UNIVERSITÀ DEGLI STUDI DI PADOVA

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

Bactlife: simulatore per comunità batteriche - sviluppo del pacchetto Python

Laureando:

ALESSANDRO LUCCHIARI

Matr. 1219584

Relatore:

ING. MASSIMO BELLATO, PHD

Correlatore:

CHIAR.MA PROF.SSA BARBARA DI CAMILLO

DOTT. MARCO CAPPELLATO

ANNO ACCADEMICO 2021/2022

DATA DI LAUREA: 23 SETTEMBRE 2022

A papà.

Abstract

La presenza e la diffusione delle popolazioni batteriche in ogni ecosistema presente sulla Terra, rende di fondamentale importanza il loro studio, volto alla comprensione del loro importante ruolo a livello di sistema. Nonostante le numerose scoperte effettuate in questo ambito nell'ultimo secolo, si è ancora distanti dalla totale comprensione dei meccanismi che governano le interazioni tra comunità batteriche, con sostanze nutritive (tossiche e non) e in generale con l'ambiente che popolano. Lo sviluppo di un simulatore di popolazioni batteriche risulta perciò di fondamentale importanza per la comunità scientifica al fine di predire i comportamenti di più specie batteriche coesistenti nello stesso ambiente in risposta agli stimoli esterni, quali: introduzione di sostanze nutritive e di nuove specie batteriche, modifica delle caratteristiche di una o più comunità, introduzione di sostanze nutritive e di nuove specie batteriche, cambiamento del PH e della temperatura. Con questo obiettivo è stato sviluppato un simulatore di comunità batteriche basato sul modello ad agenti multipli, ossia entità autonome in grado di interagire, in base alle proprie caratteristiche, con le altre entità presenti e con l'ambiente.

Quest'elaborato si pone come obiettivo di ampliare le potenzialità del simulatore, standardizzandone la struttura ed implementandone una prima release per renderlo condivisibile e facilmente fruibile a qualsiasi utente, indipendentemente dalla preparazione informatica posseduta.

Per quanto riguarda la fase di sviluppo di tale software si è scelto di sfruttare le potenzialità di GitLab, un servizio di gestione di *repository* che permette di lavorare in team contemporaneamente allo stesso progetto, facilitando la distribuzione e il mantenimento del software sviluppato. Grazie all'utilizzo di quest'ultimo strumento, si è riusciti a sviluppare un pacchetto Python contenente il simulatore di comunità batteriche. Tale pacchetto è organizzato in moduli e file di esempio con una guida all'utilizzo del pacchetto stesso sono presenti nel *repository*.

Il pacchetto può essere utilizzato come prima versione di uno strumento in grado di predire lo sviluppo di comunità batteriche in base alle caratteristiche imposte ad ogni specie ed in base ai metaboliti presenti nello spazio considerato come ambiente. Questo software, grazie alla sua modularità, funge da punto di partenza da cui sviluppare un tool che permetta di integrare ipotesi sempre più precise sull'evoluzione delle comunità batteriche e con funzionalità sempre più avanzate da proporre all'utente finale. Lo sviluppo di questo

pacchetto ha rappresentato un fondamentale passaggio nell'evoluzione del simulatore, specialmente in ottica di sviluppi futuri, che ora possono essere portati avanti su file organizzati in modo standardizzato.

I prossimi aspetti su cui si può lavorare potrebbero essere: (1) la creazione di un database di specie batteriche e metaboliti rilevanti con le relative caratteristiche di interazione, (2) l'ampliamento dell'ambiente da unidimensionale a bidimensionale, (3) la modellizzazione e l'implementazione di un flusso di sostanze nutritive che si muove nell'ambiente considerato, (4) il miglioramento delle prestazioni del simulatore in termini di tempo di esecuzione, (5) la validazione tramite esperimenti in laboratorio dei risultati previsti dal software nella sua esecuzione.

Sommario

Capitolo 1: Simulatore agent-based di comunità batteriche

Lo sviluppo di questo lavoro di tesi prosegue le attività iniziate e presentate nell'elaborato di tesi triennale in ingegneria biomedica "Implementazione di un simulatore per comunità microbiche basato su un modello multi-agente" [1] che si poneva come obiettivo la creazione di uno scheletro di un simulatore predittivo modulare in grado di riprodurre l'evoluzione di specie batteriche in base alle caratteristiche dello spazio in cui si trovano ed al proprio metabolismo.

In questo primo capitolo verrà spiegata l'importanza dello studio e della comprensione delle comunità batteriche in virtù della loro diffusione e dei possibili sviluppi biomedicali. Si introdurrà il modello scelto per rappresentare le interazioni tra le popolazioni batteriche e una prima versione del simulatore proposta in un precedente lavoro di tesi basata su tale modellizzazione. Particolare attenzione verrà posta alle classi, alle funzioni e alle variabili usate. Infine, verranno analizzati i principali limiti della versione preliminare del simulatore; il loro superamento ha rappresentato l'obiettivo primario del lavoro presentato in questo manoscritto e nell'elaborato di Rebecca Sara
'Bactlife: simulatore per comunità batteriche – sviluppo di interfaccia grafica in Dash', con la quale ho collaborato su questo progetto e che si è occupata del lato front-end dello sviluppo, ossia della GUI.

1.1: Introduzione al modello e prima versione del codice

Il ruolo rivestito dalle specie batteriche in natura è fondamentale. Esse partecipano attivamente a numerosissimi meccanismi biologici: per esempio, sono responsabili del funzionamento, della stabilità, della sostenibilità e della produttività dell'ecosistema forestale; l'attività dei loro enzimi riveste infatti un ruolo chiave nel ciclo del carbonio, controllando elementi cruciali per questo ecosistema come la diversità delle piante e la decomposizione [2]. Le specie batteriche condividono, inoltre, la capacità di adattarsi a pressoché tutti gli ambienti naturali, tra cui anche il corpo umano; la loro presenza si può apprezzare nel microbiota intestinale: una complessa comunità batterica specifica per ogni individuo, fondamentale per la salute e per la comprensione dei meccanismi di numerose malattie come l'obesità ed i disturbi metabolici ad essa associati [3], nonché la sepsi [4], considerata uno dei più importanti problemi riguardanti la salute pubblica mondiale. Altro esempio è lo studio delle comunità microbiche negli animali che possono generare meccanismi in grado di contrastare le malattie dell'essere umano, come per esempio il COVID-19 [5]. Risulta evidente quanto e perché le specie batteriche al giorno d'oggi rappresentano un campo di studio fondamentale

per la comunità scientifica. Nonostante l'importanza delle comunità batteriche in natura, le conoscenze su di esse sono ancora tuttavia limitate e rendono difficile la totale comprensione dei meccanismi di interazione in contesti complessi come quelli dell'apparato digerente umano.

Sviluppare un simulatore in grado di modellizzare efficacemente e di predire il loro comportamento in base agli stimoli offerti, quali nutrienti presenti e altre specie batteriche, anche da conoscenze disaccoppiate per ogni singola specie anziché a livello di sistema potrebbe essere di fondamentale importanza per prevenire malattie importanti come quelle sopra citate. Tale progetto permetterebbe di progredire nell'apprendimento del loro ruolo e del loro modo di interagire tra popolazioni batteriche differenti e con l'ambiente ospitante.

Il primo passo necessario per sviluppare un simulatore di comunità batteriche è la scelta del modello, esistono infatti diversi approcci al problema (si veda la Figura 1) che si dividono in: (1) meccanicistici (la cui struttura matematica fa riferimento a rigorose ipotesi biologiche e parametri ottenuti dalla sperimentazione diretta), (2) empirici (con struttura matematica non direttamente interpretabile fisicamente), (3) integrati (ottenuti combinando i parametri del modello empirico con l'approccio meccanicistico). Un'altra classificazione dei modelli si basa sull'obiettivo della simulazione. Si individuano così: (1) i modelli ecologici che pongono l'accento sull'analisi delle dinamiche delle popolazioni senza considerare le interazioni molecolari che sono presenti (modelli generalizzati di Lotka-Volterra [9, 10] e modelli regressivi *data-driven* [11, 12, 13]), (2) i modelli microbo-effettore (tra cui i *genome-scale models* GEMs) che si interessano principalmente a comprendere il ruolo delle molecole effettrici (metaboliti e tossine) nello sviluppo delle comunità batteriche e nelle interazioni tra loro, (3) i modelli *agent-based* (ABM). Quest'ultimi utilizzano entità denominate agenti, aventi caratteristiche specifiche per ogni tipo di agente in grado di fornire simulazioni stocastiche che emergono dal comportamento dei singoli agenti in un ambiente condiviso. I modelli *agent-based* si basano su una caratterizzazione spaziale in termini di relazioni fra gli agenti, su un criterio di prossimità (considerando ciò che ogni agente può percepire nella sua zona limitrofa) e sulle regole di comportamento, che descrivono il modo di interazione tra agente e zona circostante.

La scelta del modello da adottare per questo simulatore è ricaduta sui modelli *agent-based*, considerando come agenti le specie batteriche presenti in un vettore unidimensionale. Si è preferito tale modellizzazione perché il tipo di interazione tra agenti che si viene a creare nel corso della simulazione è una conseguenza e non una causa delle caratteristiche degli agenti

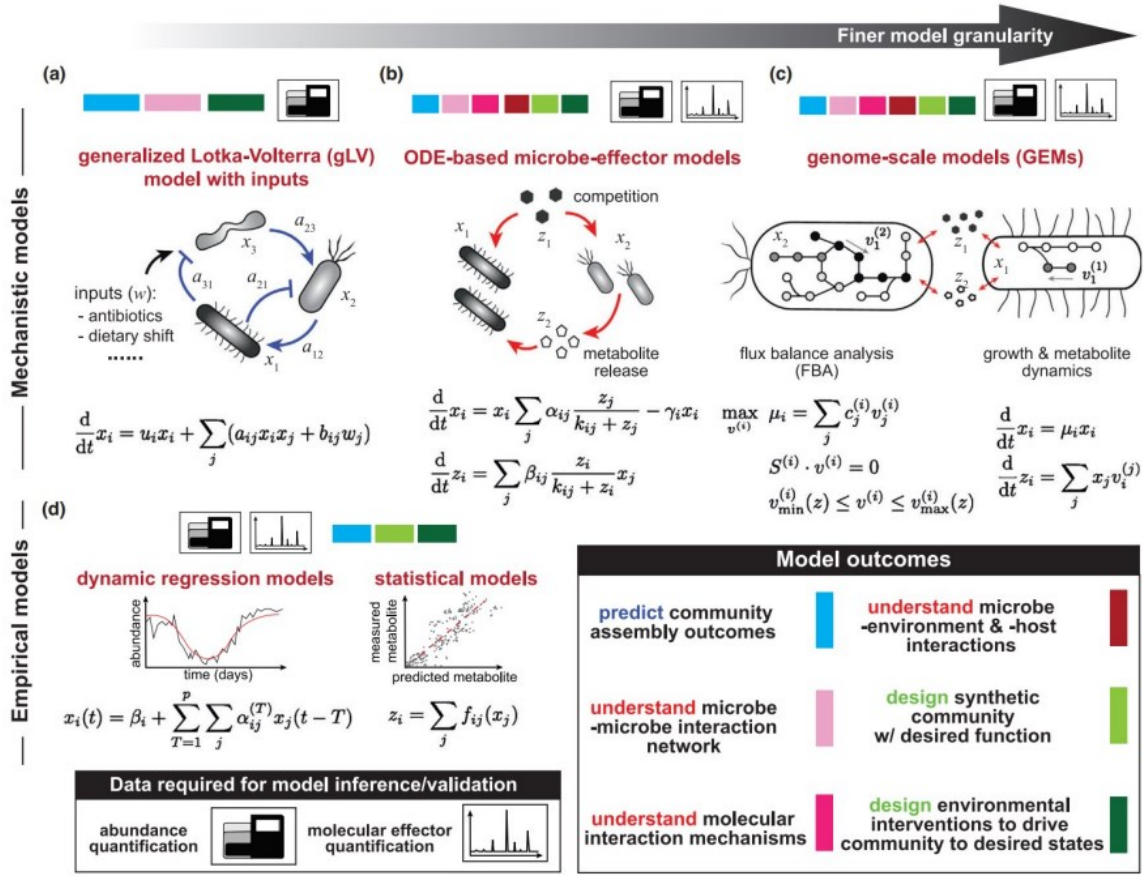


Figura 1: Schema rappresentante i vari modelli a confronto, immagine presa da Qian et al., Curr. Opin. Microbiol., 2021

stessi. Inoltre tali caratteristiche, specifiche per ogni specie, sono più facili da quantificare in modo affidabile e risulta facile manipolare la struttura ad oggetti, modificando, eliminando e aggiungendo fenomeni alla modellizzazione. Quest'ultima caratteristica, affiancata alla possibilità di modificare i diversi parametri della simulazione per esprimere fenomeni quali le variazioni dell'ambiente e delle caratteristiche degli agenti considerati, rende preferibile la scelta del modello ABM rispetto agli altri modelli sopra citati.

Creare un simulatore *agent-based*, in grado di predire in maniera sempre più precisa l'evoluzione delle specie batteriche assunte come agenti in relazione all'interazioni con i metaboliti e con gli agenti stessi presenti nell'ambiente circostante, sarebbe utile per permettere analisi accurate senza necessariamente effettuare sperimentazioni. Questo cambiamento porterebbe notevoli vantaggi in termini di conoscenze, tempo, risorse e rischi in caso di specie batteriche potenzialmente patogeniche.

