University of Camerino

Computer Science (LM-18)
Distributed Calculus and Coordination

The Role of Levodopa and Carbidopa in Parkinson's Disease Using Repast4Py: A Simulation Approach

Group:
Dr. Francesco Finucci

Supervisor:
Prof. Emanuela Merelli
Prof. Stefano Maestri

Anno accademico 2023/2024

Contents

Chapter 1

Introduction

Parkinson's disease (PD) stands as one of the most intricate neurological conditions, affecting millions worldwide with its progressive motor impairments and a spectrum of non-motor symptoms. Characterized by the degeneration of dopaminergic neurons in the substantia nigra pars compacta (SNpc) region of the brain, PD's pathophysiology involves a complex interplay between genetic predispositions, environmental factors, and the immune system. While significant strides have been made in understanding its etiology and progression, the precise mechanisms underlying its onset and progression remain elusive.

This report embarks on a pioneering exploration into PD's intricate dynamics, employing Multi-Agent Systems (MAS) as a sophisticated computational framework to simulate and elucidate the communication channels between the substantia nigra and the peripheral immune system. By integrating principles from neuroscience, immunology, and artificial intelligence, our study endeavors to unravel the intricate dialogue between these systems, shedding light on potential therapeutic targets and avenues for intervention.

Understanding Parkinson's Disease: A Multifaceted Perspective

Parkinson's disease manifests through a myriad of symptoms, ranging from tremors, bradykinesia, and rigidity to non-motor impairments such as cognitive decline,

sleep disturbances, and autonomic dysfunction. At its core lies the progressive degeneration of dopaminergic neurons in the SNpc, leading to dopamine deficiency in the striatum and subsequent disruption of motor control circuits. While this neurodegenerative process is well-documented, the underlying triggers and propagating factors remain subjects of intense investigation.

Emerging evidence implicates the immune system in PD pathogenesis, hinting at a bidirectional communication network between the central nervous system (CNS) and the peripheral immune milieu. Microglia, the resident immune cells of the brain, undergo a phenotypic switch in PD, adopting an activated, pro-inflammatory state that exacerbates neuronal demise. Concurrently, peripheral immune cells infiltrate the brain, further amplifying neuroinflammation and neuronal vulnerability. However, the precise mechanisms orchestrating this immune response and its modulation by the substantia nigra remain enigmatic.

Modeling Parkinson's Disease Dynamics: The Role of Multi-Agent Systems

Central to our investigation is the utilization of Multi-Agent Systems (MAS) as a powerful computational paradigm to simulate PD dynamics at multiple scales. MAS encapsulates a distributed system of autonomous agents, each endowed with unique behaviors, knowledge, and interactions. By abstracting the intricate interplay between the substantia nigra, peripheral immune cells, and environmental cues into computational entities, MAS facilitates the exploration of emergent phenomena and system-level dynamics that transcend reductionist approaches.

Our MAS framework integrates neuroinflammatory signaling cascades, dopaminergic neuron degeneration, and immune cell infiltration dynamics within a unified simulation environment. Agents representing microglia, astrocytes, T cells, and dopaminergic neurons interact through local and long-range communication channels, responding dynamically to stimuli and perturbations akin to PD pathology. Leveraging principles from network science, machine learning, and systems biology, our model aims to capture the nuanced interdependencies and feedback loops underlying PD progression.

In the pursuit of unraveling Parkinson's disease's enigmatic pathophysiology, this study harnesses the integrative power of Multi-Agent Systems to elucidate the complex dialogue between the substantia nigra and the peripheral immune system. By delineating the emergent properties and dynamic interactions within this intricate network, we aspire to unlock novel insights into PD etiology and identify promising avenues for therapeutic intervention. Through interdisciplinary collaboration and computational modeling, we endeavor to pave the way towards more effective treatments and ultimately, a cure for Parkinson's disease.

Chapter 2

MAS (Multi-Agent System)

A multi-agent system (MAS or "self-organized system") is a computerized system composed of multiple interacting intelligent agents. Multi-agent systems can solve problems that are difficult or impossible for an individual agent or a monolithic system to solve. Intelligence may include methodic, functional, procedural approaches, algorithmic search or reinforcement learning.

Multi-agent systems consist of agents and their environment. Typically multi-agent systems research refers to software agents.

