Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in
Ingegneria Biomedica

Bactlife: simulatore per comunità batteriche – sviluppo di interfaccia grafica in Dash

Relatore:
ING. MASSIMO BELLATO, PHD

Laureanda:
SARA REBECCA
Matr. 1228824

Correlatori:
CHIAR.MA PROF.SSA BARBARA DI CAMILLO
DOTT. MARCO CAPPELLATO

Anno Accademico 2021/2022

Abstract

I ruoli essenziali che le comunità batteriche ricoprono in numerosi ambienti sono sempre più riconosciuti e studiati, tuttavia la comprensione delle complesse reti di interazioni tra i microbi e l'ambiente è ancora molto limitata. Per questo motivo, la modellizzazione dinamica e lo sviluppo di simulazioni computazionali rappresentano strumenti promettenti per prevedere le risposte delle comunità batteriche a perturbazioni e progettare interventi con lo scopo di manipolare queste comunità a nostro vantaggio. Il simulatore codificato in linguaggio Python alla base di questo progetto è basato su un modello *Agent Based* (ABM), basato cioè su una rappresentazione tramite entità (agenti) contraddistinte da una serie di attributi che ne definiscono l'interazione con l'ambiente.

Rendere il simulatore facilmente utilizzabile da qualsiasi utente, indipendentemente dalle conoscenze informatiche possedute, ha rappresentato il principale obiettivo di questo progetto.

In questo elaborato viene presentata la progettazione e l'implementazione di un'interfaccia *user-friendly* intuitiva e di facile utilizzo, che permette ad utenti non informatici di impostare ed eseguire simulazioni facilmente personalizzabili ed esportare in un formato adatto i risultati ottenuti.

Vengono illustrate tutte le tappe che hanno permesso lo sviluppo della GUI (*Graphical User Interface*), a partire dagli strumenti di cui ci si è serviti per la gestione e alla visualizzazione dei dati, ossia le librerie Python (*Pandas*) e (*Plotly*); fino alla spiegazione dei componenti grafici scelti per l'interfaccia durante la fase di progettazione e dei meccanismi che permettono e regolano le interazioni.

La GUI, insieme al *package* Python contenente il programma, è disponibile in un *repository* GitLab, una piattaforma che permette la distribuzione di un progetto e facilita la collaborazione di un team di sviluppatori che ci lavora. Nel presente elaborato, è presente una guida all'utilizzo del simulatore, a cominciare dal *download* della *repository* e l'installazione del pacchetto.

Nonostante i progressi nell'evoluzione del simulatore, non può ancora considerarsi un prodotto finito e totalmente attendibile. Ad ogni modo, è ragionevole pensare che la struttura modulare del simulatore è sufficientemente generalizzabile da poter essere adottata in un prossimo futuro, quando lo sviluppo scientifico e tecnologico renderà possibile ottenere tutti i dati necessari all'identificazione del modello.

Alcuni sviluppi futuri che aiuterebbero ad avvicinarsi ad uno strumento sempre più

attendibile potrebbero essere: l'ampliamento della caratterizzazione spaziale in modo da rappresentare ambienti in due e tre dimensioni, l'implementazione di un flusso dinamico di sostanze nutritive, l'integrazione di *database* contenenti informazioni note sull'interazione di specie batteriche e nutrienti e la validazione dei risultati tramite il confronto con una sperimentazione in laboratorio.

Indice

Capitolo 1

Simulatore di comunità microbiche

Il presente lavoro di tesi si colloca all'interno di un progetto più ampio, finalizzato allo sviluppo di un simulatore modulare dell'evoluzione di comunità microbiche, basato su modelli multi-agente. Nello specifico, questa tesi è focalizzata sulla progettazione e sviluppo di una GUI che permetta all'utente un facile utilizzo del simulatore ed una completa gestione dei moduli che lo compongono.

In questo capitolo verranno esposti i motivi per cui le simulazioni computazionali come quella in oggetto stanno acquisendo un ruolo importante nello studio delle comunità microbiche.

Verrà poi definito il modello scelto per rappresentare il sistema biologico e le interazioni tra le entità che lo popolano, precedentemente sviluppato e presentato nel lavoro di tesi triennale in Ingegneria Biomedica "Implementazione di un simulatore per comunità microbiche basato su un modello multi-agente" di A.Calzavara [1], illustrandone la struttura ad oggetti e il funzionamento iterativo.

Infine, verrà posta l'attenzione ai limiti della prima versione del simulatore, il cui superamento sta alla base dello svolgimento di questo lavoro di tesi e di quello presentato in "Bactlife: simulatore per comunità batteriche - sviluppo del pacchetto Python" di Alessandro Lucchiari, con il quale ho collaborato nel proseguimento del progetto.

1.1 Introduzione al modello

Le comunità batteriche sono costituite da diverse specie di microrganismi che interagiscono tra di loro in un determinato ambiente, il quale offre risorse che determinano l'evoluzione della comunità. L'interazione che si instaura tra le specie può generare un ampia casistica di situazioni, a seconda di composizione della comunità e condizioni ambientali; nei casi più semplici può essere ad esempio vantaggiosa, come quando i prodotti di scarto di alcuni microorganismi fungono da nutrienti per altri, o dannosa, quando invece gli scarti si rivelano tossici per le altre specie.

I microorganismi si trovano ovunque e svolgono funzioni essenziali per la salute umana e ambientale, formando complesse reti di interazioni, anche con l'organismo o l'ambiente

ospite, responsabili di numerosi processi biologici.

Ad esempio, le comunità microbiche presenti nel tratto digestivo umano prendono il nome di "microbiota" e svolgono numerosi ruoli, tra cui il mantenimento dell'integrità strutturale della barriera mucosa dell'intestino, l'immuno-modulazione, la protezione contro i patogeni e complementano il metabolismo dell'ospite processando alcuni dei nutrienti e dei farmaci assunti [2].

La diffusione e l'importanza dei ruoli ricoperti dalle comunità microbiche negli ecosistemi, rendono particolarmente interessante il loro studio e la comprensione dei meccanismi di interazione dei microrganismi tra loro e con l'ambiente in cui si trovano, con lo scopo di prevederne l'evoluzione nel tempo e nello spazio. Tuttavia, le interazioni che caratterizzano questi meccanismi sono estremamente complesse e, nonostante i continui progressi in questo ambito da parte della comunità scientifica, manca ancora la conoscenza necessaria a definirle in modo assolutamente attendibile.

Per questi motivi, si sta diffondendo lo sviluppo di modellizzazioni e strumenti computazionali con lo scopo di validare teorie e predire il comportamento delle specie di una comunità in risposta a stimoli esterni. Realizzare un simulatore in grado di offrire queste funzionalità porterebbe enormi vantaggi: permetterebbe ad esempio di sostituire, almeno in parte, la sperimentazione diretta, andando così a risparmiare tempo e risorse nello studio delle comunità batteriche; questo porterebbe enormi vantaggi, in particolare quando si tratta di studiare specie potenzialmente nocive o di difficile propagazione in laboratorio.

Esistono diversi modelli ideati con questo scopo. Gli approcci più classici sfruttano reti di co-occorrenza ottenute da dati metagenomici, tralasciando la componente temporale e la composizione dell'ambiente [3]; non sono quindi in grado di fornire informazioni sui meccanismi molecolari che hanno generato gli scenari osservati. Un altro tipo di approccio è quello basato su metodi di *flux balance analysis* (FBA); in questo caso è richiesta la conoscenza approfondita di tutti i processi metabolici delle specie batteriche [4]. Tuttavia, la mancanza di modelli metabolici sufficientemente dettagliati rappresenta ancora un grosso problema; inoltre, analogamente ai modelli basati sulla co-occorrenza, questo approccio deve essere annidato in modelli più complessi per fornire informazioni spaziali. I modelli *Agent Based* rappresentano un tipo di approccio che descrive i sistemi rappresentando le singole entità che li compongono, dette agenti. Gli agenti interagiscono dinamicamente con l'ambiente circostante che può essere arbitrariamente modellato, in base alle regole di comportamento implementate; in seguito, come conseguenza del comportamento degli agenti, si deriva l'evoluzione temporale di una popolazione batterica in un determinato ambiente, senza il bisogno di descrivere le reti di interazione che sono difficili da identificare [4–7].

Il simulatore codificato in Python implementa un modello ABM basato su specie batteriche, nutrienti e ambiente, consentendo la completa personalizzazione dello strumento, grazie alla sua intrinseca modularità. Nello specifico, il modello intende rappresentare spazi discretizzati, che ospitano un certo numero di batteri per ogni specie e una quan-

tità definita di nutrienti che formano l'ambiente circostante. I batteri possono migrare da un'unità spaziale (cella) in un'altra, incontrando diversi metaboliti attraverso l'intero percorso spaziale. La loro crescita e sopravvivenza sono governate dal loro metabolismo, che è a sua volta funzione dei metaboliti presenti in ciascuna unità spaziale all'iterazione in esecuzione. Ogni iterazione del modello comporta il consumo e la produzione di metaboliti da parte dei batteri, calcolati per ogni unità spaziale, in base alla specifica composizione batterica. A sua volta, la crescita batterica, la morte e la possibile migrazione vengono calcolate per ciascuna specie, a seconda dei metaboliti processati localmente.

1.2 Implementazione del simulatore

1.2.1 Definizione del modello

Il modello sviluppato si basa su un ambiente lineare a celle successive, che approssima un tratto di intestino, ognuna delle quali è ospitata da diversi nutrienti con cui gli agenti, ossia le specie batteriche, possono interagire in diversi modi in base al loro metabolismo. Ogni specie batterica è caratterizzata da un nome identificativo, il tasso di crescita massimo ed il metabolismo, che determina la sua capacità di consumare, produrre od ignorare un determinato tipo di metabolita.

Inizializzazione dei parametri e dell'ambiente

Per prima cosa, sono definiti dei parametri che caratterizzano la simulazione: il numero di celle del sistema, il numero del tipo di molecole effettrici, il numero di iterazioni della simulazione (che corrispondono a ore di evoluzione della comunità) e il numero massimo iniziale di nutrienti e batteri. Vengono inoltre impostati i valori di *seed*, utili ad inizializzare il generatore di numeri casuali. In questo modo, utilizzando lo stesso valore di *seed* due volte si ottengono gli stessi risultati casuali, rendendo così la simulazione riproducibile.

La prima fase della simulazione è l'inizializzazione del sistema. Ogni cella viene inizializzata con un insieme casuale di nutrienti, mentre solo la prima cella del sistema viene popolata da un numero casuale di entità per ogni specie batterica.

Fase dinamica

La fase dinamica del simulatore consiste in un processo iterativo, in cui ad ogni ciclo viene calcolato (in una determinata cella), per ogni specie:

1. la quantità di nuovi batteri nati (la metà dei quali si sposterà nella cella successiva) in base al fattore di crescita, calcolato considerando il tasso di crescita massimo e i nutrienti consumati;

2. la quantità di batteri morti, in base al tasso di tossicità dei matabolitи nella cella e al tasso di morte basale;

3. il numero di batteri che si sposteranno nella cella successiva (impostato al 5% della popolazione).