Il simulatore considera come ambiente uno spazio unidimensionale in cui gli agenti (le specie batteriche) possono evolversi seguendo precisi passaggi iterativi: tale scelta è stata fatta ispirandosi ad un tratto di intestino, ospitante la flora batterica ed i nutrienti dovuti

all'ingerimento di cibo da parte dell'individuo. Il numero di celle e di specie batteriche di default è 5, il numero di nutrienti complessivo per ogni cella è stato scelto pari a 25 mentre il metabolismo di ogni specie nei confronti di ogni nutriente è creato randomicamente (si veda la funzione `RandomBacts` nel paragrafo successivo). Il simulatore, una volta stabiliti i parametri iniziali, procede con una fase dinamica, nella quale ogni iterazione viene assunta corrispondente ad un'ora di sviluppo delle comunità batteriche considerate. Il primo fondamentale passaggio riguarda il numero di batteri che vengono prodotti da ogni specie in ogni cella: viene calcolato il fattore di crescita basandosi sul *rate* di crescita massimo della specie e sulla presenza di nutrienti che possono essere consumati nella cella colonizzata. Inoltre, si è ipotizzato che le popolazioni batteriche abbiano la capacità di produrre metaboliti a loro volta. Una volta ottenuto il tasso di crescita della specie batterica, il 5% dei batteri presenti in ogni cella viene spostato nella cella successiva (nel caso dell'ultima cella i batteri vengono espulsi dall'ambiente). Viene poi stimato il numero di batteri in una determinata cella che saranno considerati morti in base al tasso di tossicità dei metaboliti con cui la specie è venuta a contatto e del *rate* di morte basale. Si ottiene così una simulazione completa dello sviluppo di specie batteriche, avendo preso in considerazione il consumo e la produzione di

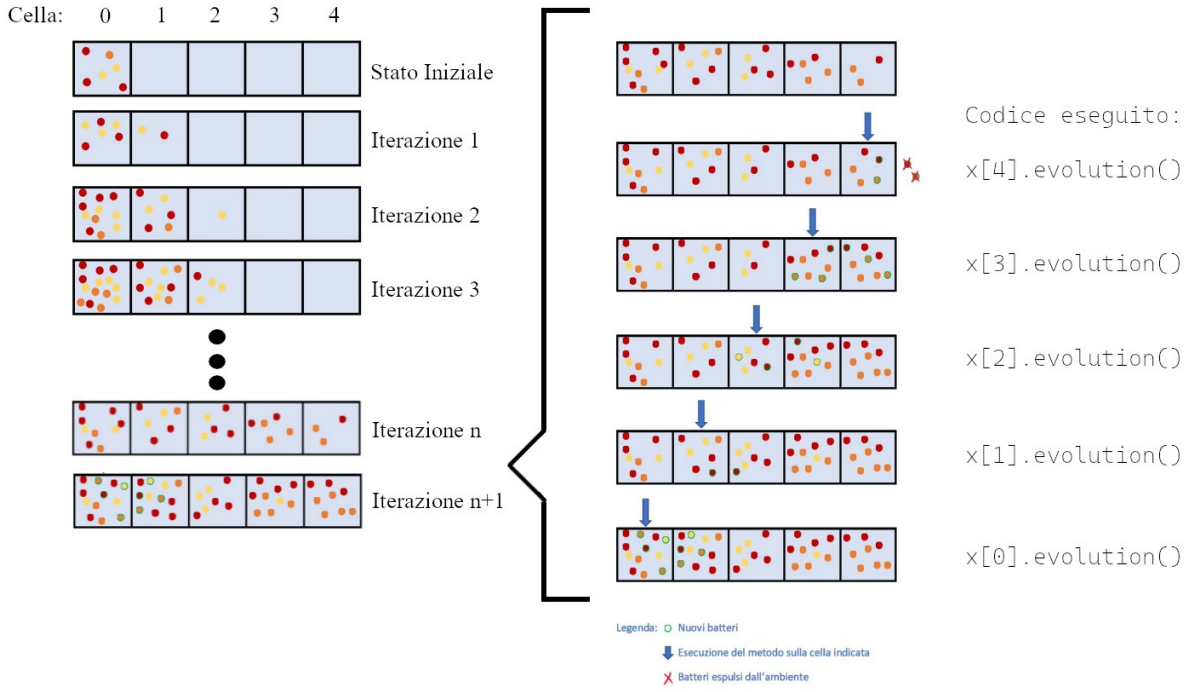


Figura 2: Schema rappresentante le n+1 iterazioni della simulazione sulla sinistra, sulla destra le modifiche effettuate dalla funzione `evolution()` in un'iterazione ad ogni cella

nutrienti, la riproduzione, la morte e lo spostamento di batteri di ogni specie (vedi Figura 2)

1.1.1: Classi e funzioni

Successivamente alla definizione del modello appena esposto, si è creata una prima versione del simulatore, sfruttando le potenzialità della programmazione ad oggetti permessa da Python. Sono state prodotte le seguenti classi e funzioni, descritte dettagliatamente in [1] e qui riassunte (vedi Figura 3), indispensabili per l'evoluzione degli agenti del modello:

Classe Bact e principali funzioni

La classe ***Bact***, rappresentante tutte le caratteristiche di una popolazione microbica, gli attributi di questa classe e i metodi di accesso vengono presentati in 1.1.2;

- Funzione `getGrowth` (appartenente alla classe ***Bact***): calcola il fattore di crescita di una popolazione microbica in base alla specie stessa, alle sue caratteristiche e ai nutrienti presenti nella cella che ospita la popolazione stessa;
- Funzione `getTox` (appartenente alla classe ***Bact***): basandosi sulle caratteristiche della specie, calcola il suo coefficiente di tossicità;

Classe Cell e principali funzioni

La classe ***Cell***, rappresentante una cella del vettore adottato come ambiente in cui si svolge la simulazione, gli attributi di questa classe e i metodi di accesso vengono presentati in 1.1.2;

- Funzione `addBact` (appartenente alla classe ***Cell***): permette di sommare al dizionario di batteri presenti in una determinata cella, un dizionario avente le stesse chiavi del dizionario di batteri passato come parametro d'ingresso della funzione;
- Funzione `death` (appartenente alla classe ***Cell***): calcola, considerando la tossicità, il numero di microbi di ogni specie che muoiono in ogni iterazione;
- Funzione `evolution` (appartenente alla classe ***Cell***): serve a simulare e a gestire lo sviluppo delle comunità microbiche, considerando il flusso da una cella a quella successiva, la morte e la riproduzione dei batteri, l'interazione con i nutrienti;
- Funzione `food_upd` (appartenente alla classe ***Cell***): serve ad aggiornare il dizionario dei nutrienti presenti in quell'elemento cella;
- Funzione `randomBacts`: utile ad inizializzare una lista di batteri con caratteristiche (vettore del metabolismo e tasso di crescita massimo) casuali

- Funzioni `printState`: stampa a video lo stato del vettore di celle ad una determinata iterazione, mettendo in evidenza il numero di batteri divisi per specie, per ogni cella;
- Funzione `graph`: genera un grafico avente per asse delle ascisse le celle che compongono l'ambiente, rappresentante il numero di batteri di ogni specie;
- Funzione `graph_met`: rappresenta lo sviluppo temporale dei metaboliti con un grafico per ogni cella.

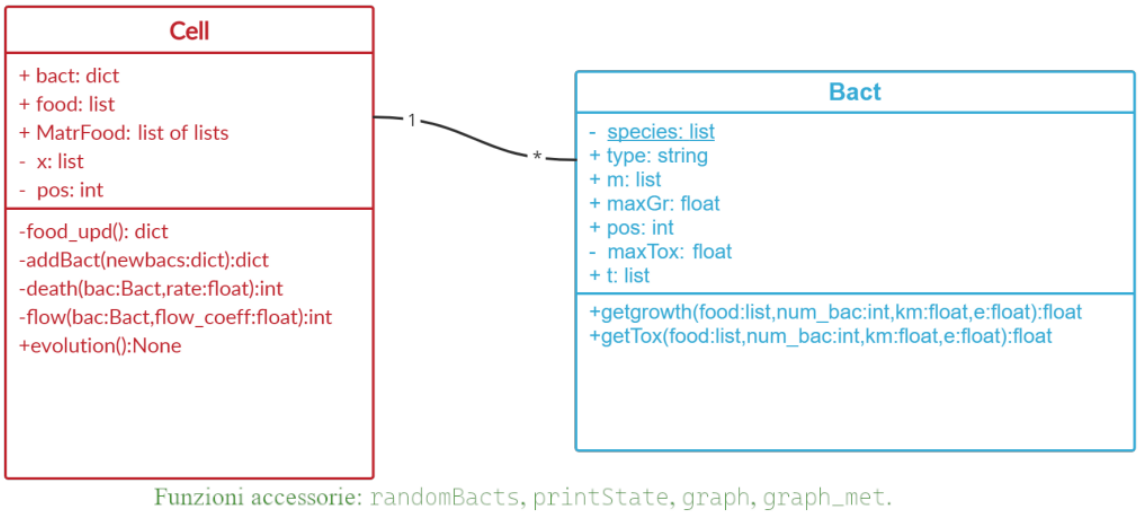


Figura 3: Schema rappresentante le classi con le rispettive variabili e funzioni, sono presentate anche le funzioni accessorie non appartenenti a nessuna classe

1.1.2: Variabili principali e metodi d'accesso

Nella Tabella 1 vengono presentati tutti gli attributi delle classi della prima versione sviluppata del simulatore, accompagnati dal nome del metodo d'accesso se presente. Si adottano, inoltre, queste convenzioni: 's' è il numero di specie batteriche, 'n' è il numero di tipi di nutrienti, 'i' è il numero di iterazioni e 'c' è il numero di celle da cui è composto il vettore. In caso di matrici, il primo valore indicato delle dimensioni è considerato come numero di righe.

Classe	**Attributo**	**Descrizione**	**Tipo**	**Dimensione**
Bact	*_species*	Vettore contenente i nomi di tutte le specie batteriche, comune a tutti gli elementi bact	Vettore di stringhe	s

	_type	Nome della specie batterica scelto dal vettore *_species* Metodo d'accesso: `type`	Stringa	-
	_m	Vettore rappresentante il metabolismo della specie batterica per ogni nutriente. Se nella casella i-esima è presente il valore: o -1 allora la specie batterica consuma il nutriente i-esimo; o 0 allora il nutriente i-esimo viene ignorato dalla specie batterica; o 1 allora la specie batterica produce il nutriente i-esimo. Metodo d'accesso: `getm`	Vettore di interi	n
	_t	Vettore rappresentante la possibile tossicità di un nutriente nei confronti dell'elemento bact. Se nella casella i-esima è presente il valore: o -1 il nutriente i-esimo è tossico per l'elemento bact; o 0 il nutriente i-esimo viene ignorato dalla specie batterica considerato. Metodo d'accesso: `gett`	Vettore di interi	n
	_maxGr	Coefficiente di massima crescita della specie batterica. Metodo d'accesso: `getmaxGr`	Float	-
	_maxTox	Coefficiente di massima tossicità dell'elemento bact.	Float	-

	_pos	Posizione della specie batterica all'interno del vettore di elementi cell. Metodo d'accesso: `getpos`	Intero	-
Cell	*_food*	Vettore contenente i nutrienti presenti nella cella. Metodo d'accesso: `getFood`	Vettore di interi	n
	_MatrFood	Matrice dei nutrienti nel tempo: la riga j-esima rappresenta il vettore _food all'iterazione j-esima Metodo d'accesso: `getMatrFood`	Matrice di interi	i × n
	_bact	Dizionario con chiavi i nomi delle specie batteriche e con valori il numero di batteri corrispondenti ad ogni specie Metodo d'accesso: `getBact`	Dizionario di interi	s
	_pos	Posizione della cella nel vettore di elementi cell.	Intero	-
	_x	Vettore delle celle usate nella simulazione	Vettore di elementi cell	c

Tabella 1: Tabella delle variabili della prima versione del simulatore.

1.2: Limiti del simulatore

Lo sviluppo del simulatore ha sempre avuto tra gli obiettivi principali quello di garantire la facilità d'utilizzo da parte di tutti gli utenti, rendendo immediata la sua condivisione e la sua comprensione, indipendentemente dalle conoscenze informatiche possedute dal fruitore finale. Nel corso dell'analisi della prima versione proposta del codice, sono emersi importanti limiti che impediscono la crescita in tal verso.

Il codice, come è stato inizialmente proposto, presenta le funzioni, le classi e un programma di test nello stesso file. Questa struttura, pur essendo necessaria in una prima fase, risulta sconveniente nel successivo sviluppo del simulatore in quanto non permette la creazione di un

pacchetto (per la definizione di pacchetto si veda 2.1: Creazione di un software in Python) in vista della sua distribuzione. Per ovviare a questo problema, si è scelto di fornire al simulatore una struttura gerarchica ben definita e standardizzata; tale struttura è necessaria anche a rendere più facile e immediata la comprensione del codice e delle sue potenzialità a tutti coloro che non hanno preso parte alla fase di sviluppo del simulatore stesso.
Inoltre, il fine ultimo del simulatore è di eseguire analisi accurate senza necessaria sperimentazione diretta; non è perciò richiesto agli utenti di questo prodotto di avere le conoscenze informatiche per poter modificare e comprendere tutti i processi del simulatore presenti a basso livello. Si è reso necessario in quest'ottica lo sviluppo di un'interfaccia grafica interattiva in grado di interporsi tra il codice sviluppato e l'utente, rendendo possibile la modifica dello stato iniziale del sistema di specie batteriche e di nutrienti e la presentazione dei dati prodotti nel corso delle iterazioni della simulazione.
Un ultimo aspetto su cui si è voluto lavorare è stata l'esportabilità dei risultati e delle impostazioni iniziali utilizzate durante l'esecuzione del programma. Nell'ambito biomedico risulta di fondamentale importanza l'analisi dei dati e la riproducibilità degli esperimenti e delle simulazioni effettuate; perciò, rendere accessibili e consultabili in un file di testo tutti i dati prodotti nelle iterazioni ha rappresentato un importante superamento dei limiti presenti nella prima versione del codice.

Capitolo 2: Strumenti per lo sviluppo

Come anticipato nell'introduzione del Capitolo 1, il superamento dei limiti riscontrati nella prima versione del codice e la risoluzione delle problematiche ad essi legate hanno rappresentato gli obiettivi nel corso dello sviluppo di questo elaborato. In particolare, si è convenuto che rendere distribuibile e utilizzabile il programma a qualsiasi utente, indipendentemente dalle conoscenze informatiche, fosse un obiettivo di primaria importanza. Per raggiungere quest'obiettivo, si è lavorato sul fornire un *packaging* del simulatore ed un'interfaccia grafica intuitiva. Mentre il secondo aspetto è stato sviluppato e presentato nel lavoro di tesi da Rebecca Sara [8]. Questo lavoro di tesi si è concentrato sul primo aspetto *back-end* ed in questo capitolo viene presentato quali siano le conoscenze da acquisire per implementare il *packaging* di un programma in linguaggio Python.
Il principale vantaggio che ha reso lo sviluppo del pacchetto per il simulatore necessario è stata la possibilità di condividerlo su larga scala a tutti gli utenti interessati tramite piattaforme online come GitLab e GitHub. Questo passaggio, ovviamente, amplia la platea di utilizzatori. La condivisione permette inoltre ad altri utilizzatori di considerare il pacchetto come un punto di partenza per un ulteriore sviluppo personale del simulatore.
Un mezzo e al tempo stesso un fine nel realizzare il pacchetto è stata la semplicità e la modularità con cui si è dovuto strutturare il simulatore per renderlo facilmente comprensibile anche ad utenti che non hanno preso parte al lavoro di creazione e sviluppo del programma stesso, ma che si affidano solo agli elementi presenti all'interno del pacchetto diffuso su GitLab (si veda in particolare il paragrafo 3.4: Guida all'utilizzo di 'Bactlife').
Lo sviluppo dell'interfaccia grafica è necessario al fine di rendere utilizzabile il pacchetto 'Bactlife' anche ad utenti senza conoscenze nell'ambito della programmazione. In relazione all'obiettivo finale di questo progetto, l'interfaccia grafica riveste il ruolo chiave di permettere a chiunque di poter simulare l'evoluzione batterica in modo semplice e intuitivo, senza dover necessariamente conoscere il linguaggio Python.

2.1: Creazione di un software in Python

Python permette di organizzare il codice in file e cartelle usando i concetti di pacchetto (*Package*) e moduli (*Modules*). I moduli sono i file Python contenenti gli elementi costitutivi del programma, principalmente funzioni, istruzioni *import* e classi. Alcuni moduli Python sono scritti in linguaggi diversi da Python, generalmente C o C++ e vengono denominati *extension module*. Tutte le informazioni accessibili presenti all'interno di tali file sono

chiamate attributi (*Attributes*). I pacchetti organizzano i moduli in una gerarchia di cartelle con lo scopo di aiutare gli utenti a capire come ogni file concorra alle finalità del progetto.

Per iniziare ad utilizzare un pacchetto all'interno del proprio progetto, il primo passo è l'installazione seguendo le indicazioni presenti nel file denominato '`Readme`', presente generalmente presente nel pacchetto stesso. L'installazione deve essere fatta da linea di comando (*shell/command window*) con lo scopo di installare tutti i moduli necessari al corretto funzionamento del pacchetto e di inserire il *package* all'interno del *Pythonpath*, un insieme di cartelle contenenti moduli importabili. La sequenza di comandi e in che cartella posizionarsi con la *command window* vengono specificati nei file presenti assieme al pacchetto, generalmente chiamato '`Readme`', come nel caso di 'Bactlife' (si veda 3.4: Guida all'utilizzo di 'Bactlife' e Appendice B: Readme.md e Tutorial.ipynb).

Nel momento in cui un modulo viene importato, Python valuta la possibile posizione del pacchetto cercando tra varie cartelle finché non trova un file avente il nome indicato seguito dal suffisso '.py'. L'ordine delle cartelle in cui il modulo viene cercato è il seguente:

- La cartella del programma che si sta creando;
- Le cartelle presenti nel *Pythonpath*;
- Le cartelle nella variabile d'ambiente *PATH environment variable*.

