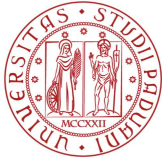


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN
INGEGNERIA BIOMEDICA

Implementazione di un simulatore per comunità microbiche basato su un modello multi-agente

Relatore:
ING. MASSIMO BELLATO, PHD
Correlatori:
CHIAR.MA PROF.SSA BARBARA DI CAMILLO
DOTT. MARCO CAPPELLATO

Laureando:
ANDREA CALZAVARA
MATR. 1222524

22 Luglio 2022 ∼ A.A. 2021-22

“Ho affermato che le matematiche sono molto utili per abituare la mente a un raziocinio esatto e ordinato; con ciò non è che io creda necessario che tutti gli uomini diventino dei matematici, ma quando con questo studio hanno acquisito il buon metodo di ragionare, essi lo possono usare in tutte le altri parti delle nostre conoscenze.”

John Locke

Abstract

In natura le comunità microbiche contengono centinaia, se non migliaia, di specie interagenti. La loro interazione con l'ambiente circostante, talvolta coincidente con i tessuti di un certo organismo ospite, è un fattore critico per poter determinare l'evoluzione di sistemi ecologici e fisiologici complessi. Per questa ragione le simulazioni computazionali stanno acquisendo un ruolo sempre più importante nell'ecologia microbica. In questo manoscritto, viene presentato un simulatore per comunità batteriche basato su un modello ad agenti multipli, entità autonome capaci di interagire tra loro e con l'ambiente. I modelli Agent-Based (ABM) fanno parte della classe dei modelli meccanicistici, la cui struttura matematica è basata su rigorose ipotesi biologiche. Questo tipo di modelli mira a descrivere i sistemi biologici con cognizione di causa, cercando di generare nuova conoscenza sull'evoluzione di un sistema, a partire da nozioni ben consolidate sui singoli costituenti che lo compongono.

L'obiettivo di questo progetto di tesi è sviluppare la versione preliminare del simulatore, implementando in Python una struttura di base che consenta già di ottenere le prime simulazioni e che sia modularmente incrementabile, sia in termini di funzionalità che di livello di dettaglio dei modelli e parametri descriventi la biologia rappresentata.

Il fine ultimo è, infatti, quello di produrre uno strumento atto a validare teorie ed ipotesi e che permetta di sostituire, in parte, la sperimentazione diretta. Questo garantirebbe enormi vantaggi dal punto di vista dell'impiego di tempo e risorse.

Ci sono ancora diversi raffinamenti da effettuare, alcuni dei quali vengono suggeriti verso la fine del manoscritto.

Si tenga conto, inoltre, che il simulatore è stato pensato per un utilizzo futuro: al momento attuale le tecnologie disponibili non consentono ancora di ottenere facilmente la conoscenza necessaria ad una definizione rigorosa di tutti i parametri degli agenti. Considerando, però, la velocità con la quale la scienza progredisce ci si aspetta che in una decina d'anni questo *gap* venga riempito e ci auguriamo che nel frattempo il simulatore raggiunga lo stato dell'arte.

In ogni caso, si sta già considerando la possibilità di rendere il simulatore un pacchetto Python con un'interfaccia *user-friendly* che sia fruibile anche ad utenti con limitate competenze di programmazione ed informatica in generale.

Indice

Abstract **v**

1 Modellizzazione di popolazioni microbiche **1**
1.1 Modelli ecologici . . . 2
1.2 Modelli microbo-effettore . . . 2
1.2.1 Modelli microbo-effettori ODE-based . . . 2
1.2.2 Genome-scale models . . . 2
1.3 Modelli agent-based . . . 3
1.3.1 Vantaggi e svantaggi dell'ABM . . . 4

2 Obiettivi **5**

3 Implementazione **7**
3.1 Definizione del modello . . . 7
3.2 Trasposizione in codice Python . . . 8
3.2.1 Classi . . . 8
3.2.2 Metodi . . . 10
3.2.3 Esecuzione del simulatore: funzione *main* . . . 14
3.2.4 Funzioni accessorie . . . 16
3.3 Raffinamenti del modello . . . 20
3.3.1 Tossicità delle molecole effettrici . . . 20
3.3.2 Flusso microbico . . . 22

4 Simulazioni significative **25**
4.1 Ambiente ostile . . . 25
4.2 Ambiente favorevole . . . 26
4.3 Sintrofia . . . 26

5 Ulteriori raffinamenti e modellizzazioni **31**
5.1 Raffinamenti . . . 31
5.2 Ulteriori modellizzazioni . . . 32

6 Conclusioni **33**

A Equazione di Hill 41

B Codice Python del simulatore 43

Capitolo 1

Modellizzazione di popolazioni microbiche

Le comunità batteriche sono diffuse ovunque sulla Terra e la loro presenza può avere effetti significativi sull'ambiente in cui si trovano. Ad esempio il microbiota intestinale umano (comunemente detto flora intestinale) svolge delle attività di vitale importanza: sono stati individuati, infatti, diversi effetti positivi sulla salute, in contrasto a condizioni patologiche come l'obesità, depressione e melanomi [1–3]. Studiare e comprendere l'evoluzione di comunità batteriche in risposta a determinati stimoli, come la somministrazione di un antibiotico, diventa dunque un tema di particolare interesse nel campo della biologia dei sistemi. Ad esempio per la previsione dell'evoluzione di una certa patologia associata al microbiota, ma anche per la progettazione di comunità batteriche ingegnerizzate con funzioni terapeutiche desiderate. Le comunità batteriche sono, però, caratterizzate da complesse reti di interazioni inter-microbiche e tra microbi e ambiente: per questo motivo creare un modello affidabile è un obiettivo particolarmente impegnativo. Pertanto, lo sviluppo di un simulatore che permetta di accrescere la nostra conoscenza sulle reti di interazione e sulle comunità microbiche in generale, è un obiettivo ambito da molti ricercatori.

Ci sono due principali tipi di approccio: i modelli meccanicistici, la cui struttura matematica è basata su rigorose ipotesi biologiche e in cui ogni parametro ha una diretta interpretazione fisiologica, ed i modelli empirici, la cui struttura matematica permette di spiegare dati sperimentali senza però adottare necessariamente struttura e parametri con una diretta interpretazione fisiologica (vedi fig. 1.1 a pagina 3). In letteratura si osserva una maggiore tendenza, però, a combinare i due approcci per ottenere dei modelli integrati con un campo di applicazione più esteso [4–6]. Questi modelli empirici si ottengono sostituendo equazioni e parametri costanti dei modelli meccanicistici con equazioni agglomerate i cui parametri sono definiti in funzione di parametri ambientali. Seguendo quanto fatto da Qian et al. [7], in seguito classificheremo i modelli sulla base del loro obiettivo principale come ecologici, microbo-effettore ed Agent-Based.

1.1 Modelli ecologici

L'obiettivo dei modelli ecologici è predire ed analizzare le dinamiche di popolazione, inclusi i cambiamenti nel tempo delle abbondanze delle singole popolazioni e come tali popolazioni si influenzano tra loro, senza considerare le interazioni molecolari che stanno alla base.

Un esempio è la classe dei modelli generalizzati di Lodka-Volterra (gLV) che, basati su equazioni differenziali ordinarie (ODEs), riescono a descrivere l'abbondanza assoluta di ogni popolazioni microbica attraverso una funzione del tempo che tiene conto del proprio tasso di crescita intrinseco, delle interazioni con le altre popolazioni e di ingressi esterni come trattamenti antibiotici [8, 9]. Una delle maggiori limitazioni è l'utilizzo di parametri costanti, che rendono il modello poco adatto ad ambienti che subiscono cambiamenti significativi nel tempo.

Un altra classe è quella dei modelli regressivi dinamici *data-driven* [10–12], la cui struttura matematica è basata su dati raccolti sperimentalmente piuttosto che su ipotesi di tipo biologico. Questi modelli, in cui la composizione della comunità dipende dalla composizione misurata a tempi precedenti, sono predittivi ma non meccanicistici: non permettono infatti una comprensione diretta dei meccanismi di interazione.

1.2 Modelli microbo-effettore

Le interazioni tra le popolazioni sono spesso dovute a molecole effettrici, come metaboliti e tossine, rilasciate e assorbite dall'ambiente condiviso. L'obiettivo di questi modelli è di comprendere il ruolo degli effettori nel mediare la crescita batterica, la formazione della comunità e molto altro.

1.2.1 Modelli microbo-effettori ODE-based

I modelli microbo-effettori basati sulle equazioni differenziali ordinarie (ODE) possono essere usati per descrivere la variazione della crescita microbica in relazione alla concentrazione extracellulare di determinati effettori chiave [13]. Questo tipo di modello permette una comprensione meccanicistica del modo in cui le interazioni ecologiche derivino da processi biochimici, come la sintrofia (fenomeno per cui una specie si nutre dei prodotti metabolici di un'altra e viceversa) [14] e la competizione (fenomeno per cui la disponibilità di un nutriente per una specie è ridotta a causa della presenza di un'altra che lo consuma) [15]. La difficoltà principale consiste nel definire oculatamente il giusto livello di complessità: all'aumentare del numero di effettori il modello offre una maggiore flessibilità a discapito, però, di un maggiore numero di parametri cinetici da identificare attraverso i dati sperimentali raccolti.

1.2.2 Genome-scale models

Un altro tipo di modello microbo-effettore sono i *genome-scale models* (GEMs), un potente strumento per l'analisi e il design di comunità microbiche. I GEMs cercano di predire tutte le reazioni metaboliche che si verificano all'interno di un microbo in virtù dell'informazione

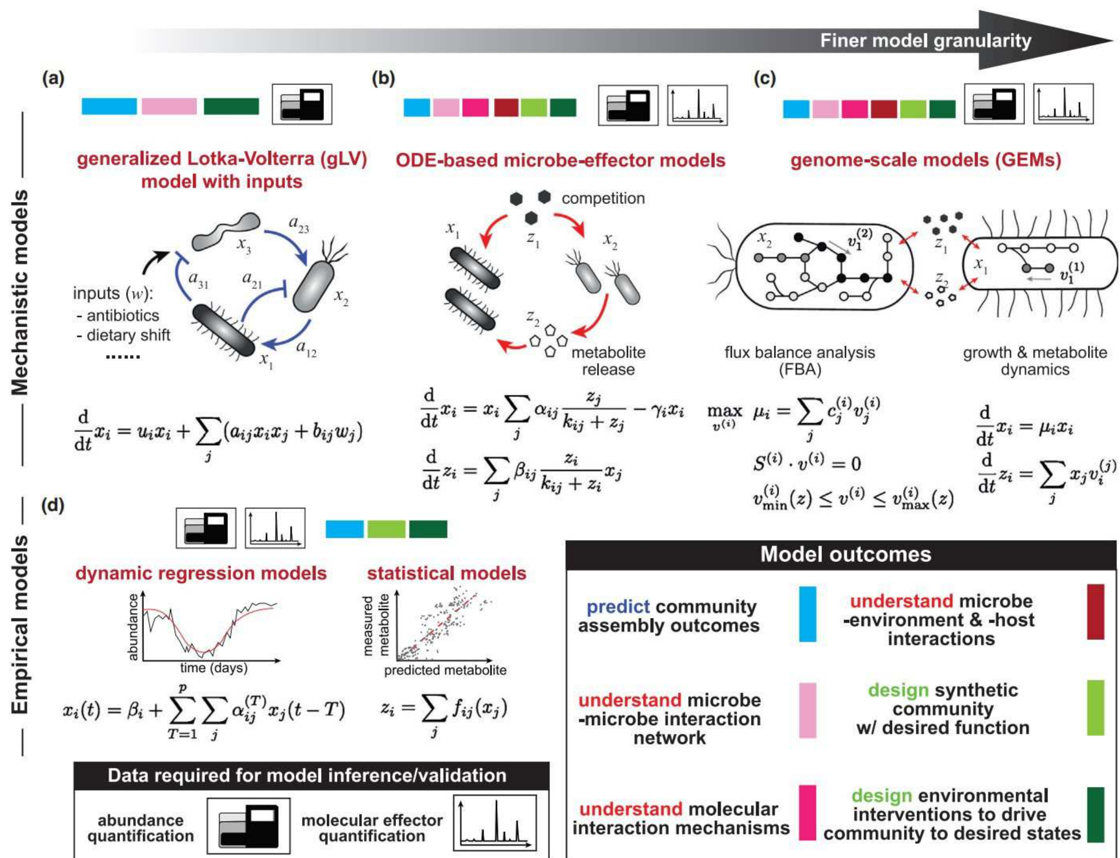


Figura 1.1: Confronto tra vari modelli, immagine presa da Qian et al., Curr. Opin. Microbiol., 2021 [7]

genomica e di evidenze sperimentali. Nella teoria, potrebbero predire la composizione di una comunità e il metabolismo di una specie senza alcun dato sperimentale. La simulazione dei GEMs è fondata sull'analisi del bilanciamento dei flussi (FBA) [16], che calcola i flussi che ottimizzano il tasso di crescita batterica dati i coefficienti stechiometrici delle reazioni metaboliche. Questa assunzione di ottimizzazione evita di dover identificare migliaia di parametri cinetici per tutte le reazioni metaboliche. Il principale problema, in questo caso, è un accurato sequenziamento, nonché una completa annotazione del genoma microbico e del suo trascrittoma (insieme delle molecole di mRNA ottenute dalla trascrizione del DNA) [17].

1.3 Modelli agent-based

La modellizzazione attraverso agenti multipli (ABM) è un approccio sempre più utilizzato nell'ambito dei sistemi dinamici e, quindi, anche nel campo della biologia dei sistemi. L'ABM si basa sull'idea che alcuni tipi di sistemi possano essere modellati rappresentando le singole entità che li compongono e attribuendo a queste un insieme di caratteristiche peculiari. Queste entità sono appunto dette agenti. Innanzitutto ogni modello di questo tipo ha una caratterizzazione spaziale, almeno implicita, in termini di relazioni fra gli agenti. In secondo luogo, gli agenti interagiscono secondo un criterio di prossimità, considerando come prossimo l'insieme delle entità che ciascun agente è in grado di percepire. Infine ogni agente segue delle regole di comportamento che definiscono come interagisce con le altre entità limitrofe e con

l'ambiente [18].