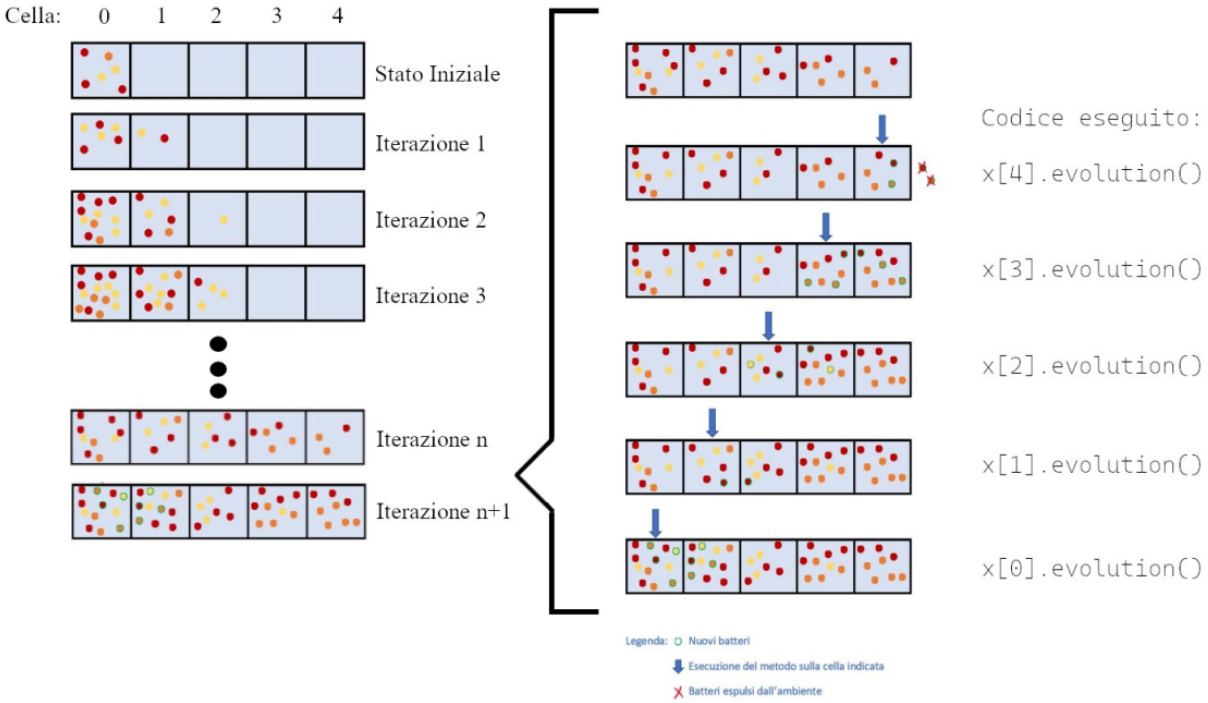


Figura 1.1: Immagine che mette in evidenza l'evoluzione di un sistema all'ultima iterazione

Il risultato è l'evoluzione completa di una popolazione batterica le cui specie, consumando e producendo i nutrienti presenti nelle celle in cui si trovano, si riproducono, muoiono e si spostano nelle celle successive (vedi Figura 1.1).

1.2.2 Classi e funzioni

Verrà ora trattata la traduzione del modello descritto sopra in codice Python. In particolare, verranno spiegate più in dettaglio le classi usate per rappresentare l'ambiente e gli agenti e le funzioni a supporto della fase dinamica del simulatore e della visualizzazione dei risultati.

Classe *Cell*

Gli oggetti della classe *Cell* rappresentano una cella dell'ambiente. Gli attributi e i metodi che caratterizzano gli oggetti della classe sono riportati in Tabella 1.1.

Tabella 1.1: Metodi ed attributi della Classe *Cell*.

attributi	*_food*	vettore dei nutrienti, contenente in ogni posizione la quantità di un determinato nutriente
	_MatrFood	matrice che tiene traccia della quantità dei nutrienti nel tempo; ogni riga rappresenta il vettore *_food* in una data iterazione
	_bact	dizionario che ha come chiavi istanze della classe Bact e come valori la quantità delle specie nella cella
	_x	vettore che contiene tutte le celle
	_pos	intero che indica la posizione nel vettore di celle
metodi	*getFood, getBact, getMatrFood*	metodi di accesso per, rispettivamente: il vettore *_food*, il dizionario *_bact* e la matrice dei nutrienti
	addBact	aggiorna il dizionario *_bact*, sommandogli un dizionario costruito allo stesso modo contenente i nuovi batteri da aggiungere in seguito all'evoluzione del sistema
	food upd	aggiorna il vettore dei nutrienti *_food*
	death	calcola la quantita di batteri morti nell'iterazione corrente
	evolution	gestisce interamente la fase dinamica, calcolando lo spostamento, la morte, la riproduzione e l'interazione con i nutrienti dei batteri presenti nella cella

Classe *Bact*

Gli oggetti della classe *Bact* rappresentano gli agenti, ossia le specie batteriche della popolazione microbica. La classe è caratterizzata da attributi e metodi in Tabella 1.2.

Tabella 1.2: Metodi ed attributi della Classe *Bact*.

attributi	*_species*	attributo di classe; una lista di stringhe che individuano le diverse specie batteriche
	_type	stringa, tra quelle in *_species*, che definisce la specie
	_m	lista di interi che definisce il metabolismo della specie rispetto ogni nutriente presente: -1 se viene consumato, 0 se viene ignorato oppure 1 se viene prodotto
	_t	vettore di 1 e 0, indicanti rispettivamente nutrienti tossici e non tossici per la specie
	_maxGr	valore che determina il tasso di crescita massimo
	_maxTox	valore che determina il tasso di tossicità massimo
	_pos	posizione nel vettore ambiente
metodi	getm, gett, getmaxGr, getpos	metodi di accesso che restituiscono, in ordine: il vettore del metabolismo, il vettore di tossicità, il coefficiente di crescita massimo e la posizione
	getgrowth	calcola il tasso di crescita della specie in una determinata cella, in base al metabolismo della specie rispetto ai nutrienti nella cella e alla loro quantità
	getTox	calcola il coefficiente di tossicità della specie, in base ai nutrienti presenti

Funzioni

Alcune funzioni implementate per l'inizializzazione degli agenti del modello ed a supporto della visualizzazione dei risultati della simulazione:

- *randomFill* restituisce una lista di interi estratti casualmente da un intervallo definito. L'intervallo è definito dai parametri in ingresso, come anche la lunghezza della lista;
- *randomBacts* restituisce una lista di oggetti *Bact* con caratteristiche aleatorie, quali il coefficiente di crescita massimo e il vettore dei metabolismi $\vec{m}$;

- *printState* stampa a schermo lo stato del sistema in ogni iterazione, ovvero la quantità delle diverse specie batteriche in ogni cella;
- *graph* rappresenta, attraverso uno *stackplot*, la distribuzione dei batteri nel tempo;
- *graph_met* rappresenta la distribuzione dei metaboliti nel sistema tramite un grafico diviso in tanti *subplot* quante sono le celle.

1.3 Limiti del simulatore

Il progetto di tesi che ha portato all'implementazione del simulatore si poneva come obiettivo a lungo termine lo sviluppo di uno strumento per l'analisi di comunità batteriche che permettesse di limitare l'utilizzo della sperimentazione diretta. É evidente che uno strumento con queste caratteristiche risulterebbe conveniente a diverse figure professionali nel campo della biologia, della medicina, e delle scienze della vita in generale. Di conseguenza, è essenziale che il simulatore sia facilmente accessibile e utilizzabile dagli utenti, a prescindere dalle competenze in ambito informatico possedute.

É importante inoltre che i risultati ottenuti da una simulazione siano restituiti in un formato adatto per essere consultati in un secondo momento, in modo da permettere l'analisi dei dati e la riproducibilità della sperimentazione.

La struttura del simulatore, nella sua prima versione, non ha le caratteristiche tali da permettere un utilizzo che soddisfi queste esigenze.

Impostare i parametri principali di una simulazione richiede di apportare modifiche al codice stesso, limitando così l'utilizzo del simulatore ad utenti con competenze informatiche e in grado di comprendere il codice scritto in linguaggio Python. Si è giunti alla conclusione che la mancanza di un'interfaccia grafica che consenta all'utente di interagire con il programma in modo più semplice ed intuitivo, rappresentasse un importante limite per il programma.

Il presente lavoro di tesi si pone come obiettivo il superamento di questo limite, attraverso l'implementazione della GUI (*Graphical User Interface*) del simulatore.

Nel raggiungere questo obiettivo, si è lavorato contemporaneamente all'implementazione di miglioramenti al codice originale e alla sua organizzazione in un *package*, con il fine di rendere l'intero programma facilmente distribuibile e condivisibile. Questo ultimo aspetto è stato implementato e presentato nell'elaborato Alessandro Lucchiari [8].

Capitolo 2

Obiettivi

Come anticipato nel precedente capitolo, l'obiettivo di questa tesi è stato quello di progettare e sviluppare un'interfaccia grafica per il simulatore presentato in [1], con lo scopo di renderlo accessibile e fruibile in modo intuitivo da ogni utente.

La Graphical User Interface (GUI) è l'interfaccia grafica che riproduce il codice in back-end e permette all'utente di interagire con il computer. A differenza dell'interfaccia a righe di comando, nella quale i comandi devono essere impartiti tramite istruzioni testuali e quindi richiedono un minimo di conoscenza informatica, l'interfaccia grafica permette un utilizzo più naturale tramite componenti grafici come pulsanti, menu e icone; l'implementazione di un'interfaccia grafica user-friendly per il simulatore di comunità batteriche risulta quindi la soluzione migliore per renderlo facilmente utilizzabile a tutti gli utenti.

La GUI deve comprendere tutte le funzionalità offerte dal simulatore, ma renderne l'esperienza d'uso più semplice per l'utente. In particolare, lo scopo è quello di facilitare l'impostazione dei parametri iniziali di una simulazione, che prima richiedeva di mettere mano direttamente al codice in linguaggio Python, nonché un'efficace interfacciamento tra codice sottostante la simulazione e GUI stessa. In questo modo, ogni utente è in grado di impostare ed eseguire una simulazione di comunità batteriche in pochi e semplici passi (per una guida all'utilizzo della GUI si veda il Capitolo 5).

Capitolo 3

Strumenti per lo sviluppo

In questo capitolo vengono presentati gli strumenti che hanno permesso lo sviluppo dell'interfaccia grafica e la sua distribuzione.

Inizialmente verranno introdotti i mezzi che hanno favorito la collaborazione al progetto e permesso la sua distribuzione, ossia Git e GitLab.

Successivamente verranno presentati gli strumenti utili alla realizzazione dell'interfaccia grafica, per cui è stato necessario prima di tutto acquisire le conoscenze alla base dell'elaborazione e la visualizzazione dei dati. Queste due funzionalità sono offerte, rispettivamente, dalle librerie Python *Pandas* e *Plotly*.

3.1 Git

Git è un sistema che consente il controllo delle versioni e la collaborazione di più utenti ad un unico progetto contemporaneamente. Un sistema di controllo di versione (VSC), registra i cambiamenti effettuati nel tempo ad uno o più file e permette di ripristinarli ad una versione precedente, revisionare le modifiche fatte nel tempo, vedere gli autori delle modifiche e molto altro. Ogni volta che si effettuano delle modifiche ad un file si assegna un nome e una descrizione a questa nuova versione, questa operazione si chiama `commit`.

GitLab e GitHub sono piattaforme web basate su Git per la gestione di repository, dove è possibile caricare progetti per favorire la collaborazione di un team di sviluppatori. Nello sviluppo del *package* "Bactlife" e della GUI è stato scelto GitLab.

Se si vuole collaborare ad un progetto esistente su GitLab, per prima cosa ci si deve procurare una copia della repository in cui è contenuto. Digitando sulla *command window* il comando `git clone [url]` (dove url è l'indirizzo web della repository), si crea una directory sul proprio computer, che contiene un'istantanea dell'ultima versione dei file del progetto. É possibile aggiungere nuovi file alla directory di lavoro con `git add <file>`.

Ogni volta che si effettuano delle modifiche ad un file questo assume lo stato *modified* (gli stati sono visibili tramite il comando `git status`), si può assegnare un nome e una descrizione a questa nuova versione, eseguendo il commit:

```
git commit -m ‘‘commit message’’
```

Per visualizzare la cronologia dei commit si utilizza il comando `git log`.

Eseguire il `commit` dei file non apporta modifiche nel repository originale in remoto ma solo nella directory locale, mentre quando si desidera caricare online la propria versione (fare il *push*) si esegue il comando:

```
git push origin master
```
(da origin a master)

Una funzionalità offerta da Git particolarmente utile per la collaborazione di più utenti allo stesso progetto è la possibilità di creare *branch* (rami), ossia diverse versioni dello steso progetto sviluppate in maniera indipendente l'una dall'altra. Il ramo principale di un progetto è detto `master` e ci si può spostare da un ramo all'altro usando `git checkout <new-branch>`. Una volta apportate le modifiche desiderate, si possono unificare due versioni in un unico salvataggio con `git merge <branch>`. Ad esempio, se ci si trova sul ramo `master`, il comando `git merge test` unisce il ramo "test" al `master` se le due versioni sono compatibili, altrimenti evidenzia le righe nei documenti presenti su entrambi i branch che generano conflitto poiché versioni alternative provenienti da una stessa versione d'origine.

