Language change: parameter exploration

Solution | Agent-based modelling, Konstanz, 2024

Henri Kauhanen

14 May 2024

Setting up

First, we load the module we have already written:

```
include("VariationalLearning.jl")
using .VariationalLearning
```

> **Note**
>
> If VSCode complains about the module, remove the line `module VariationalLearning` and the last `end` statement from the source file `VariationalLearning.jl`. Then load the contents of the file by simply issuing `include("VariationalLearning.jl")` (without the `using` command).

We also need to load Plots to be able to do the visualizations:

```
using Plots
```

And we also need StatsBase for the `mean` function, which we will use to calculate population averages:

```
using StatsBase
```

In the lecture, we noted that, in order to both carry out an interaction between two agents and summarize the population's state over one time step, we had to use a `begin ... end` block. Like this:

```
history = [begin
             interact!(rand(pop), rand(pop))
             average_p(pop)
           end for t in 1:1_000_000]
```

Alternatively, we can wrap the contents of the block in a function, and then call that function. This leads to neater code for the array comprehensions later, so that's the strategy I will follow in this solution. Accordingly, I define:

```
function average_p(x::Array{VariationalLearner})
  return mean([a.p for a in x])
end

function step!(x::Array{VariationalLearner})
  interact!(rand(x), rand(x))
  return average_p(x)
end
```

Our goal in this exercise is to see what varying a number of model parameters does. Since the steps for running the simulation stay the same, it makes sense to define a function that takes the model parameters as arguments and then runs a whole simulation for a desired number of steps, returning the population's history (defined as the sequence of average values of `p` over time). So that we don't get confused over which argument to this function represents which model parameter, I am going to use keyword arguments here. You can read more about them in this week's lecture.

```
function simulate(; N, p, gamma, P1, P2, maxtime)
  # initialize a population
  pop = [VariationalLearner(p, gamma, P1, P2) for i in 1:N]

  # step the population until maxtime
  history = [step!(pop) for t in 1:maxtime]

  # return history
  return history
end
```

I am also going to wrap my plotting code in a custom function, for similar reasons:

```
function my_plot(x)
  plot(1:100:length(x), x[begin:100:end], label=false)
  ylims!(0.0, 1.0)
end
```

```
my_plot (generic function with 1 method)
```

This function is set up so that we plot every 100th time step (otherwise there will be so much stuff to plot that both plotting and displaying the figures will be very slow). Additionally, the function sets the y-axis limits to 0 and 1.

> Tip
>
> Whenever there is something that needs to be repeated a number of times in your code, it is very probably a good idea to wrap that something in a function! The benefits are:
>
> 1. You only need to write the function definition once
> 2. Partly because of the above, you are less likely to make mistakes (compare alternative strategy: you copy-paste a block of code, then later decide that the code needs to be modified somehow, but forget to modify all instances of the same block of code)
> 3. You can use arguments to modulate how the function does its thing
>
> Make a habit of writing functions whenever the opportunity arises.

Unless otherwise noted, I am going to run each simulation for 100,000 steps, and set that with the following global variable:

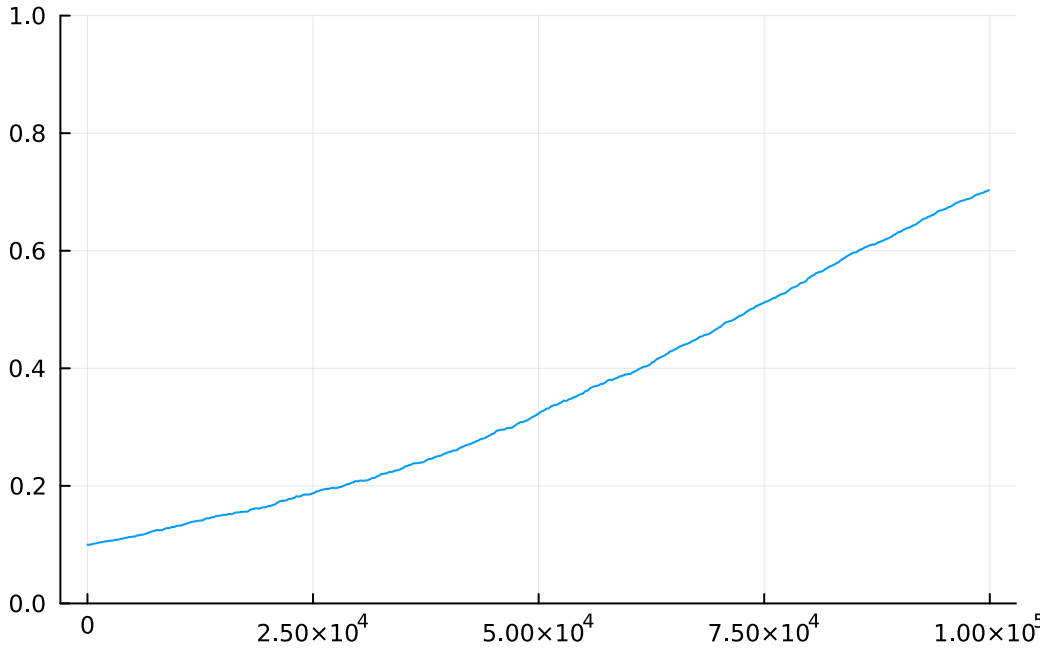
```
maxt = 100_000
```

```
100000
```

Effect of `N` (population size)

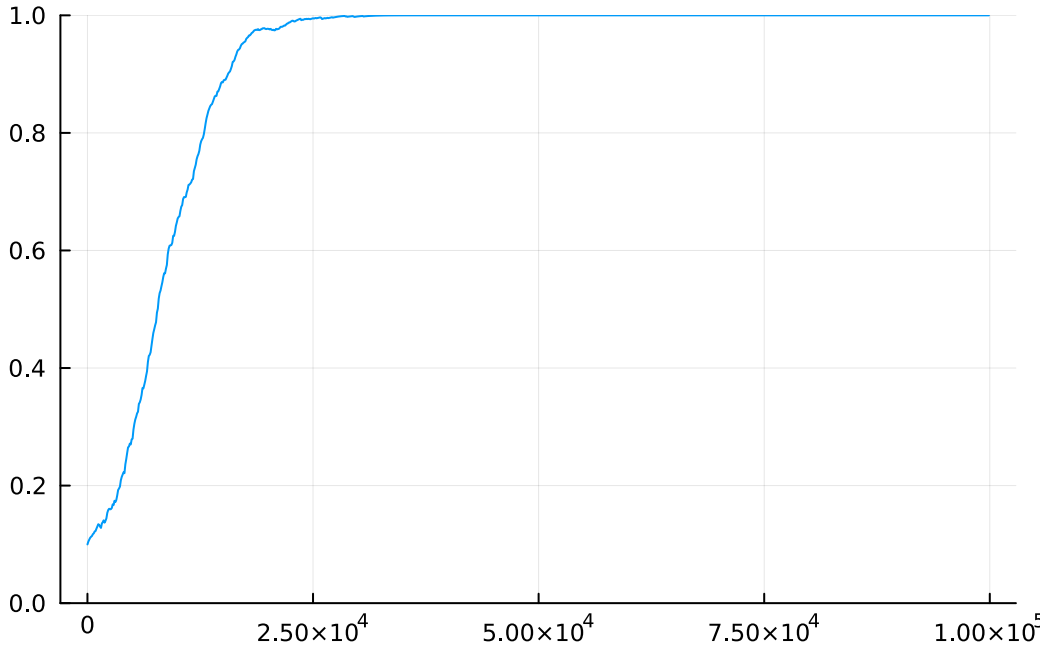
Population of 100 agents:

```
simu1 = simulate(N = 100, p = 0.1, gamma = 0.01, P1 = 0.4, P2 = 0.1, maxtime = maxt)
my_plot(simu1)
```



Population of 10 agents:

```
simu2 = simulate(N = 10, p = 0.1, gamma = 0.01, P1 = 0.4, P2 = 0.1, maxtime = maxt)
my_plot(simu2)
```