1.3.1 Vantaggi e svantaggi dell'ABM

Innanzitutto questo tipo di modellizzazione permette un approfondimento meccanicistico sulle interazioni a livello molecolare tra microbi e tra microbi e ambiente. In futuro, questo potrebbe consentire la progettazione di comunità batteriche ingegnerizzate con determinate caratteristiche desiderate, come ad esempio la produzione di metaboliti che inducono nell'ospite la resistenza a specie microbiche nocive.

Si noti che, a differenza di altri modelli, negli ABM il tipo di interazione che si instaura tra i microbi è una conseguenza delle caratteristiche dei singoli agenti e non una ipotesi definita a priori. Ad esempio, fenomeni come la sintrofia e la competizione non sono la causa delle interazioni tra le specie, come accade in altri modelli, ma emergono piuttosto come conseguenza dell'interazione tra agenti e tra agenti e ambiente.

Dal punto di vista informatico, si può pensare un agente come un oggetto, i cui attributi e metodi definiscono il suo comportamento all'interno del sistema. La struttura modulare che ne deriva è facilmente implementabile, ad esempio nel caso si scoprano nuovi fenomeni da modellizzare. Altri tipi di modelli, invece, necessiterebbero di cambiamenti più radicali.

Un altro punto di forza è la flessibilità: infatti attraverso i diversi di parametri di simulazione si riescono a catturare una vasta gamma di fenomeni, come variazioni dell'ambiente o del microbiota.

Il principale svantaggio di questo tipo di approccio, invece, è senz'altro la necessità di un gran numero di parametri che spesso risultano difficili da misurare o definire. La mancanza di conoscenza è, quindi, il principale limite dei modelli Agent-Based. Tuttavia esistono già diversi *database*, come ad esempio *BacDive*, dai quali è possibile reperire alcuni di questi parametri. Col progredire della scienza e delle tecnologie, questi verranno gradualmente arricchiti e, quindi, risulta chiaro come un approccio di questo tipo sia un strategia vincente.

Capitolo 2

Obiettivi

L'obiettivo fondamentale del progetto su cui nasce questa tesi è l'implementazione in Python di un simulatore che descriva l'evoluzione temporale di comunità microbiche attraverso un struttura ad agenti multipli. Lo scopo di questo lavoro di tesi è creare un simulatore predittivo altamente modulare che sia facilmente implementabile grazie alla sua struttura ad oggetti. In concreto, quello che ci siamo prefissati di sviluppare è la versione preliminare di un simulatore che sia comunque in grado di generare dei risultati verosimili ed interessanti.

Tuttavia, si consideri che il simulatore è stato pensato per un utilizzo futuro, in quanto il livello di conoscenza attuale non è tale da permettere un'adeguata accuratezza nella definizione degli agenti.

Il fine ultimo di questo progetto è, in ogni caso, creare uno strumento che permetta di effettuare delle analisi accurate senza la necessità di alcuna sperimentazione diretta, andando a risparmiare tempo e risorse. Questo si potrebbe rivelare utile anche nello studio di specie batteriche particolarmente nocive, in modo da ridurre i rischi.

In futuro, il simulatore si propone di essere un mezzo efficace per validare teorie ed ipotesi e, di conseguenza, per approfondire la nostra conoscenza sui processi ecologici e biochimici che determinano l'evoluzione di una comunità microbica.

Capitolo 3

Implementazione

La costruzione del simulatore è avvenuta in due fasi successive: la prima di definizione dei formalismi di base e la seconda di raffinamento del modello al fine di renderlo più vicino alla realtà.
La prima parte è stata suddivisa in due sottofasi:

- la definizione del giusto livello di astrazione;
- la sua traduzione nel linguaggio di programmazione ad alto livello Python.

Nel paragrafo 3.1 si procede alla descrizione del modello, mentre nel 3.2 sarà descritta la sua implementazione in codice Python.

3.1 Definizione del modello

Innanzitutto, in un modello ad agenti multipli si devono definire con esattezza la caratterizzazione spaziale e gli agenti che popolano l'ambiente così definito. La scelta è stata quella di sviluppare un modello in cui i batteri colonizzano un ambiente costituito da una serie di celle successive con propagazione monodimensionale lineare, come può essere un tratto di colon.
Ogni cella conterrà un insieme casuale di metaboliti iniziali dal quale i diversi agenti presenti, ossia le diverse specie batteriche, potranno ricavare l'energia necessaria a duplicarsi, in base al loro specifico metabolismo. Ogni specie batterica è definita da un nome identificativo, dal suo preciso metabolismo, dal tasso di crescita massimo e dalla cella occupata nell'ambiente.
In particolare nel simulatore il numero di celle, specie batteriche e tipo di nutrienti metabolizzabili di *default* è 5. Questi parametri, ovviamente, possono essere modificati secondo le diverse necessità dell'utente.
Il metabolismo di ogni specie batterica è definito sulla base della sua capacità di consumare, produrre o ignorare un determinato tipo di molecola effettrice. Si precisa che in questa tesi non verrà mai utilizzato un numero elevato di specie batteriche o metaboliti e verranno, quindi, sempre tenuti i parametri di *default*: infatti, sebbene sia più realistico, l'aumento delle specie e metaboliti renderebbe l'analisi dei risultati e la valutazione delle potenzialità del simulatore molto più complesse, nonché inaffidabili, dal momento che i parametri relativi ai metabolismi

sono fittizi.
La fase dinamica del simulatore si basa su cicli di iterazione successivi, in ognuno dei quali si calcola:

1. il fattore di crescita di ciascuna specie presente in una determinata cella;
2. la quantità di nuovi batteri nati;
3. la colonizzazione della cella successiva da parte della metà dei neo-nati.

Per ciascuna specie (in una determinata cella) il tasso di crescita è calcolato tenendo conto del proprio fattore di crescita massimo e del proprio metabolismo in relazione ai nutrienti disponibili in quella cella. Nel caso si stia considerando l'ultima cella, i nuovi batteri che sarebbero dovuti essere trasferiti in quella successiva si considerano espulsi dall'ambiente.
Riassumendo, ad ogni iterazione, il *pool* di batteri presenti in ogni cella consuma e produce nutrienti, permettendo loro di moltiplicarsi e fluire verso la cella successiva (vedi fig. 3.1).

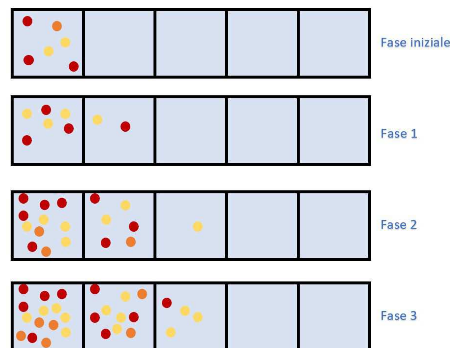


Figura 3.1: Esempio dell'evoluzione di una comunità batterica modellizzata

3.2 Trasposizione in codice Python

La traduzione in linguaggio Python del modello appena descritto rappresenta il fulcro dell'intero progetto.
È importante, quindi, effettuare una scelta oculata delle classi e dei rispettivi attributi e metodi da implementare. In aggiunta è necessario pensare con scrupolo la realizzazione della fase dinamica del simulatore, cercando di rendere il codice il più modulare e chiaro possibile, in vista di eventuali successive modifiche.

3.2.1 Classi

Si è scelto di utilizzare due classi (vedi fig. 3.2 a pagina 10):

- *Cell*, che rappresenta una cella dell'ambiente lineare di crescita;
- *Bact*, che rappresenta l'agente, ossia una specifica popolazione microbica presente in una cella.

Classe *Cell*

La classe *Cell*, come ogni classe, è definita dai suoi attributi e dai suoi metodi. Per ogni istanza *Cell*, gli attributi sono:

- il vettore dei nutrienti **f**, una lista contenente in ogni posizione un intero che indica la quantità di un determinato nutriente nella cella;
- *MatrFood*, una matrice che tiene traccia delle variazioni temporali dei metaboliti nella cella;
- il dizionario dei batteri *bac_diz*, che indica la quantità di ogni specie microbica. In particolare, le chiavi sono istanze *Bact*, mentre i valori sono interi che specificano le corrispondenti quantità di batteri presenti nella cella;
- il vettore ambiente **x**, una lista che contiene le diverse istanze *Cell*;
- l'indice *pos* della posizione nel vettore ambiente, definito da un intero.

I metodi sono i seguenti:

- *getBact*, metodo di solo accesso che restituisce in uscita un dizionario che ha come chiavi le stringhe contenenti i nomi identificativi dei tipi batterici e come valori il numero intero di batteri corrispondenti nella cella;
- *getFood*, metodo di solo accesso che restituisce in uscita il vettore dei nutrienti **f** della cella;
- *getMatrFood*, metodo di solo accesso che restituisce in uscita la matrice MatrFood;
- *addBact*, metodo *mutator* che permette di aggiornare il dizionario dei batteri della cella, dando in ingresso un dizionario costruito allo stesso modo ma contenente, per ogni tipologia batterica, il numero di nuovi batteri da aggiungere;
- *food_upd*, metodo che aggiorna il vettore dei nutrienti **f** della cella;
- *death*, metodo che calcola la quantità di microbi morti nell'iterazione corrente per ogni specie;
- *evolution*, metodo che gestisce interamente la fase dinamica del simulatore, richiamando i metodi precedenti.

Classe *Bact*

Lo stato di un singolo oggetto della classe è definito dai suoi attributi, che sono:

- la stringa *type*, che definisce il tipo di batterio;
- il vettore **m**, una lista di interi che definisce il metabolismo della specie batterica rispetto ai nutrienti presenti (vedi fig. 3.4 a pagina 12);

- la variabile *maxGr*, un numero (float) che definisce il tasso di crescita massimo;
- *pos*, un intero che indica la posizione nel vettore ambiente **x** della cella che colonizza;
- *maxTox* ed il vettore **t**, due attributi riportati per completezza, ma che saranno descritti in seguito, poiché frutto di una raffinazione del modello iniziale.

Inoltre, la classe *Bact* presenta anche l'attributo di classe *species*, cioè una lista di stringhe, ciascuna delle quali definisce un diverso tipo batterico.
In seguito si riportano i metodi:

- *type, getm, getmaxGr, getPos*, metodi di solo accesso che restituiscono in uscita i corrispettivi attributi;
- *getgrowth*, metodo che implementa l'algoritmo di calcolo del fattore di crescita per quella specie in una determinata cella;
- *getTox*, metodo che verrà discusso in seguito, poiché parte di un raffinamento successivo.

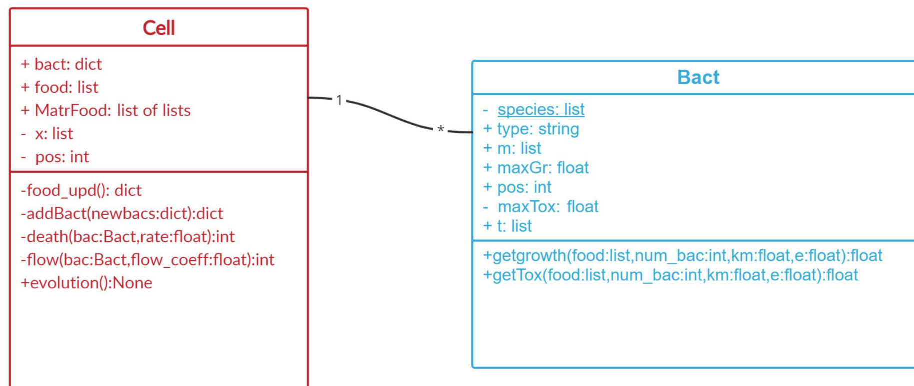


Figura 3.2: Diagramma UML delle classi (da notare che alcuni metodi presenti verranno trattati in seguito)

3.2.2 Metodi

In questo paragrafo i metodi citati vengono descritti in maniera più approfondita.

Metodo *getgrowth*

Questo metodo della classe *Bact* calcola e restituisce in uscita il valore del tasso di crescita di una specie batterica in una determinata cella.
Siano μ_{max} il tasso di crescita massimo, f_{tot} la quantità totale di nutrienti che il batterio ha consumato (data dalla funzione *food_upd*, vedi sec. 3.2.2 nella pagina successiva), n_{bac} il numero di batteri di quella specie nella cella, km ed η i parametri dell'equazione di Hill (vedi app. A)

che modellizza il tasso di crescita.
Allora, si definisce il tasso di crescita μ come segue:

$$\mu = \begin{cases} \dfrac{\mu_{max}}{1+\left(\dfrac{km \cdot n_{bac}}{f_{tot}}\right)^{\eta}} & \text{se } f_{tot} > 0 \\ 0 & \text{se } f_{tot} = 0 \end{cases}$$

In assenza di nutrimenti consumati il tasso di crescita sarà nullo e questo è consistente con la realtà: infatti i batteri non sarebbero in grado di ricavare l'energia necessaria alla duplicazione.
La scelta dell'equazione di Hill è dovuta ad un'analisi empirica della crescita di una popolazione batterica, basata sulle seguenti considerazioni (vedi fig. 3.3):

1. all'aumentare di f_{tot} il tasso di crescita aumenta fino ad un valore di saturazione, detto μ_{max};

2. all'aumentare di n_{bac} sono necessari più nutrienti per raggiungere lo stesso tasso di crescita.

