Introduction to Simulation: Complex social dynamics in a few lines of code

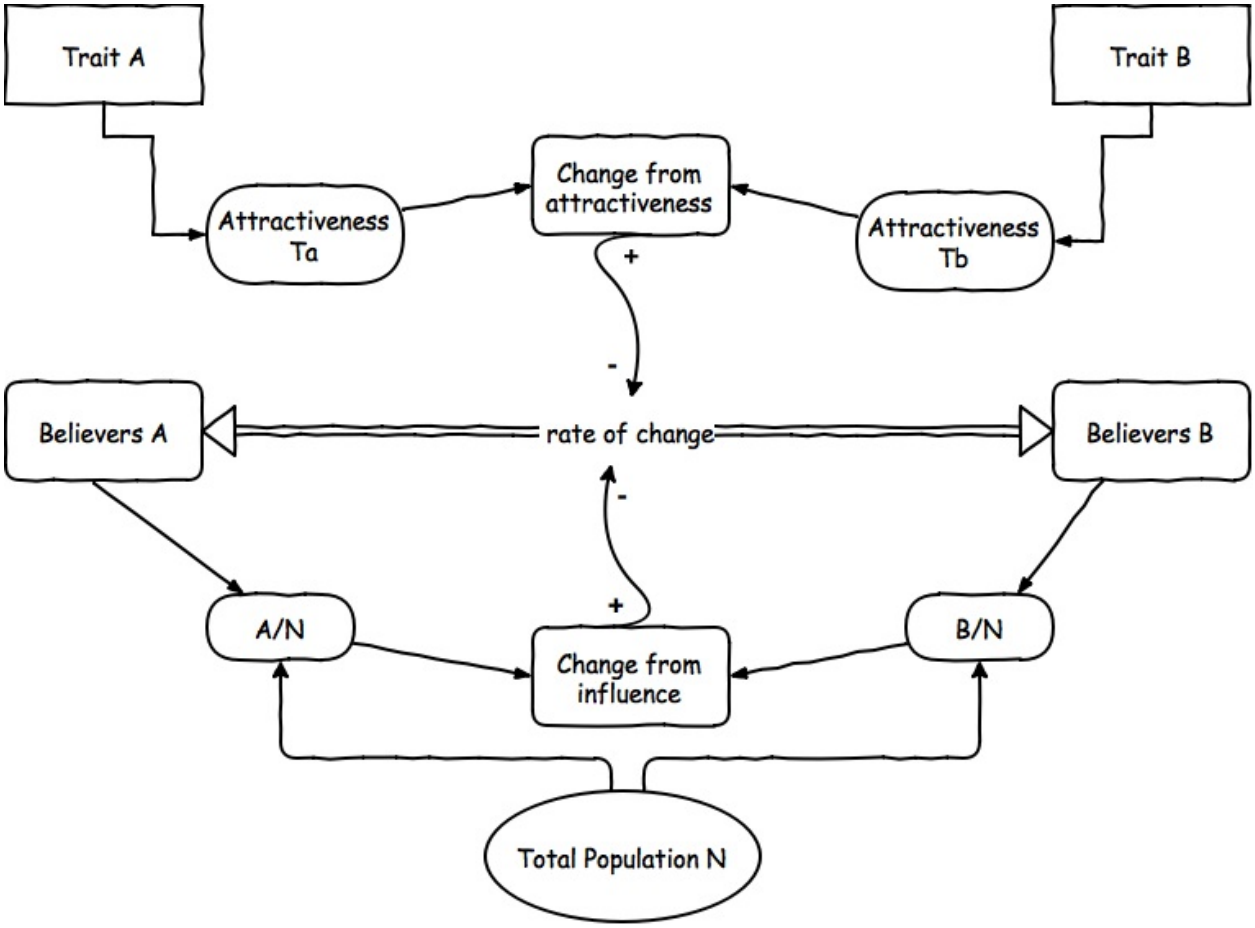
We will create a model depicting competition between two cultural traits within a common population. This is a typical cultural dynamics scenario where individuals must adopt one option amongst two or more **mutually exclusive** options (i.e. religion, elections, football teams, ...). In this case we are interested in situations when you have to choose one option (e.g., you cannot practice two religions), but more complex versions with individuals adopting more than one trait can easily be developed.