3.2 Pandas

Pandas è una libreria Python usata per l'analisi, l'esplorazione e la manipolazione di dati. Fornisce strutture dati che permettono un'interazione intuitiva con i dati in formato tabellare o sequenziale. Di seguito ne riporto le principali caratteristiche e funzionalità, soffermandomi sugli aspetti che più sono stati utili per lo scopo di questo elaborato.

Installazione ed import

Se Python e PIP (un tool che permette di cercare, scaricare ed installare package Python che si trovano sul *Python Package Index*) sono già installati nel sistema, è sufficiente eseguire il seguente comando dalla command window:

```
py -m pip install pandas
```
(per gli utenti Windows)

```
python3 -m pip install pandas
```
(per gli utenti Unix/MacOS)

In fase di import viene spesso utilizzato un alias, per il pacchetto Pandas si è scelto di utilizzare l'alias "pd":

```
Import pandas as pd
```

Strutture dati

Le due principali strutture dati in Pandas sono `Series` e `DataFrame`.

Una `Series` è un array monodimensionale che può contenere dati di ogni tipo (interi, float, stringhe, oggetti, etc.). Gli elementi sono etichettati da un indice e sono accessibili sia come in una lista (l'indice indica la posizione della lista) che come in un dizionario (l'indice è utilizzato come chiave primaria del dizionario). Una `Series` può essere creata tramite la funzione `pd.Series()` passando una lista, eventualmente specificando gli indici, oppure un dizionario, in questo caso le chiavi diventano gli indici.

```
>>> s = pd.Series({'a':1,'b':34,'c':7.8,'d':0})
>>> s
a     1.0
b    34.0
c     7.8
d     0.0
dtype: float64
>>> s[2]
7.8
>>> s['a']
1.0
```

Listing 3.1: Creazione di una Series ed estrazione degli elementi

Un `DataFrame` è una struttura dati bidimensionale. Le colonne sono `Series` e possono contenere dati di tipo diverso tra loro. Analogamente alle `Series`, un `DataFrame` è creato tramite la funzione `pd.DataFrame()` che accetta diversi tipi di input, tra cui dizionari di `Series` e dizionari di liste. Un `DataFrame` ha indici sia per le righe (index) che per le colonne (columns). Si può accedere agli elementi in moltissimi modi, a seconda delle esigenze; in Tabella 3.1 vengono elencati i principali.

Tabella 3.1: Accesso agli elementi di un DataFrame.

selezione	sintassi	risultato
colonna	`df[col]`	Series
righe tramite indice	`df.loc[label]`	Series
righe tramite posizione	`dfiloc.[label]`	Series
subset tramite vettore booleano	`df[bool\_vec]`	DataFrame

Funzionalità

Si possono effettuare diversi tipi di operazioni sulle strutture dati appena introdotte:

statistica: si possono effettuare facilmente e velocemente analisi statistiche su set di dati di grandi dimensioni ad esempio trovare la media `s.mean()` e la deviazione standard `s.std()` di un vettore di dati.

test logici: servono per filtrare un dataset estraendo solo gli elementi che soddisfano una determinata condizione. Ad esempio, il comando `s < 10` applicato alla serie `s` produce una maschera, ossia una Series con valori `True` dove la condizione è verificata e `False` altrove. Le maschere servono a filtrare la serie originale:

```
>>> s < 10
a     True
b    False
c     True
d     True
dtype: bool
>>> s[s < 10]
a    1.0
c    7.8
d    0.0
dtype: float64
```

Listing 3.2: Esempio di operazione su Series.

import and storage: Pandas permette di importare dati da file di molti formati diversi (excel, json, csv, etc.) grazie alle funzioni `read_*`. Ad esempio, se si dispone di dati in formato json, si può facilmente creare un `DataFrame` tramite il comando `df = pd.read_json()`. Analogamente, tramite i metodi `to_*` è possibile memorizzare dati in vari formati [9]. Nella GUI presentata in questo elaborato, cliccando un pulsante è possibile esportare un `DataFrame` sotto forma di file excel, con più *sheets*, rappresentante parametri e risultati della simulazione effettuata.

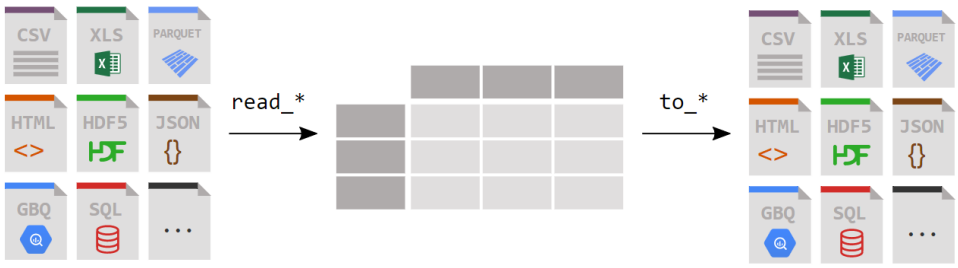


Figura 3.1: Lettura e scrittura di *dataframes* in diversi formati. Immagine adattata da [9]

3.3 Plotly

Plotly è una libreria Python che permette di di creare, manipolare e visualizzare molti tipi di grafici, che coprono un'ampia gamma di casi d'uso statistici, finanziari, geografici,

scientifici e tridimensionali [10]. In particolare, consente di creare visualizzazioni interattive ed animate integrabili in applicazioni web scritte interamente in Python, usando Dash (vedi Capitolo 4. Implementazione).
Si può installare `Plotly` usando il comando `pip` dalla command window:

```
py -m pip install plotly                    (per gli utenti Windows)

python3 -m pip install plotly               (per gli utenti Unix/MacOS)
```

Strutture dati

Le figure sono istanziazioni della classe `Figure`, del modulo `plotly.graph_oblects`, oggetti con una struttura ad albero i cui nodi sono chiamati attributi. I due attributi "padri" sono `data` e `layout` Figura 3.2.

- `data` è una lista di tracce (dizionari) che contengono l'informazione sui dati ed il tipo di grafico. Ogni traccia deve contenere l'attributo `type` che ne descrive il tipo, ad esempio "scatter" (per grafici di dispersione, a linee, ad area) o "bar" (per grafici a barre).

- `layout` è un dizionario contenente attributi che controllano la configurazione di parti della figura non relative ai dati, come dimensioni, titolo, leggenda, assi, etc.

Gli attributi possono essere modificati anche successivamente all'istanziazione dell'oggetto `Figure` tramite metodi come `.update_layout()` e `.add_trace()`.

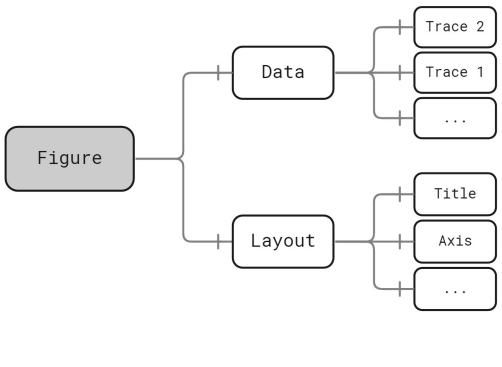


Figura 3.2: Struttura di un oggeto di tipo *Figure*

Creare ed aggiornare figure

Il procedimento più facile e veloce per la creazione delle figure più comuni è usare il modulo `plotly.express` (solitamente importato con l'alias "px"), contenente funzioni che creano intere figure in una sola chiamata. Ogni funzione utilizza internamente il modulo

`graph_objects` e la figura che restituisce è un'istanziazione della classe `Figure`, dunque può essere poi modificata usando gli stessi metodi.
Le funzioni per la realizzazione di grafici fornite da Plotly Express permettono di visualizzare diversi tipi di dati forniti in input, tra cui `DataFrame` Pandas. Per estrarre un grafico a partire da dati contenuti in un `DataFrame` è sufficiente fornirlo in input alla funzione scelta, insieme ad alcuni parametri che caratterizzano il *plot*. L'esempio di seguito mostra la creazione di un semplice grafico a partire da un `DataFrame`.
La funzione necessaria per creare un grafico a barre raggruppate è `px.bar`. Gli input sono:

- *df*: un DataFrame contenente i dati da plottare
- *x = "Medals"*: serve a settare l'asse x. In questo caso con dati (unici) contenuti nella colonna *Medals*
- *y = "Count"*: serve a settare l'asse x. In questo caso con dati (unici) contenuti nella colonna *Count*
- *color = "Sex"*: con questo argomento si specifica che il grafico sarà a barre raggruppate, utile a confrontare i dati di diversi gruppi di una stessa categoria. Ad esempio, nel caso mostrato si vogliono confrontare le quantità di medaglie d'oro, argento e bronzo ottenute dagli uomini con quelle ottenute dalle donne. Attribuendo "Sex" all'attributo *color* si indica alla funzione "bar" di raggruppare le barre in base ai valori (presi unicamente) della colonna, quindi confrontando i valori contenuti nelle righe corrispondenti.

```
>>> import pandas as pd
>>> import plotly.express as px
>>> df=pd.DataFrame({'Medals':['Gold','Silver','Bronze','Gold','Silver',
    'Bronze'],
...                  'Sex':['Male','Male','Male','Female','Female','
    Female'],
...                  'Count':[2,5,8,3,7,5]})
>>> df
   Medals     Sex  Count
0    Gold    Male      2
1  Silver    Male      5
2  Bronze    Male      8
3    Gold  Female      3
4  Silver  Female      7
5  Bronze  Female      5
>>> fig=px.bar(df,x='Medals',y='Count',color='Sex')
>>> fig.show()
```

Listing 3.3: Esempio di codice per generare un grafico a barre.

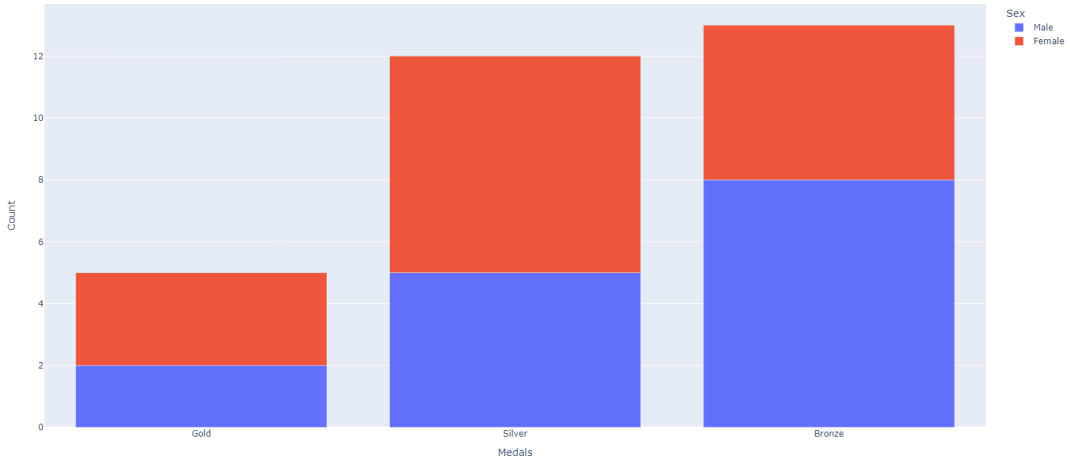


Figura 3.3: Grafico a barre ottenuto da un Dataframe

Le funzioni di Plotly Express supportano moltissimi altri argomenti per la personalizzazione delle figure, qui di seguito verranno citati quelli più significativi per l'implementazione della GUI presentata in questo elaborato; per ulteriori possibili implementazioni è possibile consultare la documentazione in [10].

- *facet_row*: serve nella creazione di più subplot che hanno lo stesso set di assi, dove ogni subplot rappresenta un subset di dati.
- *animation_frame*: serve per creare figure animate in base al subset di dati scelto (solitamente una serie di dati temporali come anni, date, etc.)

Plolty e Dash

La libreria Plotly è perfettamente compatibile con dash, il framework scelto per l'implementazione dell'interfaccia grafica del simulatore. Ogni figura realizzata con Plotly può essere visualizzata in un'app Dash, passando l'oggetto *Figure* che restituisce una funzione Plotly Express come argomento `figure` del componente grafico `dcc.Graph` di Dash (si veda Capitolo 4 per la spiegazione dei componenti grafici).