Questa lista è consultabile ed espandibile tramite il modulo importabile 'sys.path' e gli stessi metodi usati per gestire una lista in Python, come nel seguente esempio:

```
>>> import sys

>>> sys.path.append('/ufs/utente/lib/python')
```

Una volta installato, si può procedere all'importazione che si differenzia in base a che cosa si vuole installare: singoli moduli oppure tutti gli attributi presenti nel pacchetto.

Nel primo caso si può specificare quale parte del pacchetto importare usando un punto separativo tra il nome del *package* e quello del *subpackage*/modulo:

`import` *`packagename.modulename`* (-si veda il successivo esempio di importazione).

In alternativa a questa forma, si può combinare '`import`' con il comando '`from`'. Quest'ultimo seleziona la posizione all'interno del pacchetto dalla quale verranno importati gli attributi specificati subito dopo l'''`import`':

`from` *`packagename.modulename`* `import` *`function_name`*

Per importare tutti gli attributi presenti si sostituisce il nome della specifica funzione con un asterisco: `from mymodule import *`. L'*import* di tutte le funzioni tramite * presenta dei pericoli per la facilità di comprensione e di gestione del programma: non si ha più il controllo di che cosa si sta importando e si rischia di complicare la fase di *debugging* data la quantità di nomi di variabili che vengono automaticamente importate in questo modo.

Per esempio, ci si pone nella situazione di aver scaricato un pacchetto in grado di implementare varie funzionalità grafiche tramite gli attributi presenti e di voler usare nel proprio programma la funzione 'edges' contenuta in `noise.py` . Questo pacchetto sfrutta la struttura gerarchica mostrata nel Codice 1 per differenziare le classi e le funzioni in base al loro scopo, formando così 3 cartelle: formats, effects e filters.

```
graphics/                                Top-level package
          __init__.py                    Initialize the graphics package
          formats/                       Subpackage for file format conversions
                __init__.py
                gifread.py
                gifwrite.py
                ...
          effects/                       Subpackage for graphics effects
               __init__.py
               speckle.py
               swirl.py
               invert.py
               ...
          filters/                       Subpackage for filters
               __init__.py
               noise.py
               mask.py
```

Codice 1: Struttura del pacchetto considerato nell'esempio.

I comandi che possono essere eseguiti per importare la funzione desiderata sono i seguenti:

1. 'import graphics.filters.noise.edges'
2. 'from graphics.filters.noise import edges'
3. 'from graphics.filters.noise import *'

Ottenendo risultati sensibilmente diversi:

- Mentre le righe di codice 1 e 2 provocano l'*import* solo della funzione '`edges`', 3 importa (oltre che la funzione '`edges`' anche) tutti gli attributi contenuti nel file `noise.py`
- A seguito dell'esecuzione della riga 1, la funzione '`edges`' potrà essere utilizzata solo specificando sempre il percorso di file effettuato dall' '`import`': `graphics.filters.noise.edges(args)`
- A seguito dell'esecuzione della riga 2, la funzione '`edges`' potrà essere utilizzata semplicemente eseguendo il comando: *`edges(args)`*

È utile ricordare che per rendere più semplice e intuitivo l'utilizzo degli attributi importati, Python permette di rinominare localmente le funzioni e/o i moduli. Quest'azione può essere fatta in fase di *import* oppure in righe successive e si avvale dell'operatore '`as`' o dell'operatore di assegnazione '='. Nel primo caso le righe di codice che permettono l'importazione e ridenominazione sono le seguenti:

`import` *`packagename.modulename.functionname`* `as` *`myfunction`*

`from` *`packagename.modulename`* `import` *`functionname`* `as` *`myfunction`*

Nel secondo caso invece prima viene importata la funzione e poi in una successiva riga viene rinominata:

`import` *`packagename.modulename.functionname`*

`myfunction=packagename.modulename.functionname`

La differenza tra i due metodi non riguarda semplicemente l'operatore utilizzato ma soprattutto il nome assunto dalla funzione nelle righe di codice successive: mentre dopo aver rinominato l'attributo tramite '`as`', esso risulta richiamabile solo col nuovo nome imposto, nel caso di utilizzo dell'operatore di assegnazione viene attribuito alla funzione un nome aggiuntivo, risultando dunque richiamabile sia col nome precedente, sia con quello nuovo.

2.2: Sviluppo con GIT

Un altro importante strumento adoperato durante lo sviluppo di questo elaborato è stato GIT: un software volto alla gestione di *repository* Git (vedi paragrafo 2.3) e che fornisce un sistema di controllo di versione (VSC) del progetto al quale si sta lavorando, indipendentemente dal linguaggio di programmazione utilizzato per quest'ultimo. Git è solitamente eseguibile a linea di comando e permette di tenere traccia e di gestire tutte le modifiche che vengono svolte al

codice sorgente: scegliendo quale versione usare come punto di partenza per un ulteriore sviluppo, occupandosi di più versioni dello stesso progetto parallelamente (tramite l'uso di *branch*), lavorando contemporaneamente ad una o a più versioni, salvando in locale tutta la storia del codice. Il singolo utente o il team di sviluppatori per sfruttare le potenzialità di questo strumento possono avvalersi di una piattaforma web che ne permette l'utilizzo, come GitLab oppure GitHub.

Per esempio, si ipotizza che esista un progetto salvato in un *repository* Git e che di questo progetto siano già state create 3 versioni. L'utente che ora vuole andare a modificare una delle versioni del codice può scaricare localmente non solo il file modificato più recentemente ma l'intero archivio del progetto, completo di tutta la sua storia. Ora l'utente può proseguire lo sviluppo da una qualsiasi versione e successivamente salvarlo e caricarlo online, andando così ad aggiungere un nuovo salvataggio nel ramo della versione da cui si è partiti. Il tipo di sistema di controllo adottato da piattaforme come Git (rappresentato dallo schema presente nella Figura 4) viene definito distribuito in quanto ad ogni utente viene fornito l'intero archivio dei salvataggi appartenente al server. Questa caratteristica garantisce che, in caso di perdita di dati nel server, essi possano essere recuperati caricando la versione del *repository* più recente scaricata localmente.

2.3: Distribuzione del software in GitLab

Il sistema di controllo Git si basa sulla gestione di *repository*, ossia delle cartelle *.git* contenenti tutti i file e tutte le versioni precedenti del progetto creato. Nel paragrafo precedente è stato messo in luce quanto questo software sia utile nell'organizzazione dei dati nel corso dell'elaborazione di un software, permettendo le operazioni di creazione di *branch* e di *package sharing* in forma di *repository*; un'altra caratteristica fondamentale di Git è la possibilità di collaborare in team allo stesso progetto contemporaneamente e senza creare conflitti. Questo sistema di controllo risulta molto flessibile grazie ai differenti servizi web utilizzabili per gestire i file del progetto e le loro versioni precedenti, come GitHub e GitLab. Nello sviluppo di questo software si è scelto di utilizzare GitLab.

GitLab è una piattaforma web permette di eseguire tutte le attività utili alla creazione di un

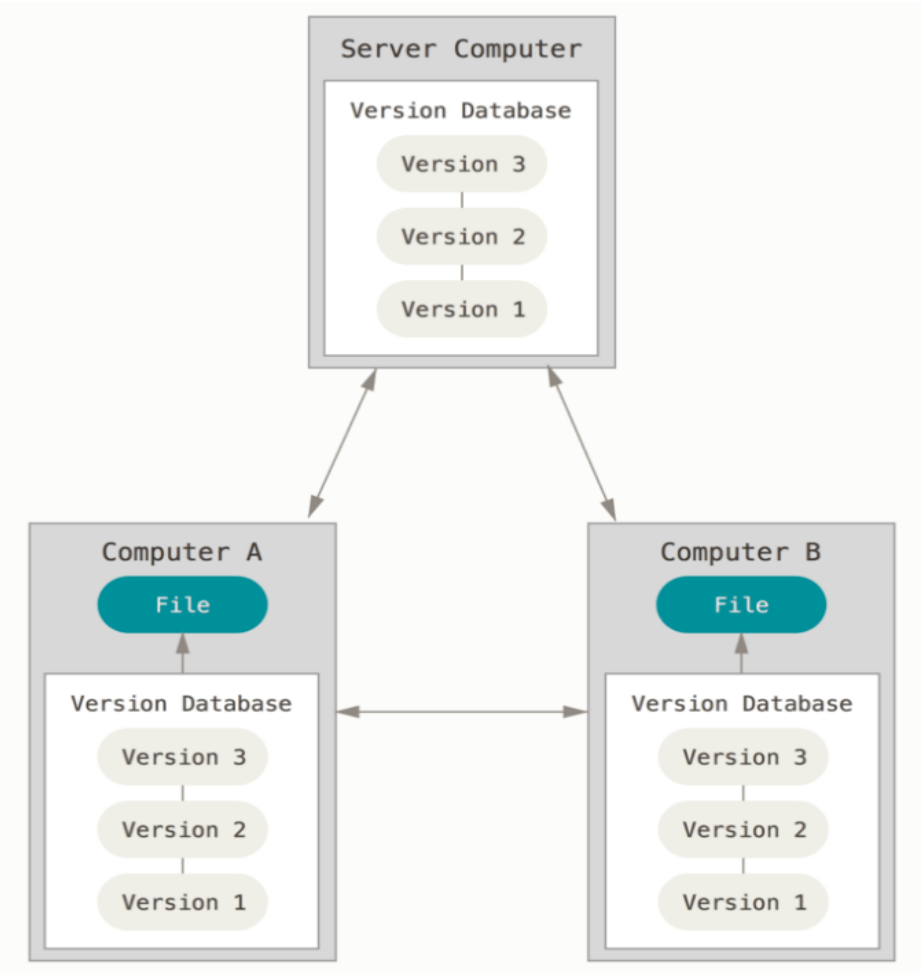


Figura 4: Schema di condivisione e di gestione di una cartella *.git* utilizzando un sistema di controllo di versione distribuito. Immagine tratta da [14]

software: dalla sua pianificazione alla sua gestione e modifica. Permette a tutti gli utenti che partecipano di: scaricare in locale il codice presente nel *repository* (`pull`), caricare online la

versione del codice creata localmente (`push`), dividere il progetto in più rami da sviluppare in maniera diversa tra loro, decidere su quale ramo e versione lavorare, operare in parallelo contemporaneamente su una o più versioni del progetto (usando i *branches*), unire le modifiche effettuate in un unico progetto (`merge`).

Ad esempio, si ipotizza di essere nella situazione in cui due utenti (A e B) vogliono lavorare allo stesso progetto usando come servizio di gestione del file Gitlab. Dopo essersi registrati entrambi su di esso, l'utente A crea e sceglie il nome del *repository* nel quale si elaborerà il codice. In questa prima fase si può decidere se rendere pubblica o privata la cartella e si può decidere chi parteciperà allo sviluppo (in questo caso l'utente B) e di includere o meno uno o più file utili per la fase di condivisione del progetto, quali `Readme.md` e `License.txt`. Una volta creato il *repository*, si procede alla clonazione della cartella tramite l'uso di chiavi SSH oppure HTTPS, cosicché sia A che B possano lavorarci localmente tramite la *command window* messa a disposizione da GitLab (GitBash o GitGUI), dando effettivamente inizio allo sviluppo del software desiderato. Una tipica situazione che può venirsi a creare e che mette in evidenza le potenzialità di GitLab è la seguente: si ipotizza che A abbia modificato 3 volte il codice sorgente, creando così 3 salvataggi; B vuole lavorare su altri aspetti del progetto e per farlo inizia dalla seconda versione prodotta da A. In questo caso, B scarica localmente la versione desiderata utilizzando la chiave di clonazione specifica di quel progetto fornita da GitLab e digitando sulla *command window* posizionata sulla cartella che si desidera ospiti il progetto:

```
git clone git_key
```

Oppure, nel caso in cui la cartella sia già presente nel computer ma non sia aggiornata:

```
git pull
```

Si ipotizza poi che B voglia continuare a modificare un ramo di salvataggi specifico (*branch*), in questo caso il comando da eseguire nella *command window* è:

```
git checkout origin/'branchname'
```

Ora l'utente può modificare potenzialmente tutti i file presenti nella cartella che è stata appena scaricata localmente. In questo esempio si ipotizza che B svolga delle correzioni al file '`init.py`' e '`license.txt`' e che crei il documento '`index.docx`'. Una volta svolte tutte le modifiche desiderate, digiti per sapere lo stato dei salvataggi:

```
git status
```

Ottenendo il risultato presente nella Figura 5, in cui si evidenzia come i primi due file sono stati modificati mentre il terzo è stato aggiunto.

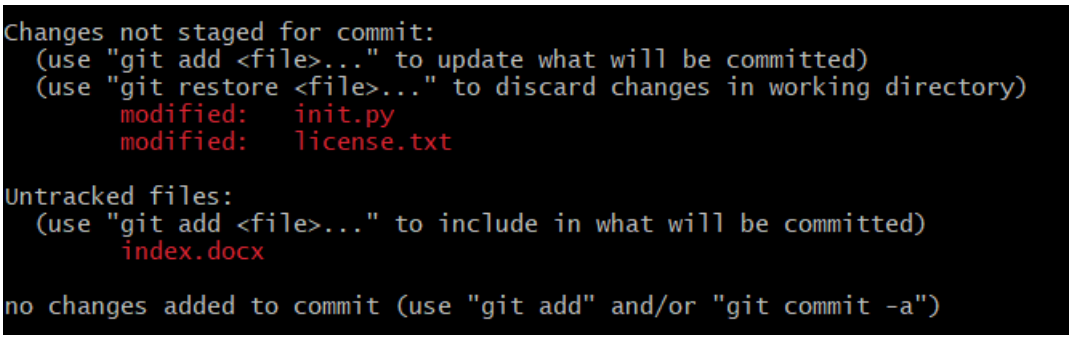


Figura 5: Schermata di GitBash risultante a seguito delle modifiche descritte e del comando 'git status'.

Si può notare come GitBash suggerisca i prossimi comandi da eseguire (`add` o `restore`). In questo caso B decide di salvare le modifiche localmente tramite `add`:

```
git add init.py
```

```
git add license.txt
```

```
git add index.docx
```

E successivamente di salvare le modifiche con un nome specifico della versione:

```
git commit -m newindex
```

Infine B può caricare online le modifiche effettuate:

```
git push
```

Sarà così visibile un nuovo salvataggio sul ramo scelto anche all'utente A, che avrà la possibilità di lavorare a sua volta dalla versione che preferisce. Nel momento in cui si deciderà di unire le versioni appartenenti a due rami diversi in un unico salvataggio, si potrà fare senza creare conflitti grazie al comando:

```
git merge branchname
```

Alla luce di questo esempio, risulta evidente quanto questo servizio sia risultato utile nell'implementazione di un software, lavorando in team parallelamente.

Capitolo 3: Implementazione di ‘Bactlife’

Nel seguente capitolo vengono presentate e analizzate tutte le tappe che hanno permesso, partendo dalla prima versione proposta in Calzavara [1], la creazione e la distribuzione del pacchetto ‘Bactlife’. In particolare, vengono inizialmente presentati la struttura gerarchica fornita alle varie parti del codice e i criteri con i quali essa è stata organizzata. Successivamente si espongono le modifiche apportate alla struttura introdotta, al fine di rendere il pacchetto distribuibile ed importabile, ed al codice stesso, per fornire al simulatore funzionalità aggiuntive e per rendere il pacchetto compatibile con l’interfaccia grafica sviluppata. Infine, vengono presentati il processo di caricamento del pacchetto su Gitlab e la creazione di tutti i file che permettono ad un utente esterno di informarsi sulle potenzialità del simulatore, sul funzionamento generale dei moduli e su come scaricare e installare ‘Bactlife’ sul proprio computer.

