**IBM FOR BACTERIAL COMMUNTY BREEDING**

(MITRI LAB, DMF UNIL)

By Björn Vessman and Pablo Guridi Fernández

Contents

[Scheme of the code of my main model 2](#_Toc80914960)

[Important data structures and variables (reinitialised at each round) 3](#_Toc80914961)

[Main data structures for the model to work 3](#_Toc80914962)

[Other important variables (used in disassembly) 3](#_Toc80914963)

[Relevant list within a round 3](#_Toc80914964)

[Relevant dictionaries within a round only filled in if “weplothere” 3](#_Toc80914965)

[Abbreviations when looping (might not always apply) 4](#_Toc80914966)

[Use of random seeds 4](#_Toc80914967)

[How results will be stored 5](#_Toc80914968)

[General results of each repeat 5](#_Toc80914969)

[Include the following dataframes saved as csv: 5](#_Toc80914970)

[Also include some variables in pickle format at some rounds: 7](#_Toc80914971)

[Results within a round (“per\_round”) only if “weplothere”: 7](#_Toc80914972)

[How to run the model……………………………………………………………………………………………………………………7](#_Toc80914968)

[How to run the model in the cluster: 8](#_Toc80914974)

[How I modify the main model for the cluster 8](#_Toc80914976)

[Bash script to call the python script in the cluster(fb.sh) 8](#_Toc80914977)

[How I really run in the cluster 9](#_Toc80914978)

CODE STRUCTURE (“#%%” used just to define sections in spyder IDE)

Import packages

# Scheme of the code of my main model

Chose parameters and conditions, set random seed generator (rng(seed))

Create functions (generate species, mutate, calculate deg score, count items in the grid…)

Generate species and its order for all the repeats

For each propagation method (if we run more than one in the same script):

For each repeat: #For more detail of how the model works see the algorithm in overleaf (paper)

Reset random generator 🡪 rng(seed)

For each round:

For each time step:

For each tube:

For each population (=strain):

Consume nutrients, degrade toxins, activate

For each population (in the original length):

Replicate

For each population (in reverse order):

Die

Shuffle grid

Save things and make plots within a round (if “weplothere”)

PROPAGATION

No selection (“n”)

Propagule (“p”): traditional one

Propagule invasion (“pi”) : any species can immigrate, 1 immigrant per tube

Propagule (“pip”) : any species can immigrate, 1 +poisson(0.5) immigrants/tube

Propagule (“pi2”) : only species not pressent in the tube /metacomm immigrate, 1 imm/tube immigrants/tube

Migrant pool (“m”): traditional one

Mp invasion (“mi”) : any species can immigrate, 1 immigrant per tube

Mp (“mip”) : any species can immigrate, 1 +poisson(0.5) immigrants/tube

Mp (“mi2”) : only species not pressent in the tube immigrate, 1 imm/tube immigrants/tube

Disassembly (“d”): traditional one

Disassembly2(“d2):decoupling emigration and immigration: 1 + Poisson(0.5)

Disassembly 2 old (“d2\_old”) : same as above but first immigrate then emigrate

Save information over rounds (data frames, grid at the beginning of a round, dicts…)

# Important data structures and variables (reinitialised at each round)

**“weplothere”**🡪 When true, we plot and store some results within rounds (indicated in red in the text below). By default, I will save rounds 0-4 and the last one, the periodicity in between can be chosen with the variable “**save\_periodicity**”. This helps a bit with efficiency and also, mainly helps with reducing the amount of data to save.

Say that it helps for efficiency (although more in the previous version of the model), it also saves to reduce the amount of info we store (like save plots within rounds and the grid only at some rounds).

## Main data structures for the model to work

* **“grid”**: list of lists 🡪 [[tube1],[tube2],…] at the same time if we zoom in tube1 = [{strain1}{strain2},…] #this is the list where cells/strains are stored
* **“nuts”**: list of lists 🡪 each list are the nutrients corresponding to the tube (or community) in the same position in “grid”
* **“toxs”:** list of lists 🡪 each list are the toxins corresponding to the tube in the same position in “grid”

## Other important variables (used in disassembly)

* **“st\_repo”** 🡪 dictionary, each key corresponds to one species, and the corresponding value is a list with the last version of strains of that species. This last version of species is only updated in disassembly, but in the
* **“sp\_in\_play”** 🡪 list of all the species present in a metacommunity, if one species is present in many tubes, would appear as many times in “sp\_in\_play”, so usually I use set(sp\_in\_play) just to know all the species that are present in this particular repeat. This is not really relevant because right now we impose that the 15 species need to be present in the metacommunity, but could matter if we change that condition (so in disassembly we don’t reintroduce species not present at the beginning of the repeat)
* **“sp\_we\_have”🡪** equivalent to “sp\_in\_play” but just including the species present in the metacommunity of selected tubes in a particular round.