Capitolo 4

Implementazione

Dash è una libreria opensource Python ideata appositamente per lo sviluppo di applicazioni web e particolarmente adatta per app d'analisi e visualizzazione di dati. Sebbene le app Dash vengano visualizzate tramite browser Web, non è necessario scrivere codice JavaScript mentre è sufficiente conoscere solo i tag base di HTML [11].

Setup

Installando `dash` dal terminale, usando il comando `pip install dash`, verrà installata automaticamente anche la libreria `Plolty`. Altre installazioni necessarie sono:

- `pandas`, richiesta da `plotly.express` che viene utilizzata per la realizzazione delle figure nel seguito
- `dash-bootstrap-components` una libreria per la personalizzare del layout della GUI.

In questo capitolo viene illustrata la struttura generale di un'app Dash e, successivamente, verranno elencate le tappe intraprese per la creazione della GUI del pacchetto "Bactlife".

4.1 Struttura

Innanzitutto si devono importare tutte le librerie ed i moduli necessari, successivamente si inizializza l'app creando un'istanza `Dash` e assegnandole una variabile globale [12], nel seguente modo:

```
app = dash.Dash(__name__)
```

per conferire all'app un tema (stile) specifico, si può usare l'argomento aggiuntivo `external_stylesheets` ed impostare un tema tra quelli disponibili in `dash-bootstrap-components` [12], ad esempio:

```
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.LUX])
```

Dopodiché si passa alla struttura principale, che è composta essenzialmente da due elementi costitutivi, che verranno approfonditi nei prossimi sottocapitoli dedicati: il *Layout* che ne descrive l'aspetto e le *Callback Functions* che ne determinano l'interattività [12].

Infine, le seguenti righe di codice:

```
if __name__ == "__main__":
    app.run_server(debug=True)
```

rendono possibile eseguire l'app nel server locale. L'argomento `debug=True` aggiorna automaticamente il browser Web quando si apportano modifiche al codice, quindi non è necessario aggiornare il browser ogni volta[12].

4.1.1 Layout

Il layout di un'app Dash ne determina l'aspetto visivo. É un albero gerarchico di componenti , per cui ogni opzione configurabile è disponibile come attributo (`children`, `style`, `id`, etc.) [11]. Con componenti si intendono sia le parti testuali sia gli elementi grafici come figure, tabelle, etc. I moduli di Dash che permettono la creazione di componenti del layout sono:

- *Dash HTML Components* (`dash.html`): contiene una classe per ogni tag e argomenti per tutti gli attributi HTML. Serve a creare e definire lo stile di contenuti come intestazioni e paragrafi.
- *Dash Core Components* (`dash.dcc`): modulo che serve a generare componenti interattivi come grafici, menù e sliders.
- *Dash Bootstrap Components* (importata come `dbc`): libreria che fornisce componenti grafici e temi (*stylesheets*) che permettono di migliorare l'aspetto generale dell'app, standardizzando il formato di intestazioni, pulsanti, menù, etc.

4.1.2 Callback functions

Le *Callback Function* rendono interattiva l'interfaccia realizzata con Dash. Sono funzioni che permettono l'aggiornamento dinamico delle proprietà dei componenti definiti precedentemente nel layout.

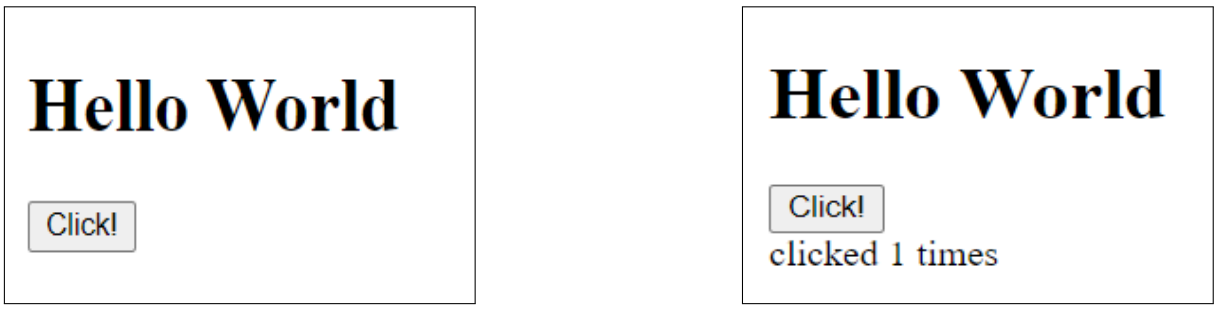
Gli Input e Output delle funzioni sono gli argomenti del decoratore `@app.callback`. Quest'ultimo serve a comunicare a Dash di richiamare la funzione (definita subito dopo) ogni volta che il componente in Input cambia, e conseguentemente aggiornare il componente in Output. Quando ci si riferisce ai componenti del layout in una *Callback Functions* si utilizza l'`id`, attributo identificatore di un componente.

Facendo riferimento all'esempio illustrato di seguito, nel layout si sono definiti: un titolo, un pulsante e un contenitore di testo vuoto. La funzione `update_output` viene chiamata ogni volta che la proprietà `n_clicks` (il numero di click) del componente con

l'`id` "my-input" (il pulsante) viene aggiornata, modificando la proprietà `children` del componente che ha l'`id` "my-output" in base a ciò che viene restituito dalla funzione Figura 4.1.

```python
from dash import Dash, dcc, html, Input, Output
import dash_bootstrap_components as dbc

app = Dash(__name__) #inizializzazione

app.layout = html.Div([
    html.H1("Hello World"), #intestazione
    html.Div([
        dbc.Button("Click!",id="my-input") #pulsante
    ]),
    html.Div(id="my-output")]) #contenitore (inizialmente vuoto)

@app.callback(
    Output(component_id="my-output", component_property="children"),
    Input(component_id="my-input", component_property="n_clicks"))
def update_output(n_clicks): #quando si clicca il pulsante, viene restituito il conteggio di click aggiornato
    if n_clicks:
        return "clicked ",str(n_clicks)," times"

if __name__ == "__main__":
    app.run_server(debug=True)
```

Listing 4.1: Esempio di utilizzo di callback



(a) prima di cliccare il pulsante

(b) dopo aver cliccato il pulsante

Figura 4.1: Esempio di interazione

Le *Callback Function* possono avere molteplici Output e Input. Spesso in questi casi risulta utile sapere quale dei componenti ha attivato la funzione utilizzando `dash.callback_context` all'interno della funzione, che fornisce informazioni sullo stato dei componenti in input.

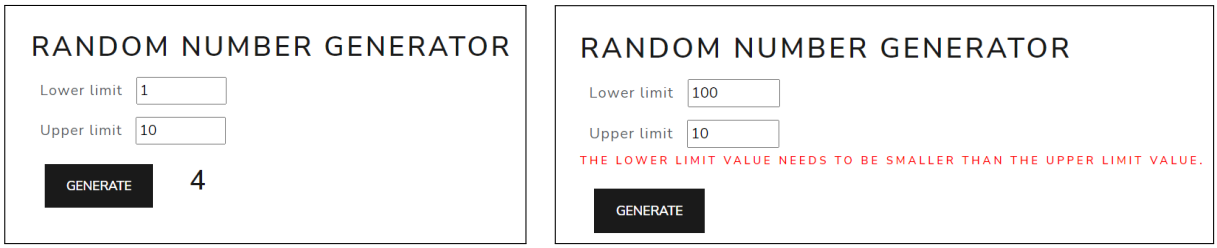
In alcuni casi risulta più funzionale fare in modo che le *Callback Functions* leggano gli stati di più componenti contemporaneamente, ma non si aggiornino fino a chè non viene dato un comando particolare, si pensi ad esempio a modelli simili alla compilazione

di moduli. Esiste per questo un terzo argomento oltre a `Input e Output`, ossia `State`. Funziona allo stesso modo: ha bisogno dell'`id` e della proprietà del componente a cui si riferisce. `State` fornisce informazioni senza attivare le funzioni, che verranno attivate solamente quando una delle proprietà dei componenti dentro `Input` viene aggiornata.

Di seguito è mostrato un esempio del suo utilizzo: un generatore di numeri casuali che richiede in input i limiti dell'intervallo per l'estrazione, che vengono utilizzati dalla funzione per restituire un numero solo dopo aver cliccato il pulsante *generate*. Inoltre, visualizza un messaggio di errore se il valore inserito come limite inferiore è maggiore di quello per il limite superiore (si veda la Figura 4.2). É riportata solamente la *callback function* che implementa l'interazione, che fa riferimento agli elementi di layout facilmente intuibili dagli identificativi.

```
@app.callback(
    Output("error","children"),
    Output("output","children"),
    Input("generate-button","n_clicks"),
    State("lower-limit","value"),
    State("upper-limit","value"))
def generate_random_number(n_clicks,lower_limit,upper_limit):

    #se il limite inferiore è maggiore del superiore, visualizza
    messaggio d'errore
    if lower_limit > upper_limit:
        return [html.H6(["The lower limit value needs to be smaller than
    the upper limit value."],
                        style={"color":"red"}),None]

    #altrimenti, genera un numero casuale nell'intervallo e lo
    visualizza
    else:
        random_number=randint(lower_limit,upper_limit)
        return [None,html.H1(random_number)]
```

Listing 4.2: Esempio di utilizzo di *State* come argomento



(a) numero casuale tra 1 e 10 **(b)** appare il messaggio d'errore perché 100 è maggiore di 10

Figura 4.2: Utilizzo di State per un generatore di numeri casuali

L'ordine degli argomenti all'interno del decoratore `app.callback` è:

1. `Output`;

2. `Input`;

3. `State` (eventualmente).

4.2 Sviluppo della GUI

Dopo aver presentato la struttura e l'implementazione generali di un'interfaccia grafica in Dash, verrà trattata ora l'implementazione della GUI per il pacchetto "Bactlife".

4.2.1 Progettazione

Prima della realizzazione computazionale di una GUI è necessario considerare chi ne farà uso e con quali scopi, quindi comprendere le esigenze dell'utente e le funzionalità desiderate. La progettazione consiste nell'individuare il modo più adeguato per rappresentare queste funzionalità, dunque specificare le modalità di input, l'interazione con i vari componenti e la visualizzazione dei risultati dell'elaborazione.

Pertanto, il primo passo della realizzazione della GUI per il pacchetto "Bactlife" è stato analizzare gli scopi del suo utilizzo. Di seguito verrà presentata l'analisi delle esigenze dell'utente e delle funzionalità richieste per la GUI.

Il simulatore di comunità batteriche, lanciato da riga di comando, esegue una simulazione e mostra a schermo i risultati ottenuti con specifici parametri, ossia un certo numero di celle, di nutrienti, di interazioni, etc. I parametri della simulazione sono modificabili solamente accedendo al codice in linguaggio Python, di conseguenza sarebbero necessarie competenze informatiche e una conoscenza approfondita del codice stesso per poter personalizzare una simulazione. Dal momento che il primario obiettivo della creazione della GUI era quello di rendere l'utilizzo del simulatore possibile anche ad utenti inesperti, la possibilità di modificare i parametri di una simulazione in modo semplice ed intuitivo è stata ritenuta un'importante e necessaria funzionalità da includere. Si è scelto di dare la possibilità all'utente di modificare i seguenti parametri:

- Numero di celle;
- Numero di nutrienti;
- Massimo numero di batteri iniziali (per specie);
- Massimo numero di nutrienti iniziali (per tipo);
- Numero di iterazioni;
- Valori di seed;
- Interazione di ogni specie con i nutrienti.

Per quanto riguarda la rappresentazione dei risultati di una simulazione, si è deciso di mantenere il formato originale, ossia la visualizzazione della distribuzione di batteri e nutrienti nel sistema attraverso dei grafici. Inoltre, anche grazie a dei miglioramenti apportati al codice del simulatore (spiegati in dettaglio in [8]), è stato possibile aggiungere delle ulteriori funzionalità come scegliere di rappresentare i dati in versione normalizzata o visualizzare solo una determinata porzione del sistema.

Un'altra funzionalità che si è ritenuto importante implementare è stata la possibilità di esportare i risultati di una simulazione, la cui mancanza rappresentava un limite del simulatore originale. Per rendere questa funzione intuitiva, si è deciso di utilizzare un pulsante, che quando viene cliccato scarica i dati della simulazione sotto forma di file excel, organizzato in diversi fogli.