3.1: Struttura gerarchica di ‘Bactlife’

La progettazione della struttura gerarchica ha rappresentato il primo importante passo nella realizzazione del pacchetto ‘Bactlife’. Dalla situazione presente nella prima versione del codice, si è posto l’obiettivo di dividere l’insieme delle classi e delle funzioni del programma in vari file: i moduli di ‘Bactlife’. I criteri seguiti in questa fase di suddivisione sono stati:

- Assegnare a moduli differenti le classi e le funzioni con scopi differenti;
- Facilitare l’importazione e il richiamo degli attributi favorendo l’utilizzo di nomi semplici ed evocativi.

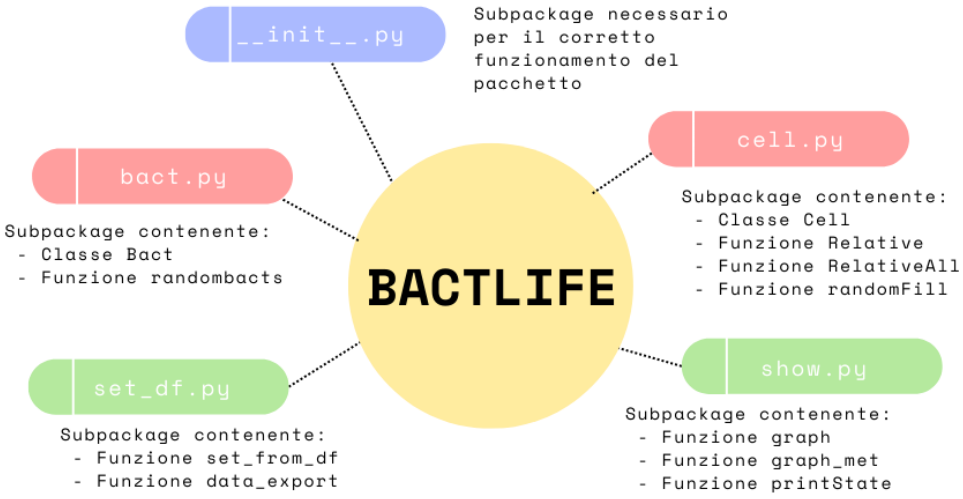


Figura 6: Diagramma rappresentante i *subpackage* componenti Bactlife

Entrambi i criteri adottati presuppongono che la struttura del pacchetto, rappresentata nella Figura 6 presenti tutte le funzionalità del codice in modo semplice e intuitivo, sia nel contenuto dei moduli che nella loro organizzazione gerarchica.

3.1.1: bact.py

Nel file '`bact.py`' sono stati inseriti gli attributi che gestiscono la creazione e la caratterizzazione di ogni specie batterica. Tali elementi sono: la classe ***Bact*** e la funzione `randomBacts`, utile a fornire caratteristiche randomiche di interazione con i nutrienti ad una specie batterica.

Il modulo che si sta analizzando necessita per il suo funzionamento dell'importazione della funzione `randomFill` presente in '`cell.py`' e dei moduli *numpy* e *random*. Essi, perciò, vengono importati nelle prime righe del modulo stesso.

La classe ***Bact*** nel corso dell'implementazione del pacchetto 'Bactlife' e dello sviluppo dell'interfaccia grafica ha subito modifiche rispetto al codice della prima versione del simulatore (precedentemente presentata in 1.1.1 Classi e Funzioni). Analizzando la classe si può notare che il vettore *_species*, gli attributi: *_type*, *_m*, *_t*, *_maxGr*, *_maxTox*, *_pos*, i metodi d'accesso `type`, `getm`, `gett`, `getmaxGr` e `getpos` e il metodo `getTox` sono rimasti invariati. Le modifiche svolte hanno coinvolto i metodi della classe ***Bact*** e hanno provocato l'inserimento del metodo `set_m`, del metodo `set_t` (utili per la modifica delle caratteristiche della specie batterica) e la rimozione del metodo `getGrowth`.

3.1.2: cell.py

Il file '`cell.py`' contiene la classe e le funzioni in grado di gestire le celle che compongono il vettore in cui viene eseguita la simulazione e tutti i dati delle specie batteriche che popolano ogni cella. Si trovano in questo file la classe ***Cell*** e le funzioni `Relative`, `RelativeAll` e `randomFill`.

Per eseguire correttamente tutti gli attributi presenti nel modulo '`cell.py`' è necessario importare alcuni moduli esterni, nello specifico: *math*, *randint* dal pacchetto *random* e *multivariate_hypergeom* dal pacchetto *scipy.stats*.

Rispetto alla versione precedente della classe ***Cell***, quella implementata in 'Bactlife' presenta notevoli cambiamenti che conferiscono nuove e più complete funzionalità all'intero simulatore. Partendo dall'analizzare gli attributi di un oggetto generico 'Cell', si può notare che, oltre a *_food*, *_MatrFood*, *_bact*, *_pos* e *_x*, sono stati aggiunti:

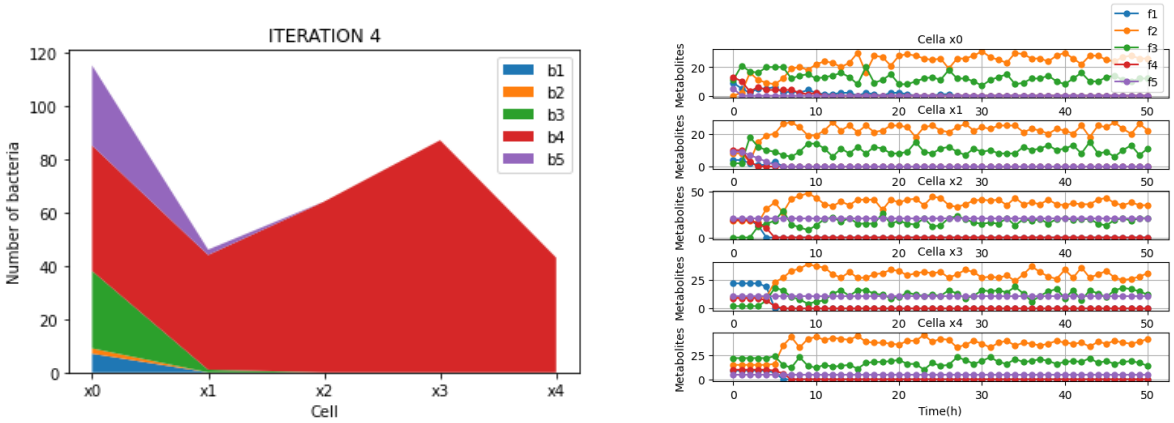
- *_MatrGrowth*, lista di coefficienti di crescita assunti dalla cella ad ogni iterazione;
- *_bactCell*, dizionario che presenta la percentuale di ogni specie batterica rispetto al numero di batteri presenti nell'oggetto cella considerato;
- *_bactAll*, dizionario che presenta la percentuale di ogni specie batterica rispetto al numero totale di batteri presenti in tutto il vettore di celle.

Per ogni attributo appena elencato è stato aggiunto un metodo di accesso: `getMatrGrowth`, `getBactCell`, `getBactAll`. I metodi della versione iniziale di ***Cell*** presentata precedentemente continuano ad esserci ma nel pacchetto 'Bactlife' risultano arricchiti dalla presenza del metodo `getGrowth` e del metodo `RelUpd`. Il primo proviene dalla rimozione appena presentata dalla classe ***Bact***; il secondo metodo invece è stato creato ad hoc per ottenere le versioni normalizzate (*_bactCell* e *_bactAll*) del dizionario di batteri.

3.1.3: set_df.py e show.py

I moduli '`set_df.py`' e '`show.py`' completano le funzionalità del pacchetto 'Bactlife'. Essi sono composti solamente da funzioni con lo scopo, nel caso di '`show.py`', di presentare i dati di ogni iterazione usando anche supporti grafici, nel caso di '`set_df.py`', di permettere l'interazione tra il programma e l'interfaccia grafica tramite l'uso di *dataframe*.

I moduli che sono necessari da importare in questi due file sono: *matplotlib.pyplot*, *pandas*, *numpy* e tutti gli attributi contenuti in `cell.py` e `bact.py`.




```
ITERATION 4
x0: {'b1': 7, 'b2': 2, 'b3': 29, 'b4': 47, 'b5': 30}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 43, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 64, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 87, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 43, 'b5': 0}
```

Figura 7: Nel grafico a sinistra viene presentato il numero (asse y) di ogni specie batterica in ogni cella (asse x) ad una determinata iterazione (in questo caso si è considerata la quarta). Nel grafico a destra vengono presentati gli andamenti temporali di ogni nutriente in ogni cella. In basso è presente l'output prodotto ad ogni iterazione dalla funzione printState.

Il file '`show.py`' racchiude in sé tutte le funzioni necessarie alla rappresentazione grafica dei dati di ogni iterazione (si veda la Figura 7); queste funzioni erano già presenti nella prima

versione del codice presentata in [1] e sono `graph`, `graph_met` e `printState`. I risultati prodotti da tali funzioni sono presentati nelle figure seguenti:
Il file '`set_df.py`' presenta due funzioni create durante lo sviluppo di 'Bactlife' e della sua interfaccia grafica:

- `set_from_df`: avente lo scopo di poter far modificare all'utente le caratteristiche delle specie batteriche interagendo solo con l'interfaccia grafica;
- `data_export`: funzione che crea un file excel contenente i dati notevoli della simulazione;

3.2: Nuovi File

Nei paragrafi precedenti è stata messa in evidenza la struttura dei moduli che compongono 'Bactlife'. Risulta evidente che la prima versione del codice proposta differisce da tale pacchetto non solo per l'organizzazione delle funzioni e per l'aggiunta dell'interfaccia grafica ma anche per le modifiche eseguite per creare un *package* distribuibile. Oltre a quest'ultime, sono state implementate delle funzionalità che permettono un efficiente interconnessione tra codice di esecuzione ed elaborazione dati del simulatore (*back end*) ed interfaccia grafica per l'utente (*front-end*). Concretamente questi miglioramenti sono stati perseguiti con la creazione di file accessori e di funzioni aggiunte ai moduli del pacchetto.

3.2.1: File accessori

Esistono dei file che devono essere contenuti all'interno del pacchetto per garantirne il suo corretto funzionamento; durante lo sviluppo di 'Bactlife', si è reso dunque necessario creare ed inserire tali file in punti specifici.

- '`__init__.py`' deve essere contenuto all'interno di ogni cartella del pacchetto. Nel caso più semplice, può essere anche vuoto ma, in generale, contiene istruzioni da eseguire al momento dell'importazione del pacchetto. Oltre alla funzione appena descritta, questo file risulta indispensabile nel comunicare che la cartella corrente in cui si trova dev'essere considerata come un *package*.
- '`setup.py`': in questo file Python viene prima importata e poi eseguita la funzione `setup` dal pacchetto *setuptools*, in particolare vengono attribuite ai parametri di tale funzione le caratteristiche tecniche (come per esempio: nome, versione, licenza, descrizione, pacchetti richiesti…) del pacchetto che si vuole distribuire.
- '*__all__*' e '*__name__*': il primo è una lista rappresentante l'insieme di attributi che vengono importati a seguito di una possibile esecuzione del comando `from` *packagename* `import *`

Il secondo è una variabile stringa presente in ogni modulo e che contiene il nome del modulo stesso, ad eccezione del caso in cui il modulo non sia importato ma eseguito direttamente come programma principale: a '__*name*__' viene assegnato '__main__'. Quest'ultima caratteristica è molto utile per testare il modulo, aggiungendo (come nel codice della figura sottostante) una condizione `if` in grado di discernere l'utilizzo che si sta facendo del modulo e svolgere delle righe di *test-code* (si veda Codice 2 in basso).

```
def f(x):
      y=x**x
      return y

if __name__==”main”:
      print “testing…”
      print “passing the value 2”
      z=f(2)
      print “the function returns”,z
```

Codice 2: Righe di codice in cui si definisce una funzione e, sfruttando le proprietà di __name__, si inseriscono del righe di *test-code*. Codice adattato da [15] a pagina 192

3.2.2: Funzioni aggiunte

Funzioni set

I metodi che sono stati aggiunti alla prima versione della classe ***Bact*** sono `set_m` e `set_t`. Essi hanno lo scopo di assegnare rispettivamente al vettore *m* e al vettore *t* della classe ***Bact*** una lista di interi ciascuno. Tale lista è ottenuta da un *dataframe* proveniente dall'uso dell'interfaccia grafica implementata e viene passata come parametro nei due metodi.

Per sovrascrivere ai vettori *m* e *t* con le liste che compongono il *dataframe*, è stata inserita, all'interno del modulo `set_df.py`, la funzione `set_from_df`. Questa funzione, che accetta come parametro in ingresso il *dataframe* stesso, seleziona per ogni specie batterica la colonna che ha come per intestazione 'i' e 't' ('i' indica il vettore del metabolismo, 't' invece il vettore delle tossicità della popolazione) seguiti dal tipo di specie. Tali colonne sono usate come parametri in ingresso per le funzioni `set_m` e `set_t` appena presentate.

getGrowth

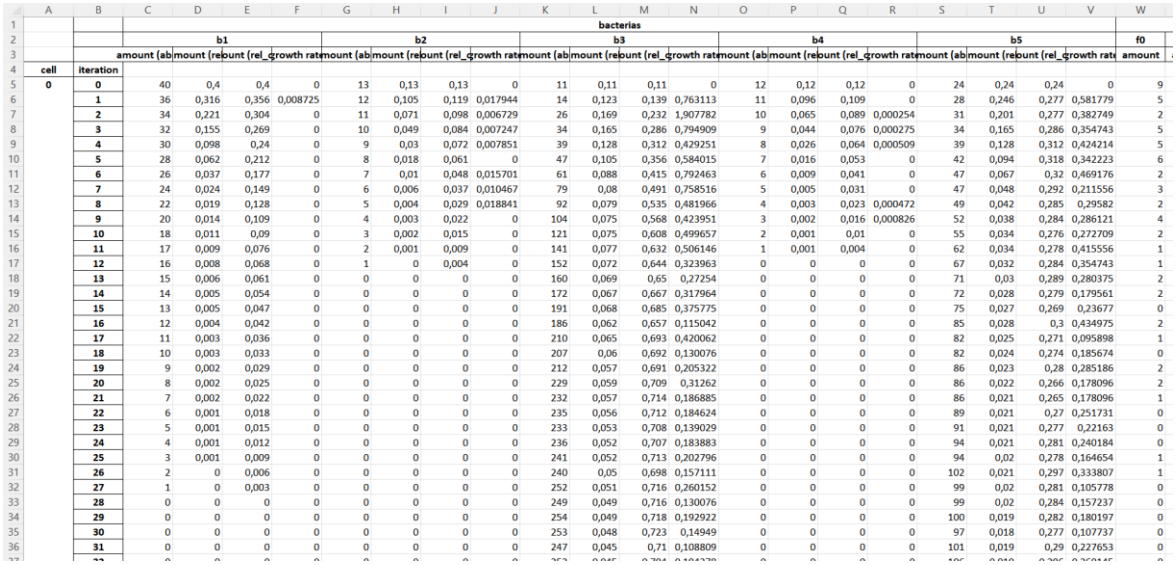
Il metodo `getGrowth` ha subito uno spostamento nel corso del progetto, passando da essere un metodo della classe ***Bact*** ad esserne uno della classe ***Cell***. Questa modifica è dovuta alle caratterizzazioni che si è voluto dare a queste due classi. ***Bact*** con i suoi attributi e metodi deve riuscire a rappresentare le specie batteriche e le loro caratteristiche; la classe ***Cell***, invece, possiede la capacità di estrarre informazioni dalle specie batteriche e dai metaboliti presenti in una specifica cella. Da questo punto di vista, si è ritenuto più appropriato attribuire il metodo `getGrowth` alla classe ***Cell*** piuttosto che a ***Bact***.