## Relevant list within a round

* **“pop\_0”** 🡪 list of dictionaries. They store population of each tube at the beginning of the round (one dictionary for each tube, species as keys, populations as values)
* **“sp\_tube”** 🡪 list of lists. They store species in each tube at the beginning of the round (tm=0), species are ordered by numerical order of species.
* **“deg\_list”** 🡪 store degradation score of each community at the of the round (tm=end).
* **“nut\_list”** 🡪 store nutrient consumption score of each community at the end of the round
* **“auc\_tot”** 🡪 list of lists. Each list includes total population of each tube (all species together) **at each time step** (this is different from the 4 lists above that only have one value at the first or last time step)**.** This would not be needed if we have “dict\_grid” defined below, as more detailed information is stored there, but “dict\_grid” is only stored if “weplothere”.

## Relevant dictionaries within a round only filled in if “weplothere”

* **“dict\_nuts”** 🡪 each tube is a key, its value is a list of lists each corresponding to one of the nutrients (in order, 0:3). A list indicates the amount of each nutrient at each time step.
* **“dict\_toxs”** 🡪 each tube is a key, its value is a list of lists each corresponding to one of the toxins (in order, 0:3). A list indicates the amount of each toxin at each time step.
* **“dict\_grid”** 🡪 each tube is a key, its value is a list of lists each of them corresponds to one of the species initially present in the tube. The lists are always ordered by numerical order of species identity 1🡪 15 (which is the same order in which they appear in “sp\_tube” below).

# Abbreviations when looping (might not always apply)

* tm = time step
* rd = round
* rp = repeat
* cell = every item inside a community, it used to correspond to a cell in previous versions of the model, but right now it is not just a cell anymore but a population (all the cells of a particular strain).
* c = index corresponding to “cell”
* tube = bacterial community (really the data structure with its content)
* t = index corresponding to the tube
* j = nutrient
* k = toxin (sometimes it also has a no related use referring to a key of a dictionary(“k” would be the key and “v” the value))

# Use of random seeds

For one propagation:

Seed = 22 #the seed we add serves to generate the species and each combination

**rng = np.random.default\_rng(seed)**

🡪Generate species

🡪Generate species combinations for all the repeats

For repeat in range(N\_repeats):

**rng = np.random.default\_rng(seed)** # reinitialise the rng seed for each repeat

For each round:

For each time step:

Consume, degrade, divide, die.

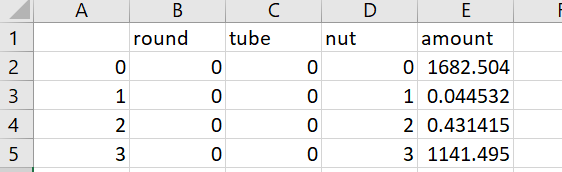
# How results will be stored

* At the beginning we should specify the path where to store the results **(“enter path”).** Within that folder the script will create:
  + Folder named as the species seed (just the number e.g. “22”)
    - Folder for the propagation (“d\_s”, “m\_s”, “p\_s”, “d\_r”, “d\_s” , and so on, see the abbreviations used for each propagation in the first page of these document. After the letter corresponding to the method we add “\_s” for selection treatment and “\_r” for random control. I call “n\_r” to the random control”)
      * INDEX\_REP: pickle file containing the combination of species per tube at the beginning of each repeat.
      * INFO.txt: file containing the information of the initial parameter values we chose for this particular run.
      * Species (just the species in pickle format or as txt), they are the same for each seed, so to have them for each method is a bit redundant, at the beginning was a bit more to check that everything is fine.
      * “timer.txt” indicating the time each repeat take, in seconds, it id cumulative so if we take timer\_lastrep.txt, we know the total time which might be useful to run in the cluster
      * Folder for the results of each repeat , inside
        + **Results within a round**
        + **General results of this repeat**

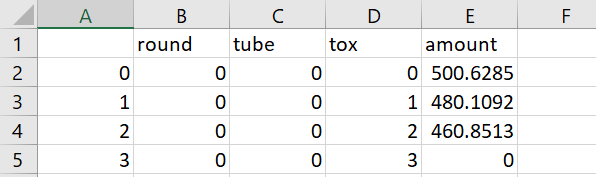
## General results of each repeat

### Include the following dataframes saved as csv:

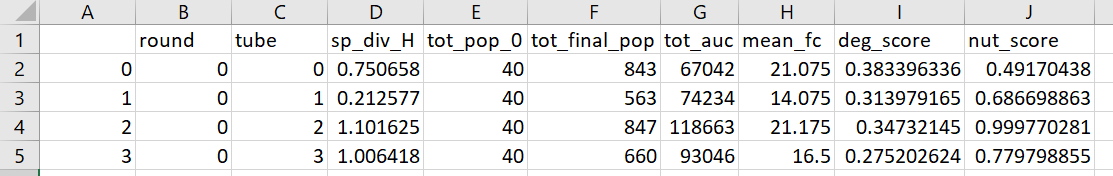
* **df\_nuts**. Stores the nutrient amounts at the end of each round in each tube. Each nutrient will appear in a different row.