4.2.2 Implementazione

Avendo ora chiare le funzionalità richieste, si può pensare all'implementazione vera e propria dell'interfaccia. Di seguito verranno presentate le scelte implementative messe in atto per soddisfare gli obiettivi posti.

L'aspetto visivo dell'interfaccia grafica deve essere progettato in modo da venire incontro all'utente, con lo scopo di garantire un'esperienza di utilizzo il più possibile naturale e intuitiva. La progettazione del layout della GUI è stata dunque pensata con questi obiettivi. Ciò significa che la scelta dei componenti visivi, la loro dimensione e la loro posizione nella pagina non è stata casuale, ma frutto di considerazioni sulle esigenze dell'utente.

L'aspetto dell'interfaccia grafica proposta è raffigurato in Figura 4.3.

Parametri di Simulazione

La metà superiore dell'interfaccia è dedicata al settaggio dei parametri della simulazione. Da sinistra a destra:

- *Seeds.* In questa sezione è possibile:

  1. modificare i valori di *seed* tramite caselle di input o generare dei valori random con il pulsante *Random*. Nello specifico, `Seed 1` è relativo alla libreria `random` e `Seed 2` è relativo alla libreria `numpy`. Il primo valore influenza l'inizializzazione casuale dei vettori $\vec{m}$ (metabolismo) e $\vec{t}$ (tossicità) delle specie batteriche, la quantità di entità per specie nella cella iniziale e il numero di nutrienti nell'ambiente, mentre il secondo incide nella definizione del tasso di crescita massimo di ogni specie batterica;
  2. scegliere una simulazione (e quindi dei parametri preimpostati) tra alcune più significative da un menù a tendina;
  3. eseguire la simulazione ed esportare i risultati ottenuti cliccando i rispettivi pulsanti *Run* e *Download*.

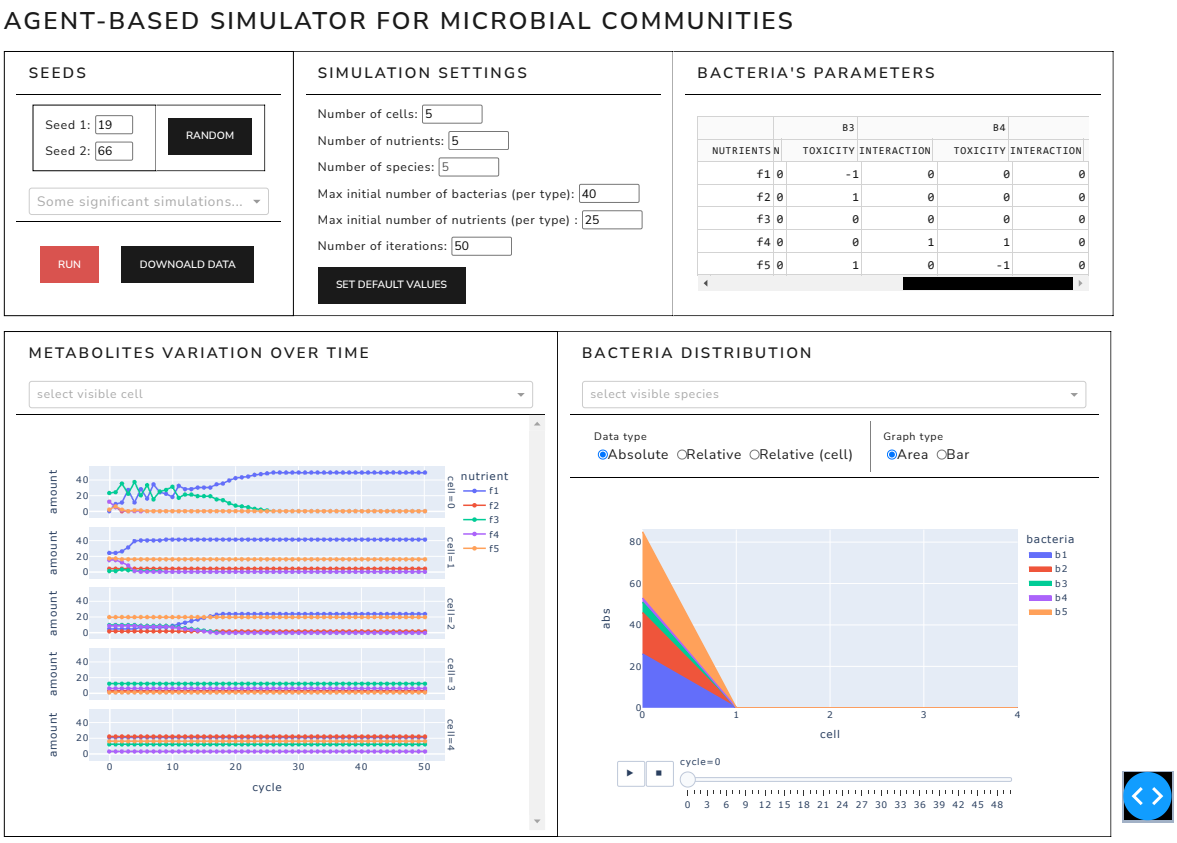


Figura 4.3: Aspetto visivo dell'interfacca grafica sviluppata

```
1     dcc.Dropdown(id='simulation', #identificatore
2                  options=['commensalism', #opzioni del menù a
    tendina
3                           'hostile environment',
4                           'favorable environment'],
5                  placeholder='Some significant simulations...', #
    testo segnaposto
6                  style={'fontSize':18}) #modifica lo stile
7
```

Listing 4.3: Esempio: implementazione di un menù a tendina

```
1     dbc.Button('Downoald data',          #label
2                id='btn_download_data',   #identificatore
3                style={'margin':'15px'}) #stile
4
```

Listing 4.4: Esempio: implementazione di un pulsante

- *Simulation settings.* In questa sezione si possono modificare il numero di celle, il numero di nutrienti, il numero di specie (interazione disabilitata al momento, si veda Capitolo 6 dove si parla dei limiti di "Bactlife"), il massimo numero di batteri e nutrienti iniziali (per tipo) e di cicli della simulazione. L'interazione è permessa

da caselle di input, realizzate con il componente `dcc.Input`, nelle quali l'utente può usare le frecce per incrementare o diminuire il valore oppure inserirlo tramite la tastiera.

```
    dcc.Input(id='input_num_nut', #identificatore
              type='number', #vincola il tipo di input accettato
              min=1, #vincola il minimo valore accettato
              value=5, #valore di default
              style={'width':'80px',  #modifica lo stile
                     'height':'25px',
                     'margin':'5px'})

```

Listing 4.5: Esempio: implementazione di una casella di input

- *Bacteria's Parameters.* In questa sezione è presente una tabella, realizzata con il componente `dash_table`, in cui è rappresentata l'interazione di ogni specie batterica con tutti i nutrienti del sistema. Per ogni specie la colonna *Interaction* racchiude le informazioni sul metabolismo, e la colonna *Toxicity* mostra quali nutrienti sono tossici per quella data specie. Ogni cella della tabella è modificabile, come se fossero delle caselle di input.

Visualizzazione Risultati

La metà inferiore dell'interfaccia è invece dedicata alla visualizzazione dei risultati della simulazione.

- *Metabolites Variation Over Time.* Il grafico a sinistra rappresenta la variazione dei metaboliti nel tempo ed è suddiviso in tanti subplots quante sono le celle del sistema. Un menù a tendina permette di scegliere di selezionare solo un sottoinsieme del sistema (celle).

- *Bacteria Distribution.* In questa sezione è presente un grafico che rappresenta i dati riguardanti la distribuzione del batteri nel tempo. Tramite un menù a tendina, è possibile selezionare un sottoinsieme di specie di cui visualizzare i risultati. Inoltre, tramite dei componenti `dcc.RadioItems` (un insieme di pulsanti), si può interagire con il grafico nei seguenti modi:

  1. scegliendo il tipo di dati visualizzati (frequenze assolute, normalizzate rispetto il sistema o rispetto la cella);
  2. selezionando il tipo di grafico che li rappresenta (uno *stackplot* o un grafico a barre);
  3. osservando lo sviluppo nel tempo tramite uno *slider*.

```
dcc.RadioItems(id='amt_type',  #identificatore
               options=[{'label':'Absolute','value':'abs'}, #
    opzioni
                        {'label':'Relative','value':'rel_all'},
                        {'label':'Relative (cell)','value':'
    rel_cell'}],
               value='abs',  #valore di default
               inline=True,  #per allineare in orizzontale
               labelStyle={'marginRight':'5px',  #stile
                           'marginLeft':'5px',
                           'fontSize':18})
```

Listing 4.6: Esempio: implementazione di *RadioItems*

Callback Functions

Le funzioni interattive dell'interfaccia appena descritte sono state implementate tramite le seguenti *Callback Function*:

- *random_simulation* annulla la selezione nel menù delle simulazioni quando vengono randomizzati i valori *seed*

- *set_parameters* permette il collegamento tra più componenti. In prima chiamata, crea la tabella dei parametri usando la funzione `parameters_table`, usando i dati degli oggetti *Bacts* ricevuti in Input, elaborati in un *dataframe*. I cambiamenti apportati sono diversi a seconda di quale componente ha attivato la funzione. La tabella viene aggiornata ogni volta che vengono modificati i valori *seed* o il numero di nutrienti e specie. Inoltre, modifica i valori di seed quando viene cliccato il pulsante *Random* o scelta una simulazione specifica.

- *set_default_values* reimposta i parametri della sezione *Simulation Settings* nei valori di *default*

- *main* è la funzione dentro la quale vengono creati gli oggetti ed elaborati i dati della simulazione. I risultati sono organizzati in *dataframe* che vengono salvati (tramite il componente dash `dcc.Store`) per poter essere poi accessibili ad altre Callback Function. Si usa l'argomento `State` del decoratore per registrare tutti i parametri della simulazione, che viene eseguita solamente quando si interagisce con i componenti in Input, ovvero i pulsanti *Run* e *Download* (se si vuole anche scaricare il file excel con i risultati). Quando viene eseguita la simulazione, la funzione `main` chiama le funzioni `df_bacs` e `df_nut` per creare *dataframe* che contengono, rispettivamente: (1) informazioni sulla quantità e il tasso di crescita dei metaboliti in ogni cella e per ogni iterazione e (2) informazioni sulla quantità di nutrienti in ogni cella nel tempo. Questi *dataframe* vengono memorizzati in formato json in un componente

`dcc.Store`, in questo modo possono essere usati come input in altre *Callback Function* senza ripetere i calcoli. Infine, se ad innescare la funzione è stato il pulsante *Download*, viene creato un file excel che racchiude, in due *sheet* (fogli) separati, tutti i parametri della simulazione e i risultati ottenuti. Il file excel viene caricato in un componente `dcc.Download`, che permette di scaricare i file contenuti nella sua proprietà *data*.

```
>>> df_bacs(bd,n_it,x)
    cycle  abs_amt  rel_cell  rel_all  growth_rate bacteria  cell
0       0       23     0.173    0.173       0.0000       b1     0
1       0        0     0.000    0.000       0.0000       b1     1
2       0        0     0.000    0.000       0.0000       b1     2
3       1       21     0.149    0.128       0.0250       b1     0
4       1        1     0.043    0.006       0.0000       b1     1
..    ...      ...       ...      ...          ...      ...   ...
28      9      171     0.643    0.265       0.0000       b5     1
29      9       98     0.790    0.152       0.0000       b5     2
30     10      146     0.535    0.211       0.3766       b5     0
31     10      188     0.644    0.272       0.0000       b5     1
32     10       99     0.780    0.143       0.0000       b5     2

[165 rows x 7 columns]

>>> df_nut(x)
    cycle  amount nutrient  cell
0       0       5       f1     0
1       1      13       f1     0
2       2      19       f1     0
3       3      18       f1     0
4       4      18       f1     0
..    ...     ...      ...   ...
6       6       0       f5     2
7       7       0       f5     2
8       8       0       f5     2
9       9       0       f5     2
10     10       0       f5     2

[165 rows x 4 columns]
```

Listing 4.7: Struttura dei due *dataframe*. df_bacs riceve in input: un dizionario che registra la quantità di batteri di ogni specie (chiavi) in ogni cella e iterazione, il numero di iterazioni e il vettore di celle. df_nut ha come input solamente il vettore di celle

- *options_update* aggiorna le opzioni dei menù a tendina dei grafici, quando vengono modificate le quantità di celle e nutrienti tramite input.