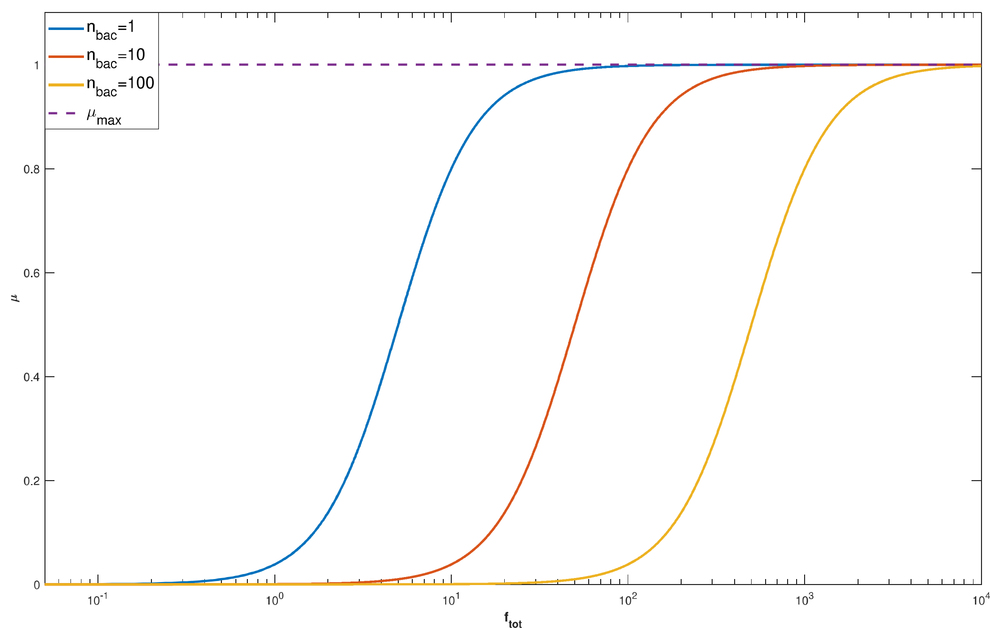


Figura 3.3: Tasso di crescita con equazione di Hill a parametri km ed η costanti

La scelta dei parametri km ed η, comuni per tutte le specie batteriche della comunità, deve essere effettuata all'inizio di ogni simulazione.
A seguito di diverse simulazioni sono stati scelti $km = 0.5$ e $\eta = 1$ come valori di *default*.

Metodo *food_upd*

Questo metodo della classe *Cell* implementa l'algoritmo di aggiornamento dei nutrienti nella cella dovuto all'azione del microbiota, che si compone di due fasi distinte: la prima di consumo e la seconda di produzione dei nutrienti.

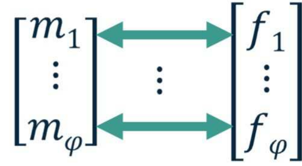


Figura 3.4: Relazione tra vettore dei metabolismi della specie i e vettore dei nutrienti della cella

1. **Consumo**

   Siano n_i il numero di batteri della specie i, f_j la quantità di nutriente j-esimo nella cella e $m_{i,j}$ il corrispondente metabolismo della specie batterica i (il vettore dei nutrienti della cella **f** e quello dei metabolismi di ogni specie **m** sono della stessa dimensione per costruzione, vedi fig. 3.4). Allora, durante la fase di consumo, per ogni nutriente j, viene verificato se la sua abbondanza è sufficiente per tutta la comunità batterica, ossia se:

   $$f_j > \left| \sum_{i=1} (n_i \cdot m_{i,j})|_{m_{i,j}<0} \right| \tag{3.1}$$

   e in tal caso:

   $$f_j^{new} = f_j + \sum_{i=1} (n_i \cdot m_{i,j})|_{m_{i,j}<0}$$

   Se, invece, per il metabolita j la condizione 3.1 non fosse verificata, oltre ad imporre $f_j^{new} = 0$ si deve scegliere un criterio per definire come la quantità di nutriente f_j venga distribuita tra le varie specie batteriche che lo consumano.
   A questo proposito si è pensato di utilizzare un campionamento ottenuto da una distribuzione ipergeometrica multivariata, che è una densità di probabilità discreta. In particolare questa distribuzione definisce la probabilità che, dopo N estrazioni senza reimmissione da un'urna contenente D palline (con $D \geq N$) di M colori $c_1, c_2, ...c_M$ di cui si conosce esattamente il numero, l'insieme di palline estratto sia esattamente $\{n_{c_1}, n_{c_2}, ...n_{c_M}\}$ ove $n_{c_1}, n_{c_2}, ...n_{c_M} \in \mathbf{N}$ indicano il numero di palline del corrispondente colore a pedice (vedi fig. 3.5 nella pagina successiva). Nel nostro caso, il numero di estrazioni N indica la quantità di nutriente j da distribuire, mentre i colori delle palline rappresentano le diverse specie che compongono il microbiota della cella.
   Dunque, nell'ipotesi che la probabilità di una specie batterica di consumare il nutriente dipenda solamente dalla quantità relativa della specie stessa in relazione alla comunità presente nella cella, il campionamento dalla distribuzione ci fornisce un plausibile smistamento del nutriente.
   Infine, l'algoritmo tiene traccia dei nutrienti complessivamente consumati dalla specie i-esima tramite un dizionario, le cui chiavi sono le istanze *Bact* presenti nella cella e i cui valori sono la misura cumulativa dei nutrienti da esse consumati $f_{tot,i}$. Questo dizionario, restituito in uscita dal metodo, ci fornirà le informazioni necessarie a calcolare il fattore di crescita per ogni specie batterica (attraverso il metodo *getgrowth* visto precedentemente).

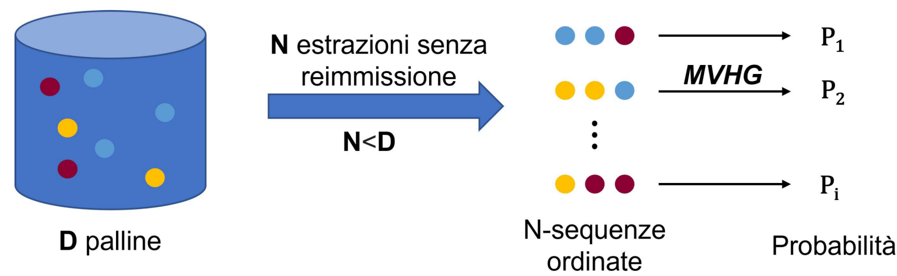


Figura 3.5: Esempio di multivariata ipergeometrica (MVHG)

2. **Produzione**

   Segue poi la produzione dei nutrienti: durante questa fase viene tenuto conto del principio di conservazione di massa delle molecole effettrici, la cui mancanza avrebbe portato alla possibilità di generare una fonte illimitata di risorse e, di conseguenza, un aumento smisurato delle dimensioni delle popolazioni batteriche.
   Abbiamo quindi considerato che i nutrienti prodotti siano in egual numero rispetto a quelli consumati in modo da evitare una produzione anomala di metaboliti; tuttavia questa modellizzazione è ancora da perfezionare in quanto non tiene conto dell'energia utilizzata dalle specie batteriche per la propria crescita che imporrebbe $\mathbf{f}_{prodotti} < \mathbf{f}_{consumati}$. Quindi, per ogni specie batterica, il simulatore:

   (a) calcola il totale dei nutrienti consumati e lo divide tra i vari nutrienti che la specie batterica può produrre, in maniera proporzionale al rispettivo metabolismo di quel nutriente prodotto;

   (b) nel caso siano presenti nutrienti consumati residui, attraverso la distribuzione multivariata ipergeometrica l'algoritmo li divide tra i nutrienti prodotti, con maggiore probabilità per i nutrienti con metabolismi maggiori.

   Il motivo dietro all'utilizzo di questa distribuzione di probabilità è che ci permette di ripartire i nutrienti consumati tra i vari metaboliti che la specie può produrre, con probabilità pesata dai rispettivi metabolismi di produzione.

Metodo *death*

Il metodo *death* della classe *Cell* calcola e restituisce in uscita il numero di batteri per tipo che, nell'iterazione corrente, saranno destinati a morire. Sia d_i il numero di batteri della specie i che moriranno, n_i il numero di batteri della medesima specie ad inizio ciclo e λ il tasso di morte basale, uguale per ogni specie, che di default è 0.027 (tasso di morte del batterio *Escherichia coli* [19]). Allora, la funzione lineare che modellizza la morte è:

$$d_i(n_i) = \lceil \lambda \cdot n_i \rceil \quad \forall i \tag{3.2}$$

Si noti che, dovendo essere il d_i un intero, il risultato viene approssimato per eccesso al minore intero successivo (come indicato dal simbolo "$\lceil \quad \rceil$"). Questa accortezza è necessaria per

permettere l'estinzione delle popolazioni microbiche: l'approssimazione per eccesso, infatti, implica sempre la morte di almeno un batterio, anche nel caso di popolazioni di dimensioni ridotte.
Come vedremo in seguito, questo metodo può essere perfezionato con l'aggiunta di effetti legati alla produzione di metaboliti tossici da parte di altre specie batteriche.

Metodo *evolution*

Il metodo *evolution* della classe *Cell* è il fulcro del simulatore: gestisce, infatti, l'evoluzione della comunità batterica, calcolando nascite e morti e coordinando lo smistamento.
Innanzitutto viene aggiornato il vettore dei nutrienti **f** attraverso il metodo *food_upd* che darà in uscita il dizionario dei nutrienti consumati (vedi par. 3.2.2 a pagina 11). Il vettore dei nutrienti appena aggiornato viene aggiunto come colonna alla matrice *MatrFood*, così da tenere traccia delle variazioni nel tempo dei metaboliti nella cella.
Poi, per ogni specie batterica i della cella:

1. viene calcolato il fattore di crescita μ_i dando in ingresso il dizionario dei nutrienti consumati al metodo *getgrowth*;

2. viene calcolato il numero batteri morti d_i che viene, quindi, sottratto al totale;

3. viene calcolato il numero di nuovi batteri n_i^{new} dal numero di batteri presenti ad inizio ciclo n_i attraverso la formula $n_i^{new} = \mu_i \cdot n_i$;

4. metà dei nuovi batteri $\frac{n_i^{new}}{2}$ vengono aggiunti alla cella corrente, mentre l'altra metà viene trasferita alla cella successiva attraverso il metodo *addBact* o, nel caso si stia considerando l'ultima cella, viene eliminata simulando l'uscita dei batteri dal sistema.

Si noti che il numero di batteri deve sempre essere intero: nello smistamento dei neo-nati, infatti, viene utilizzata la divisione intera.
Chiaramente lo spostamento dovrà essere raffinato modellando le proprietà di *swarming* (rapido spostamento collettivo su superfici solide o semi-solide) di ogni singola specie, oltre che possibili effetti legati alla *chemiotassi* (movimento dei batteri legato a stimoli chimici) ed agli effetti legati ai movimenti peristaltici (laddove si intenda modellare un tratto intestinale).

3.2.3 Esecuzione del simulatore: funzione *main*

Il *main* è il corpo dello script, ossia quella parte del codice sorgente che richiama metodi e funzioni per mettere in esecuzione il programma. Nel nostro caso, è composto di una prima parte di definizione dei parametri della simulazione, una seconda parte di inizializzazione di oggetti, vettori e dizionari necessari all'esecuzione ed un'ultima parte di simulazione vera e propria.
Inoltre, considerando che al momento non si ha la conoscenza necessaria a definire i diversi parametri delle specie batteriche, la simulazione farà riferimento a popolazioni microbiche con

caratteristiche aleatorie.
Quindi, i parametri da impostare sono:

- il numero di celle nel quale è suddividere l'ambiente;
- il numero del tipo di molecole effettrici in ogni cella, che definisce la lunghezza del vettore dei nutrienti **f** e dei metabolismi **m** (vedi fig. 3.4 a pagina 12);
- la durata temporale dell'evoluzione della comunità batterica, che corrisponde al numero di cicli della simulazione;
- parametri per il collaudo con comunità batteriche casuali, quali il numero massimo iniziale di batteri per singola specie.

Le variabili che vengono allocate di conseguenza sono:

- la lista di oggetti *Bact* con caratteristiche casuali;
- dizionari che, per ogni cella, definiscono le quantità di microbi contenuti (le chiavi saranno oggetti di tipo *Bact* e i valori le rispettive quantità presenti nella cella);
- il vettore dei nutrienti **f** presenti in ogni cella;
- gli oggetti *Cell*, inizializzati in virtù dei corrispondenti vettore dei nutrienti **f** e dizionario dei batteri;
- il vettore ambiente **x**, ossia la lista contenente le istanze *Cell*.

A questo punto la fase dinamica della simulazione consiste in un algoritmo molto semplice, che racchiude la complessità all'interno del metodo *evolution* della classe *Cell* (vedi lst. 3.1 e fig. 3.6 nella pagina successiva) e di alcune funzioni ausiliarie (vedi par. 3.2.4 nella pagina seguente).

Listing 3.1: Pseudocodice dell'algoritmo

```
mostra caratteristiche batteri (metabolismi e tasso di crescita basale)
mostra stato iniziale sistema (distribuzione di batteri)
per numero di cicli impostato:
    per tutte le celle (dall'ultima alla prima):
        cella.evolution()
    mostra stato del sistema
mostra variazioni temporali dei metaboliti nel sistema
```

Il motivo dietro alla scelta di richiamare il metodo dalla cella in ultima posizione nel vettore ambiente alla prima è semplice: in questo modo evitiamo di considerare i batteri appena nati e trasferiti dalla cella precedente nel calcolo dei nuovi batteri della cella stessa. Infatti, dei microbi appena formati avranno bisogno di tempo e nutrienti per svilupparsi e raggiungere nuovamente la fase di scissione.

