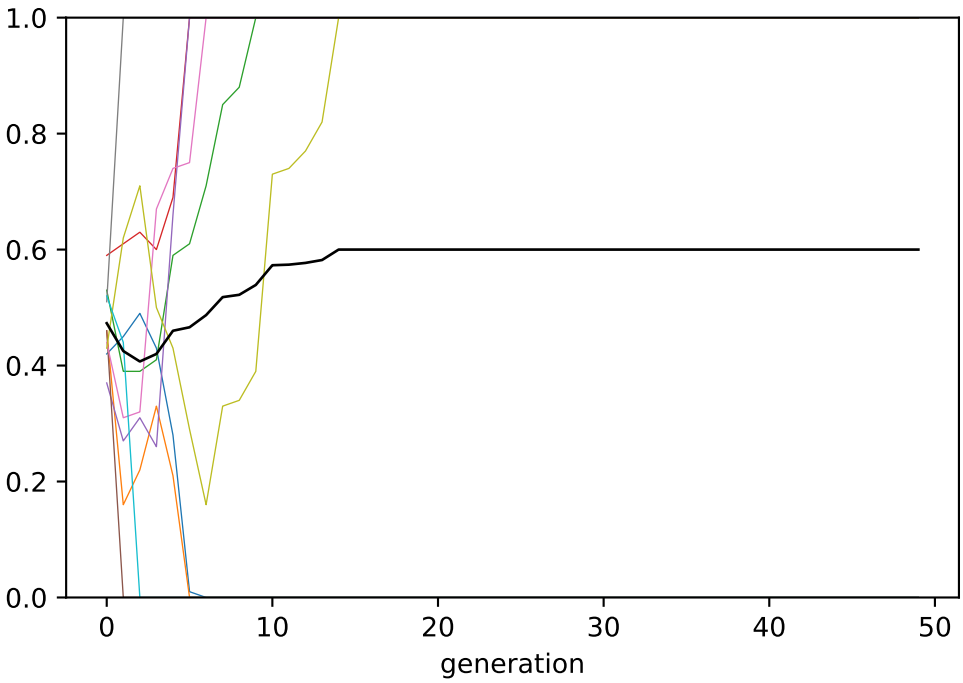
```
if sum(p_demonstrator) > 0:
    ps = p_demonstrator / p_demonstrator.sum()
    demonstrator_index = rng.choice(np.arange(N), size=N, p=ps, replace=True)
    population["trait"] = population.loc[demonstrator_index, "trait"].values
```

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

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

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

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

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

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

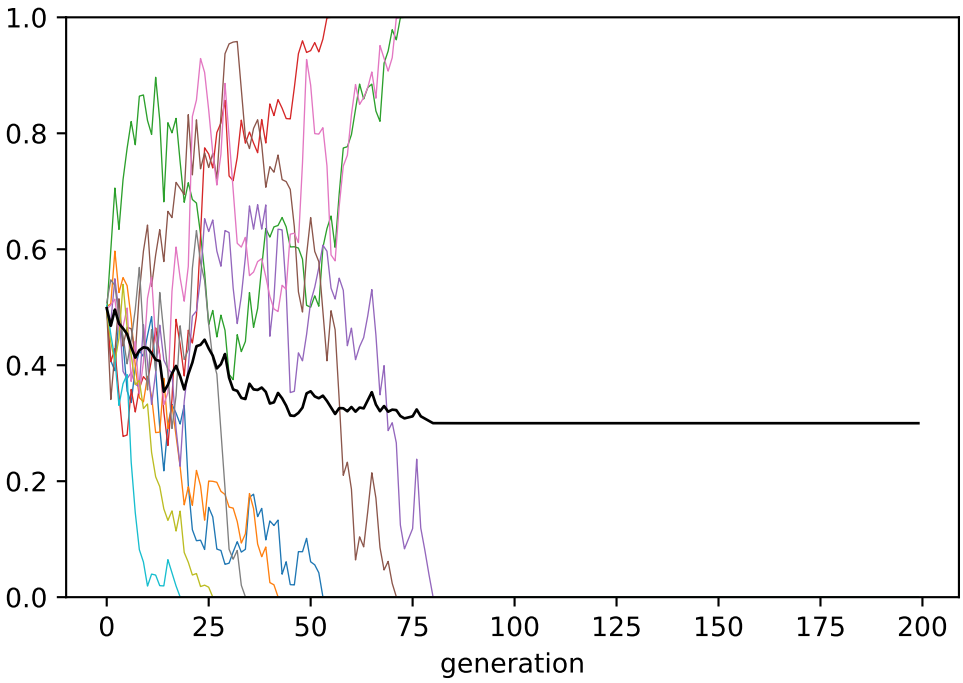
```
                # Get p and put it into output slot for this generation t and run r
                output.loc[r * t_max + t, "p"] = population[ population["trait"] == "A" ].

    return output
```

```
data_model = biased_transmission_demonstrator(N=100, p_s=0.05, p_low=0.0001, p_0=0.5, t_ma
```

```
plot_multiple_runs(data_model)
```




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




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

    ...

    return output
```

```
data_model = biased_transmission_demonstrator_2(N=100, p_s=0.1, p_low=0.0001, p_0=0.5, t_m
```

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.