- *graph_nut* restituisce il grafico della distribuzione dei nutrienti. Riceve in input (1) i dati contenuti nel `dcc.Store`, forniti in output dalla funzione `main`, e (2) le celle selezionate nel menù. Il *dataframe* dei nutrienti, opportunamente filtrato in base alle celle selezionate, è fornito in input alla funzione `px.line`. L'oggetto *Figure* così creato viene restituito in output nel componente `dcc.Graph`, che renderizza il grafico nell'interfaccia.

– *graph_bact* restituisce il grafico rappresentante la distribuzione dei batteri nel tempo. Il meccanismo è analogo a quello di `graph_nut`, in più il filtraggio del *dataframe* tiene conto anche del tipo di dati selezionato (assoluti o normalizzati) e la funzione per creare la figura è scelta tra `px.bar` e `px.area` a seconda del tipo di grafico selezionato.

Capitolo 5

Come utilizzare la GUI

La GUI è disponibile nel *repository* GitLab insieme al pacchetto e altri file accessori. Questi ultimi hanno lo scopo di illustrare le funzionalità e i meccanismi del simulatore e guidare l'utente all'installazione del pacchetto e al suo utilizzo con e senza GUI.

Questo capitolo si propone come guida all'utilizzo dell'interfaccia grafica, illustrando le tappe necessarie per l'esecuzione di una simulazione.

Installazione e lancio della GUI

Le istruzioni per il download e l'installazione dell'intero pacchetto, che comprende la GUI, sono contenute nel file `Readme.md` su GitLab, sotto il paragrafo *"How to install Bactlife"* (si veda la figura Figura 5.1 per il procedimento da seguire).

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. Warning: if you have both Python2 and Python3 installed on your system, use the pip3 and python3 commands instead of the pip and python mentioned below'.
4. Install all dependencies listed in `requirements.txt`.

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

5. Install the package **bactlife**

```
pip install -e .
```

Figura 5.1: Estratti del Readme del *repository* GitLab

Dopo di che si può procedere con l'esecuzione del file `app.py`, questa operazione lancia un server web locale che ospita l'interfaccia grafica. Apparirà a schermo il suo indirizzo web e per visualizzarla è sufficiente aprirlo in un browser.

L'aspetto dei componenti dell'interfaccia e la loro funzione sono stati spiegati in modo approfondito in sezione 4.2.

Simulazione

Il primo passo per lanciare una simulazione è determinare i parametri che la caratterizzano, tramite le diverse componenti presenti nella metà superiore della schermata. Nonostante non ci siano dei vincoli sull'ordine scelto per modificare i parametri, è indicato procedere da sinistra verso destra, quindi decidere per primi i valori di *seed* (o alternativamente una simulazione preimpostata), poi i parametri ambientali come il numero di celle, le iterazioni, etc. e per ultimi i parametri delle specie batteriche. Questo perché alcuni parametri sono collegati tra loro da un rapporto causa-effetto tramite *Callback Functions*, ad esempio: modificando i valori di *seed*, si genera da capo la tabella dei parametri dei batteri , quindi si perderebbero tutti i dati inseriti se si invertisse l'ordine di inserimento.

Una volta determinati tutti i parametri necessari, per vedere i risultati della simulazione la si deve avviare cliccando il pulsante *Run*.

Risultati e Interazione

I risultati della simulazione sono visibili nei due grafici in basso alla schermata, rappresentanti le distribuzioni nel tempo di nutrienti e metaboliti. Mentre il primo è statico e diviso in tanti *subplot* quante sono le celle del sistema, il secondo è un grafico animato in cui ogni *frame* rappresenta la situazione del sistema in una specifica iterazione. Per visualizzare la completa evoluzione nel tempo della distribuzione dei batteri basta cliccare il tasto *play* (▸) oppure cliccare e scorrere con il mouse sullo *slider*.

Grazie alle funzionalità interattive implementate, si può cambiare la modalità di visualizzazione dei risultati in base alle proprie esigenze e in modo facile e intuitivo. Ad esempio, scegliendo le opzioni *Relative* (per *Data type*) e *Bar* (per *Graph type*) i risultati sulla distribuzione dei metaboliti saranno ora normalizzati e visualizzati in un grafico a barre invece che uno *stackplot*.

Esportazione

Cliccando il pulsante *Download* viene scaricato un file excel che riporta sia il *setup* della simulazione, quindi tutti i parametri impostati, sia i risultati ottenuti riguardo le distribuzioni di nutrienti e batteri, organizzati in un unico *dataframe* come quello in Figura 5.2
.

		bacterias																				nutrients				
		b1				b2				b3				b4				b5				f0	f1	f2	f3	f4
		abs	rel	rel_tocell	growth rate	abs	rel	rel_tocell	growth rate	abs	rel	rel_tocell	growth rate	abs	rel	rel_tocell	growth rate	abs	rel	rel_tocell	growth rate	amount	amount	amount	amount	amount
cell	iteration																									
0	0	14	0,15	0,151	0	22	0,24	0,237	0	14	0,15	0,151	0	14	0,15	0,151	0	29	0,31	0,312	0	9	0	13	17	20
	1	13	0,09	0,113	0,03600541	20	0,14	0,174	0,00388593	13	0,09	0,113	0,0034271	13	0,09	0,113	0,09045913	56	0,38	0,487	2,03309997	17	13	13	3	13
	2	12	0,03	0,075	0,03744563	18	0,05	0,113	0,00285497	12	0,03	0,075	0	12	0,03	0,075	0,07855662	105	0,3	0,66	1,90603122	13	9	18	2	17
	3	11	0,02	0,055	0,01950293	17	0,03	0,085	0,0024981	11	0,02	0,055	0,00391668	11	0,02	0,055	0,03553752	149	0,24	0,749	0,99053591	9	5	21	2	22
	4	10	0,01	0,047	0,02753355	16	0,02	0,075	0,00190331	10	0,01	0,047	0,00731114	10	0,01	0,047	0,08779857	168	0,19	0,785	0,42619332	7	5	23	1	23
	5	9	0,01	0,038	0,05363307	15	0,02	0,063	0,00199848	9	0,01	0,038	0	9	0,01	0,038	0,16584175	196	0,19	0,824	0,49722554	2	13	21	6	17
	6	8	0,01	0,031	0,03120469	14	0,01	0,055	0,00210366	8	0,01	0,031	0	8	0,01	0,031	0,17156043	218	0,21	0,852	0,38120624	2	9	24	5	19
	7	7	0,01	0,027	0,06240939	13	0,01	0,049	0,006813	7	0,01	0,027	0,00548336	7	0,01	0,027	0,1990101	229	0,22	0,871	0,25079358	15	18	13	8	5
	8	6	0,01	0,019	0,01733594	12	0,01	0,037	0,00315549	6	0,01	0,019	0,00996974	6	0,01	0,019	0,17305226	291	0,25	0,907	0,70106896	6	13	19	5	16
	9	5	0	0,014	0,06240939	11	0,01	0,031	0,0066616	5	0	0,014	0	5	0	0,014	0,1990101	327	0,26	0,926	0,40191074	12	19	12	6	10
	10	4	0	0,01	0	10	0,01	0,024	0	4	0	0,01	0	4	0	0,01	0,16584175	394	0,28	0,947	0,56708367	0	14	22	3	20
	11	3	0	0,008	0,06240939	9	0,01	0,023	0,0066616	3	0	0,008	0	3	0	0,008	0,1990101	380	0,28	0,955	0,08577141	10	18	13	4	14
	12	2	0	0,005	0,06240939	8	0,01	0,018	0,0066616	2	0	0,005	0	2	0	0,005	0,1990101	424	0,29	0,968	0,39241819	9	21	11	3	15
	13	1	0	0,002	0,06240939	7	0,01	0,015	0,0066616	1	0	0,002	0	1	0	0,002	0,1990101	455	0,3	0,978	0,30632645	8	23	9	2	17
	14	0	0	0	0,06240939	6	0	0,013	0,0066616	0	0	0	0	0	0	0	0,1990101	474	0,3	0,988	0,24076184	7	24	7	1	20
	15	0	0	0	0	5	0	0,01	0,0066616	0	0	0	0	0	0	0	0	482	0,3	0,99	0,18671326	6	24	5	0	24
	16	0	0	0	0	4	0	0,008	0	0	0	0	0	0	0	0	0	477	0,29	0,992	0,13890106	0	24	6	0	29
	17	0	0	0	0	3	0	0,007	0,0066616	0	0	0	0	0	0	0	0	441	0,28	0,993	0	4	24	2	0	29
	18	0	0	0	0	2	0	0,005	0	0	0	0	0	0	0	0	0	429	0,27	0,995	0,1018814	0	24	3	0	32
	19	0	0	0	0	1	0	0,003	0,0066616	0	0	0	0	0	0	0	0	396	0,26	0,997	0	2	24	1	0	32
	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	377	0,25	1	0,05718094	0	24	1	0	34
	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	348	0,24	1	0	0	24	1	0	34
	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	321	0,23	1	0	0	24	1	0	34
	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	296	0,22	1	0	0	24	1	0	34
	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	274	0,21	1	0	0	24	1	0	34
	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	253	0,2	1	0	0	24	1	0	34
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	5	20	23	9
	1	0	0	0	0	1	0,01	0,032	0	0	0	0	0	0	0	0	0	30	0,21	0,968	0	3	5	20	23	9

Simulation parameters | Simulation data

Figura 5.2: Foglio del file excel che riporta i risultati della simulazione

Capitolo 6

Conclusioni

In questo elaborato è presentata la progettazione e lo sviluppo di una *Graphical User Interface* (GUI) con Dash, una libreria Python per lo sviluppo di applicazioni web, a supporto di un simulatore di comunità batteriche basato su un modello ad agenti multipli (*Agent Based*).

Per prima cosa, sono state analizzate la struttura e le funzionalità del simulatore. Ne vengono poi messi in luce i limiti, focalizzandosi in particolare sulla mancanza di un'interfaccia *user-friendly*, che ne impedisce l'utilizzo ad utenti con ridotte conoscenze informatiche.

Successivamente, sono state illustrate le fasi che hanno portato all'implementazione della GUI, a partire dagli strumenti che hanno reso possibile la realizzazione del progetto e la sua condivisione, fino alla progettazione e implementazione vera e propria dell'applicazione web.

L'interfaccia grafica del simulatore è caricata in un *repository* GitLab insieme al pacchetto Python "Bactlife" contenente una versione aggiornata del programma.

Benché l'obiettivo di rendere il simulatore accessibile e di facile utilizzo da parte di tutti gli utenti interessati sia stato raggiunto, esso presenta ancora delle mancanze. Nello specifico, manca la possibilità di modificare il numero di specie batteriche della popolazione ed il tempo di esecuzione del programma risulta in certi casi eccessivo.

L'obiettivo a lungo termine è realizzare uno strumento valido per lo studio delle comunità microbiche. Alcune implementazioni che risulterebbero utili a tale scopo sono:

1. l'utilizzo da parte dei batteri dell'energia estratta dai nutrienti, che ora vengono totalmente utilizzati per produrre metaboliti trascurando il dispendio energetico;

2. l'introduzione di un flusso di materia esterno che influenzi la dinamica di nutrienti e tossine;

3. ampliare la caratterizzazione spaziale per consentire simulazioni in due o tre dimensioni;

4. l'impiego di un database da cui ricavare i parametri di interazione delle specie batteriche [13–16].

Infine, come prospettiva futura, per poter verificare l'attendibilità dei risultati ottenuti, si può pensare di validare il simulatore confrontando una simulazione con una co-coltura batterica reale su una comunità ristretta, in bio-reattore; quest'ultimo è infatti approssimabile ad un ambiente costituito da una sola cella di cui si può quindi avere un controllo preciso sia per quanto riguarda i nutrienti, sia per quanto riguarda la popolazione batterica, se opportunamente selezionata.