Funzioni Relative

Per aumentare le funzionalità e i dati che possono essere visualizzati nell'interfaccia grafica è stato necessario aggiungere il metodo `RelUpd` alla classe ***Cell***. Nella sua esecuzione, tale metodo utilizza due funzioni presenti sempre in `cell.py`: `Relative` e `RelativeAll`; esse hanno lo scopo di generare dei dizionari aventi per chiavi i nomi delle specie batteriche e per valori le percentuali delle abbondanze delle specie batteriche presenti nell'ambiente di sviluppo. Mentre la prima funzione divide il numero di batteri di una specie in una determinata cella per la somma dei batteri presenti nella cella stessa, `RelativeAll` necessita di avere come parametro d'ingresso anche il vettore x (lista di oggetti cell), in quanto calcola la percentuale rispetto al numero totale di batteri presenti nell'insieme di celle. Combinando l'esecuzione di queste due funzioni, il metodo `RelUpd` è in grado di ottenere dizionari delle specie batteriche presenti in una data cella con valori normalizzati rispetto alla cella stessa o all'intero vettore. Questi dizionari vanno a comporre gli attributi *_bactCell* e *_bactAll* che, come si è visto precedentemente, hanno la forma di liste in cui ogni elemento è un dizionario rappresentante le percentuali di specie batteriche presenti in una specifica iterazione.

export

Infine, si è ritenuto necessario implementare la possibilità di salvare localmente in un foglio di calcolo i dati di una simulazione effettuata ritenuta di particolare interesse, un esempio del file prodotto è presentato nella Figura 8. Per fare ciò è stata creata, e inserita nel modulo `set_df.py`, la funzione `data_export`. Accettando come parametri in ingresso: (1) un dizionario rappresentante il numero di batteri di ogni specie microbica (considerata come chiave) in ogni cella e iterazione, (2) il numero di iterazioni, (3) la lunghezza del vettore di celle, (4) la lunghezza del vettore dei metaboliti, (5) il vettore di celle. Questa funzione



		bacterias																				
		b1				b2				b3				b4				b5				f0
		amount (ab	mount (re	ount (rel_	growth rat	mount (ab	mount (re	ount (rel_	growth rat	mount (ab	mount (re	ount (rel_	growth rat	mount (ab	mount (re	ount (rel_	growth rat	mount (ab	mount (re	ount (rel_	growth rat	amount
cell	iteration																					
0	0	40	0,4	0,4	0	13	0,13	0,13	0	11	0,11	0,11	0	12	0,12	0,12	0	24	0,24	0,24	0	9
	1	36	0,316	0,356	0,008725	12	0,105	0,119	0,017944	14	0,123	0,139	0,763113	11	0,096	0,109	0	28	0,246	0,277	0,581779	5
	2	34	0,221	0,304	0	11	0,071	0,098	0,006729	26	0,169	0,232	1,907782	10	0,065	0,089	0,000254	31	0,201	0,277	0,382749	2
	3	32	0,155	0,269	0	10	0,049	0,084	0,007247	34	0,165	0,286	0,794909	9	0,044	0,076	0,000275	34	0,165	0,286	0,354743	5
	4	30	0,098	0,24	0	9	0,03	0,072	0,007851	39	0,128	0,312	0,429251	8	0,026	0,064	0,000509	39	0,128	0,312	0,424214	5
	5	28	0,062	0,212	0	8	0,018	0,061	0	47	0,105	0,356	0,584015	7	0,016	0,053	0	42	0,094	0,318	0,342223	6
	6	26	0,037	0,177	0	7	0,01	0,048	0,015701	61	0,088	0,415	0,792463	6	0,009	0,041	0	47	0,067	0,32	0,469176	2
	7	24	0,024	0,149	0	6	0,006	0,037	0,010467	79	0,08	0,491	0,758516	5	0,005	0,031	0	47	0,048	0,292	0,211556	3
	8	22	0,019	0,128	0	5	0,004	0,029	0,018841	92	0,079	0,535	0,481966	4	0,003	0,023	0,000472	49	0,042	0,285	0,29582	2
	9	20	0,014	0,109	0	4	0,003	0,022	0	104	0,075	0,568	0,423951	3	0,002	0,016	0,000826	52	0,038	0,284	0,286121	4
	10	18	0,011	0,09	0	3	0,002	0,015	0	121	0,075	0,608	0,499657	2	0,001	0,01	0	55	0,034	0,276	0,272709	2
	11	17	0,009	0,076	0	2	0,001	0,009	0	141	0,077	0,632	0,506146	1	0,001	0,004	0	62	0,034	0,278	0,415556	1
	12	16	0,008	0,068	0	1	0	0,004	0	152	0,072	0,644	0,323963	0	0	0	0	67	0,032	0,284	0,354743	1
	13	15	0,006	0,061	0	0	0	0	0	160	0,069	0,65	0,27254	0	0	0	0	71	0,03	0,289	0,280375	2
	14	14	0,005	0,054	0	0	0	0	0	172	0,067	0,667	0,317964	0	0	0	0	72	0,028	0,279	0,179561	2
	15	13	0,005	0,047	0	0	0	0	0	191	0,068	0,685	0,375775	0	0	0	0	75	0,027	0,269	0,23677	0
	16	12	0,004	0,042	0	0	0	0	0	186	0,062	0,657	0,115042	0	0	0	0	85	0,028	0,3	0,434975	2
	17	11	0,003	0,036	0	0	0	0	0	210	0,065	0,693	0,420062	0	0	0	0	82	0,025	0,271	0,095898	1
	18	10	0,003	0,033	0	0	0	0	0	207	0,06	0,692	0,130076	0	0	0	0	82	0,024	0,274	0,185674	0
	19	9	0,002	0,029	0	0	0	0	0	212	0,057	0,691	0,205322	0	0	0	0	86	0,023	0,28	0,285186	2
	20	8	0,002	0,025	0	0	0	0	0	229	0,059	0,709	0,31262	0	0	0	0	86	0,022	0,266	0,178096	2
	21	7	0,002	0,022	0	0	0	0	0	232	0,057	0,714	0,186885	0	0	0	0	86	0,021	0,265	0,178096	1
	22	6	0,001	0,018	0	0	0	0	0	235	0,056	0,712	0,184624	0	0	0	0	89	0,021	0,27	0,251731	0
	23	5	0,001	0,015	0	0	0	0	0	233	0,053	0,708	0,139029	0	0	0	0	91	0,021	0,277	0,22163	0
	24	4	0,001	0,012	0	0	0	0	0	236	0,052	0,707	0,183883	0	0	0	0	94	0,021	0,281	0,240184	0
	25	3	0,001	0,009	0	0	0	0	0	241	0,052	0,713	0,202796	0	0	0	0	94	0,02	0,278	0,164654	1
	26	2	0	0,006	0	0	0	0	0	240	0,05	0,698	0,157111	0	0	0	0	102	0,021	0,297	0,333807	1
	27	1	0	0,003	0	0	0	0	0	252	0,051	0,716	0,260152	0	0	0	0	99	0,02	0,281	0,105778	0
	28	0	0	0	0	0	0	0	0	249	0,049	0,716	0,130076	0	0	0	0	99	0,02	0,284	0,157237	0
	29	0	0	0	0	0	0	0	0	254	0,049	0,718	0,192922	0	0	0	0	100	0,019	0,282	0,180197	0
	30	0	0	0	0	0	0	0	0	253	0,048	0,723	0,14949	0	0	0	0	97	0,018	0,277	0,107737	0
	31	0	0	0	0	0	0	0	0	247	0,045	0,71	0,108809	0	0	0	0	101	0,019	0,29	0,227653	0

Figura 8: Esempio di tabella che viene creata per esportare i dati significativi della simulazione.

produce un *dataframe* in cui vengono riportati tabulati i dati ottenibili da tutte queste fonti di informazioni.

Le modifiche effettuate hanno permesso di migliorare il simulatore in termini di coerenza nella struttura, rendendola più intuitiva, e con maggiori funzionalità. Inoltre, il software risulta pronto per modifiche future e per una maggiore integrazione con l'interfaccia grafica.

3.3: Creazione e caricamento del pacchetto

Successivamente all'inserimento dei file appena citati (`__init__.py` e `setup.py`), ci si è occupati della realizzazione di un pacchetto facilmente distribuibile ([6] e [7]). Per far ciò è stato necessario installare innanzitutto i pacchetti di distribuzione, ossia degli archivi che vengono caricati nell'indice del pacchetto Python e possono essere installati usando il comando `pip` nella *command window*:

```
py -m pip install --upgrade build          (versione per Windows)
python3 -m pip install --upgrade build     (versione per Unix/macOS)
```

Successivamente, si può andare a creare la versione distribuibile del pacchetto, eseguendo il seguente comando dalla *command window* posizionata nella stessa cartella in cui si trova il file `setup.py`:

```
py -m build                                (versione per Windows)
python3 -m build                           (versione per Unix/macOS)
```

Questo comando, una volta completato, produce come output una cartella denominata ‘dist’ in cui sono presenti due file (si veda Figura 9), necessari per la condivisione del pacchetto in formato compresso.

```
dist/
├── example_package_YOUR_USERNAME_HERE-0.0.1-py3-none-any.whl
└── example_package_YOUR_USERNAME_HERE-0.0.1.tar.gz
```

Figura 9: Cartella e file che vengono creati a seguito dei comandi sopracitati, immagine riadattata da [6]

Si è terminato così il processo di creazione del pacchetto e si può procedere al suo caricamento sulla piattaforma GitLab.

Mantenendosi posizionati nella cartella contenente il file `setup.py` del pacchetto creato, si apre la *command window* di GIT e tramite il comando `git status` si ottiene in output la lista di modifiche che sono state effettuate rispetto alla versione precedente. Per ogni file che è stato aggiunto o modificato è stato necessario eseguire il seguente codice: `git add file_name.py` , in tal modo vengono salvati localmente i cambiamenti svolti. Si è proseguito poi con l’attribuzione del nome della versione del progetto appena creata usando la scrittura: `git commit -m version_name` e con il caricamento online della stessa: `git push` .

Il pacchetto risulta in questo modo caricato sulla pagina del progetto GitLab precedentemente creata ed è scaricabile da qualsiasi utente, indipendentemente dall’essere utente o meno del sito.

3.4: Guida all’utilizzo di ‘Bactlife’

Per rendere il caricamento concettualmente completo è stato necessario scrivere e caricare dei file che riuscissero a presentare in maniera intuitiva e allo stesso tempo esaustiva lo scopo, le potenzialità e i meccanismi del pacchetto ‘Bactlife’. Questi file, presenti su GitLab assieme al *package* stesso, guidano l’utente nell’installazione e nell’uso del programma indipendentemente dal sistema operativo posseduto.

Readme.md

Il principale file che fornisce istruzioni sull’installazione e sulle caratteristiche generali del pacchetto è `Readme.md`. Questo file di testo è scritto in linguaggio Markdown ed è suddiviso in questi paragrafi:

- *How to install Bactlife*: una breve guida al download e all'installazione del pacchetto. Il simulatore può essere scaricato sottoforma di file Zip oppure clonando la cartella (questa seconda opzione è praticabile solo dagli utenti registrati su GitLab) e per funzionare ha bisogno dell'ambiente di programmazione Python. Una volta installati entrambi, ci si posiziona nella cartella creata dall'estrazione del file zip e si scrive nella *command window*:

  ```
  pip install -r requirements.txt
  ```

  Tale comando installa tutti i moduli elencati nel file `requirements.txt` e che sono necessari al corretto funzionamento di 'Bactlife'.

  Il processo d'installazione del pacchetto si conclude scrivendo sulla *command window*:

  ```
  pip install -e .
  ```

- *How to run Bactlife*: il pacchetto è stato implementato garantendo la possibilità sia di usare l'interfaccia grafica che di interagire direttamente col codice Python e i risultati da esso prodotti. Inoltre, vengono riportati anche due codici esemplificativi di questi due utilizzi, rispettivamente: `app.py` e `test.py`
- *How to run a simulation without GUI*: il file `test.py` presenta un codice che importa e utilizza 'Bactlife' senza ricorrere all'interfaccia grafica; i risultati prodotti da tale programma sono presenti nella Figura 7.
- *About the app* (paragrafo composto da *How to use the app* e *Screenshots*): come per il file `test.py`, viene esposto come lanciare e come interpretare i risultati di `app.py`. L'interfaccia che viene proposta è rappresentata nella Figura 10.

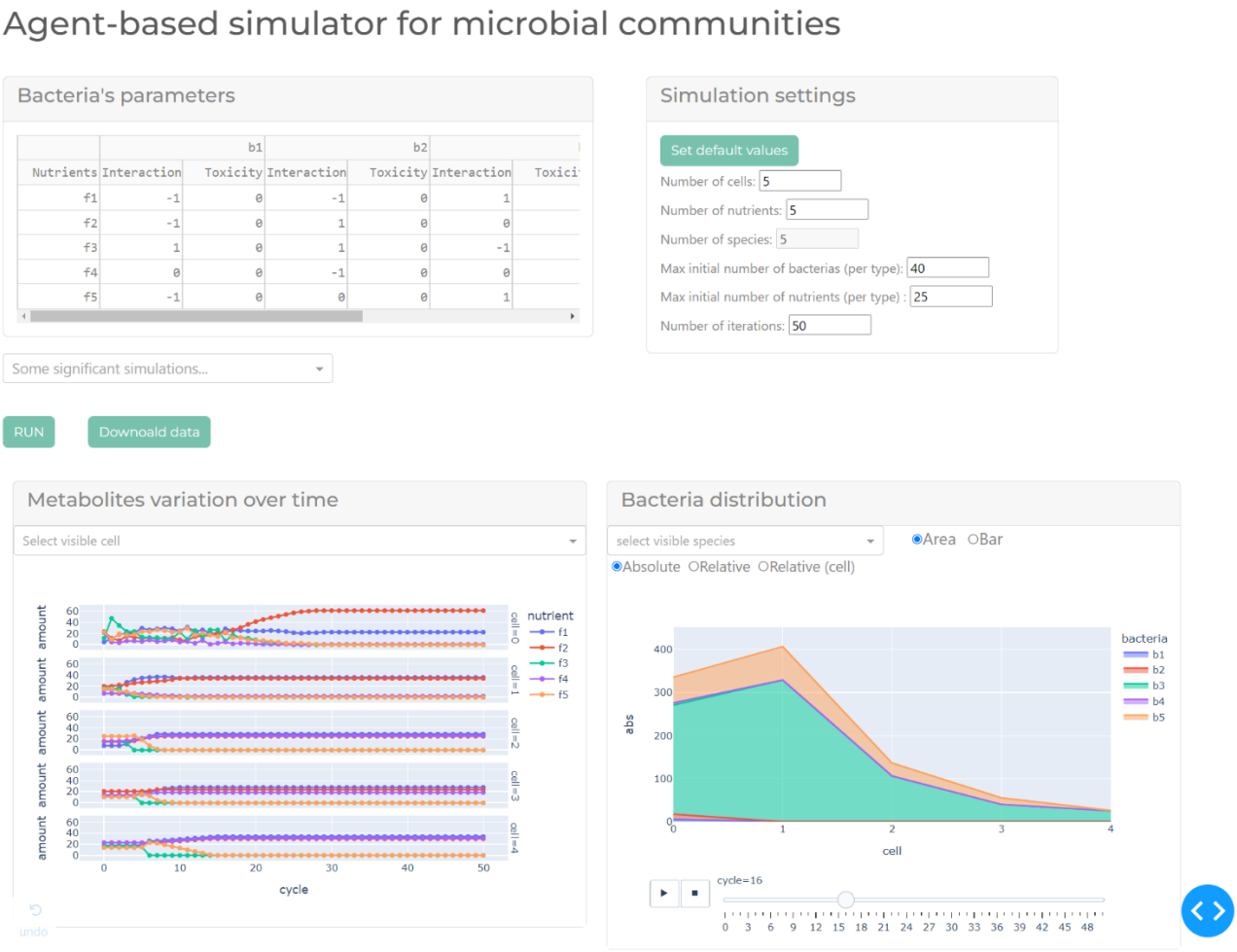
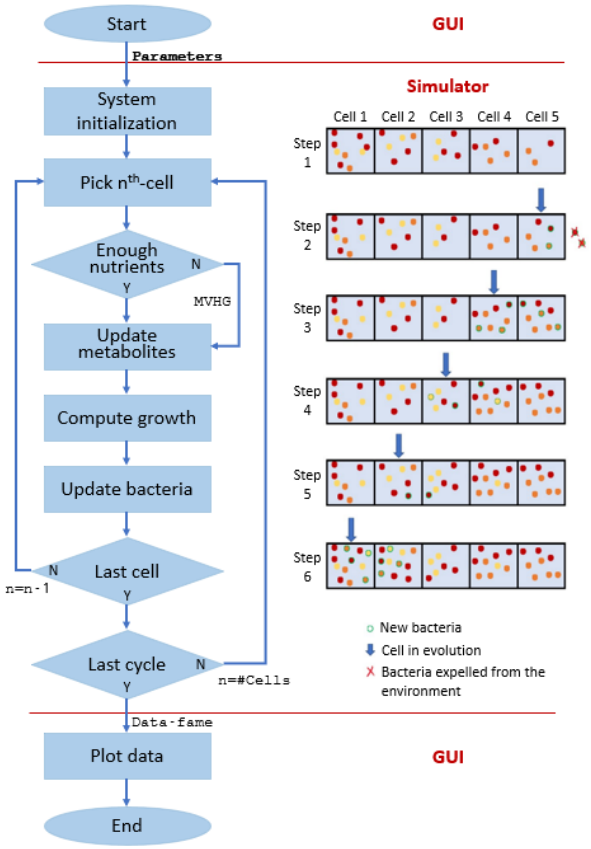