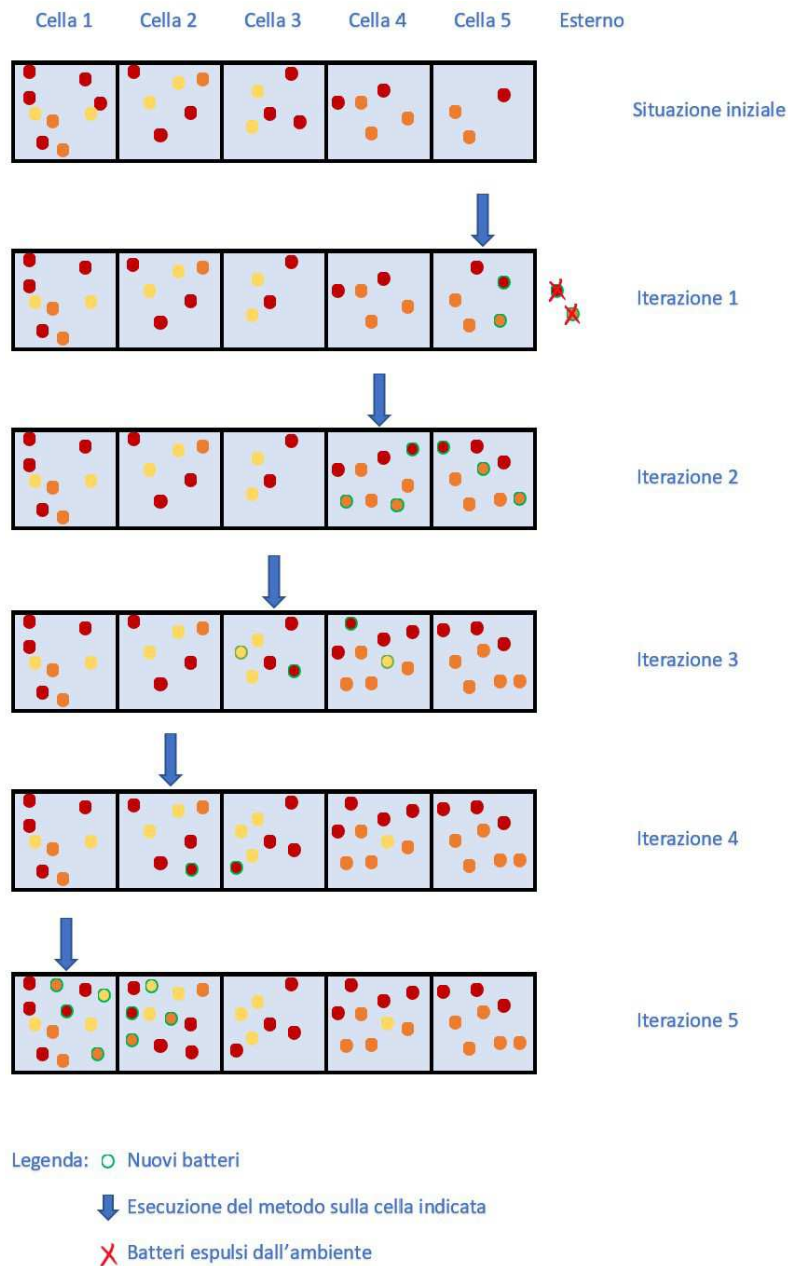


Figura 3.6: Illustrazione dell'algoritmo

3.2.4 Funzioni accessorie

Nell'implementazione del *main* sono state utilizzate alcune funzioni accessorie:

- *randomBacts*, una funzione che inizializza e restituisce in uscita una lista di batteri con caratteristiche casuali, quali il vettore dei metabolismi **m**, ottenuto mediante la funzione *randomFill*, e il tasso di crescita massimo.

- *randomFill*, una funzione che inizializza e restituisce in uscita una lista di lunghezza desiderata i cui elementi sono numeri interi estratti casualmente da un intervallo definito dai parametri in ingresso;

- *printState*, una funzione che stampa a schermo la quantità di batteri per specie contenuta in ogni cella;

- *graph*, una funzione che grafica, attraverso uno stackplot, la distribuzione dei batteri all'interno del sistema;
- *graph_met*, una funzione che grafica la distribuzione spazio-temporale dei nutrienti nel sistema durante la simulazione.

Vediamo ora nel dettaglio le più complesse.

Funzione *randomBacts*

Questa funzione, che richiede in ingresso i parametri di lunghezza del vettore dei metabolismi *len_m* e la posizione iniziale dei batteri nel vettore ambiente, restituisce in uscita una lista di oggetti *Bact* con caratteristiche aleatorie, quali il vettore dei metabolismi **m** e il tasso di crescita massimo μ_{max}. Il primo è definito casualmente per ogni specie attraverso la funzione *randomFill* che, con i giusti parametri, genera una lista di numeri interi campionati tra -1 e +1 dove:

- -1 indica che il nutriente corrispondente (vedi fig. 3.4 a pagina 12) può essere consumato dalla specie;
- 0 indica che il nutriente corrispondente viene ignorato dalla specie;
- +1 indica che il nutriente corrispondente può essere prodotto dalla specie.

Si noti che questa scelta è stata fatta per semplicità: è possibile, infatti, impostare dei metabolismi con modulo diverso da 0 o 1, ponendoli ad esempio proporzionali a parametri stechiometrici. Infine, viene effettuato un controllo per evitare la creazione di metabolismi irrealistici: si procede ad un assegnazione casuale di 1 e/o -1 nel caso *randomFill* abbia generato una lista che non li contenga. Il tasso di crescita massimo μ_{max}, invece, si ricava dal *doubling time* t_2 (in ore) attraverso la formula $\mu_{max} = \frac{\ln(2)}{t_2}$. Il *doubling time* è a sua volta ottenuto dal campionamento casuale da una distribuzione lognormale di parametri $(\mu, \sigma) = (2.8, 1.9745)$, come quella riportata in fig. 3.7 nella pagina seguente).
Il motivo dietro alla scelta di questa distribuzione con questi parametri è prettamente empirico: la curva che ne deriva è molto simile a quella della distribuzione dei tempi di raddoppio delle popolazioni microbiche. Infatti:

- la distribuzione lognormale non è definita nel semiasse negativo delle ascisse e non può, quindi, assumere valori negativi;
- la maggior parte dei batteri ha un tempo di raddoppio simile a quello del batterio *E. coli*, cioè di circa 20 minuti. In termini matematici vogliamo che la moda μ_0 sia 20 min.

Manipolando la formula per la moda della lognormale $\mu_0 = e^{\mu - \sigma^2}$ e considerando una media pari a $2.8h$, si ottiene che $\sigma \simeq 1.9745$.

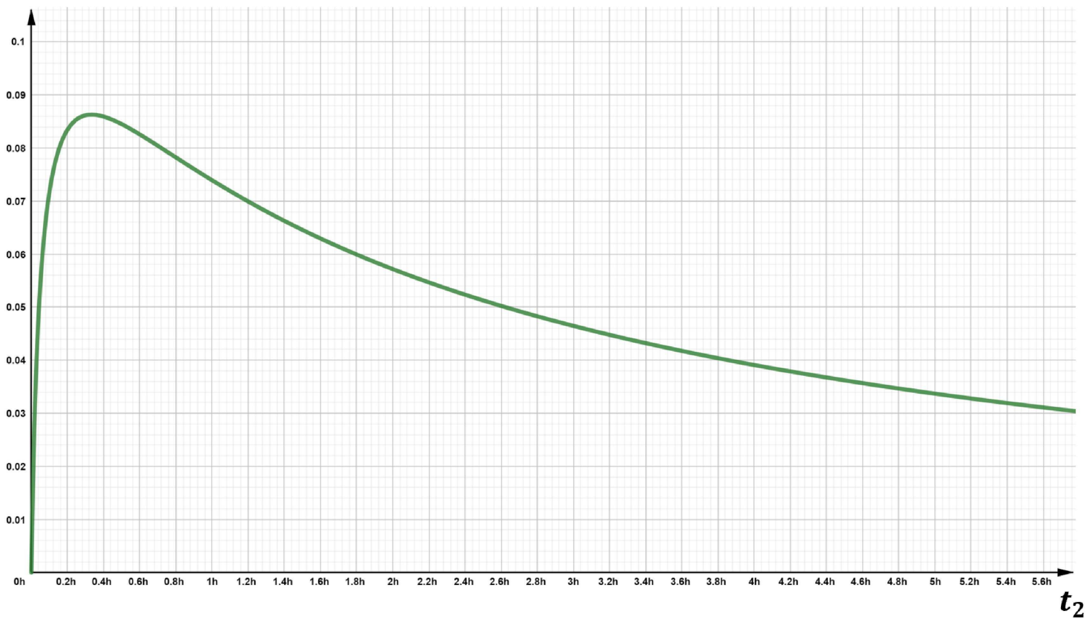


Figura 3.7: Densità di probabilità della distribuzione lognormale di parametri $(\mu, \sigma) = [2.8, 1.9745]$

Funzione *graph*

La funzione *graph* permette all'utente di visualizzare l'andamento della simulazione nel tempo. In particolare permette di visualizzare a schermo, attraverso uno stackplot (vedi fig. 3.8), lo stato ecologico corrente del sistema simulato, ossia il numero e il tipo di batteri in ogni cella ad una determinata iterazione.

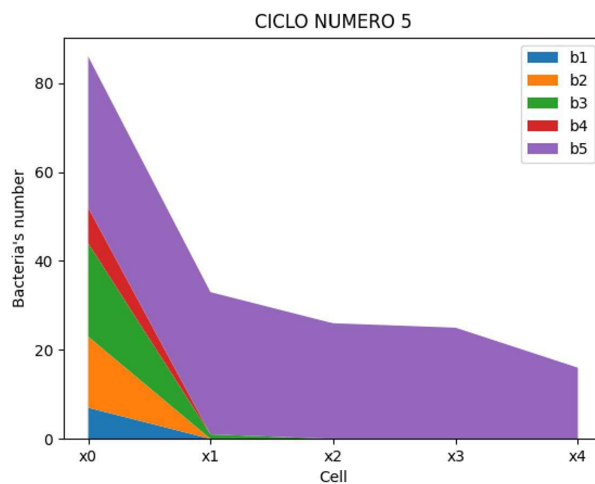


Figura 3.8: Esempio di stackplot di una simulazione

L'algoritmo con il quale questo avviene è il seguente:

1. Innanzitutto viene inizializzato un nuovo dizionario che abbia come chiavi i tipi batterici (stringhe) e come valori le liste delle quantità microbiche corrispondenti nelle diverse celle (vedi esempio 3.3 con 3 celle e 2 specie batteriche);

$$\{\text{'bac1'} : [n_{cella1}, n_{cella2}, n_{cella3}], \text{'bac2'} : [m_{cella1}, m_{cella2}, m_{cella3}]\} \tag{3.3}$$

2. Dal nuovo dizionario vengono successivamente derivate due variabili: un vettore che ne contiene le chiavi ed una matrice che contiene i rispettivi valori. La matrice, di dimensione $len(bac_diz) \times len(\mathbf{x})$, di fatto associa ad ogni riga un tipo batterico e ad ogni colonna una determinata cella (vedi fig. 3.9).

$$\begin{bmatrix} n_{1,0} & \cdots & n_{1,j} \\ \vdots & \ddots & \vdots \\ n_{i,0} & \cdots & n_{i,j} \end{bmatrix}$$

Batterio i → ; ↑ Cella j

Figura 3.9: Matrice da passare come argomento alla funzione stackplot

Il primo vettore viene utilizzato per generare la legenda, mentre la matrice viene passata come argomento alla funzione *stackplot* del modulo *Matplotlib* e il grafico viene mostrato a schermo.

Funzione *graph_met*

La funzione *graph_met* permette all'utente di visualizzare la variazione nel tempo dei metaboliti nel sistema. L'algoritmo con il quale questo avviene sfrutta l'attributo *MatrFood* delle istanze *Cell*, che tiene traccia delle variazione nel tempo dei metaboliti nella cella (vedi fig. 3.10). Ogni colonna j, rappresenta il vettore dei nutrienti $\mathbf{f}$ della cella al tempo j. In particolare la prima colonna rappresenta la distribuzione dei metaboliti al tempo 0, ossia lo stato iniziale. Ogni riga, invece, mostra come varia nel tempo un determinato metabolita. Fornendo in *input* questa matrice alla funzione *plot* del modulo *Matplotlib* viene visualizzato un unico grafico con una curva di colore differente per ogni metabolita. Svolgendo questa operazione più volte, la funzione *graph_met* genera, in un unica figura, un subplot per ogni cella. Questo permette di visualizzare contemporaneamente l'evoluzione dei vari nutrienti nell'intero sistema (vedi fig. 3.11 nella pagina successiva).

$$\begin{bmatrix} f_{1,0} & \cdots & f_{1,t} \\ \vdots & \ddots & \vdots \\ f_{\varphi,0} & \cdots & f_{\varphi,t} \end{bmatrix}$$

Nutriente φ → ; ↑ Iterazione t

Figura 3.10: Attributo MatrFood per il plot delle variazioni temporali dei metaboliti

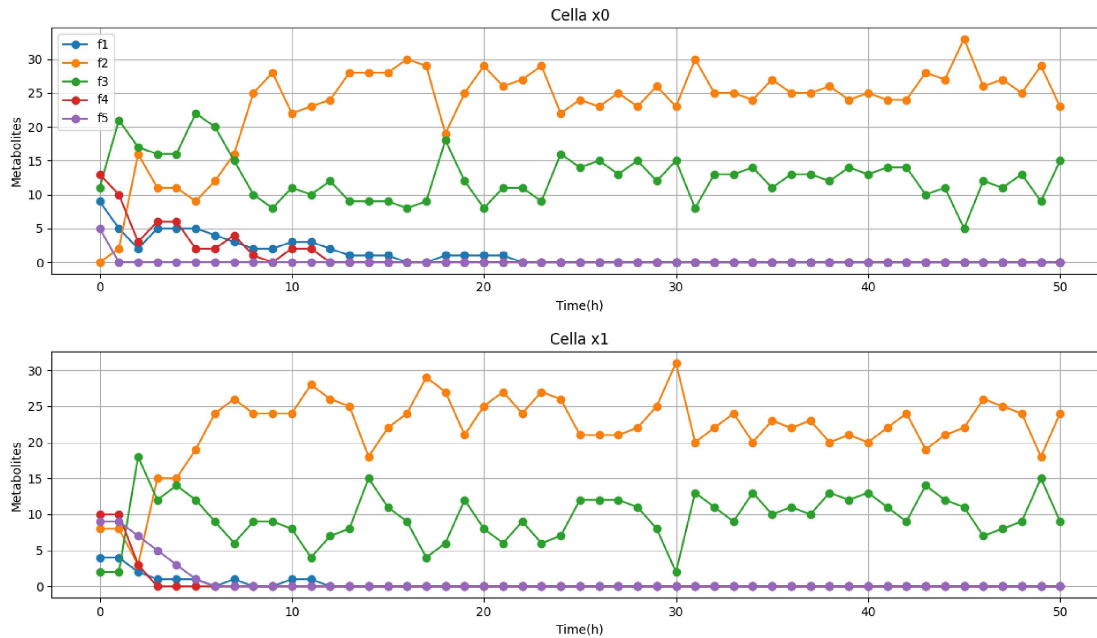


Figura 3.11: Evoluzione nel tempo dei nutrienti in un sistema a due celle e cinque nutrienti

3.3 Raffinamenti del modello

Ogni modello, per definizione, include assunzioni, ipotesi e condizioni tali per cui è sempre possibile un margine di miglioramento. Ad esempio si possono ridurre le ipotesi semplificative, andando a considerare la presenza di fenomeni precedentemente trascurati. Nel nostro caso specifico i raffinamenti effettuati sono stati:

- l'introduzione del fenomeno di tossicità di alcune molecole effettrici;
- l'introduzione del fenomeno di flusso monodirezionale di batteri.

Vediamoli ora nel dettaglio.