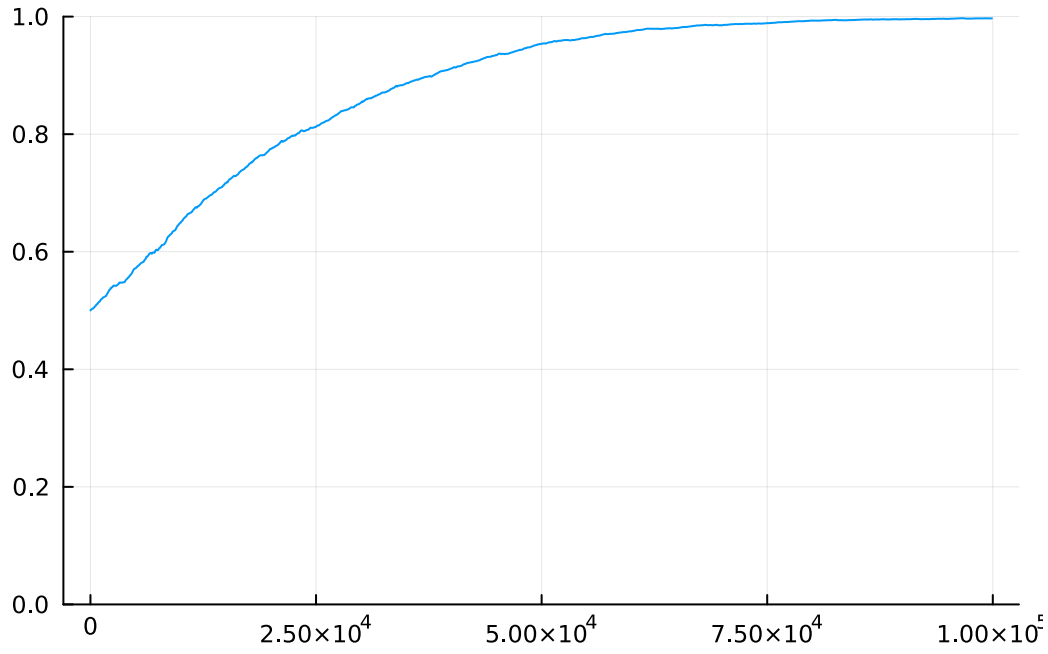


Observation: in both populations, the average of `p` increases along an S-shaped curve. However, in the smaller population, this happens faster.

Effect of `p` (initial value of `p`)

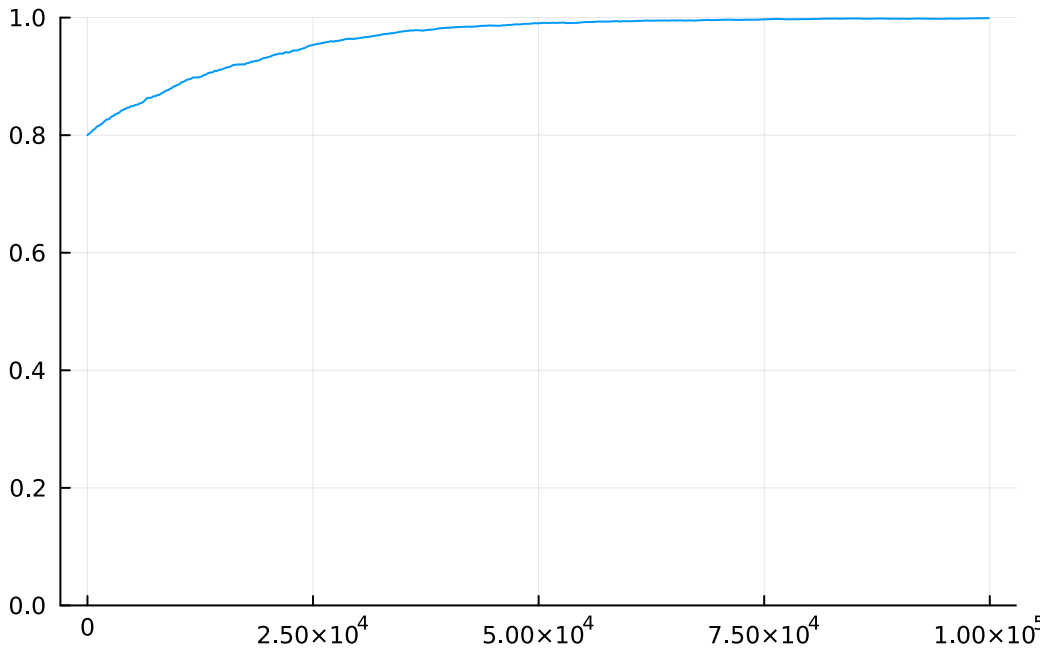
All learners start with $p = 0.5$:

```
simu3 = simulate(N = 50, p = 0.5, gamma = 0.01, P1 = 0.4, P2 = 0.1, maxtime = maxt)
my_plot(simu3)
```



All learners start with $p = 0.8$:

```
simu4 = simulate(N = 50, p = 0.8, gamma = 0.01, P1 = 0.4, P2 = 0.1, maxtime = maxt)
my_plot(simu4)
```