Figura 10: Immagine rappresentante l'interfaccia grafica sviluppata. In alto si possono consultare e modificare le caratteristiche di ogni specie batterica rispetto ogni nutriente (a sinistra) e le caratteristiche generali della simulazione. In basso vengono presentati i grafici prodotti dalla simulazione: lo sviluppo temporale dei nutrienti in ogni cella (a sinistra) e a destra l'andamento delle popolazioni batteriche nello spazio (asse x) e nel tempo (si può selezionare l'iterazione interessata).

Tutorial.ipynb

Questo file Jupyter Notebook presenta il codice, i risultati e le relative spiegazioni ad un esempio di utilizzo del pacchetto 'Bactlife' senza l'ausilio dell'interfaccia grafica. In primo luogo, vengono mostrate le righe di comando per importare tutte le componenti utilizzate del simulatore e viene presentata l'immagine 'introduction.png' (Figura 11), contenente: il diagramma di flusso dell'algoritmo seguito da 'Bactlife' durante la sua esecuzione, la rappresentazione grafica dell'esecuzione della funzione `evolution` ad ogni cella partendo

dall'ultima fino alla prima, l'output prodotto dalla funzione `printState()` ad ogni iterazione.




```
INITIAL STATE
x0: {'b1': 11, 'b2': 6, 'b3': 37, 'b4': 36, 'b5': 40}
x1: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 1
x0: {'b1': 10, 'b2': 5, 'b3': 35, 'b4': 59, 'b5': 36}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 26, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 2
x0: {'b1': 9, 'b2': 4, 'b3': 33, 'b4': 55, 'b5': 34}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 47, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 22, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 3
x0: {'b1': 8, 'b2': 3, 'b3': 31, 'b4': 51, 'b5': 32}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 45, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 58, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 37, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 4
x0: {'b1': 7, 'b2': 2, 'b3': 29, 'b4': 47, 'b5': 30}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 43, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 64, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 87, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 43, 'b5': 0}
```

Figura 11: Immagine rappresentante l'output prodotto dal programma (sulla destra) e il diagramma di flusso (sulla sinistra) seguito dal simulatore e dall'interfaccia grafica.

Viene successivamente introdotto lo scopo e la struttura del pacchetto scaricabile, presentando i moduli che lo compongono e le funzioni con le classi presenti in ciascuno di essi. Particolare attenzione è stata prestata all'introduzione delle variabili utilizzate all'interno delle classi (vedi Tabella 2).

Classe di appartenenza (Modulo)	**Nome Variabile**	**Descrizione**	**Tipo**	**Dimensione**
Bact (in bact.py)	_species	Vettore contenente i nomi di tutte le specie batteriche, comune a tutti gli elementi bact	Vettore di stringhe	s
	_type	Nome della specie batterica scelto dal vettore _species	Stringa	-

	_m	Vettore rappresentante il metabolismo della specie batterica per ogni nutriente. Se nella casella i-esima è presente il valore: o -1 allora la specie batterica consuma il nutriente i-esimo; o 0 allora il nutriente i-esimo viene ignorato dalla specie batterica; o 1 allora la specie batterica produce il nutriente i-esimo.	Vettore di interi	n
	_t	Vettore rappresentante la possibile tossicità di un nutriente nei confronti dell'elemento bact. Se nella casella i-esima è presente il valore: o -1 il nutriente i-esimo è tossico per l'elemento bact; o 0 il nutriente i-esimo viene ignorato dalla specie batterica considerato.	Vettore di interi	n
	_maxGr	Coefficiente di massima crescita della specie batterica.	Float	-
	_maxTox	Coefficiente di massima tossicità dell'elemento bact.	Float	-
	_pos	Posizione della specie batterica all'interno del vettore di elementi cell.	Intero	-
Cell (in cell.py)	_food	Vettore contenente i nutrienti presenti nella cella.	Vettore di interi	n
	_MatrFood	Matrice dei nutrienti nel tempo: la riga j-esima rappresenta il vettore _food all'iterazione j-esima	Matrice di interi	i × n
	_MatrGrowth	Vettore con i coefficienti di crescita assunti nel tempo dalle specie batteriche presenti all'interno della cella. La riga j-esima rappresenta i	Vettore di float	i × s

		coefficiente di crescita delle specie batteriche all'iterazione j-esima		
	_bact	Dizionario con chiavi i nomi delle specie batteriche e con valori il numero di batteri corrispondenti ad ogni specie	Dizionario di interi	s
	_bactCell	Dizionario _bact con valori normalizzati rispetto al numero totale di batteri presenti nella cella.	Dizionario di float	s
	_bactAll	Dizionario _bact con valori normalizzati rispetto al numero totale di batteri presenti nel vettore di celle.	Dizionario di float	s
	_pos	Posizione della cella nel vettore di elementi cell.	Intero	-
	_x	Vettore delle celle usate nella simulazione	Vettore di elementi cell	c

Tabella 2: Tabella delle variabili del simulatore aggiornata a seguito delle modifiche svolte.

Nella sezione '`Test.py`', viene analizzata l'esecuzione e il relativo significato dell'omonimo file presente assieme al pacchetto 'Bactlife'. Sono impostati i parametri principali della simulazione (*len_x* lunghezza del vettore, *len_m* lunghezza del vettore dei metaboliti, *max_bac* e *max_f* il numero massimo ad inizio simulazione rispettivamente di batteri di una specie e di nutrienti per tipo, *n_it* il numero di iterazioni che saranno simulate tenendo conto che ogni iterazione rappresenta un'ora di evoluzione batterica) e vengono creati le caratteristiche dei batteri in maniera casuale (numericamente e rispetto all'interazione con i metaboliti).

Terminata questa prima parte di presentazione di '`Test.py`' contrassegnata come 'Initial State', si procede all'analisi della vera e propria esecuzione del programma in 'Simulation'. Innanzitutto sono stampate a video le caratteristiche delle comunità batteriche appena create. Successivamente viene eseguita la proiezione dello stato iniziale del sistema presentando numericamente e graficamente ogni specie batterica presente per ogni cella. Segue quindi la simulazione dell'evoluzione di queste comunità batteriche tramite l'utilizzo iterativo della funzione `evolution` ad ogni cella e la presentazione numerica e grafica (grazie nuovamente a `printState()` e `graph()`) dei risultati di ciascuna iterazione. Si conclude l'esecuzione di

`Test.py` con il grafico ottenuto dalla funzione `graph_met()` rappresentante lo sviluppo nel tempo delle quantità di ogni metabolita in ogni cella.

Requirements.txt

In questo file di testo vengono elencati tutti i moduli e le loro versioni necessari al corretto funzionamento del pacchetto 'Bactlife'. Se si è seguita correttamente la procedura presentata nel paragrafo '*How to install Bactlife*' del file `Readme.md`, allora tali pacchetti sono già stati installati usando la scrittura:

```
pip install -r requirements.txt
```

dalla *command window* aperta sulla cartella precedentemente scaricata.

Questo file contiene la scrittura riportata nella Figura 12.

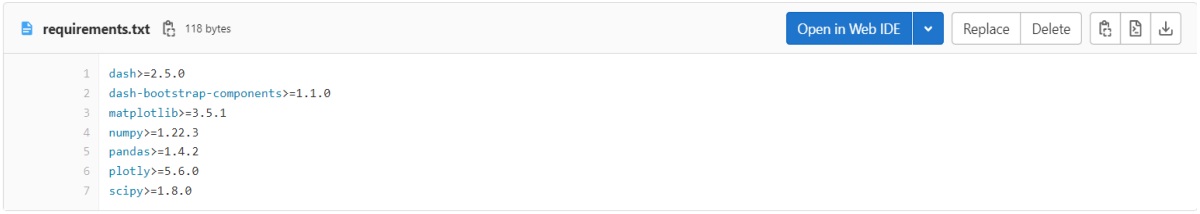


Figura 12: File requirements.txt.

Capitolo 4: Conclusioni

Questo elaborato si propone di presentare come è stato implementato un pacchetto in linguaggio Python di un simulatore di comunità batteriche, analizzando tutte le fasi che hanno portato a questo stadio tale progetto. Innanzitutto viene illustrata l'importanza e il ruolo che le popolazioni batteriche rivestono in natura e, in particolar modo, nel corpo umano. Si è proceduto con la identificazione delle problematiche riguardo gli studi e la sperimentazione in laboratorio. A seguito di queste motivazioni, è di notevole interesse la realizzazione di uno strumento predittivo in grado di simulare con precisione il comportamento microbico in risposta a stimoli esterni.

La modellizzazione scelta sfrutta l'idea di attribuire caratteristiche a priori a delle entità e di considerare tutti i fenomeni che si generano come conseguenza di interazione tra le stesse entità e tra entità e ambiente. Questo tipo di approccio è detto 'modello *Agent-Based*' (ABM). Successivamente viene presentato lo scheletro del simulatore creato, evidenziandone le principali funzionalità e i limiti. In particolare ci si focalizza sugli aspetti che hanno motivato la creazione di un pacchetto contenente una versione più completa del programma, quali la possibilità di: (1) rendere il programma modulare e strutturato, (2) condividere tale progetto con qualsiasi figura scientifica interessata e infine (3) fornire all'utente importanti funzionalità, come l'esportazione dei risultati e dei parametri iniziali.

I principali strumenti che hanno reso possibile lo sviluppo di 'Bactlife' sono stati Python, specialmente per la capacità di generare e gestire pacchetti condivisibili, e GitLab, che ha permesso un'efficace gestione dei file in tutte le fasi del progetto. Quest'ultimo strumento è risultato particolarmente utile nell'agevolare il lavoro sugli stessi elementi da parte mia e di Rebecca Sara, la quale si è occupata in contemporanea allo sviluppo di un'interfaccia grafica *user-friendly* del simulatore.

Nonostante i passi avanti presentati in quest'elaborato, 'Bactlife' presenta dei limiti, come: non permettere all'utente di modificare il numero di specie presenti nella simulazione e impiegare troppo tempo nell'esecuzione del programma. Altre funzionalità interessanti che potrebbero essere implementate in futuro sono: (1) la modellizzazione di un bolo di sostanze nutritive che attraversa l'ambiente di sviluppo, (2) l'espansione dell'ambiente stesso in due dimensioni, (3) l'integrazione del programma con un database di specie batteriche e sostanze nutritive notevoli. In ottica futura si può pensare di validare i risultati prodotti da tale strumento con i risultati provenienti da esperimenti svolti in laboratorio.

Bibliografia

[1] A. Calzavara, Implementazione di un simulatore per comunità microbiche basato su un modello multi-agente. Tesi di laurea Triennale in Ingegneria biomedica, Università degli Studi di Padova, 2022.

[2] L. Dinca et al., Microbial soil biodiversity in beech forests of European mountains. *Canadian Journal of Forest Research*, 2021.

[3] P. Gerard, Gut microbiota and obesity. *Cellular and Molecular Life Sciences*, 2016.

[4] S. Tong-wen et al., Intestinal Microbiota in Sepsis. *Intensive Care Research*, 2022.

[5] A. Alqathama et al., The vital role of animal, marine, and microbial natural products against COVID-19. *Pharmaceutical Biology*, 2022.

[6] Sito web: https://packaging.python.org/en/latest/tutorials/packaging-projects/

[7] Sito web: https://python-packaging-tutorial.readthedocs.io/en/latest/setup_py.html

[8] S. Rebecca, Bactlife: simulatore per comunità batteriche – sviluppo di interfaccia grafica in Dash. Tesi di laurea Triennale in Ingegneria biomedica, Università degli Studi di Padova, 2022.

[9] D. Davar et al., Fecal microbiota transplant overcomes resistance to anti–pd-1 therapy in melanoma patients. *Science*, 2021.

[10] S. Mukherjee et al., Bacterial quorum sensing in complex and dynamically changing environments. *Nature Reviews Microbiology*, 2019.

[11] Y. Qian et al., Towards a deeper understanding of microbial communities: integrating experimental data with dynamic models. *Current Opinion in Microbiology*, 2021.

[12] A. R. Ives et al., Estimating community stability and ecological interactions from time-series data. *Ecological monographs*, 2003.

[13] P. Wang et al., Robust growth of escherichia coli. *Current biology*, 2010.

[14] Sito web: https://git-scm.com/book/it/v2/Per-Iniziare-Il-Controllo-di-Versione

[15] S. Maruch et al., Python for Dummies. 2006.