3.3.1 Tossicità delle molecole effettrici

La presenza di alcune molecole può generare un incremento della mortalità basale per una determinata specie batterica: questo fenomeno è detto tossicità. A livello di codice l'implementazione si è suddivisa in diverse parti:

1. è stata aggiornata la classe *Bact* con l'aggiunta di due attributi, ossia il coefficiente di tossicità massima μ_t^{max} e il vettore delle tossicità $\mathbf{t}$, e il metodo *getTox*, che calcola il coefficiente di tossicità effettiva μ_t.
2. di conseguenza è stata aggiornata la funzione *randomBacts*, in modo da generare casualmente anche i nuovi attributi aggiunti;
3. è stato aggiornato il metodo *death*, sulla base del nuovo fenomeno considerato.

Tossicità nella funzione *randomBacts*

La necessità di due parametri aggiuntivi nell'inizializzazione di un istanza della classe *Bact* si riflette nella funzione accessoria *randomBacts*. Il coefficiente di tossicità massima, uguale per ogni specie, è stato impostato di default a $\mu_t^{max} = 0.973$. Questo valore è stato deciso in modo da ottenere il valore unitario come estremo superiore del coefficiente moltiplicativo nella funzione 3.5 nella pagina seguente: questo implica che, per ogni specie i, il numero di batteri morti d_i non possa superare il numero di batteri presenti nella cella n_i, quindi vale sempre $d_i \leq n_i$. Il vettore delle tossicità $\mathbf{t}$, invece, è creato in maniera casuale per ogni specie a partire dal vettore dei metabolismi $\mathbf{m}$: nello specifico, ogni nutriente ignorato o prodotto ha il 5% di probabilità di causare fenomeni di tossicità. Il risultato è un vettore $\mathbf{t}$ di 1 e 0, indicanti rispettivamente nutrienti tossici e non tossici, della stessa lunghezza del vettore dei nutrienti $\mathbf{f}$ (vedi fig. 3.12). Come per il vettore dei metabolismi questa scelta è stata fatta per semplicità: è possibile, dunque, impostare anche tossicità di modulo superiore ad 1, ma che siano pur sempre numeri interi.

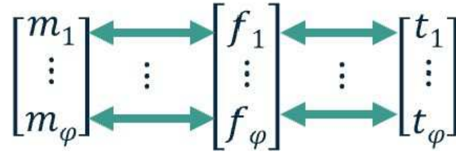


Figura 3.12: Relazione biunivoca tra i vettori dei metabolismi **m** e delle tossicità **t** della classe *Bact* e il vettore dei nutrienti della cella **f**

Metodo getTox

Il metodo *getTox* è un metodo della classe *Bact* che calcola e restituisce in uscita il coefficiente di tossicità della specie in una determinata cella μ_t. Siano n_{bac} e $\mathbf{t}$ rispettivamente il numero di batteri e il vettore delle tossicità della specie, $\mathbf{f}$ il vettore dei nutrienti della cella e μ_t^{max} il coefficiente di tossicità massima. Sia t_{tot} la quantità complessiva di nutrienti tossici per la specie, calcolata come $t_{tot} = \mathbf{f} \cdot \mathbf{t}$ e siano km ed η due parametri della curva impostati di default a $km = 3500$ ed $\eta = 1$. Allora il coefficiente di tossicità è dato dalla formula:

$$\mu_t = \begin{cases} \dfrac{\mu_t^{max}}{1 + \left(\dfrac{km \cdot n_{bac}}{t_{tot}}\right)^{\eta}} & \text{se } t_{tot} > 0 \\ 0 & \text{se } t_{tot} = 0 \end{cases} \tag{3.4}$$

La scelta di utilizzare una funzione di Hill (vedi app. A a pagina 41) deriva da considerazioni empiriche analoghe a quelle fatte per il metodo *getgrowth* (vedi par. 3.2.2 a pagina 10). Specificatamente:

- in assenza di sostanze tossiche per la specie il suo coefficiente di tossicità risulta nullo;
- all'aumentare di t_{tot} il coefficiente di tossicità aumenta fino al valore di saturazione μ_t^{max};

- all'aumentare di n_{bac} sono necessarie più sostanze tossiche per raggiungere lo stesso coefficiente di tossicità.

Tossicità nel metodo *death*

L'incremento della mortalità dovuto alla tossicità viene introdotto nel simulatore attraverso il metodo *death* della classe *Cell*, nel quale viene modificata la formula per il calcolo dei batteri inattivati (vedi eq. 3.2 a pagina 13). In particolare viene introdotto il coefficiente di tossicità della specie i $\mu_{t,i}$ calcolato attraverso il metodo *getTox*. Il numero di batteri morti nella cella risulta quindi essere:

$$d_i(n_i) = \lceil (\lambda + \mu_{t,i}) \cdot n_i \rceil \quad \forall i \tag{3.5}$$

3.3.2 Flusso microbico

I batteri possono trovarsi a generare un flusso netto in una determinata direzione sotto l'azione di una forza motrice esterna. Nel nostro modello di ambiente lineare a celle successive si è scelto di simulare un flusso nella direzione crescente degli indici delle celle. Questo fenomeno agisce sinergicamente con la migrazione delle specie nella stessa direzione dovuta allo smistamento dei batteri generati dalla duplicazione (si veda punto 4 del par. 3.2.2 a pagina 14). A livello di codice il fenomeno è stato implementato attraverso il metodo *flow* della classe *Cell*.

Metodo *flow*

Il metodo in questione calcola e restituisce in uscita il numero di batteri della specie i n_i^{flow} che sarà trasferito nella cella successiva in funzione del numero di batteri della stessa specie presenti nella cella all'inizio del ciclo n_i. Sia ρ la percentuale di batteri che fluisce, allora :

$$n_i^{flow} = [\rho \cdot n_i] \quad \forall i \tag{3.6}$$

Si noti che questo metodo viene invocato ad ogni ciclo, in quanto viene richiamato nel metodo *evolution* con il valore di *default* $\rho = 0.05$.

Aggiornamento del metodo *evolution*

Per far fronte a questo fenomeno, l'algoritmo del metodo *evolution* della classe *Cell* ha subito delle modifiche. Innanzitutto viene inizializzato attraverso un ciclo *for* un dizionario che abbia come chiavi gli oggetti *Bact* e come valori i numeri corrispondenti di batteri da trasferire nella cella successiva n_i^{flow}. Lo spostamento effettivo avviene sottraendo tali quantità dal dizionario dei batteri della cella corrente e aggiungendole a quello della cella successiva tramite il metodo *addBact* sviluppato appositamente. Inoltre, questo algoritmo di trasferimento è analogamente utilizzato per lo spostamento alla cella successiva dei batteri neo-nati. Si ricordi che nel caso si stia considerando l'ultima cella, i batteri da trasferire vengono eliminati, simulandone l'uscita dal sistema (vedi fig. 3.6 a pagina 16).

Considerando questi ulteriori fenomeni, il numero di batteri della specie i-esima al tempo t nella cella j è definito come:

$$n_{i,j}(t+1) = n_{i,j}(t) - d_{i,j}(t) + \left(\frac{n_{i,j}^{new}(t)}{2} + \frac{n_{i,j-1}^{new}(t)}{2} \right) + \left(n_{i,j-1}^{flow}(t) - n_{i,j}^{flow}(t) \right) \quad \forall i,j \tag{3.7}$$

Capitolo 4

Simulazioni significative

Dal momento che il livello di conoscenza attuale non consente di determinare con esattezza tutti i parametri necessari a specificare le caratteristiche delle specie batteriche, noi non siamo in grado di definire in modo preciso gli attributi degli agenti.
Per questo, al fine di collaudare il nostro simulatore, abbiamo generato delle specie batteriche aleatorie, i cui parametri sono stati definiti basandosi su considerazioni di carattere statistico e dati medi disponibili in letteratura.
Dopo aver valutato un ampio numero di simulazioni, ne ho selezionate alcune che descrivono possibili comportamenti di comunità batteriche e che presentano delle caratteristiche particolarmente interessanti e non triviali.

4.1 Ambiente ostile

Da una comunità composta da specie microbiche con tassi di crescita massimi piuttosto bassi, ci si aspetta che le dimensioni delle diverse popolazioni fatichino ad aumentare. Questo è proprio quello che si evince da questa simulazione: come possiamo vedere, infatti, in meno di 30 ore la comunità si è estinta (vedi fig. 4.1 a pagina 27). Solitamente è la specie con il tasso di crescita più elevato ad avere il sopravvento, ossia b_5 in questo caso specifico.
Tuttavia, come possiamo notare, questo non accade: uno dei motivi è sicuramente l'alta concentrazione del metabolita f_5 nella cella x_1, tossico per quella specie. Dal grafico dei metaboliti, inoltre, si può notare come i nutrienti nelle celle x_2, x_3 e x_4 rimangano inalterati fino al termine della simulazione: questo perché la comunità non riesce mai a popolare quelle celle e a consumarne le risorse.
Le altre popolazioni, invece, non riescono a nutrirsi abbastanza per raggiungere un tasso di crescita sufficientemente elevato da superare il tasso di morte basale, andando incontro ad una rapida estinzione.

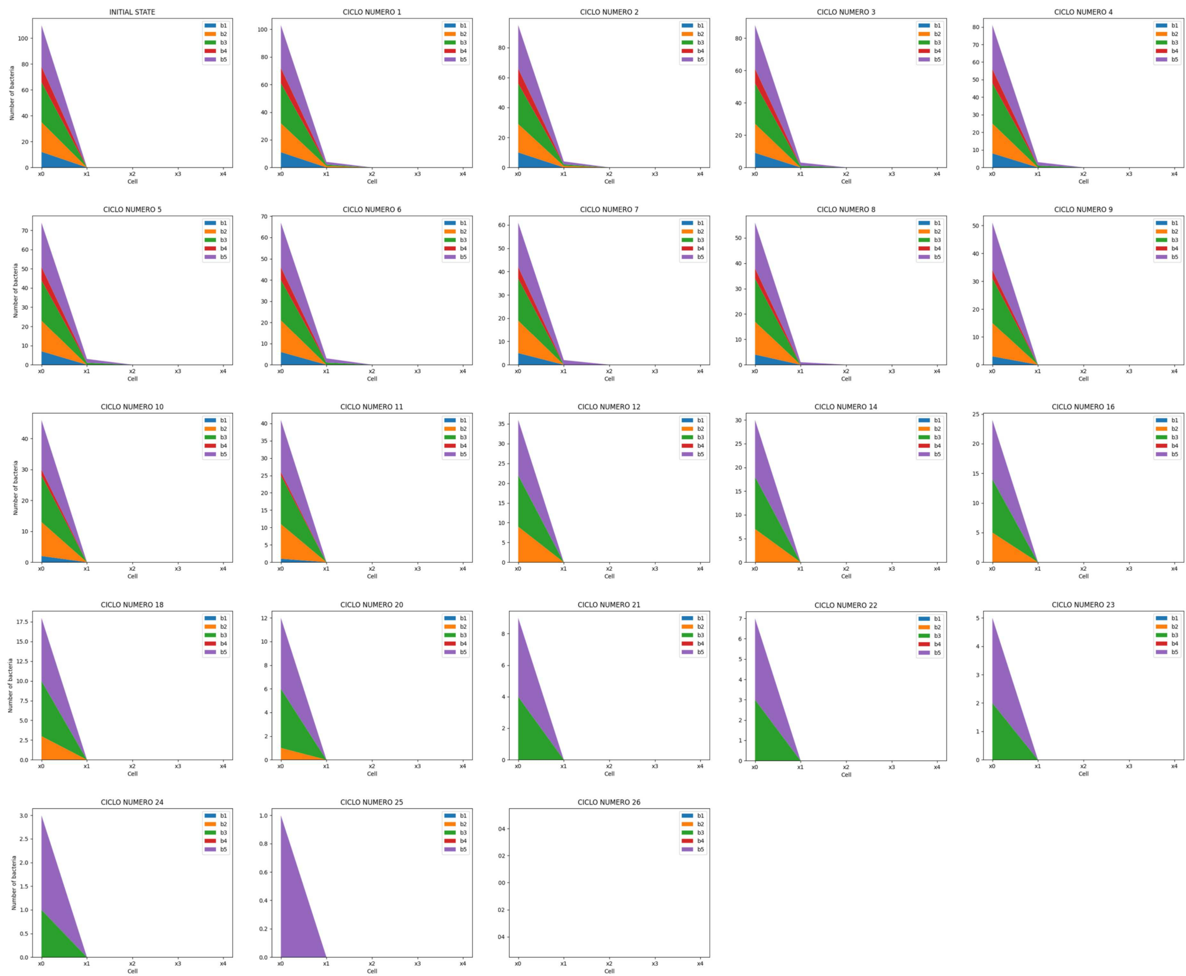
4.2 Ambiente favorevole

In questa simulazione sorgono diversi aspetti importanti. Dai grafici si evince subito che la specie b_4 ha delle caratteristiche vincenti, che le permettono in poche ore di prendere il sopravvento.
In primis, possiamo notare che il suo tasso di crescita massimo sia molto maggiore rispetto agli altri. Inoltre, dal grafico dei metaboliti vediamo come dalla cella x_2 si sviluppi un trend positivo molto forte: questo è dovuto al fatto che in quella cella sono presenti in grandi quantità i metaboliti f_1 ed f_5, consumati proprio dalla specie b_4. Al terzo ciclo, infatti, con l'arrivo della specie nella cella x_2, si può notare nel grafico dei metaboliti un crollo repentino dei due nutrienti.
Nella cella, conseguentemente alla produzione della specie b_4, vi è un aumento della concentrazione dei metaboliti f_2, f_3 ed f_4. Si noti, però, che già al quarto ciclo la loro quantità rimane costante: la specie b_4, infatti, non avendo più nutrienti da consumare, non può produrre molecole effettrici (vedi fig. 4.2 a pagina 28).
Le altre specie, invece, oltre ad avere dei tassi di crescita massimi più bassi, hanno anche più difficoltà a consumare i metaboliti: essendo ben più numerosa, la specie b_4 consuma la maggior parte dei nutrienti, limitando la possibilità delle altre specie di nutrirsi. Questo fenomeno è detto competizione.