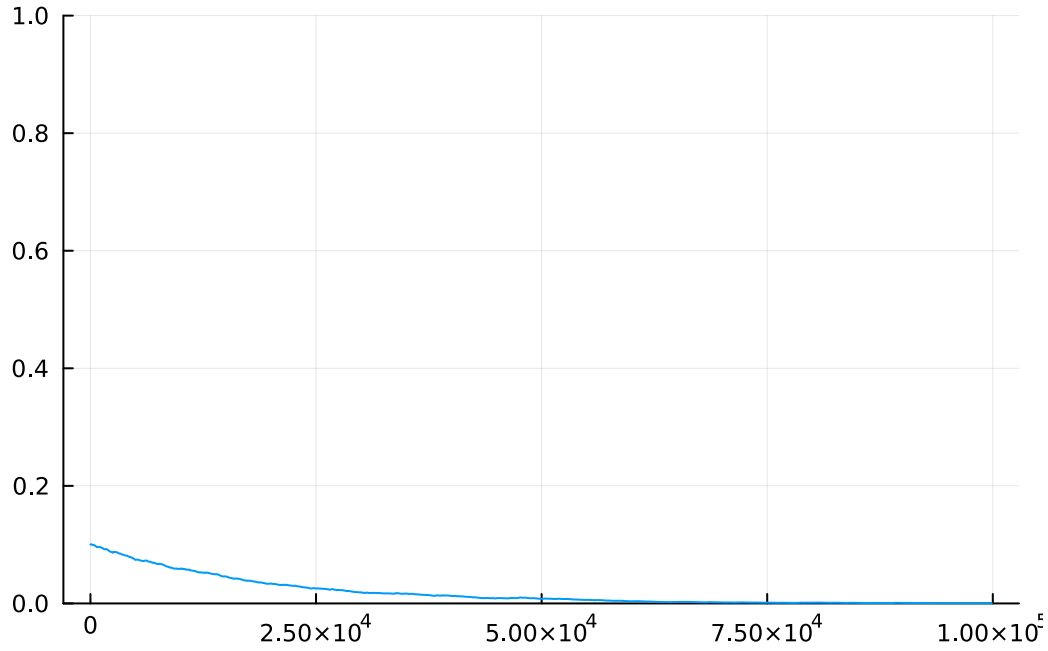


Observation: in both cases, the average of `p` increases over time. This behaviour does not seem dependent on the initial value of `p` in the learners.

Effect of `P1` and `P2` (the amounts of evidence for grammars G_1 and G_2)

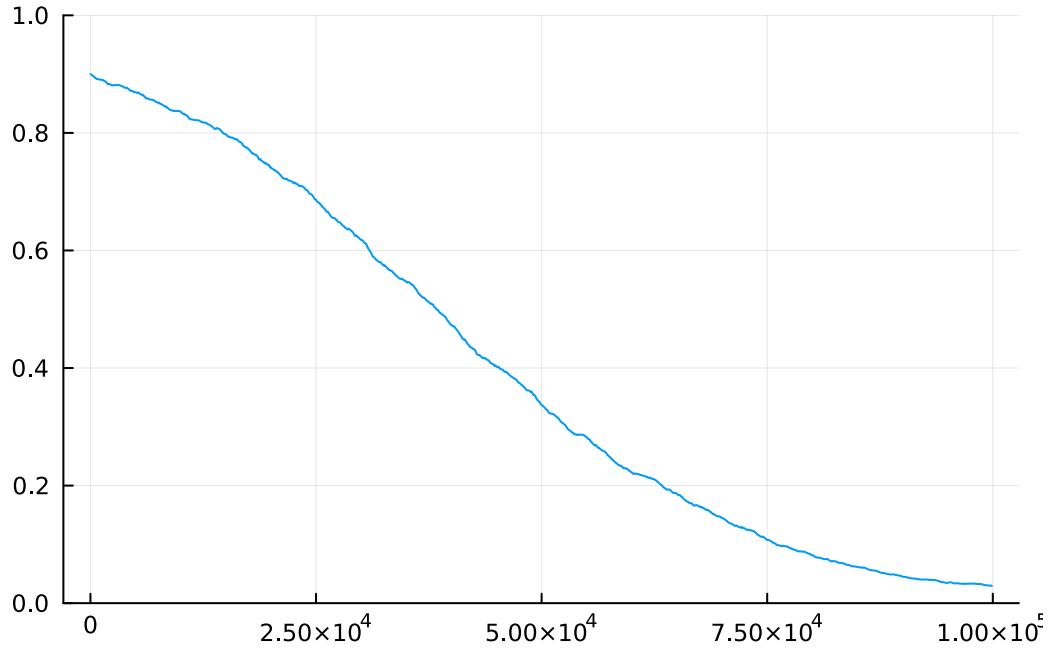
The values of `P1` and `P2` reversed:

```
simu5 = simulate(N = 50, p = 0.1, gamma = 0.01, P1 = 0.1, P2 = 0.4, maxtime = maxt)
my_plot(simu5)
```