Bibliografia

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

[2] Sai Manasa et al. Jandhyala. «Role of the normal gut microbiota». In: *World journal of gastroenterology* 21 (2015), pp. 8787–8803. DOI: 10.3748/wjg.v21.i29.8787.

[3] Karoline Faust et al. «Microbial Co-occurrence Relationships in the Human Microbiome». In: *PLOS Computational Biology* 8.7 (lug. 2012), pp. 1–17. DOI: 10.1371/journal.pcbi.1002606. URL: https://doi.org/10.1371/journal.pcbi.1002606.

[4] Yili Qian, Freeman Lan e Ophelia S Venturelli. «Towards a deeper understanding of microbial communities: integrating experimental data with dynamic models». In: *Current Opinion in Microbiology* 62 (2021), pp. 84–92. ISSN: 1369-5274. DOI: https://doi.org/10.1016/j.mib.2021.05.003. URL: https://www.sciencedirect.com/science/article/pii/S1369527421000631.

[5] Robert Marsland et al. «The Community Simulator: A Python package for microbial ecology». In: *PLOS ONE* 15.3 (mar. 2020), pp. 1–18. DOI: 10.1371/journal.pone.0230430. URL: https://doi.org/10.1371/journal.pone.0230430.

[6] Pahala Gedara Jayathilake et al. «A mechanistic Individual-based Model of microbial communities». In: *PLOS ONE* 12.8 (ago. 2017), pp. 1–26. DOI: 10.1371/journal.pone.0181965. URL: https://doi.org/10.1371/journal.pone.0181965.

[7] Eugen Bauer et al. «BacArena: Individual-based metabolic modeling of heterogeneous microbes in complex communities». In: *PLOS Computational Biology* 13.5 (mag. 2017), pp. 1–22. DOI: 10.1371/journal.pcbi.1005544. URL: https://doi.org/10.1371/journal.pcbi.1005544.

[8] A. Lucchiari. «Bactlife: simulatore per comunità batteriche - sviluppo del pacchetto Python». Tesi triennale in Ingegneria Biomedica. Università degli Studi di Padova, 2022.

[9] *Pandas documentation*. https://pandas.pydata.org/docs/getting_started/index.html.

[10] *Pandas documentation*. https://plotly.com/python/getting-started/.

[11] *Dash Python User Guide*. https://dash.plotly.com/.

[12] *How To Build An App*. https://medium.com/innovation-res/how-to-build-an-app-using-dash-plotly-and-python-and-deploy-it-to-aws-5d8d2c7bd652.

[13] Carola et al. Söhngen. «BacDive–the Bacterial Diversity Metadatabase.» In: *Nucleic acids research* 42 (2014), pp. 592–599. DOI: 10.1093/nar/gkt1058. URL: https://doi.org/10.1093/nar/gkt1058.

[14] Ross et al Overbeek. «The SEED and the rapid annotation of microbial genomes using Subsystems Technology (RAST)». In: *Nucleic acids research* 42.3 (2014), pp. 206–214. DOI: 10.1093/nar/gkt1226. URL: https://doi.org/10.1093/nar/gkt1226.

[15] Kutt L. et al. Magnúsdóttir S. Heinken A. «Generation of genome-scale metabolic reconstructions for 773 members of the human gut microbiota.» In: *Nat Biotechnol* 35 (2017), pp. 81–89. DOI: 10.1038/nbt.3703. URL: https://doi.org/10.1038/nbt.3703.

[16] Disz T. et al. Brettin T. Davis J. «RASTtk: A modular and extensible implementation of the RAST algorithm for building custom annotation pipelines and annotating batches of genomes». In: *Scientific Reports* 5 (2015). DOI: 10.1038/srep08365. URL: https://doi.org/10.1038/srep08365.

Appendice A

Codice Python dell'interfaccia

```
##import bactlife package
from bactlife.cell import *
from bactlife.bact import *
from bactlife.set_df import *

##import necessary dependencies
import dash
from dash import Dash,html,dcc,Input,Output,State,dash_table,callback_context
import dash_bootstrap_components as dbc
from dash.exceptions import PreventUpdate
import plotly.express as px
import json


##create a dataframe with data on bacterias amount
##@param bd dict with bact species as keys and number of bacteria for each iter and cell
##@param n_it number of iterations
##@param x list of cells
def df_bacs(bd,n_it,x):
    frames_b=[]
    pos=0
    for bac in bd:
        rel_cell=[]
        rel_all=[]
        for j in range(n_it+1):
            for i in range(len(x)):
                rel_cell.append(x[i].getBactCell()[j][bac])
                rel_all.append(x[i].getBactAll()[j][bac])
                if j==0:
                    growth=[0]*len(x)
                else:
                    growth.append(x[i].getMatrGrowth()[j-1][pos])
        row=bd[bac]
        num_cell=len(x)
        c=list([i]*len(x) for i in range(n_it+1))
        bd_p={'cycle':sum(c,[]),
              'abs':row,
              'rel_cell':rel_cell,
              'rel_all':rel_all,
              'growth rate':growth,
              'bacteria':[bac]*len(row),
              'cell':list(range(len(x)))*(n_it+1)}
        frames_b.append(pd.DataFrame(bd_p))
        pos+=1
```

```
    df_b=pd.concat(frames_b)
    return df_b

def df_nut(x):
    frames_n=[]
    for I in range(len(x)):
        MF=numpy.matrix(x[I].getMatrFood()).transpose()
        for r in range(len(MF)):
            row=MF[r,:].tolist()[0]
            d={'cycle':list(range(len(row))),
               'amount':row,
               'nutrient':['f'+str(r+1)]*len(row),
               'cell':[I]*len(row)}
            frames_n.append(pd.DataFrame(d))
    df_n=pd.concat(frames_n)
    return df_n

##create a dash table from a dataframe of toxicity and metabolism for each nutrient and
    for each type of bacteria
##param bacts list of bacts
##param len_m number of nutrients
##@return table dash table created from dataframe
def parameters_table(bacts,len_m):
    d=dict()
    d['Nutrients']=list('f'+str(i+1) for i in range(len_m))
    cols=[{'name':['','Nutrients'],'id':'Nutrients'}]
    for b in bacts:
        d['Interaction '+str(b.type())]=b.getm()
        d['Toxicity '+str(b.type())]=b.gett()
        cols.append(
            {'name':[str(b.type()),'Interaction'],
             'id':'Interaction '+str(b.type()),
             'type':'numeric'})
        cols.append(
            {'name':[str(b.type()),'Toxicity'],
             'id':'Toxicity '+str(b.type()),
             'type':'numeric'})
    df=pd.DataFrame(d)
    table=dash_table.DataTable(
        id='parametri_met_table',
        columns=cols,
        data=df.to_dict('records'),
        merge_duplicate_headers=True,
        editable=True,
        fixed_columns={'headers': True, 'data': 1},
        fixed_rows={'headers':True,'data':0},
        style_table={'overflow':'auto',
                     'maxWidth':'100%',
                     'maxHeight':'300px',
                     'width':'600px'},
        style_cell={'minWidth':'100px',
                    'width':'100px',
                    'maxWidth':'100px'})
    return table


##app inizialitazion
app=Dash(__name__,
         external_stylesheets=[dbc.themes.LUX],
         show_undo_redo=True)
app.title='Agent based simulator'
server=app.server
```

```
106
107
108 ##app layout
109 app.layout=html.Div([
110     dcc.Store(id='seed_values',storage_type='memory'),
111     dcc.Store(id='main-data',storage_type='memory'),
112     html.Div([html.H1(['Agent-based simulator for microbial communities'],
113             style={'margin':'20px'})]),
114     html.Div(id='print1'),
115     html.Div(
116         [dbc.ListGroup(
117             [dbc.ListGroupItem(
118                         dbc.ListGroup(
119                             [dbc.ListGroupItem(html.Div(html.H4('Seeds'))),
120                              dbc.ListGroupItem(
121                                  dbc.ListGroup(
122                                      [dbc.ListGroupItem(
123                                          [html.Label('Seed 1: '),
124                                           dcc.Input(id='seed1',
125                                                     type='number',
126                                                     min=0,
127                                                     style={'width':'50px',
128                                                            'height':'25px',
129                                                            'margin':'5px'}),
130                                           html.Label('Seed 2: '),
131                                           dcc.Input(id='seed2',
132                                                     type='number',
133                                                     min=0,
134                                                     style={'width':'50px',
135                                                            'height':'25px',
136                                                            'margin':'5px'})]),
137                                       dbc.ListGroupItem(
138                                           dbc.Button('Random',id='random_seed',
139                                                      style={'marginTop':'9px'}))],
140                                      horizontal=True,
141                                      style={'margin':'5px'})),
142                              dbc.ListGroupItem(
143                                  dcc.Dropdown(id='simulation',
144                                               options=['commensalism',
145                                                        'hostile environment',
146                                                        'favorable environment'],
147                                               placeholder='Some significant simulations
    ...',
148                                               style={'fontSize':18})),
149                              dbc.ListGroupItem(
150                                  [html.Div(
151                                      [dbc.Button('RUN',id='run-btn',color='danger',
152                                                  style={'margin':'15px'}),
153                                       dbc.Button('Downoald data',id='btn_download_data',
154                                                  style={'margin':'15px'}),
155                                       dcc.Download(id='download_data')])],
156                                  style={'marginTop':'10px'})],
157                             flush=True,
158                             style={'width':'350px'})),
159              dbc.ListGroupItem(
160                  dbc.ListGroup(
161                      [dbc.ListGroupItem(html.Div(html.H4('Simulation settings'))),
162                       dbc.ListGroupItem(html.Div(html.Div(
163                           [html.Div(
164                               [html.Label('Number of cells: '),
165                                dcc.Input(id='input_num_cell',
166                                          type='number',
```

```
                                        min=1,
                                        value=5,
                                        style={'width':'80px',
                                               'height':'25px',
                                               'margin':'5px'})]),
                            html.Div(
                                [html.Label('Number of nutrients: '),
                                 dcc.Input(id='input_num_nut',
                                           type='number',
                                           min=1,
                                           value=5,
                                           style={'width':'80px',
                                                  'height':'25px',
                                                  'margin':'5px'})]),
                            html.Div(
                                [html.Label('Number of species: '),
                                 dcc.Input(id='input_num_species',
                                           type='number',
                                           min=1,
                                           value=5,
                                           disabled=True,
                                           style={'width':'80px',
                                                  'height':'25px',
                                                  'margin':'5px'})]),
                            html.Div(
                                [html.Label('Max initial number of bacterias (per type):')
    ,
                                 dcc.Input(id='num_max_bac',
                                           type='number',
                                           min=1,
                                           value=40,
                                           style={'width':'80px',
                                                  'height':'25px',
                                                  'margin':'5px'})]),
                            html.Div(
                                [html.Label('Max initial number of nutrients (per type):')
    ,
                                 dcc.Input(id='num_max_nut',
                                           type='number',
                                           min=1,
                                           value=25,
                                           style={'width':'80px',
                                                  'height':'25px',
                                                  'margin':'5px'})]),
                            html.Div(
                                [html.Label('Number of iterations: '),
                                 dcc.Input(id='num_cicli',
                                           type='number',
                                           min=1,
                                           value=50,
                                           style={'width':'80px',
                                                  'height':'25px',
                                                  'margin':'5px'})]),
                            html.Div(
                                [dbc.Button('Set default values',id='def-val')],
                                style={'marginTop':'10px'})]))],
                    flush=True,
                    style={'width':'470px'})),
            dbc.ListGroupItem(
                dbc.ListGroup(
                    [dbc.ListGroupItem(html.Div(html.H4("Bacteria's parameters"))),
                     dbc.ListGroupItem(html.Div(id='parametri_met',
```

```
                                                    style={'marginTop':'20px'}))],
                    flush=True,
                    style={'width':'550px'}))],
            horizontal=True)],
        style={'margin':'20px'}),
    html.Div(
        [dbc.ListGroup(
            [dbc.ListGroupItem(
                dbc.ListGroup(
                    [dbc.ListGroupItem(html.H4("Metabolites variation over time")),
                     dbc.ListGroupItem(
                         dcc.Dropdown(id='visible_cell',
                                      value=['all'],
                                      multi=True,
                                      clearable=False,
                                      placeholder='select visible cell')),
                     dbc.ListGroupItem(
                         [dcc.Loading(id='loading_graph1',
                                      type='circle',
                                      children=[dcc.Graph(id='met_graph')])],
                         style={'height':'550px',
                                'overflowY': 'scroll'})],
                    style={'width':'700px'},
                    flush=True)),
             dbc.ListGroupItem(
                 dbc.ListGroup(
                     [dbc.ListGroupItem(html.H4("Bacteria distribution")),
                      dbc.ListGroupItem(
                          dcc.Dropdown(id='visible_bacts',
                                       multi=True,clearable=False,value=['all'],
                                       placeholder='select visible species')),
                      dbc.ListGroupItem(
                          [dbc.ListGroup(
                              [dbc.ListGroupItem(
                                  [html.Small('Data type'),
                                   dcc.RadioItems(id='amt_type',
                                                  options=[{'label':'Absolute',
                                                            'value':'abs'},
                                                           {'label':'Relative',
                                                            'value':'rel_all'},
                                                           {'label':'Relative (cell)',
                                                            'value':'rel_cell'}],
                                                  value='abs',
                                                  inline=True,
                                                  labelStyle={'marginRight':'5px',
                                                              'marginLeft':'5px',
                                                              'fontSize':18})],
                                  style={'border':'1px','borderRight':'0.3px  grey solid'
    }),
                               dbc.ListGroupItem(
                                   [html.Small('Graph type'),
                                    dcc.RadioItems(id='graph_tipe',
                                                   options=['Area','Bar'],
                                                   value='Area',
                                                   inline=True,
                                                   labelStyle={'marginRight':'5px',
                                                               'marginLeft':'5px',
                                                               'fontSize':18})],
                                   style={'border':'1px','borderLeft':'0.3px  grey solid'
    })],
                              horizontal=True,
                              flush=True)]),
```

```
                        dbc.ListGroupItem(
                            dcc.Loading(id='loading_graph2',
                                        type='circle',
                                        children=[dcc.Graph(id='bat_graph')]))],
                       style={'width':'700px'},
                       flush=True))],
            horizontal=True)],
        style={'margin':'20px'}))

##Callback Functions

##updates the simulation to "None" when seed values are randomized
@app.callback(
    Output('simulation','value'),
    Input('random_seed','n_clicks'))
def random_seed(click):
    return None

##updates the parameters table every time the seed values are changed;
##updates seed values when num_nut is changed or button "random" is clicked
@app.callback(
    Output('seed1','value'),
    Output('seed2','value'),
    Output('parametri_met','children'),
    Input('simulation','value'),
    Input('input_num_nut','value'),
    Input('input_num_species','value'),
    Input('random_seed','n_clicks'),
    Input('seed1','value'),
    Input('seed2','value'),)
def set_parameters(simulation,num_nut,num_species,random_seed,seed1,seed2):
    inp=callback_context.triggered_id ##trigger input id
    ## the trigger input are the seed values:  update the output table
    if inp in ['seed1','seed2']:
        seed(seed1)
        numpy.random.seed(seed2)
        bacts=randomBacts(num_nut,0)
        table=parameters_table(bacts,num_nut)
        return [dash.no_update,dash.no_update,table]

    ## the trigger input is the 'RANDOM' button:  randomize seed values and update them
    elif inp=='random_seed':
        seed()
        numpy.random.seed()
        s1=randint(0,100)
        s2=randint(0,100)
        seed(s1)
        numpy.random.seed(s2)
        bacts=randomBacts(num_nut,0)
        table=parameters_table(bacts,num_nut)
        return [s1,s2,json.dumps(data),table]

    ## the trigger input is 'input_nut_nut' or 'input_num_species': update the output
    table
    elif inp in ['input_num_nut','input_num_species']:
        seed(seed1)
        numpy.random.seed(seed2)
        bacts=randomBacts(num_nut,0)
        table=parameters_table(bacts,num_nut)
        return [dash.no_update,dash.no_update,table]