4.3 Sintrofia

In questa simulazione vedremo due popolazioni della comunità instaurare un rapporto di sintrofia, ossia quel particolare tipo di interazione per la quale una specie batterica consuma i metaboliti prodotti da un'altra e viceversa. Questo aspetto si può osservare confrontando i vettori dei metabolismi delle specie b_3 e b_5, ma anche dal grafico dei metaboliti (vedi fig. 4.3 a pagina 29).
Inoltre, dopo un determinato intervallo di tempo, le dimensioni delle due popolazioni risulteranno essere in un rapporto circa costante, come si può osservare negli ultimi pannelli.
La cosa interessante è che questo tipo di interazione non è definita a priori, come accade in altri modelli, ma risulta essere una proprietà emergente da meccanismi indipendenti (metabolismi).
La principale criticità che sorge da questa simulazione, invece, è il fatto che nel modello abbiamo considerato che la totalità dei nutrienti consumati venisse utilizzata per la produzione di molecole effettrici, trascurando la metabolizzazione delle risorse per lo sviluppo cellulare. Questo è ben visibile nel grafico: i metaboliti (arancione e verde) oscillano nel tempo mantenendo, però, la loro somma costante.
Questo permette alle specie b_3 e b_5 di continuare a nutrirsi e sopravvivere, senza mai andare incontro all'estinzione.

```
METABOLISMI:
b1  :  [1, -1, 0, 0, -1] , mu_max=[0.00149675] , t= [1, 0, 0, 0, 0]
b2  :  [1, 0, 0, -1, 0] , mu_max=[0.1057667] , t= [0, 0, 0, 0, 0]
b3  :  [1, 1, -1, -1, 1] , mu_max=[0.03950546] , t= [0, 0, 0, 0, 0]
b4  :  [1, 1, -1, 0, 0] , mu_max=[0.01885167] , t= [0, 0, 0, 0, 0]
b5  :  [1, -1, -1, 1, 1] , mu_max=[0.20013075] , t= [0, 0, 0, 0, 1]
```



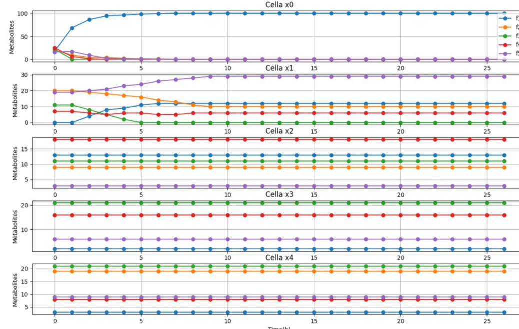


Figura 4.1: Simulazione in ambiente ostile

```
METABOLISMI:
b1  :  [0, -1, 0, 1, -1] , mu_max=[0.00735953] , t= [0, 0, 0, 0, 0]
b2  :  [-1, 1, -1, -1, -1] , mu_max=[0.02863148] , t= [0, 0, 0, 0, 0]
b3  :  [-1, -1, -1, 1, 0] , mu_max=[0.02080694] , t= [0, 0, 0, 0, 0]
b4  :  [-1, 1, 1, 1, -1] , mu_max=[4.30706937] , t= [0, 0, 0, 0, 0]
b5  :  [-1, 1, -1, 0, 0] , mu_max=[0.35897197] , t= [0, 0, 0, 0, 0]
```

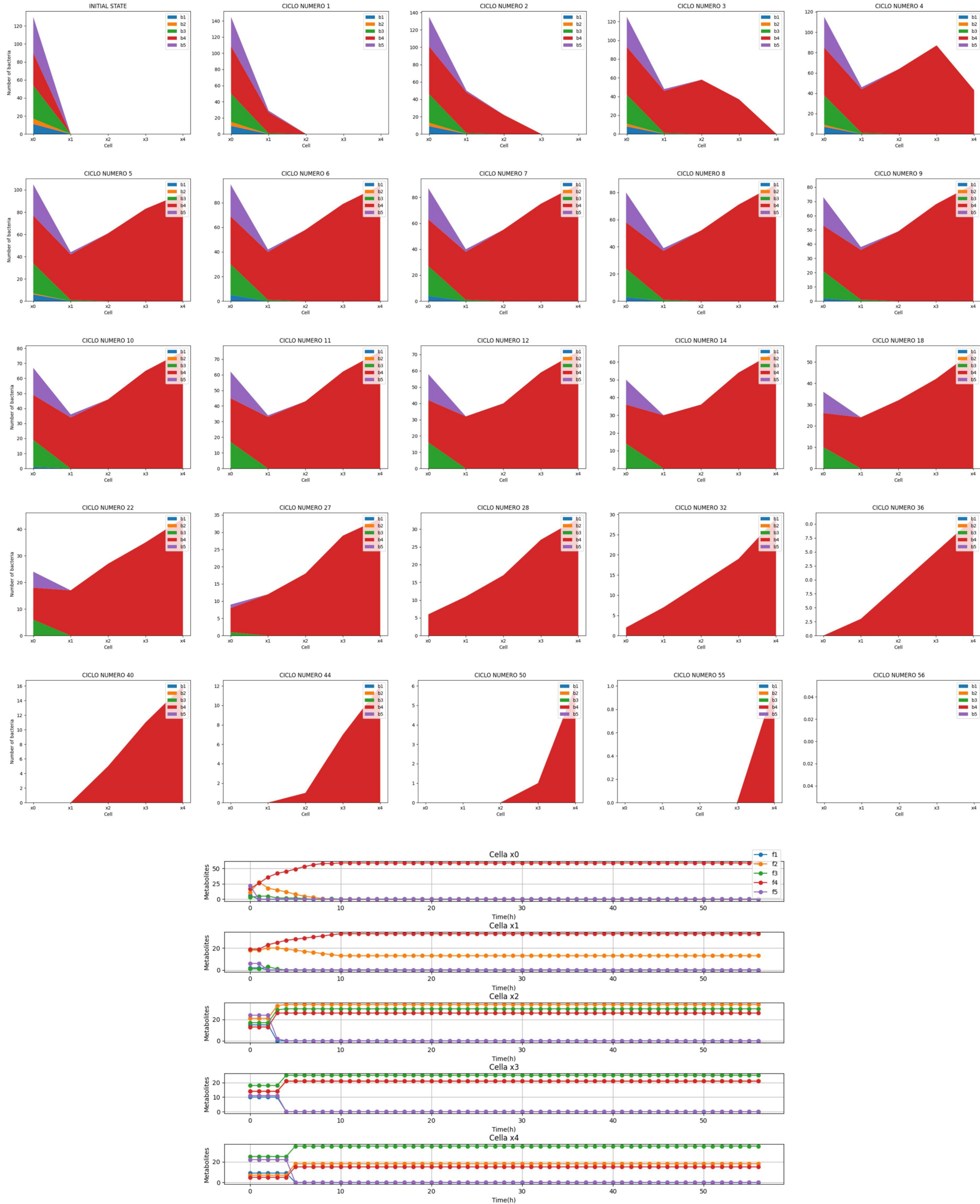


Figura 4.2: Simulazione in ambiente favorevole

```
METABOLISMI:
b1  :  [1, 0, 1, 0, -1] , mu_max=[0.09597913] , t= [0, 0, 0, 0, 0]
b2  :  [-1, -1, 0, 1, -1] , mu_max=[0.04710321] , t= [0, 0, 0, 0, 0]
b3  :  [-1, 1, -1, 0, 0] , mu_max=[2.86167257] , t= [0, 0, 0, 0, 0]
b4  :  [-1, -1, 1, 1, 0] , mu_max=[0.00165289] , t= [0, 0, 0, 1, 0]
b5  :  [-1, -1, 1, -1, 0] , mu_max=[1.45444683] , t= [0, 0, 0, 0, 0]
```

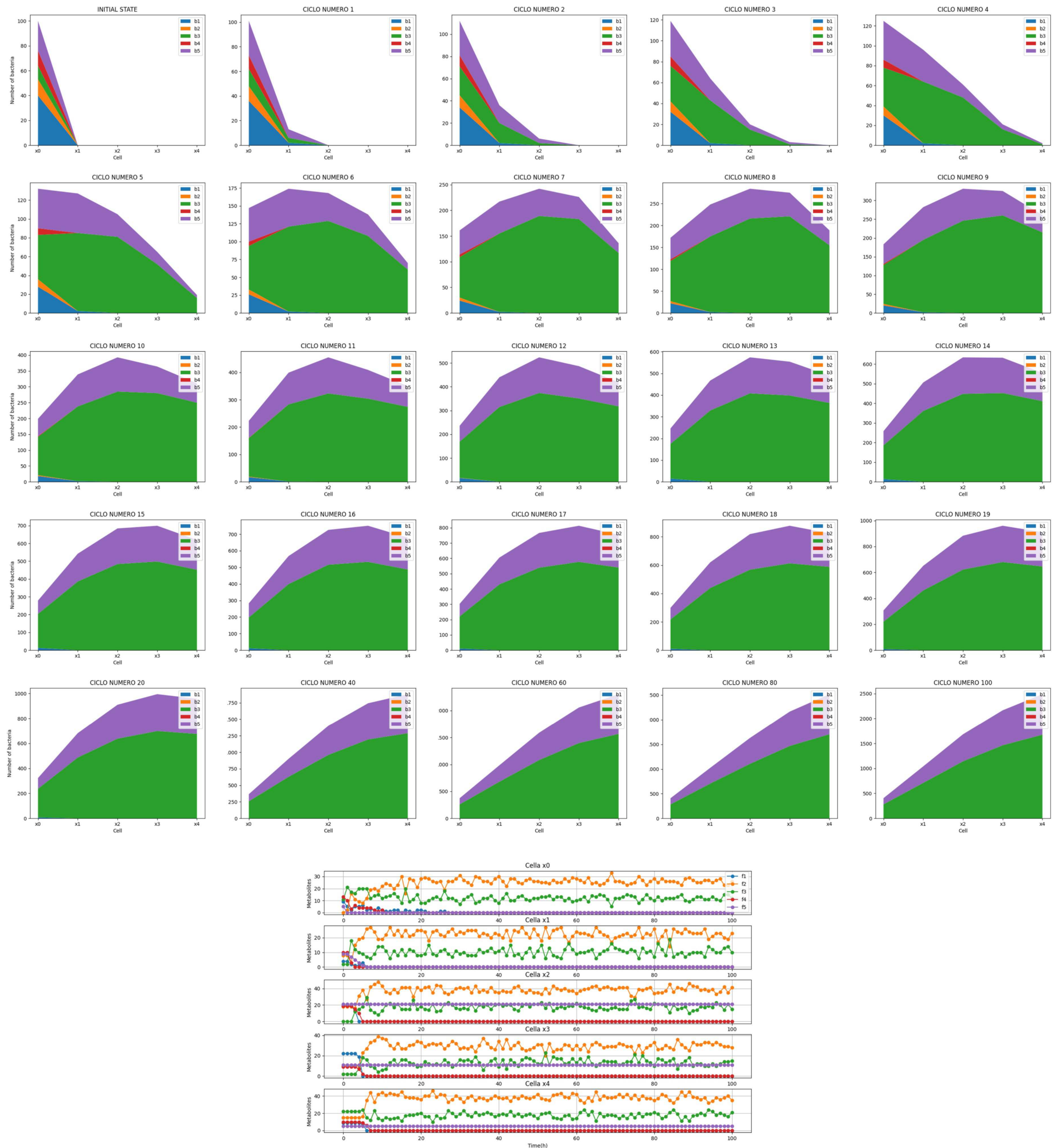


Figura 4.3: Simulazione con popolazioni in rapporto sintrofico

Capitolo 5

Ulteriori raffinamenti e modellizzazioni

Dal punto di vista spaziale il nostro modello permette di descrivere comunità microbiche in un ambiente lineare, ossia con una dimensione prevalente sulle altre, implementato attraverso una coda di celle spaziali concatenate dentro cui i batteri possono muoversi monodimensionalmente. All'interno di questo ambiente le singole popolazioni si moltiplicano sulla base delle molecole effettrici, come metaboliti e tossine presenti nelle vicinanze, ma indipendentemente dalle altre specie. Ciò non significa che una specie batterica avrebbe la stessa evoluzione in assenza di altre specie: queste, infatti, apportano delle modifiche per quanto riguarda la disponibilità delle molecole effettrici, attraverso processi come la sintrofia o la competizione. Il modello, inoltre, prevede un flusso monodirezionale del microbiota dovuto, ad esempio, ad una forza motrice esterna (peristalsi).

Comunque, come ogni modello, esso presenta un certo livello di astrazione: alcuni fenomeni sono stati semplificati, mentre altri sono stati addirittura trascurati.

5.1 Raffinamenti

Innanzitutto si potrebbero rendere specifici per ogni specie batterica quei parametri che, nel modello, sono stati considerati generali per tutta la comunità batterica. In particolare si fa riferimento al tasso di morte basale e ai parametri km ed η delle equazioni di Hill per il calcolo del tasso di crescita e del coefficiente di tossicità.

In secondo luogo sarebbe utile rendere l'implementazione spaziale più generale, in modo da consentire delle simulazioni in due o tre dimensioni. Questo causerebbe la necessità di definire nuove regole di movimento, eventualmente basate su considerazioni di carattere stocastico. Oltretutto, il movimento dei batteri è legato anche, oltre che agli agenti meccanici esterni quali peristalsi, alla chemiotassi, ossia la propensione a migrare verso "zone" più ricche di molecole "utili", come nutrienti o molecole segnale secrete da batteri della stessa specie.

Per quanto riguarda le molecole effettrici, potrebbe essere utile introdurre la possibilità di avere una certa dinamica di nutrienti e tossine, ad esempio dovuta ad un flusso di materia esterno, quale l'ingestione di un bolo alimentare e la conseguente attivazione dei moti peristalti-

ci intestinali. In aggiunta si potrebbe pensare di permettere l'approvvigionamento di queste molecole durante la simulazione stessa.

Un altro concetto che sarebbe importante implementare è l'utilizzo di energia estratta dai nutrienti per la crescita della specie. Nel modello attuale si considera solamente il principio di conservazione di massa delle molecole effettrici: il totale dei nutrienti consumati viene interamente suddiviso tra i metaboliti prodotti, trascurando che una parte di materia viene trasformata in energia per la crescita dei singoli microbi, il loro movimento ed il metabolismo stesso dei nutrienti.