Starting from $p = 0.9$:

```
simu6 = simulate(N = 50, p = 0.9, gamma = 0.01, P1 = 0.1, P2 = 0.4, maxtime = maxt)
my_plot(simu6)
```



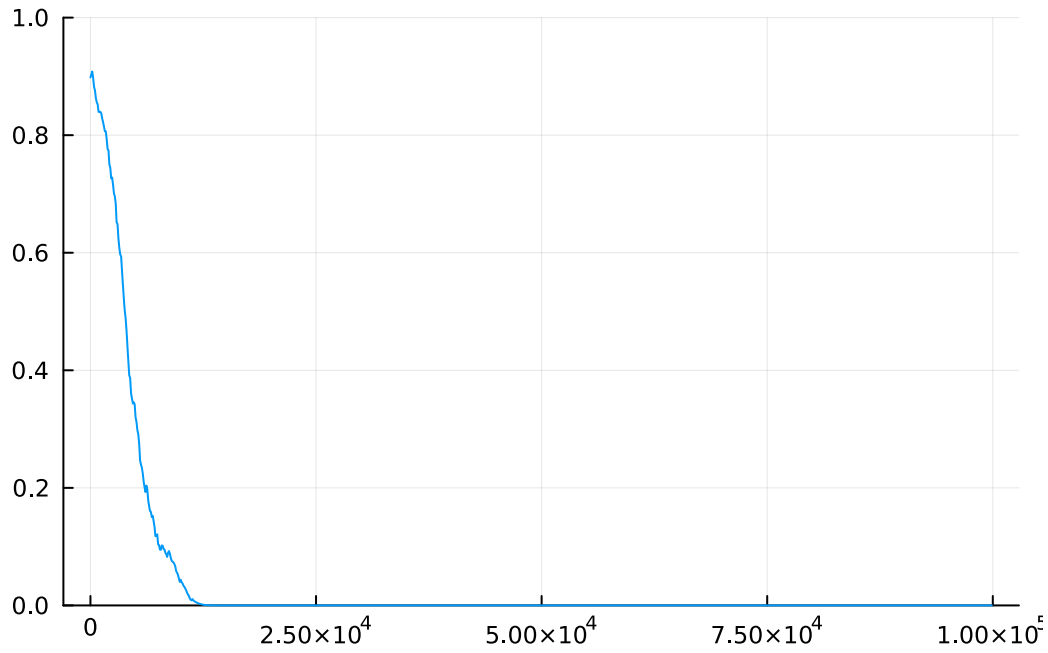
Observation: average p decreases. This makes sense, because now G_2 has more evidence for it than G_1 (`P2 = 0.4` and `P1 = 0.1`).

Effect of γ (learning rate)