Individuals can change their choice over time. The decision is based on the payoff of each trait. This payoff is a measure of the relative interest of the trait, based on:

a) how many people exhibits the trait and

b) the attractiveness of the trait.

An example of this dynamic could be a competition between two different religions. The number of people practicing a belief makes this belief more appealing. However, some beliefs could be intrinsically more interesting for some individuals so part of the population could adopt them even if they are a minority. Finally, social norms are not static so the attractiveness of specific beliefs can vary over time.



Time is divided in discrete steps starting at t = 0.

At each step t the two populations A_t and B_t are updated as individuals move from A to B and from B to A. Take intoa ccout that the value here can be negative meaning that more people move from B to A.

$$A_{t+1} = A_t + \Delta AB$$

$$B_{t+1} = B_t - \Delta AB$$

Let's see how we can express it in code.

First, we need to define the number of individuals in the population. Say we want to start with 100 people. The text after a hash symbol is just a comment so you can skip it for now, but in general it is a good idea to keep documenting the code as you write.

In [1]:

```
N = 100   # total population size
```

Then, we decide on how many believers of each cultural option (religion) we want to start with.

In [2]:

```
A = 65    # initial number of believers A
B = N - A # initial number of believers A
```

Finally, we want to update these quantities at every time step depending on some variation:

In [3]:

```
t = 0
MAX_TIME = 100
while t < MAX_TIME:
    A = A + variation
    B = B - variation

    # advance time to next iteration
    t = t + 1
```

Type the code in the code tab. Keep in mind that indents are important.

Run the code by hitting F5 (or click on a green triangle). What happens?

Well, nothing happens or, rather, we get an error.

```
NameError: name variation is not defined
```

Indeed, we have not defined what do we mean by 'variation'. So let's calculate it based on the population switching trait based on a comparison between payoffs. For example if B has higher payoff then A then we should get something like this: \begin{equation} \Delta{A\to B} = A \cdot (payoff{A\to B} - payoff_{B\to A}) \end{equation}

So the proportion of population A that switches to B is proportional to the difference between payoffs. As we mentioned the payoff of a trait is determined by the population exhibiting the competing trait as well as its intrinsic attractiveness.

To define the payoff we need to implement the following competition equations:

$$Payoff_{B\to A} = \frac{A_t}{N}\frac{T_A}{(T_A + T_B)}$$

$$Payoff_{A\to B} = \frac{B_t}{N}\frac{T_B}{(T_A + T_B)}$$

Let's look at the equations a bit more closely. The first term is the proportion of the entire population N holding a particular cultural trait ($\frac{A_t}{N}$ for A and $\frac{B_t}{N}$ for B). While the second element of the equations is the balance between the attractiveness of both ideas (T_A and T_B) expressed as the attractiveness of the given trait in respect to the total 'available' attractiveness ($T_A + T_B$).

You have probably immediately noticed that these two equations are the same in structure and only differ in terms of what is put into them. Therefore, to avoid unnecessary hassle we will create a 'universal' function that can be used for both. Type the code below at the beginning of your script:

In [4]:

```
def payoff(believers, Tx,Ty):
    proportionBelievers = believers/N
    attraction = Tx/(Ty + Tx)
    return proportionBelievers * attraction
```

Let's break it down a little. First we define the function and give it the input - the number of believers and the two values that define how attractive each cultural option is.

```
def payoff(believers, Tx, Ty):
```

Then we calculate two values:

1. percentage of population sharing this cultural option

```
proportionBelievers = believers/N
```

2. how attractive the option is

```
attraction = Tx/(Ty + Tx)
```

Look at the equations above, the first element is just the $\frac{A_t}{N}$ and $\frac{B_t}{N}$ part and the second is this bit: $\frac{T_A}{(T_A+T_B)}$. Finally we return the results of the calculations.

```
return proportionBelievers * attraction
```