Appendice A: Moduli di 'Bactlife'

__init__.py

```
from random import *
import pandas as pd
import pdb
import matplotlib.pyplot as plt
import numpy
from dash import dash_table
```

bact.py

```
from bactlife.cell import randomFill
from random import *
import numpy
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

##upgrade metabolism vector
#@param m new metabolism vector
    def set_m(self,m):
        self._m=m
        return None

##upgrade toxicity vector
#@param t new toxicity vector
    def set_t(self,t):
        self._t=t
        return None

##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

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


##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_prod==False:
                m[randint(0,len_m-1)]=1
```

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


        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
```

cell.py

```
import math
from random import randint
from scipy.stats import multivariate_hypergeom as MVHG

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._MatrGrowth=list() #trace growth rate in time
        self._bact=bact #dict={bact_obj:number of bacteria}
        self._bactCell=list() #trace cell-relative number of bacteria
        self._bactAll=list() #trace relative 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 list of growth rate
    def getMatrGrowth(self):
        return self._MatrGrowth

##@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

##@return a list of dict of bacteria
    def getBactCell(self):
        return self._bactCell

##@return a list of dict of bacteria
    def getBactAll(self):
        return self._bactAll

#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
            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[ba
c]#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)

##calculates the growth factor
#@param food the eaten food dictionary
#@param bac bact_obj of bacteria we are considering
#@param km Hill parameter
#@param e Hill parameter
#return growth factor(float in [0,maxGr) interval)
    def getgrowth(self,food,bac,km=0.5,e=1):
```

```
        num_bac=self._bact[bac]
        growth=0 #not eating=>nothing changes
        food_quantity=0
        for i in range(len(bac._m)): #for each nutrient
            if bac._m[i]<0: #for each consumed nutrient
                food_quantity+=food[i]
        if food_quantity!=0:

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



##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
        growth_cell=list()
        for bac in self._bact:
            growth=self.getgrowth(eaten[bac],bac)
            growth_cell.append(growth)
            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
        self._MatrGrowth.append(list(growth_cell))
        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

##update of relative and cell-relative list of dict of bacteria
    def RelUpd(self):
        bact=self.getBact()
        bact_rel=Relative(bact)
        bact_all=RelativeAll(bact,self._x)
        self._bactCell.append(bact_rel)
        self._bactAll.append(bact_all)
        return

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

##create a dict with relative values of input dict
#@param dict input dict
#@return a relative dict
def Relative(dic):
    keys=dic.keys()
    values=dic.values()
    tot=sum(values)
    new_dic=dict()
    for key in keys:
        if tot!=0:
            new_dic[key]=round(dic[key]/tot,3)
        else:
            new_dic[key]=0
    return new_dic

##create a dict with vector-relative values of input dict
```

```
#@param dict input dict
#@param x cells vector
#@return a vector-relative dict
def RelativeAll(dic,x):
    allbact=0
    for j in range(len(x)):
        bact=x[j].getBact()
        allbact=allbact+sum(bact.values())
    keys=dic.keys()
    new_dic=dict()
    for key in keys:
        if allbact!=0:
            new_dic[key]=round(dic[key]/allbact,3)
        else:
            new_dic[key]=0
    return new_dic

##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
```

set_df.py

```
import pandas as pd
from bactlife.bact import *
from bactlife.cell import *
import numpy

##update all the metabolism and toxicity vectors
```

```
#@param df dataframe with metabolism and toxicity values for each nutrient
and bacteria type
#@param bacts list of bacts
def set_from_df(df,bacts):
    for bact in bacts:
        head_i='i '+bact.type()
        head_t='t '+bact.type()
        m=list(df[head_i])
        bact.set_m(m)
        t=list(df[head_t])
        bact.set_t(t)
    return

##create a dataframe filled with bact and cell data
#@param bd dict with bact species as keys and number of bacteria for each
iter and cell
#@param n_it number of iter
#@param len_x length of cell vector
#@param len_m the length of the metabolism vector
#@param x cell vector
#@return df2 dataframe
def data_export(bd,n_it,len_x,len_m,x):
    rows=pd.MultiIndex.from_product([list(range(len_x)),list(range(n_it+1))],
                                    names=['cell','iteration'])
    cols1=pd.MultiIndex.from_product([['bacterias'],list(bd.keys()),
                                     ['amount (abs)','amount (rel)','amount
(rel_cell)','growth rate']])
    cols2=pd.MultiIndex.from_product([['nutrients'],list('f' + str(i) for i
in range(len_m)),['amount']])
    df2=pd.DataFrame(None,index=rows,columns=cols1)
    df=df2.join(pd.DataFrame(None,index=rows,columns=cols2))
    b=0
    for bac in bd:
        row=bd[bac]
        amt_b=numpy.array(numpy.array_split(row,n_it+1)).transpose()
        for cell in range(len(amt_b)):
```

```
            rel_cell=[]
            rel_all=[]
            for i in range(n_it+1):
                rel_cell.append(x[cell].getBactCell()[i][bac])
                rel_all.append(x[cell].getBactAll()[i][bac])
            gr=numpy.array(x[cell].getMatrGrowth()).transpose()[b].tolist()
            gr.insert(0,0)
            df.loc[cell,('bacterias',bac,'amount
(abs)')]=amt_b[cell].tolist()
            df.loc[cell,('bacterias',bac,'amount (rel)')]=rel_all
            df.loc[cell,('bacterias',bac,'amount (rel_cell)')]=rel_cell
            df.loc[cell,('bacterias',bac,'growth rate')]=gr
        b+=1
    for I in range(len_x):
        MF=numpy.matrix(x[I].getMatrFood()).transpose()
        for r in range(len(MF)):
            row=MF[r,:].tolist()[0]
            df.loc[I,('nutrients','f'+str(r),'amount')]=row
    return df
```

show.py

```
import matplotlib.pyplot as plt
##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.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
```

Appendice B: Readme.md e Tutorial.ipynb

Readme.md

README.md

How to install Bactlife

1. Download the git repository as a ZIP file, then unzip it. This will create a directory (folder) named after the GitHub repository. If you are a git user, you can clone the git repository
2. Install python (version >=3.10.2) (www.python.org/downloads/).
3. Open the command window in the folder you just created. You can do this by typing `cmd` in the address bar.
4. Install all dependencies listed in `requirements.txt`.

```
pip install -r requirements.txt
```

5. Install the package **bactlife**

```
pip install -e .
```

How to run Bactlife

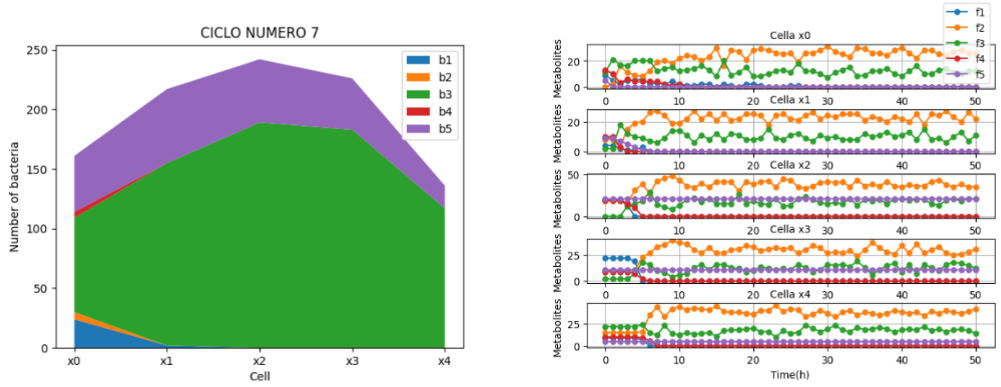
Bactlife is designed for simulating experiments on microbial communities. The package can be used without the GUI, as in the example file `test.py`, or it can be used with the app (a graphical interface created with Dash), as in the example file `app.py`.

How to run a simulation without GUI

The `test.py` file is an example of how to use the simulator without the GUI. After running `test.py`, the output will be:

- the initial state of every species
- the amount of bacteria in every iteraton
- a graph that shows, through a stackplot, the distribution of *bacteria* within the system
- a graph that shows the distribution of *nutrients* within the system

For explanations of the main classes and functions used in Bactlife, see the Jupyter notebook `tutorial.ipynb`.



About the app

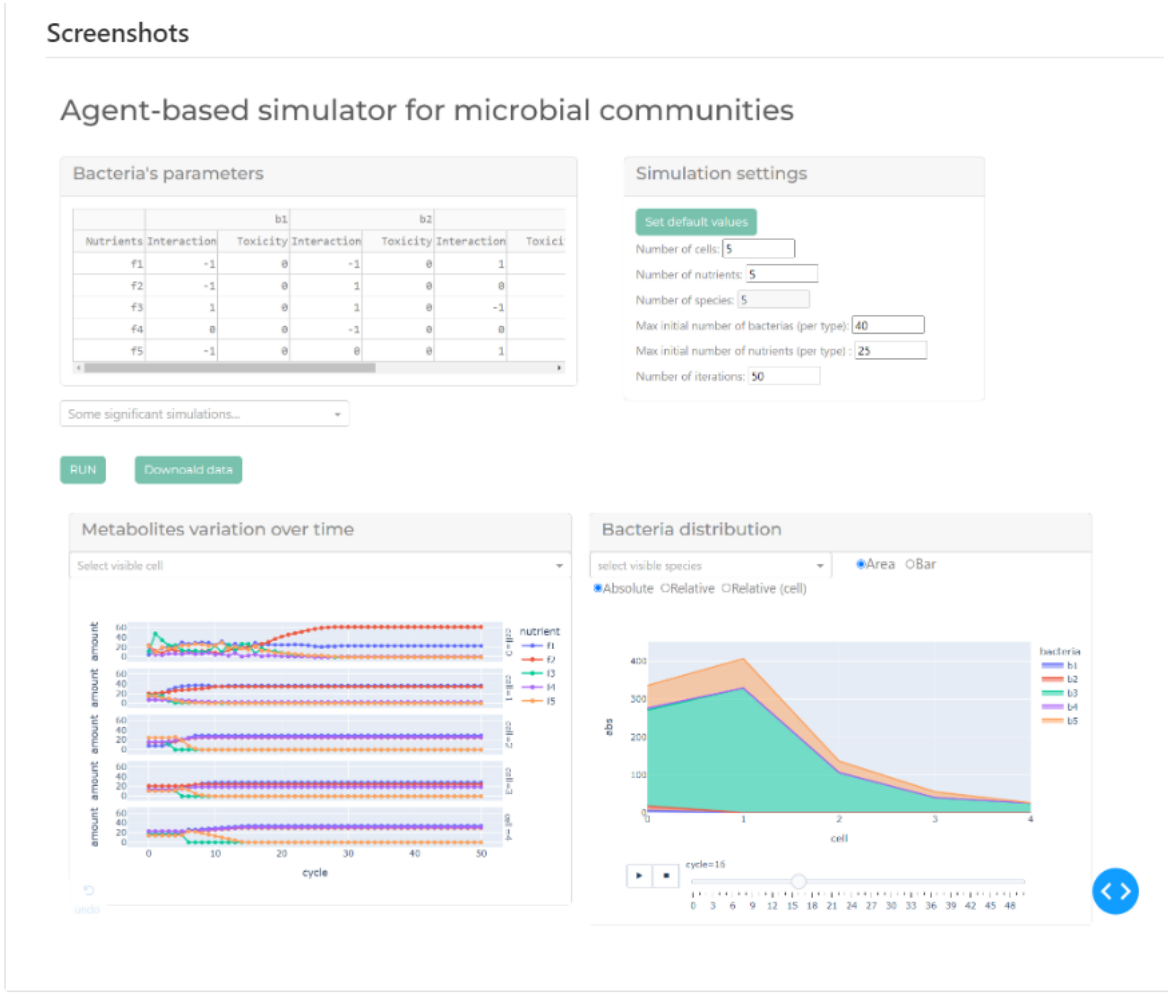
This app represents a simulator that describes the temporal evolution of microbial communities through a multiple agent structure.

How to use the app

Run `app.py` to launch a local Dash server to host the Dash app. A link will appear in your console; click this to use the Dash app.

```
python app.py
```

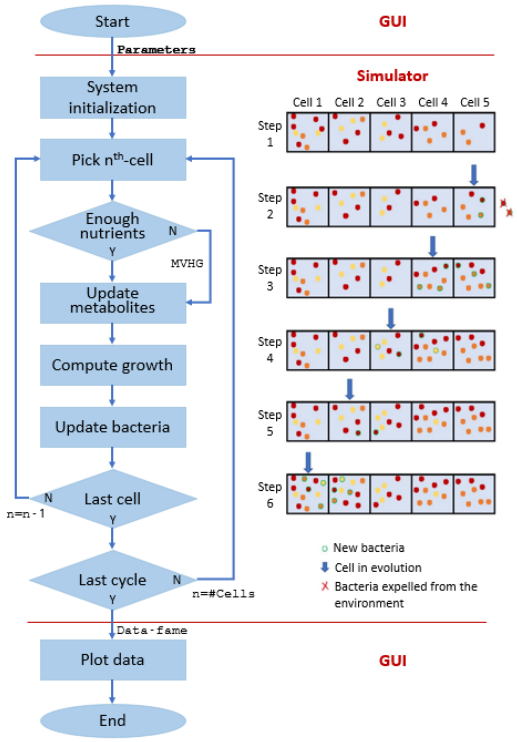
Screenshots



Tutorial.ipynb

```
from IPython.display import Image
from bactlife.cell import *
from bactlife.bact import *
from bactlife.show import *
```

```
Image('images/introduction.png')
```




```
INITIAL STATE
x0: {'b1': 11, 'b2': 6, 'b3': 37, 'b4': 36, 'b5': 40}
x1: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 1
x0: {'b1': 10, 'b2': 5, 'b3': 35, 'b4': 59, 'b5': 36}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 26, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 2
x0: {'b1': 9, 'b2': 4, 'b3': 33, 'b4': 55, 'b5': 34}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 47, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 22, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 3
x0: {'b1': 8, 'b2': 3, 'b3': 31, 'b4': 51, 'b5': 32}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 45, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 58, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 37, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
ITERATION: 4
x0: {'b1': 7, 'b2': 2, 'b3': 29, 'b4': 47, 'b5': 30}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 43, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 64, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 87, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 43, 'b5': 0}
```

Introduction

Bactlife is a agent-based simulator for microbial communities, it aims to simulate bacterial evolution in the digestive tract (one-dimensional) following food ingestion. The package consinsts of **two main classes** (which describe the characteristics of the bacterial species and of the cells created) and their methods and functions, organized in four distinct parts. These are:

[`bactlife.bact`] consists of class `bact` and the function `randomBacts`, which creates random characteristics for each species;

[`bactlife.cell`] consists of class `cell`, the functions `relative` and `relativeAll`, which make the values of the input dictionaries relative, and the function `randomFill` creates a list of n integers beetween two input numbers;

[`bactlife.set_df`] with the functions `set_from_df` and `data_export` updates all the metabolism and toxicity vectors using the values of an input dataframe and creates a dataframe filled with bact and cell data;

[`bactlife.show`] using `graph`, `graph_met` and `printState`, generates in output the nutrients graph and the bacteria graph (images in README.md 'How to run a simulation without GUI') and for each iteration the amount of bacteria for every species (figure above on the right).

Before getting into details of the functions, there is a summary of the main variables used by Bactlife (table below) and the simulation framework (figure above on the left).

```
Image('images/table.png')
```

Class (Module)	Variable	Description	Type	Dimension
Bact (bact.py)	_species	List of each species' name, it's present in every bact element	List of strings	s
	_type	Name of the species, it's an element of _species list	String	-
	_m	List representing the metabolism of the species. If the i-th element has the value: o -1 then the species consumes i-th nutrient; o 0 then the species ignores the i-th nutrient; o 1 then the species produces the i-th nutrient	List of integers	n
	_t	List representing the possibility of a nutrient being toxic to a species. If the i-th element has the value: o -1 then the i-th nutrient is toxic to the species; o 0 thn species ignores the i-ith	List of integers	n
	_maxGr	Maximum species' growth rate	Float	-
	_maxTox	Maximum toxicity rate of the species	Float	-
	_pos	Index of the cell occupied by the species	Integer	-
Cell (cell.py)	_food	List of cell's nutrients	List of integers	n
	_MatrFood	List of _food in time: the j-th row represents _food list at j-th iteration	List of lists	i × n
	_MatrGrowth	List of growth rate in time: j-th row represents the growth rates of the species at j-th iteration.	List of list	i × s
	_bact	Dictionary with the names of the species as key value and the amounts of the species as dictionary values.	Dictonary of integers	s
	_bactCell	_bact dictionary with values normalized by the total amount of bacteria in the cell.	Dictionary of float	s
	_bactAll	_bact dictionary with values normalized by the total amount of bacteria in the list of cell elements	Dictionary of float	s
	_pos	Index of the cell	Integer	-
	_x	List of cell elements	List of cell elements	c

Test.py

Initial state

In the first two lines, the *seed* functions ensure the reproducibility of samples created. Then we set main simulation parameters: len_x, len_m, max_bac, max_f and **n_it** (the number of cycles, **each iteration** in Bactlife is equivalent to **one hour of evolution** of the bacterial community). We generate a list of bact elements with randomly features, created using randomBacts function. The initial cell (pos=0) is populated by bacterial species, each with a random amount of bacteria. *d* is a useful variable for creating a dataframe to use the GUI. Also food is initialized with random amount in every cell created of the vector.