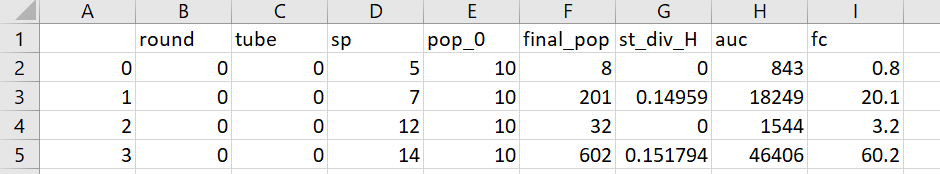
* **df\_toxs.** Stores the toxins amounts at the end of each round. Same configuration as “df\_nuts”.



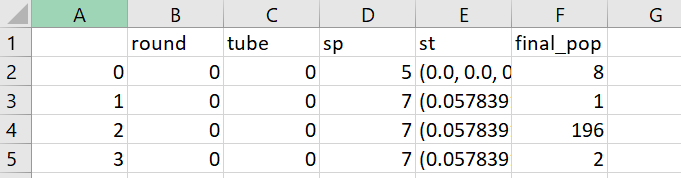
* **df\_grid.** The main one I use for plotting. This stores at each round and for each tube:
  + Diversity at the species level. Shanon index.
  + Total initial population (adding all the species)
  + Total final population.
  + Total area under the curve (AUC), just the sum of the total population at each time step (here I haven´t remove the initial population corresponding area yet)
  + Mean of the fold change of each species.
  + Degradation score.
  + Nutrient consumption score.



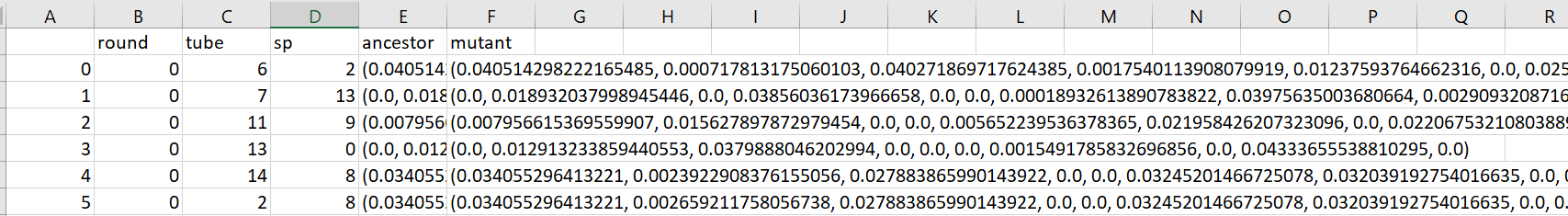
* **df\_sp.** For each species in each tube at eac round, it stores:
  + initial population
  + final population
  + strain diversity (shanon index)
  + Area under the curve #only in weplothere round
  + Fold change #only in weplothere round



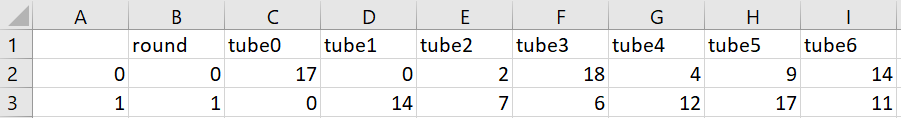
* **df\_st.** For each round, tube, species, it stores the final population of each strain. A strain is defined by a tuple of its f values (actually when converting the numpy data frame into a .cvs, the tuples are converted into string, so take that into account that when using this information (f values from this data frame), I need to turn the string back into a tuple)



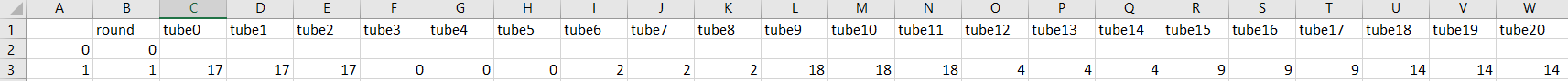
* **df\_mutants**. Not really needed. I tried to make the muller plots at some point, and I needed like an ancestry, which mutant is coming from which ancestor. So here I basically store for each round ant tube, every time a mutation occurs, from which ancestor the mutant is arriving. It ends up being huge though and not really useful…



* **df\_tubes.** It stores which tubes are selected at the end of each round. Each selected tube appears in a new column.