Voila! We have implemented the equation into Python code. Now, let's modify the main loop to call the function - we need to do it twice to get the payoff for changing from A to B and from B to A. This is repeated during each iteration of the loop, so each time we can pass different values into it. To get the payoffs for switching from A to B and from B to A we have to add the calls to 'payoff' at the beginning of our loop:

In [5]:

```
while t < MAX_TIME:

    variationBA = payoff(A, Ta, Tb)
    variationAB = payoff(B, Tb, Ta)

    A = A + variation
    B = B - variation

    # advance time to next iteration
    t = t + 1
```

That is, we pass the number of believers and the attractiveness of the traits. The order in which we pass the input variables into a function is important. In the B to A transmission, B becomes 'believers', while Ta becomes 'Tx' and Tb becomes 'Ty'. In the second line showing the A to B transmission, A becomes '_believers', Tb becomes 'Tx' and Ta becomes 'Ty'.

The obvious problem after this change is that we have not defined the 'attractiveness' of each trait. To do so add their definitions at the beginning of the script around other definitions (N, A, B, MAX_TIME, etc).

In [6]:

```
Ta = 1.0            # initial attractiveness of option A
Tb = 2.0            # initial attractiveness of option B
alpha = 0.1         # strength of the transmission process
```

We can now calculate the difference between the perceived payoffs during this time step. To do so, we need to first see which one did better (A or B).

In [7]:

```
    difference = variationBA - variationAB
```

Now, if the difference between the two is negative then we know that during time step B is more interesting than A. On the contrary, if it is positive then A seems better than B. What is left is to see how many people moved based on this difference between payoffs. We can express it in the main while loop, like this:

In [8]:

```
    # B -> A
    if difference > 0:
        variation = difference*B
    # A -> B
    else:
        variation = difference*A
    # update the population
    A = A + variation
    B = B - variation
```

We can use an additional term to have a control over how strong the transmission from A to B and back is - we will call it alpha (α). This parameter will multiply change before we modify populations A and B:

In [9]:

```
    variation = alpha*variation
```

And we need to add it to the rest of the parameters of our model:

In [10]:

```
# temporal dimension
MAX_TIME = 100
t = 0               # initial time

# init populations
N = 100          # population size
A = 65           # initial population of believers A
B = N-A             # initial population of believers B

# additional params
Ta = 1.0            # initial attractiveness of option A
Tb = 2.0            # initial attractiveness of option B
alpha = 0.1         # strength of the transmission process
```

You should bear in mind that the main loop code should be located after the definition of the transmission function and the initialization of variables (because they are used here). After all the edits you have done it should look like this:

In [11]:

```
while t < MAX_TIME:
    # calculate the payoff for change of believers A and B in the current time step
    variationBA = payoff(A, Ta, Tb)
    variationAB = payoff(B, Tb, Ta)
    difference = variationBA - variationAB

    # B -> A
    if difference > 0:
        variation = difference*B
    # A -> B
    else:
        variation = difference*A

    # control the pace of change with alpha
    variation = alpha*variation

    # update the population
    A = A + variation
    B = B - variation

    # advance time to next iteration
    t = t + 1
```

OK, we have all the elements ready now and if you run the code the computer will churn all the numbers somewhere in the background. However, it produces no output so we have no idea what is actually happening. Let's solve this by visualising the flow of believers from one option to another. First, we will create two empty lists. Second, we will add there the initial populations. Then, at each timestep, we will add the current number of believers to these lists and finally, at the end of the simulation run we will plot them to see how they changed over time. Start with creating two empty lists. Add the following code right after all the variable definitions at the beginning of the code before the while loop:

In [12]:

```
# initialise the list used for plotting
believersA = []
believersB = []

# add the initial populations
believersA.append(A)
believersB.append(B)
```

The whole initialisation/definition block at the beginning of your code should look like this:

In [13]:

```
# initialisation
MAX_TIME = 100
t = 0               # initial time
N = 100           # population size
A = 65              # initial proportion of believers A
B = N-A             # initial proportion of believers B

Ta = 1.0            # initial attractiveness of option A
Tb = 2.0            # initial attractiveness of option B
alpha = 0.1         # strength of the transmission process

# initialise the list used for plotting
believersA = []
believersB = []

# add the initial populations
believersA.append(A)
believersB.append(B)
```

We just added the initial number of believers to their respective lists. However, we also need to do this at the end of each time step. Add the following code at the end of the while loop - remember to align the indents with the previous line!