```

```
    ## the trigger input is 'simulation': update seed values (based on the chosen simulation), storage data and output table
    elif inp=='simulation':
        if simulation=='commensalism':
            s1=5
            s2=2
        if simulation=='favorable environment':
            s1=7
            s2=20
        if simulation=='hostile environment':
            s1=6
            s2=4
        if simulation==None :
            if seed1==None or seed2==None:
                seed()
                numpy.random.seed()
                s1=randint(0,100)
                s2=randint(0,100)
            else:
                s1=seed1
                s2=seed2
        seed(s1)
        numpy.random.seed(s2)
        bacts=randomBacts(num_nut,0)
        table=parameters_table(bacts,num_nut)
        return  [s1,s2,table]


##update all paarameters to default when the 'default' button has been clicked
##or when a simulation is selected
@app.callback(
    Output('input_num_cell','value'),
    Output('input_num_nut','value'),
    Output('input_num_species','value'),
    Output('num_max_bac','value'),
    Output('num_max_nut','value'),
    Output('num_cicli','value'),
    Input('def-val','n_clicks'),
    Input('simulation','value'))
def set_default_values(def_click,simulation):
    inp=callback_context.triggered_id
    if inp=='def-val' or simulation!=None :
        len_x=5
        len_m=5
        len_s=5
        max_bac=40
        max_f=25
        n_it=50
        return [len_x,len_m,len_s,max_bac,max_f,n_it]
    else:
        raise PreventUpdate

##create cells and bacts;
##stores data when 'run' button is clicked
##downloads data when 'download' button is clicked
@app.callback(
    Output('main-data','data'),
    Output('download_data','data'),
    Input('run-btn','n_clicks'),
    Input('btn_download_data','n_clicks'),
    State('parametri_met','children'),
    State('input_num_cell','value'),
```

```
    State('input_num_nut','value'),
    State('input_num_species','value'),
    State('num_max_bac','value'),
    State('num_max_nut','value'),
    State('num_cicli','value'),
    State('seed1','value'),
    State('seed2','value'))
def main(run,dwnld,par_data,len_x,len_m,len_s,max_bac,max_f,n_it,s1,s2):
    inp=callback_context.triggered_id
    seed(s1)
    numpy.random.seed(s2)

    x=[None]*len_x #cells vector
    bacts=randomBacts(len_m,0)#random bacteria list

    if par_data:
        data_par=par_data['props']['data']
        df_data_par=pd.DataFrame(data_par)
        set_from_df(df_data_par,bacts) #modifica i vettori m e t di ogni specie in base
    ai dati in tabella

    bac_diz=dict() #dict of bacteria for first cell
    bac_null=dict() # dict of bacteras for empty cells
    for bac in bacts:
        bac_diz[bac]=randint(0,max_bac)
        bac_null[bac]=0

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

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


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

    bd=dict()
    for bac in x[0].getBact():
        bd[bac]=[]*len(x)
    for i in range(len_x):
        x[i].RelUpd()
        for bac,num in x[i].getBact().items():
            bd[bac].append(num)

    for z in range(n_it):
        for i in range(1,len(x)+1):# evolution of cells
            x[-i].evolution()
        for i in range(len(x)):
            x[i].RelUpd()
            for bac,num in x[i].getBact().items():  #update bd to export
                bd[bac].append(num)

    df_b=df_bacs(bd,n_it,x)
    df_n=df_nut(x)

    if inp=='btn_download_data':
        data_par=par_data['props']['data']
        df_par=pd.DataFrame(data_par)
        df_data=data_export(bd,n_it,len_x,len_m,x)
        settings={'cells':len_x,'nutrients':len_m,'species':len_s,
                  'maximum initial number of bacterias (per type):':max_bac,
```

```
                    'maximum initial number of nutrients (per type)':max_f,
                    'iterations':n_it}
        df_settings=pd.Series(settings)
        df_seeds=pd.Series({'seed 1':s1,'seed 2':s2})
        writer=pd.ExcelWriter('data.xlsx')
        pd.DataFrame(df_seeds).T.to_excel(writer,
                                          sheet_name='Simulation parameters',
                                          index=False)
        pd.DataFrame(df_settings).T.to_excel(writer,
                                             sheet_name='Simulation parameters',
                                             index=False,
                                             startrow=3)
        df_par.to_excel(writer,
                        sheet_name='Simulation parameters',
                        index=False,
                        startrow=6)
        df_data.to_excel(writer,sheet_name='Simulation data')
        writer.save()
        return dash.no_update,dcc.send_file('data.xlsx')

    datasets={
        'df_b':df_b.to_json(date_format='iso', orient='split'),
        'df_n':df_n.to_json(date_format='iso', orient='split'),
        'bd':pd.DataFrame(bd).to_json(date_format='iso',orient='split')
        }
    return json.dumps(datasets),None

##sets the 'visible cells' and 'visible bacts' dropdown menus options based on stored
    data
@app.callback(
    Output('visible_cell','options'),
    Output('visible_bacts','options'),
    Input('main-data','data'),
    State('input_num_cell','value'),
    State('input_num_species','value')
    )
def options_upd(data,num_cells,num_bacs):
    inp=callback_context.triggered
    bacterias=[{'label':'select all','value':'all'}]
    cells=[{'label':'select all','value':'all'}]
    for b in range(num_bacs):
        bacterias.append({'label':'b'+str(b+1),'value':'b'+str(b+1)})
    for c in range(num_cells):
        cells.append({'label':'cell'+str(c),'value':c})
    return cells,bacterias

##create the graph of metabolits in cells versus time
@app.callback(
    Output('met_graph','figure'),
    Output('visible_cell','multi'),
    Output('visible_cell','value'),
    Input('main-data','data'),
    Input('visible_cell','value')
    )
def graph_met(data_json,cells):
    dffs=json.loads(data_json)
    df_n=pd.read_json(dffs['df_n'],orient='split')
    if type(cells)!=int and (cells==[] or cells==None or 'all' in cells) :
        df_toplot=df_n.copy()
        multi=False
        value=None
        h=max(len(df_n.cell.unique())*100,530)
```

```
    else:
        df_toplot=df_n.copy()
        multi=False
        value=None
        if type(cells)==int:
            cells=[cells]
        mask=df_n.cell.isin(cells)
        df_toplot=df_n[mask]
        multi=True
        value=cells
        h=max(len(cells)*100,530)
    fig=px.line(df_toplot,x='cycle',y='amount',
                facet_row='cell',color='nutrient',markers=True,
                height=h,
                )
    fig.update_layout(modebar_add="hovercompare")
    return fig,multi,value

## create the graph that shows the species amount (abs,rel_all,rel_cell) in cells
@app.callback(
    Output('bat_graph','figure'),
    Output('visible_bacts','multi'),
    Output('visible_bacts','value'),
    Input('main-data','data'),
    Input('visible_bacts','value'),
    Input('graph_tipe','value'),
    Input('amt_type','value'))
def graph_bat(data_json,bacs,sel_graph_type,sel_amt_type):
    dffs=json.loads(data_json)
    df_bd=pd.read_json(dffs['bd'],orient='split')
    df_b=pd.read_json(dffs['df_b'],orient='split')
    if bacs==[] or bacs==None or 'all' in bacs:
        df_toplot=df_b
        df_toplot=df_b.copy()
        multi=False
        value=None
        df_bd['sum']=df_bd.sum(axis=1)
    else:
        if type(bacs)==str:
            bacs=[bacs]
        mask=df_b.bacteria.isin(bacs)
        df_toplot=df_b[mask]
        multi=True
        value=bacs
        sum_col=0
        for bac in bacs:
            sum_col+=df_bd[bac]
        df_bd['sum']=sum_col
    if sel_amt_type!='abs':
        max_bact_amt=1
    else:
        max_bact_amt=max(df_bd['sum'])+1

    if sel_graph_type=='Bar':
        fig=px.bar(df_toplot,x='cell',y=sel_amt_type,color='bacteria',
                   opacity=0.8,barmode='group', hover_name='bacteria',
                   hover_data={'bacteria':False,'growth rate':True},
                   animation_frame='cycle', range_y=[0,max_bact_amt])
    elif sel_graph_type=='Area':
        fig=px.area(df_toplot,x='cell',y=sel_amt_type,color='bacteria',
                    hover_data={'bacteria':False,'growth rate':True},
                    hover_name='bacteria',animation_frame='cycle',
```

```
                        range_y=[0,max_bact_amt])


    fig.update_layout(modebar_add="hovercompare")
    return fig,multi,value


if __name__ == "__main__":
    app.run_server(debug=True,
                   dev_tools_hot_reload=False)
```