5.2 Ulteriori modellizzazioni

Un altro fenomeno interessante che potrebbe aver senso modellizzare è il *quorum sensing*, un processo di comunicazione tra cellule che fa affidamento a molecole segnale chiamate autoinduttori. Raggiunta una concentrazione di soglia, gli autoinduttori vengono riconosciuti da specifici recettori che attivano un processo di regolazione dell'espressione genica. Questo processo, quindi, permette a gruppi di batteri di mutare il proprio comportamento in risposta a cambiamenti nella densità di popolazione [20]. Ad esempio, questo fenomeno permette un aumento di resistenza antibiotica, di virulenza, di scambio di materiale genetico, regola la bioluminescenza e permette la formazione di biofilm [21, 22].

Si potrebbe, inoltre, pensare di introdurre dei meccanismi di mutazione, sfruttando ad esempio l'utilizzo di algoritmi genetici (GA), un approccio euristico applicabile a molti problemi di ottimizzazione. I GA sono ispirati ai processi evolutivi e alla selezione naturale: infatti, questo tipo di approccio si basa sull'apportare piccole variazioni alle variabili in gioco e verificare se queste nuove variabili diano dei risultati "migliori" per il problema proposto. In tal caso si cancellano le vecchie variabili, e si continua il processo con le nuove verso un valore di ottimo locale. Con questo tipo di implementazione, si potrebbe sfruttare il simulatore per arrivare a progettare l'ingegnerizzazione di comunità che abbiano delle caratteristiche ottimali per determinati parametri ambientali.

Infine, in futuro si potrebbero integrare dati genomici e metagenomici (ed -omici in generale) per assegnare valori fisiologici alle caratteristiche dei diversi agenti. Si passerebbe, quindi, da un approccio *knowledge-driven* ad uno ibrido *knowledge-data-driven*, ottenendo un simulatore che potrebbe impostare i parametri delle diverse specie microbiche a partire da una banca dati in continuo aggiornamento.

Capitolo 6

Conclusioni

In primis, questo manoscritto si propone di illustrare lo stato dell'arte della modellizzazione per comunità microbiche, con particolare attenzione per i modelli Agent-Based (ABM).

Viene quindi presentato come, sotto la guida dei miei supervisori, ho progettato e sviluppato in Python lo scheletro di un simulatore altamente modulare per comunità microbiche basato su un ABM. In particolare viene trattato il modello di base al quale, in seguito, abbiamo aggiunto due *features*, quali la tossicità dei metaboliti e il flusso microbico. Tale modello riceve in ingresso diversi parametri, tra cui in particolare: il numero di possibili specie batteriche, il numero di diversi metaboliti possibili, i metabolismi che definiscono le relazioni metabolita-specie batterica e le quantità iniziali di batteri e nutrienti nelle varie celle dello spazio monodimensionale considerato. Analizzando l'uscita, ossia l'evoluzione spazio-temporale della comunità batterica e dei metaboliti, è stato possibile determinare il tipo di interazione ecologica instauratasi nell'ecosistema, come ad esempio commensalismo e ostilità ambientale, solamente sulla base della conoscenza delle relazioni metabolita-agente (microbo).

Infine vengono proposte alcune proiezioni future: da semplici raffinamenti di tipo spaziale, sia dal punto di vista della più corretta migrazione di batteri e di metaboliti (attualmente assente) che sulla specifica struttura dell'ambiente colonizzato (attualmente una sorta di tubo monodimensionale), fino ad arrivare a complessi processi biologici come il *quorum sensing*.

Nonostante la carenza di dati in letteratura e il fatto che il simulatore sia ancora in una sua versione preliminare, siamo riusciti ad ottenere comportamenti possibili di comunità batteriche. Tuttavia manca un'effettiva validazione, che potrebbe essere svolta attraverso una sperimentazione diretta su una comunità ristretta i cui parametri siano ottenibili da database curati (come *BacDive*), con successiva analisi mediante HPLC-MS (cromatografia e spettrometria di massa). Oppure si potrebbero comparare i risultati con quelli ottenuti da un altro simulatore, come ad esempio quello che ha ispirato il progetto [23].

In ogni caso, una validazione a questo punto dello sviluppo sarebbe ancora parzialmente attendibile: il simulatore, infatti, è stato progettato per un utilizzo futuro, ossia per quando sarà disponibile la tecnologia per ottenere la conoscenza necessaria a definire tutti i parametri di un modello meccanicistico *knowledge-based* come il nostro. Per contro, un simulatore come quello presentato in questo manoscritto potrebbe tornare utile, già al suo stato di parziale implementazione, per poter studiare dal punto di vista prettamente modellistico quali combinazioni

di parametri o *pattern* specifici siano *features* determinanti per poter predire una particolare evoluzione del sistema. Tale studio esula dallo scopo di questa tesi, ma potrebbe essere uno spunto interessante per attività di tesi future.

In ultimo, il simulatore verrà a breve reso un pacchetto Python con un interfaccia *user-friendly*, che ne consenta l'utilizzo anche ad utenti con ridotte competenze informatiche.

Bibliografia

[1] Peter J Turnbaugh, Micah Hamady, Tanya Yatsunenko, Brandi L Cantarel, Alexis Duncan, Ruth E Ley, Mitchell L Sogin, William J Jones, Bruce A Roe, Jason P Affourtit, et al. A core gut microbiome in obese and lean twins. *nature*, 457(7228):480–484, 2009.

[2] Peng Zheng, B Zeng, C Zhou, M Liu, Z Fang, X Xu, L Zeng, J Chen, S Fan, X Du, et al. Gut microbiome remodeling induces depressive-like behaviors through a pathway mediated by the host's metabolism. *Molecular psychiatry*, 21(6):786–796, 2016.

[3] Diwakar Davar, Amiran K Dzutsev, John A McCulloch, Richard R Rodrigues, Joe-Marc Chauvin, Robert M Morrison, Richelle N Deblasio, Carmine Menna, Quanquan Ding, Ornella Pagliano, et al. Fecal microbiota transplant overcomes resistance to anti–pd-1 therapy in melanoma patients. *Science*, 371(6529):595–602, 2021.

[4] Ryan L Clark, Bryce M Connors, David M Stevenson, Susan E Hromada, Joshua J Hamilton, Daniel Amador-Noguez, and Ophelia S Venturelli. Design of synthetic human gut microbiome assembly and butyrate production. *Nature communications*, 12(1):1–16, 2021.

[5] Jason H Yang, Sarah N Wright, Meagan Hamblin, Douglas McCloskey, Miguel A Alcantar, Lars Schrübbers, Allison J Lopatkin, Sangeeta Satish, Amir Nili, Bernhard O Palsson, et al. A white-box machine learning approach for revealing antibiotic mechanisms of action. *Cell*, 177(6):1649–1661, 2019.

[6] Karna Gowda, Derek Ping, Madhav Mani, and Seppe Kuehn. A sparse mapping of structure to function in microbial communities. *bioRxiv*, 2020.

[7] Yili Qian, Freeman Lan, and Ophelia S Venturelli. Towards a deeper understanding of microbial communities: integrating experimental data with dynamic models. *Current Opinion in Microbiology*, 62:84–92, 2021.

[8] Vanni Bucci, Belinda Tzen, Ning Li, Matt Simmons, Takeshi Tanoue, Elijah Bogart, Luxue Deng, Vladimir Yeliseyev, Mary L Delaney, Qing Liu, et al. Mdsine: Microbial dynamical systems inference engine for microbiome time-series analyses. *Genome biology*, 17(1):1–17, 2016.

[9] Richard R Stein, Vanni Bucci, Nora C Toussaint, Charlie G Buffie, Gunnar Rätsch, Eric G Pamer, Chris Sander, and Joao B Xavier. Ecological modeling from time-series inference:

insight into dynamics and stability of intestinal microbiota. *PLoS computational biology*, 9(12):e1003388, 2013.

[10] Anthony R Ives, Brian Dennis, Kathryn L Cottingham, and Stephen R Carpenter. Estimating community stability and ecological interactions from time-series data. *Ecological monographs*, 73(2):301–330, 2003.

[11] Sean M Gibbons, Sean M Kearney, Chris S Smillie, and Eric J Alm. Two dynamic regimes in the human gut microbiome. *PLoS computational biology*, 13(2):e1005364, 2017.

[12] Benjamin J Ridenhour, Sarah L Brooker, Janet E Williams, James T Van Leuven, Aaron W Miller, M Denise Dearing, and Christopher H Remien. Modeling time-series data from microbial communities. *The ISME journal*, 11(11):2526–2537, 2017.

[13] Robert MacArthur. Species packing and competitive equilibrium for many species. *Theoretical population biology*, 1(1):1–11, 1970.

[14] Chen Liao, Tong Wang, Sergei Maslov, and Joao B Xavier. Modeling microbial cross-feeding at intermediate scale portrays community dynamics and species coexistence. *PLoS computational biology*, 16(8):e1008135, 2020.

[15] Daniel A Medina, Francisco Pinto, Veronica Ortuzar, and Daniel Garrido. Simulation and modeling of dietary changes in the infant gut microbiome. *FEMS microbiology ecology*, 94(9):fiy140, 2018.

[16] Jeffrey D Orth, Ines Thiele, and Bernhard Ø Palsson. What is flux balance analysis? *Nature biotechnology*, 28(3):245–248, 2010.

[17] Ashley Byrne, Charles Cole, Roger Volden, and Christopher Vollmers. Realizing the potential of full-length transcriptome sequencing. *Philosophical Transactions of the Royal Society B*, 374(1786):20190097, 2019.

[18] Francesco Bertolotti and Luca Mari. Agent-based modeling: un'analisi critica della bibliografia e delle relazioni con la teoria generale dei sistemi. Technical report, LIUC Universita Carlo Cattaneo, 2018.

[19] Ping Wang, Lydia Robert, James Pelletier, Wei Lien Dang, Francois Taddei, Andrew Wright, and Suckjoon Jun. Robust growth of escherichia coli. *Current biology*, 20(12):1099–1103, 2010.

[20] Sampriti Mukherjee and Bonnie L Bassler. Bacterial quorum sensing in complex and dynamically changing environments. *Nature Reviews Microbiology*, 17(6):371–382, 2019.

[21] Melphine M. Harriott. Biofilms and antibiotics. In *Reference Module in Biomedical Sciences*. Elsevier, 2019.

[22] Marijke Frederix and J. Allan Downie. Chapter 2 - quorum sensing: Regulating the regulators. In Robert K. Poole, editor, *Advances in Microbial Physiology*, volume 58 of *Advances in Microbial Physiology*, pages 23–80. Academic Press, 2011.

[23] Robert Marsland, Wenping Cui, Joshua Goldford, and Pankaj Mehta. The community simulator: A python package for microbial ecology. *Plos one*, 15(3):e0230430, 2020.

Ringraziamenti

Un sentito grazie a tutti coloro che mi hanno permesso di arrivare fin qui e di portare a termine questo lavoro di tesi.

In primis, un ringraziamento speciale va al mio relatore Ing. Massimo Bellato, sempre presente, puntuale e disponibile. Grazie al percorso intrapreso insieme ho sviluppato maggiormente le mie capacità di analisi e *problem solving*, capacità che trascendono la carriera prettamente universitaria e che mi saranno sicuramente di grande aiuto nel mondo del lavoro.

Ringrazio i miei correlatori, il Dott. Marco Cappellato e la Prof. Barbara Di Camillo, per aver partecipato attivamente al progetto attraverso preziosi consigli e suggerimenti.

Non posso non menzionare mia nonna e i miei genitori che da sempre mi sostengono nella realizzazione dei miei sogni e progetti. Non finirò mai di ringraziarvi per avermi permesso di arrivare fin qui.

Grazie anche ai miei amici per essermi sempre stati vicino, anche durante questa ultima fase del mio percorso di studi. Grazie per tutti i momenti di gioia e spensieratezza passati assieme, per avermi supportato nei momenti più bui, ma soprattutto per avermi ispirato ad essere la migliore versione di me stesso.

Infine, dedico questa tesi a me stesso, ai miei sacrifici e alla mia tenacia che mi hanno permesso di raggiungere questo traguardo che spero sia il primo di una lunga serie.

Appendice A

Equazione di Hill

Un'equazione di Hill è una funzione a valori reali del tipo $y(x) = \frac{y_{max}}{1+\left(\frac{km}{x}\right)^{\eta}}$ dove:

- y_{max} indica il valore massimo asintotico della curva, detto anche valore di saturazione;
- km indica lo spostamento sull'asse delle ascisse della e, in particolare, è l'ascissa per la quale la funzione assume il valore $\frac{y_{max}}{2}$;
- η è un indicatore della pendenza con la quale la funzione raggiunge il suo valore di saturazione.