Everybody with equal γ

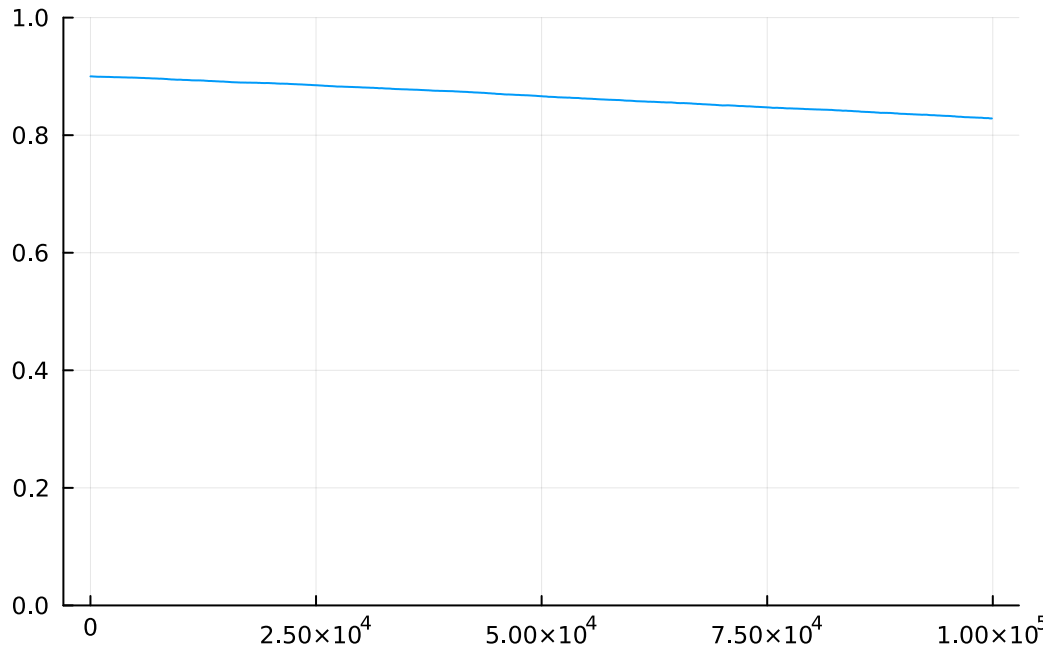
Higher value of γ:

```
simu7 = simulate(N = 50, p = 0.9, gamma = 0.1, P1 = 0.1, P2 = 0.4, maxtime = maxt)
my_plot(simu7)
```



Lower value:

```
simu8 = simulate(N = 50, p = 0.9, gamma = 0.001, P1 = 0.1, P2 = 0.4, maxtime = maxt)
my_plot(simu8)
```



Observation: higher γs lead to faster change.

Random γ

New simulation function:

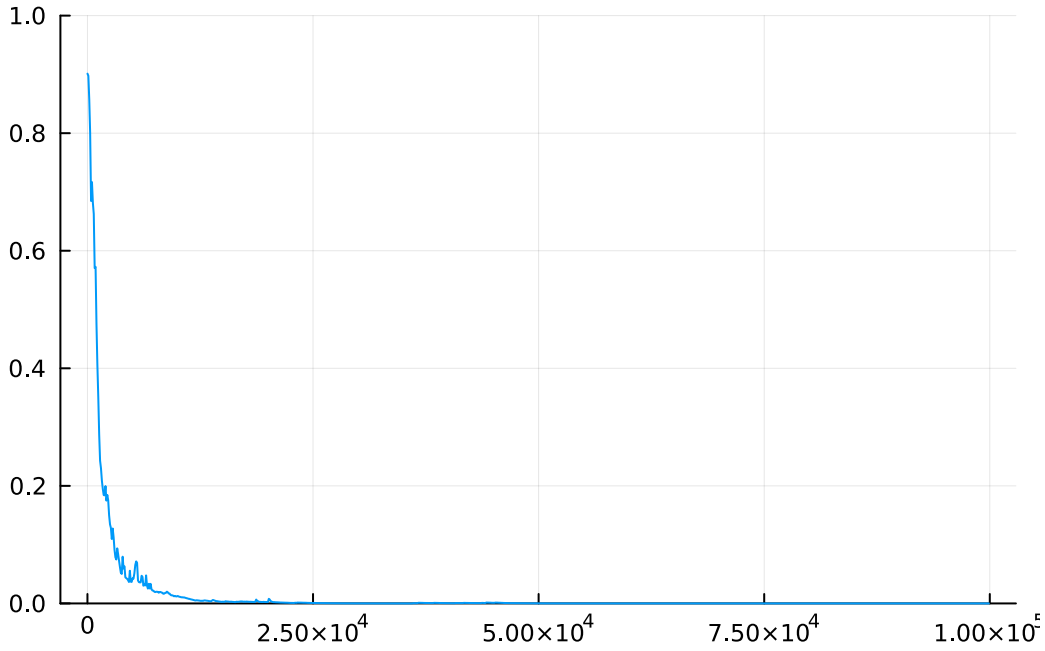
```
function simulate(; N, p, P1, P2, maxtime)
  # initialize a population
  pop = [VariationalLearner(p, rand(), P1, P2) for i in 1:N]

  # step the population until maxtime
  history = [step!(pop) for t in 1:maxtime]

  # return history
  return history
end
```

Try it:

```
simu9 = simulate(N = 50, p = 0.9, P1 = 0.1, P2 = 0.4, maxtime = maxt)
my_plot(simu9)
```



Change is again faster, with some added noise (possibly?).