* **df\_ancestry.** It stores from each tube in the previous round is coming each tube in the actual round. Each tube in a column. In migrant pool will be empty, because each tube is coming from the mixing of all the tubes in “df\_tubes”.



### Also include some variables in pickle format at some rounds:

(pickle is a package to store binary files in python, helps to save space and also to be able to directly import the python object without worrying about transformations)

* Initial “grid” if “weplothere” (species at tm=0 of that particular round)
* “sp\_tube” (defined above) if “weplothere”. These could be included in the folder “per\_round” if desired.
* Last “grid” (list in the last time step, right now I found a little mistake because it is called grid100 even though I only run 80 runs, anyway not a big deal).

Also in this folder:

* extinctions.txt: document including information of extinctions for a repeat. In this version of the model if we select a tube with not enough population (not more population than “epsilon”) I will not stop the run, but chose the following tube in the list if selection treatment or another not chosen random tube if random treatment. This however could introduce some bias in the random treatment towards tubes were population is higher, rather than totally random. Then, just to keep track of these kind of “extinctions” are anecdotic, every time it happens the file will have one more line indicating the round and how many tubes were found to not have enough species (not necessarily all the tubes without enough species at that round, but just those chosen and with not enough).

## Results within a round (“per\_round”) only if “weplothere”:

* pdf file (in some older version it was a .png) showing the populations over time for each tube or community at this round. In each plot we write on top the species that are present, the nutrients remaining and the degradation score. Each species should be coloured in a particular color.
* Pickle file containing the above defined files “dict\_grid”, “dict\_nuts” and “dict\_toxs” inside a list.

# 

# How to run the model:

This version of the model is designed to run, for all the indicated propagation methods, all the desired repeats (all this only for one species set)

At the beginning of the script (#%% Section 2) all the important parameters are chosen. Some relevant ones to make sure you have properly indicated:

* Seed = 22 #the seed to generate the species and that will be reinitialised at the beginning of each repeat.
* list\_prop = [“d3”, “p”]list of propagations to run one after another. #choose: propagule (“p”), propagule with invasion (“pip”), migrant\_pool (“m”), migrant pool with invasion (“mip”) our disassembly (“d3”), or no selection (“n”).
* list\_treat = [“s”, “s”] #chose: random sampling control (r), or selection (s) for each of the propagations in list\_prop. It has no effect in no selection treatment but still I chose “r”.
* list\_repeats = range(0,10) #it has to be a range, of the repeats we want to run

# How to run the model in unix parallelising runs:

## Call arguments externally from unix

To later call the model feeding the seed for the species set (and could also be the propagation method if we want to run in a cluster and want to parallelise more) externally I do two main changes in the model:

I use the module sys to feed variables from unix into the python script (remember to add “import sys”)

#seed will be the first external argument

seed = int(sys.argv[1])

#propagation method will be the second external argument, it should assign both the propagation and whether it is selection or random treatment.

list\_prop = ["d", "d2", "p", "m", "pip", "mip", "n", "d", "d2", "p", "m", "pip", "mip"]

list\_treat = ["s","s", "s", "s", "s", "s", "r", "r", "r", "r", "r", "r", "r"]

index\_prop = int(sys.argv[2])

treat = list\_treat[index\_prop]

prop = list\_prop[index\_prop]

#the of course “enter\_path” should be located in the cluster.

## Bash script to call the python script in our cluster(fb.sh)

#!/bin/bash

#SBATCH --nodes 1

#SBATCH --ntasks-per-node 1

#SBATCH --cpus-per-task 1

#SBATCH --mem 2G #ram to use, when doing test, I found out I don´t need a lot

#SBATCH --time 03:44:00 #time of run, adjust depending on the simulation

module load gcc python #load python

cd /path\_to\_model #move to where you want to run the simulation not needed because I always call the script from there, but in case you rather call it from somewhere else.

source /path\_to\_virtual\_environment #activate virtual environment if needed

seed=$1

method=$2

python model\_name.py $seed $method #call the script with the external variables.

## How I really run in the cluster

First go to where you want to run the script ad create a directory to store the results, inside I store the python script and fb.sh (remember to update this address in fb.sh).

Then I just loop over number of propagation methods and seeds to run the script, so I run the following:

for method in {0..12} #propagation methods, includes both 0 and 12

do

for seed in {22..26} #species sets

do

sbatch fb.sh $seed $method

sleep 1 #at some point I had interferences when creating directories, I think it is solved through but now this gives a bit of time in case we want to cancel.

done

done

check that everything is going allright:

* “sacct”
* “Squeue”