In [14]:

```
    believersA.append(A)
    believersB.append(B)
```

The whole while-loop block should now look like this:

In [15]:

```
while t < MAX_TIME:
    # calculate the payoff for change of believers A and B in the current time step
    variationBA = payoff(A, Ta, Tb)
    variationAB = payoff(B, Tb, Ta)
    difference = variationBA - variationAB

        # B -> A
    if difference > 0:
        variation = difference*B
    # A -> B
    else:
        variation = difference*A

    # control the pace of change with alpha
    variation = alpha*variation

    # update the population
    A = A + variation
    B = B - variation

    # save the values to a list for plotting
    believersA.append(A)
    believersB.append(B)

    # advance time to next iteration
    t = t + 1
```

Finally, let's plot the results. First, we will import Python's plotting library, Matplotlib and use a predifined plotting style. Add these two lines at the beginning of your script:

In [16]:

```
import matplotlib.pyplot as plt      # plotting library
plt.style.use('ggplot')              # makes the graphs look pretty
```

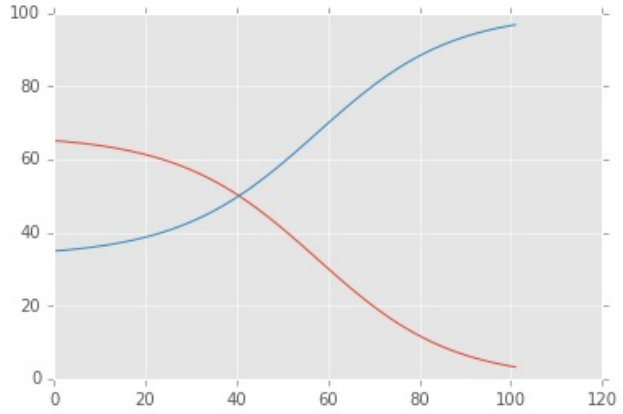
Finally, let's plot! Plotting in Python is as easy as saying 'please plot this data for me'. Type these two lines at the very end of your code. We only want to plot the results once the simulation has finished so make sure this are not inside the while loop - that is, ensure this block of code is not indented. Run the code!

In [17]:

```
# plot the results
plt.plot(believersA)
plt.plot(believersB)
```

Out[17]:

```
[<matplotlib.lines.Line2D at 0x7f0deb281908>]
```



You can see how over time one set of believers increases while the other decreases. This is not particularly surprising as the attractiveness Tb is higher than Ta. However, can you imagine a configuration where this is not enough to sway the population? Have a go at setting the variables to different initial values to see what combination can counteract this pattern.

1. set the initial value of A to 5, 10, 25, 50, 75
2. set the MAX_TIME to 1000,
3. set the Ta and Tb to 1.0, 10.0, 0.1, 0.01, etc.,
4. set alpha to 0.01, and 1.0

You can try all sorts of configurations to see how quickly the population shifts from one option to another or what are the minimum values of each variable that prevent it.

However, we can make the model more interesting if we allow the attractiveness of each option to change through time. To do so let's define a new function. Add the following line at the beginning of the while loop (remember indentation!).

In [18]:

```
    Ta, Tb = attractiveness(Ta, Tb)
```

Ta and Tb will then be modified based on some dynamics we want to model. Let's define the 'attractiveness' function. We have already done it once for the 'payoff' function so it should be a piece of cake. At each time step we will slightly modify the attractiveness of each trait using a kernel K that we will define. This can be expressed as:

$$T_{A,t+1} = T_A + K_a$$

$$T_{B,t+1} = T_B + K_b$$

K can have several shapes such as:

- Fixed traits with $K_a = K_b = 0$
- A gaussian stochastic process such as $K = N(0, 1)$
- A combination (e.g., $K_a = N(0, 1)$ and $K_b = 1/3$)

Let's start with a simple case scenario, such as a) T_a will increase each step by a fixed K_a and b) K_b is equal to zero (so T_b will be fixed over the whole simulation).