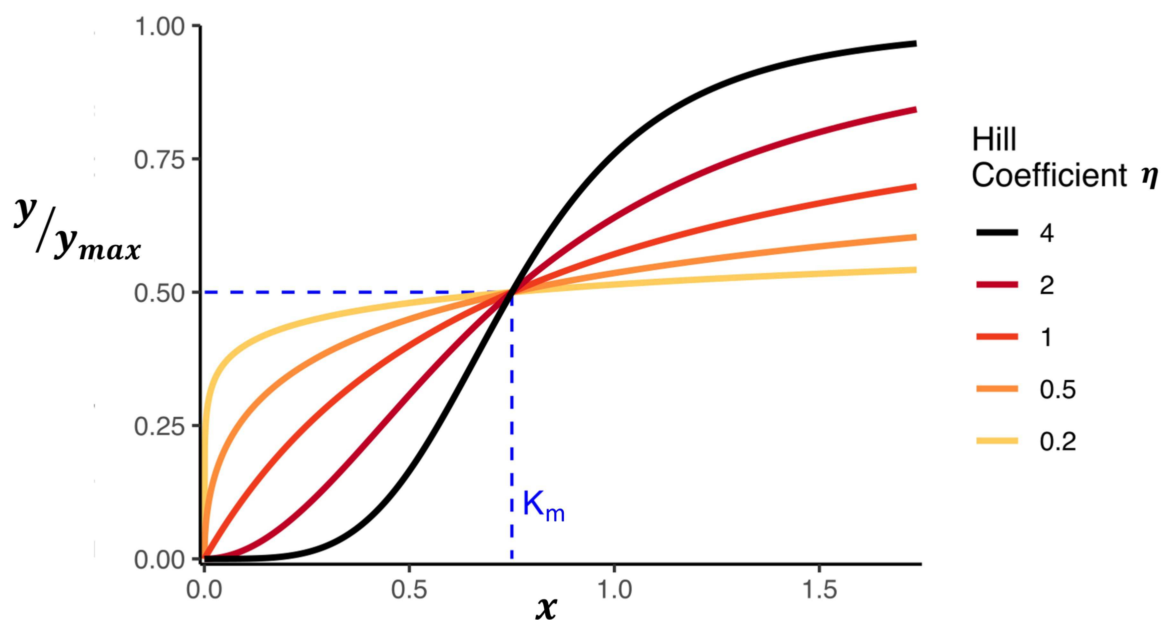


Figura A.1: Equazioni di Hill a parametro Km costante

Appendice B

Codice Python del simulatore

Listing B.1: Codice del simulatore

```
#AGENT-BASED SIMULATOR FOR MICROBIAL COMMUNITIES
#Andrea Calzavara, April 2022
class Bact:
    _species=['b1','b2','b3','b4','b5']

    def __init__(self,typ,m,t,maxGr,maxTox,pos):
        self._type=typ#type from Bact._species
        self._m=m #metabolism vector {-1=consume, 0=ignore, 1=produce}
        self._t=t #toxicity vector {0=ignored,1=toxic}
        self._maxGr=maxGr # max growth rate(float)
        self._maxTox=maxTox #max toxicity factor (float)
        self._pos=pos #position in cell vector

#Access methods

##@return type of bacteria(string)
    def type(self):
        return self._type

##@return metabolism vector of bacteria(list)
    def getm(self):
        return self._m

##@return metabolism vector of bacteria(list)
    def gett(self):
        return self._t

##@return max growth factor of bacteria(float)
    def getmaxGr(self):
        return self._maxGr

##@return position of bacteria (index) in cell vector
    def getpos(self):
        return self._pos
```

```
#methods

##calculates the growth factor
#@param food the eaten food dictionary
#@param num_bac number of bacteria of that type
#@param km Hill parameter
#@param e Hill parameter
#return growth factor(float in [0,maxGr) interval)
    def getgrowth(self,food,num_bac,km=0.5,e=1):
        growth=0 #not eating=>nothing changes
        food_quantity=0
        for i in range(len(self._m)): #for each nutrient
            if self._m[i]<0: #for each consumed nutrient
                food_quantity+=food[i]

        if food_quantity!=0:
            growth=self._maxGr/(1+ (num_bac*km/food_quantity)**e) #Hill
                equation
        else:
            growth=0
        return growth

##calculate the toxicity factor
#@param food the nutrients in the cell
#@param num_bac number of bacteria of that type
#@param km Hill parameter
#@param e Hill parameter
#return tox factor(float in [0,maxTox) interval)
    def getTox(self,food,num_bac=1,km=3500,e=1):
        tox_quantity=0
        for i in range(len(self._m)): #for each nutrient
            tox_quantity+=food[i]*self._t[i]
        if tox_quantity!=0:
            tox=self._maxTox/(1+ (num_bac*km/tox_quantity)**e) #Hill
                equation
        else:
            tox=0
        return tox

########################## Cell ####################################
class Cell:

    def __init__(self,food,bact,pos,x):
        self._food=food #nutrients list of the cell
        self._MatrFood=[list(food)] #trace nutrients in time
        self._bact=bact #dict={bact_obj:number of bacteria}
        self._pos=pos #position in cell vector
        self._x=x #cell vector
```

```
#Access methods

##@return list of metabolites
    def getMatrFood(self):
        return self._MatrFood

##@return list of metabolites
    def getFood(self):
        return self._food

##@return a dict with bacteria type (str) as keys and their quantity (int)
   as values
    def getBact(self):
        bact_diz=dict()
        for bac in self._bact:
            bact_diz[bac.type()]=self._bact[bac]
        return bact_diz

#methods

##update the cell food vector
#@return dict of eaten nutrients
    def food_upd(self):
        #food consumption
        eaten_food=dict() #initalization of eaten food dict
        for bac in self._bact:
            eaten_food[bac]=[0]*len(self._food)

        for i in range(len(self._food)): #for each nutrient
            max_eat=0
            for bac in self._bact:#for each bac
                if bac.getm()[i]<0:
                    eat=bac.getm()[i]*self._bact[bac] #food eaten by bac
                    max_eat+=eat
                    eaten_food[bac][i]=abs(eat)

            if self._food[i]>abs(max_eat): #if there's enough food
                self._food[i]-=abs(max_eat)#updating food vector

            else: #there's not enough food
                check=0
                bact_val=list(self._bact.values())
                r=0
                for r in range(len(bact_val)): #check for MVHG error
                    if bact_val[r]>0:
                        index=r #only species alive
                        check+=1
```

```
        if self._food[i]==0: #in this case the MVHG function would give an error
            eat_mvhg=[[0]*len(self._bact)]

        elif check==1: #in this case the MVHG function would give an error
            aux=[0]*len(self._bact)
            aux[index]=self._food[i]
            eat_mvhg=[aux]

        else:
            eat_mvhg=MVHG(m=bact_val,n=self._food[i]).rvs() #return matrix of size 1xlen(self._bac.values())

        self._food[i]=0 #consuming all nutrients

        j=0
        for bac in self._bact:
            eaten_food[bac][i]=eat_mvhg[0][j] #overwrite the value if there isn't enough food
            j+=1

#food production
for bac in self._bact:
    f_eat=sum(eaten_food[bac])#total of food eaten by a single type of bacteria
    #f_eat=math.floor(0.95*f_eat) #use of metabolites for growth
    m_tot=0 #number of nutrients that will be produced
    m_prod=[]#vector of metabolism for food produced
    j=0
    for j in range(len(self._food)):
        if bac.getm()[j]>0:
            m_tot+=bac.getm()[j]
            m_prod.append(bac.getm()[j])


    if f_eat>=m_tot: #weighted distribution of eaten food to each produced nutrient
        f_distr=f_eat//m_tot
        for z in range(len(self._food)):
            if bac.getm()[z]>0:
                self._food[z]+=f_distr*bac.getm()[z]
                f_eat-=f_distr*bac.getm()[z]

    if f_eat!=0: #distribution of remaining food with MVHG
        prod_mvhg=MVHG(m=m_prod,n=f_eat).rvs() #nutrients with higher metabolisms have a better chance to be produced
        q=0
        for w in range(len(self._food)):
```

```
                if bac.getm()[w]>0:
                    self._food[w]+=prod_mvhg[0][q]
                    q+=1

        return eaten_food

##adding new Bacts (given as dict) to self._bact dict
#@param newbacs a dict with this structure{bact_obj:number of bacterias}
    def addBact(self,newbacs):
        for bac in newbacs:
            self._bact[bac]=self._bact[bac]+newbacs[bac]
        return

##calculate the number of bacteria that will die in the cycle of a specific
     type of bacteria (considering toxicity)
#@param bac bact_obj of bacteria we are considering
#@param rate is the death rate
#@return the number of deaths (int)
    def death(self,bac,rate=0.027):
        num_death=(rate+bac.getTox(self._food,self._bact[bac]))*self._bact[
            bac]#number of deaths
        return math.ceil(num_death) #round to next smallest integer

##calculate the number of bacteria that will change cell
#@param bac bact_obj of bacteria we are considering
#@param flow_coeff percentage of flowing bacteria
#@return the number of flowing bacteria (int)
    def flow(self,bac,flow_coef=0.05):
        num_flow=flow_coef*self._bact[bac] #5% of population will flow to
            the next cell
        return int(num_flow)

##evolution of the cell considering bacteria flow, death, duplication and
    interaction with nutrients in the cell
    def evolution(self):
        newbacs=dict()#creating a dict for new bacts
        flowbacs=dict()#creating a dict for flowing bacts
        eaten=self.food_upd() #update food before adding new bacts
        self._MatrFood.append(list(self._food)) #update MatrFood without
            aliasing
        for bac in self._bact:
            growth=bac.getgrowth(eaten[bac],self._bact[bac])
            death_bacs=self.death(bac)
            flow_bacs=self.flow(bac)
            newbacs_number=self._bact[bac]*growth
            flowbacs[bac]=flow_bacs
            newbacs[bac]=int(newbacs_number//2) #filling newbacs dict
            self._bact[bac]+=int(newbacs_number//2) #half of newbacs will
                stay in the same cell
```

```
            self._bact[bac]-=death_bacs #death of bacteria
            self._bact[bac]-=flow_bacs #flowing bacteria
        if self._pos+1!=len(self._x):
            self._x[self._pos+1].addBact(newbacs) #adding newbacs to the
                cell in the next array position,but if it's last cell new
                bacts are ejected
            self._x[self._pos+1].addBact(flowbacs) #adding flowbacs to the
                cell in the next array position,but if it's last cell new
                bacts are ejected
        return

########################## Functions ################################

##create a random list with n integers from [inf,sup] interval
#@param inf the inferior extreme of the set
#@param sup the superior extreme of the set
#@param n the list length
#@return a new list filled with integers
def randomFill(inf,sup,n):
    newlist=[0]*n
    for i in range(n):
        newlist[i]=randint(inf,sup)
    return newlist

##create a stackplot of bacteria in cells vector
#@param x cells vector
#@param title the figure title
#@param fig the figure we want to plot on
def graph(x,title,fig):
    bac_diz=dict()
    for bac in x[0].getBact(): #create a new dict for bacs
        bac_diz[bac]=[]
    x_ax=[]
    for j in range(len(x)): #for each cell in the vector
        x_ax.append("x"+str(j)) #create x axis
        for bac,num in x[j].getBact().items(): #creating a dict to simplify
            the plot
            bac_diz[bac].append(num)

    #graph
    ax=fig.add_subplot(111)
    ax.set_title(title)
    ax.set_xlabel("Cell")
    ax.set_ylabel("Number of bacteria")
    plot_vec=[]
    legend_vec=[]
    for bac in bac_diz:
        plot_vec.append(bac_diz[bac])
        legend_vec.append(bac)
```

```
    ax.stackplot(x_ax,plot_vec)
    ax.legend(legend_vec)
    plt.pause(0.02) #time before the figure closes
    plt.clf()
    return

##plot metabolites in cells versus time
#@param x cells vector
#@param fig the figure we want to plot on
def graph_met(x,fig):
    #create legend
    len_m=len(x[0].getFood())
    food=[]
    for J in range(len_m):
        food.append('f'+str(J+1))

    #plot for each cell
    for I in range(len(x)):
        sub=fig_m.add_subplot(len(x),1,I+1)
        sub.set_title('Cella x'+str(I))
        sub.set_xlabel("Time(h)")
        sub.set_ylabel("Metabolites")
        sub.plot(x[I].getMatrFood(),'-o')
        plt.grid()
        if I+1==1: #first subplot with legend
            sub.legend(food)
    plt.show()
    return

##give as output bacteria in cell vector
#@param x cell vector
def printState(x):
    for j in range(len(x)):
        bact_num=x[j].getBact()
        print("x"+str(j)+': '+str(bact_num))

    return

##creating random characteristics for each species
#@param len_m the length of the metabolism vector
#@param pos bacteria position in cell vector
#@return a list of bacteria defined species
def randomBacts(len_m,pos):
    bacts=[]
    for typ in Bact._species:
        m=randomFill(-1,1,len_m) #metabolism list

        #for bacteria that has unrealistic metabolism vector (without >0
            and/or <0)
```

```
        ancora_prod=False; #no production
        ancora_cons=False; #no consumption
        while ancora_prod==False or ancora_cons==False:
            for z in range(len_m): #check of production
                if m[z]>0:
                    ancora_prod=True
                elif m[z]<0:
                    ancora_cons=True

                if ancora_prod and ancora_cons: #ok prod ok cons
                    break

            if ancora_cons==False:
                m[randint(0,len_m-1)]=-1 #for a new check
                ancora_prod=False

            if ancora_prod==False:
                m[randint(0,len_m-1)]=1
                ancora_cons=False



        t=[0]*len_m #toxicity vector
        for i in range(len(m)):
            if m[i]>=0 and random()<=0.05:# there is a 5% probability that
                a not consumed nutrient is toxic for bacteria
                t[i]=1
        doubling_time=numpy.random.lognormal(2.8,1.9745,1) #doubling time (
            in hours) , lognormal distribution with parameters (mu,sigma)
            chosen
        max_Gr=numpy.log(2)/doubling_time #max_growth rate
        maxTox=1-0.027 #max_toxicity rate=1-lambda
        bacts.append(Bact(typ,m,t,max_Gr,maxTox,pos))
    return bacts

########################## Main ################################
from random import *
import pdb
import matplotlib.pyplot as plt
import numpy
import math
from scipy.stats import multivariate_hypergeom as MVHG

#set the seeds (for random parameters)
#best seeds: Tier1:[2 2] [12 2] [5 2] [5 8] [8 2] [8 6] [2 8] [7 20] [8 4]
#             Tier2:[7 6] [3 8]
seed(6)
numpy.random.seed(7)
```

```
#set simulation parameters

len_x=5 #length of cells list
len_m=5 #length of metabolism (and food/toxicity) vector
n_it=0#number of cycles (1 cycle=1 hour)
max_bac=40 #max bacteria for species at start
max_f=25#max nutrients for type at start

#create vectors and dicts

x=[None]*len_x #cells vector
bacts=randomBacts(len_m,0)#random bacteria list
bac_diz=dict() #dict of bacteria for first cell
bac_null=dict() # dict of bacteras for empty cells
for bac in bacts:
    bac_diz[bac]=randint(0,max_bac)
    bac_null[bac]=0

food=randomFill(0,max_f,len_m)#random food vector for first cell

#creating cells in vector x
x[0]=Cell(food,bac_diz,0,x) #first cell

for i in range(1,len(x)): #other cells
    food=randomFill(0,max_f,len_m)
    x[i]=Cell(food,dict(bac_null),i,x)

#simulation
print('METABOLISMI:')
for bact in bacts:
    mu_max=str(bact.getmaxGr())
    print(bact.type(),' : ',bact.getm(),', mu_max='+str(mu_max),', t=',bact
        .gett())


title="INITIAL STATE"
print(title)
printState(x) #initial state
fig=plt.figure()
graph(x,title,fig)

for z in range(n_it):
    title="CICLO NUMERO "+str(z+1)
    print(title)
    for i in range(1,len(x)+1):# evolution of cells
        x[-i].evolution()
    printState(x)
    graph(x,title,fig)
fig_m=plt.figure()
```

```
graph_met(x, fig_m)
```