```
seed(7)
numpy.random.seed(20)

#set simulation parameters

len_x=5 #length of cells list
len_m=5 #length of metabolism (and food/toxicity) vector
n_it=4 #number of cycles
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 bacteria for empty cells
for bac in bacts:
    bac_diz[bac]=randint(0,max_bac)
    bac_null[bac]=0

d=dict()
d['nut']=list('f'+str(i+1) for i in range(len_m))
for b in bacts:
        d['i '+str(b.type())]=b.getm()
        d['t '+str(b.type())]=b.gett()
```

```
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

At the begging of the simulation, 'Bactlife' prints the features of each species: the **metabolism vector** (how a bact species interacts with a specific nutrient: 0 if the species ignore it, >0 if the species produces it, <0 if the species consumes it), the **max growth rate** and the **toxicity vector** (0 if the nutrient isn't toxic for the species, >0 if it is).

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

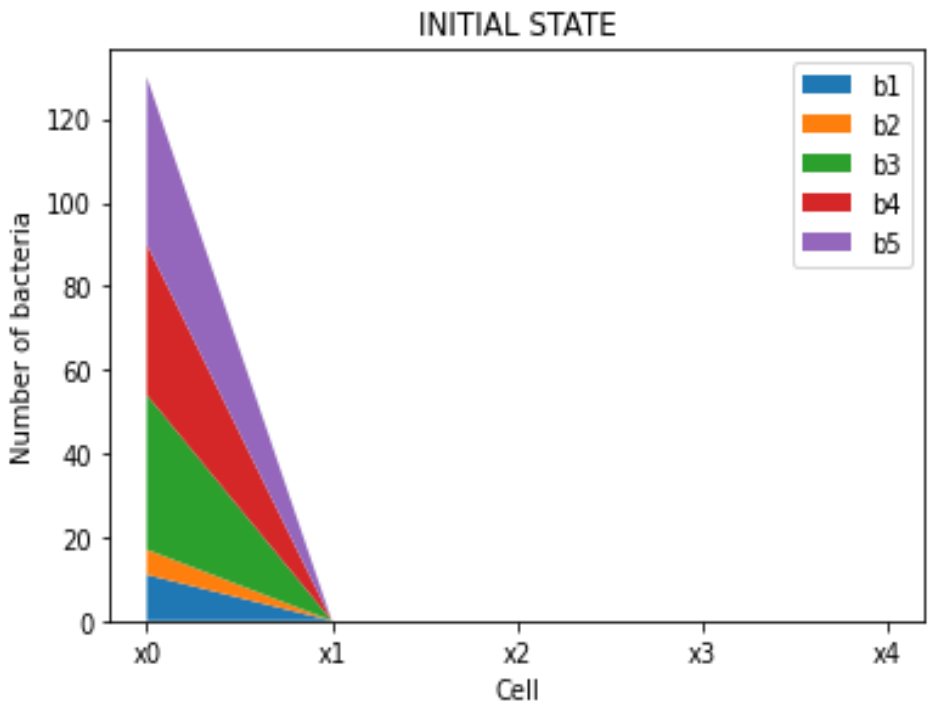
```
METABOLISM:
b1  :  [0, -1, 0, 1, -1] , mu_max=[0.00735953] , t= [0, 0, 0, 0, 0]
b2  :  [-1, 1, -1, -1, -1] , mu_max=[0.02863148] , t= [0, 0, 0, 0, 0]
b3  :  [-1, -1, -1, 1, 0] , mu_max=[0.02080694] , t= [0, 0, 0, 0, 0]
b4  :  [-1, 1, 1, 1, -1] , mu_max=[4.30706937] , t= [0, 0, 0, 0, 0]
b5  :  [-1,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

For each iteration *printState()* and *graph()* print the amount of bacteria for each species and the bacteria graph. These values can be relativized using *Relative()* and *RelativeAll()*.

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

for z in range(n_it):
```

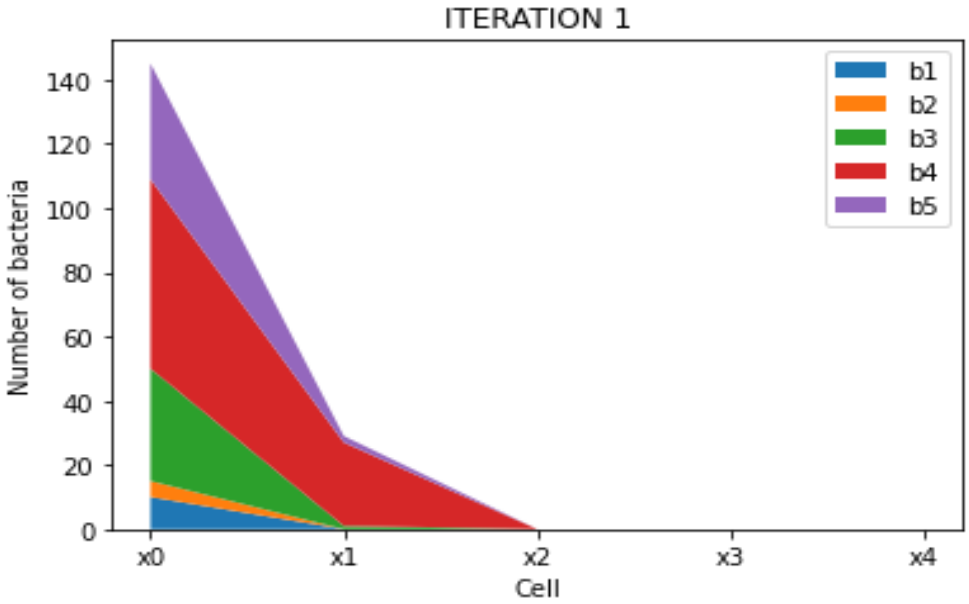
```
    title="ITERATION "+str(z+1)
    fig1=plt.figure()
    print(title)
    for i in range(1,len(x)+1):# evolution of cells
        x[-i].evolution()
    printState(x)
    graph(x,title,fig1)
```

```
INITIAL STATE
x0: {'b1': 11, 'b2': 6, 'b3': 37, 'b4': 36, 'b5': 40}
x1: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
```




```
ITERATION 1
x0: {'b1': 10, 'b2': 5, 'b3': 35, 'b4': 59, 'b5': 36}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 26, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}
```

<Figure size 432x288 with 0 Axes>



ITERATION 2

x0: {'b1': 9, 'b2': 4, 'b3': 33, 'b4': 55, 'b5': 34}

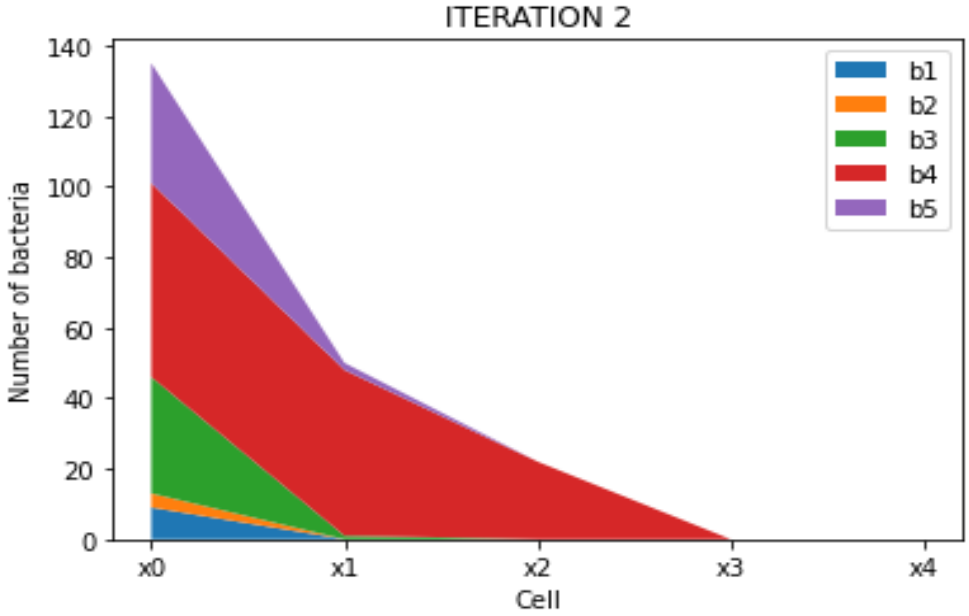
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 47, 'b5': 2}

x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 22, 'b5': 0}

x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}

x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}

<Figure size 432x288 with 0 Axes>



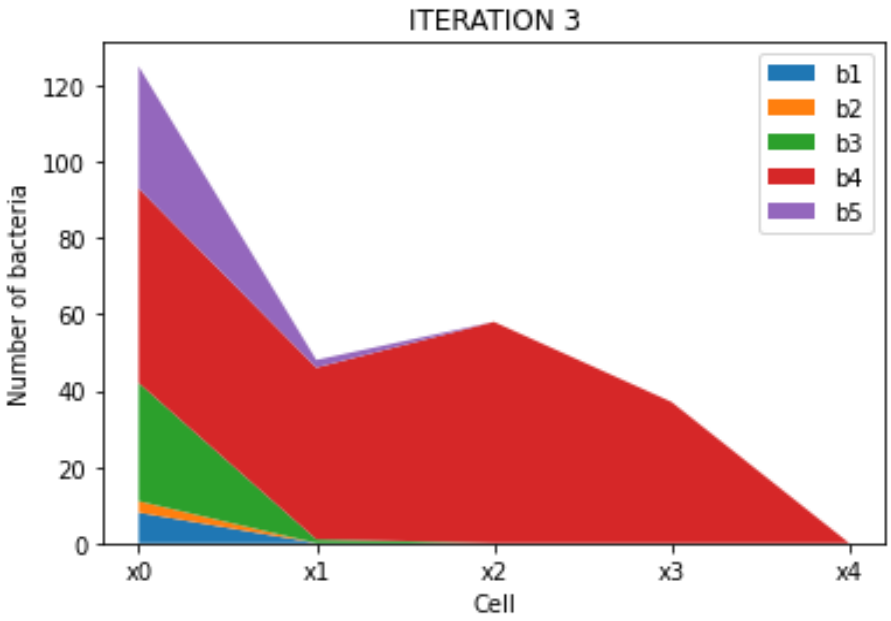
ITERATION 3

x0: {'b1': 8, 'b2': 3, 'b3': 31, 'b4': 51, 'b5': 32}

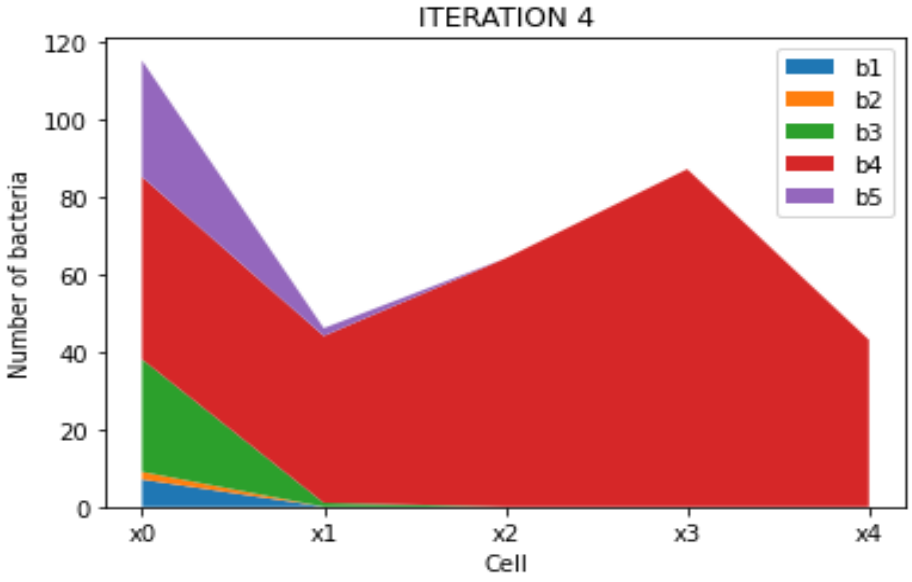
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 45, 'b5': 2}

x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 58, 'b5': 0}

```
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 37, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 0, 'b5': 0}

<Figure size 432x288 with 0 Axes>
```



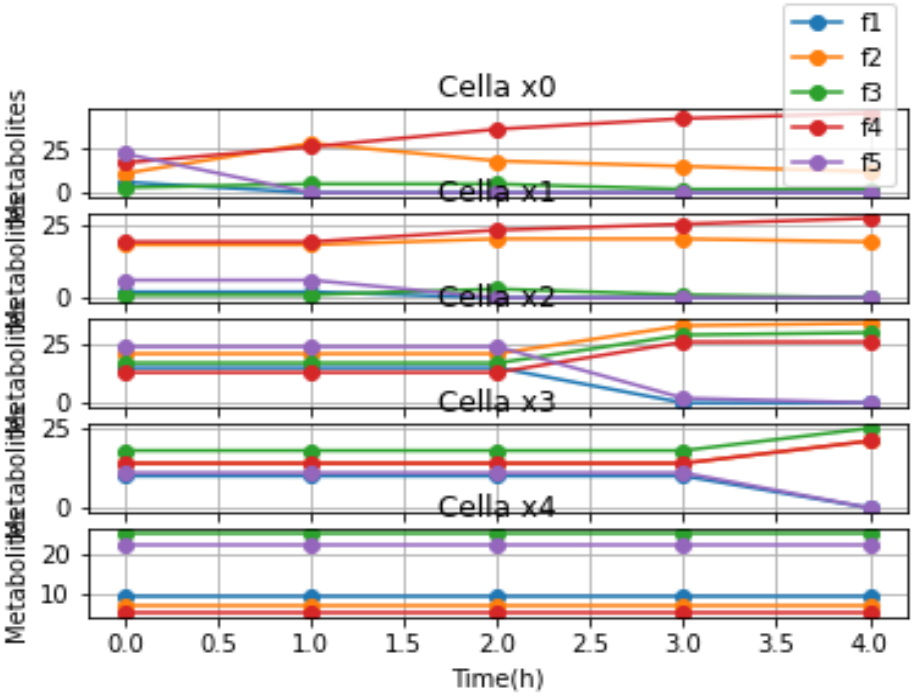

```
ITERATION 4
x0: {'b1': 7, 'b2': 2, 'b3': 29, 'b4': 47, 'b5': 30}
x1: {'b1': 0, 'b2': 0, 'b3': 1, 'b4': 43, 'b5': 2}
x2: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 64, 'b5': 0}
x3: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 87, 'b5': 0}
x4: {'b1': 0, 'b2': 0, 'b3': 0, 'b4': 43, 'b5': 0}

<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

graph_met() plots the development over time of the nutrients of each cell.

```
fig_m=plt.figure()
graph_met(x,fig_m)
```



Ringraziamenti

Un caloroso grazie a tutte le persone che mi hanno aiutato e sostenuto nell'arrivare fino a questa importante tappa del mio percorso.

Grazie al mio relatore Ing. Massimo Bellato e ai correlatori Dott. Marco Cappellato e Prof.ssa Barbara Di Camillo per avermi proposto questo lavoro di tesi, per avermi insegnato molto e per avermi seguito durante tutto il corso di questo progetto. Ho apprezzato la disponibilità, la precisione e l'interesse che hanno sempre dimostrato rispetto alle attività svolte e rispetto alle nostre impressioni nel vivere questo progetto. Ringrazio Sara Rebecca per la collaborazione svolta in questi mesi.

Grazie alla mia famiglia per essermi sempre stata vicina, per la pazienza e l'amore dimostrati. Un grazie a Giulia, per tutto. Un grazie a tutti i miei amici per aver reso il mio percorso un gioco.