In [19]:

```
def attractiveness(Ta, Tb):

    Ka = 0.01
    Kb = 0

    Ta = Ta + Ka
    Tb = Tb + Kb
    return Ta, Tb
```

First, we define the function and give it the input values.

```
def attractiveness(Ta, Tb):
```

Then, we establish how much the attractiveness of each trait changes (i.e., define K_a and K_b)

```
Ka = 0.01
Kb = 0
```

And plug them into the equations:

```
Ta = Ta + Ka
Tb = Tb + Kb
```

Finally, we return the new values:

```
return Ta, Tb
```

This is how the function is defined. The main loop will now look like this:

In [20]:

```
while t < MAX_TIME:
    # update attractiveness
    Ta, Tb = attractiveness(Ta, Tb)
    # calculate the payoff for change of believers A and B in the current time step
    variationBA = payoff(A, Ta, Tb)
    variationAB = payoff(B, Tb, Ta)
    difference = variationBA - variationAB

        # B -> A
    if difference > 0:
        variation = difference*B
    # A -> B
    else:
        variation = difference*A

    # control the pace of change with alpha
    variation = alpha*variation

    # update the population
    A = A + variation
    B = B - variation

    # save the values to a list for plotting
    believersA.append(A)
    believersB.append(B)

    # advance time to next iteration
    t = t + 1
```

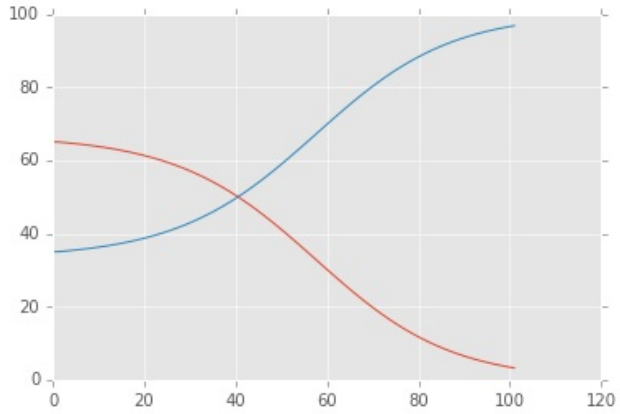
If you plot this you will get a different result than the previous:

In [21]:

```
plt.plot(believersA)
plt.plot(believersB)
```

Out[21]:

```
[<matplotlib.lines.Line2D at 0x7f0deb1c12e8>]
```



Have a go at changing the values of K_a and K_b and see what happens. Can you see any equilibrium where both traits coexist?

There are a number of functions we can use to dynamically change the 'attractiveness' of each trait. Try the following ones:

In [22]:

```
import numpy as np # stick this line at the beginning of the script alongside other 'imports'


def attractiveness2(Ta, Tb):
    # temporal autocorrelation with stochasticity (normal distribution)
    # we get 2 samples from a normal distribution N(0,1)
    Ka, Kb = np.random.normal(0, 1, 2)
    # compute the difference between Ks
    diff = Ka-Kb
    # apply difference of Ks to attractiveness
    Ta += diff
    Tb -= diff
    return Ta, Tb

def attractiveness3(Ta, Tb):
    #  anti-conformism dynamics (more population means less attractiveness)

    # both values initialized at 0
    Ka = 0
    Kb = 0

    # first we sample gamma with mean=last popSize of A times relevance
    diffPop = np.random.gamma(believersA[t])
    # we sustract from this value the same computation for population B
    diffPop = diffPop - np.random.gamma(believersB[t])

    # if B is larger then we need to increase the attractiveness of A
    if diffPop < 0:
        Ka = -diffPop
    # else A is larger and we need to increase the attractiveness of B
    else:
        Kb = diffPop

    # change current values
    Ta = Ta + Ka
    Tb = Tb + Kb

    return Ta, Tb
```

Use the functions above (just change 'attractiveness' to, e.g., 'attractiveness2' in the main code) to see what happens when we add small dynamic variations to the process. What happens when the initial conditions are changed? What if we look at a much longer time scale?