2.1 Notations of a MAS

In order to correctly describe a MAS we need to establish the notations and mathematics behind them.

Definition 1. An **agent**, denoted with Ag, is an operating system situated in an **environment** Env capable of complex and flexible actions inside said environment. An agent navigates the environment in order to achieve the goal that has been imposed on it.

In order to correctly define the agents and the environment, we first have to define some notations we are going to use.

Definition 2. We define the **set of states** E of an environment as a finite set of discrete, istantaneous states of the environment

$$E = \{e_0, e_1, ..., e_n\}.$$

Here, e_i, for $i = 1, ..., n$, represents the single states of the environment.

Once defined the set of states of an environment, we then have to define the set of actions an agent can perform.

Definition 3. The finite **set of actions**, Ac, are the actions

$$Ac = \{\alpha_0, \alpha_1, ..., \alpha_m\}$$

that an agent can perform. Here, α_i, for $i = 1, ..., m$, are the single actions.

Now that we have defined the set of actions an agent can perform, we have to define the agents.

Definition 4. We define the **set of agents** AG as

$$AG = \{Ag_0, Ag_1, ..., Ag_k\},$$

with k being the number of agents.

Definition 5. We define R as the **set of runs** over E and Ac as

$$R = \{r_0, r_1, ..., r_u\},$$

with u being the number of **runs**.

As a natural consequence, we have to define what is a run.

Definition 6. A **run** $r \in R$, of an agent Ag in an environment Env, is a sequence that intercalates a state of the environment with an action

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} ...e_{n-1} \xrightarrow{\alpha_{n-1}} e_n.$$

Let's give the last two definitions before formally defining the environment

Definition 7. We define R^{Ac} as the subset of runs that end with an action $\alpha \in Ac$. We also define R^E as the subset of runs that end with an environment state $e \in E$.

Now, we can finally define the environment

Definition 8. An **ambient** is a 3-tuple that represents a transformation function that maps the various states of the system. It is defined as

$$Env = \{E, e_0, \tau\},$$

where E is the previously defined set of possible states of the environment, $e_0 \in E$ is the starting state and $\tau \in T$ is a **state transformation function** that maps the set of runs R^{Ac} to a set of the possible states of the environment that may follow as a result of the performed action. It is defined as

$$\tau : R^{Ac} \to 2^E.$$

If $\tau(r) = 0$, then there are no possible states after the run r. This means that said action has no effect on the environment.

Finally, we can define an agent

Definition 9. We define an **agent** Ag as a function that maps a run to an action

$$Ag : R^E \to Ac.$$

Definition 10. In a multi-agent system, an **interface** can be defined as a set of rules that regulate the interaction between two or more agents. An interface is the following 4 elements tuple

$$(A_g, S_y, S_e, P_o),$$

where

- A_g are the **agents** involved;

- S_y, also called **syntax**, is the set of rules that define the format of the communication and the messages between different environments; A primitive syntax for BioAmbients could be the following[3]
  - **Names :=** n, m, p, q;
  - $\pi :=$ **Actions**. An output action from n to m is defined as $\$n!\{m\}$, an input action is defined as $\$n?\{m\}$;
  - $\$:=$ **Directions**. Basically defines if the communication happens between different environments, locally etc..
    Intra-ambient communication is labeled as *local*, **Inter-siblings** as $s2s$, **parent-to-child** as $p2c$ and **child-to-parent** as $c2p$;
  - $M, N :=$ **Capabilities**.
    - $enter\, n$: Synch entry;
    - $accept\, n$: Synch accept;
    - $exit\, n$: Synch exit;
    - $expel\, n$: Synch expel;
    - $merge+\ n$: Synch merge with;
    - $merge-\ n$: Synch merge into;
  - $P, Q :=$ **Processes**.
    - $(new\, n)P$: restriction;
    - $P|Q$: composition;
    - $!P$: replication;
    - $[P]$: Ambient (membrane);
    - $\pi.P$: Communication prefix;
    - $M.P$: Capability prefix;

  - $\sum_{i \in I} \pi_i.P_i$: Communication Choice;
  - $\sum_{i \in I} M_i.P_i$: Capability Choice;

- S_e, also called **semantics**, is the set of rules that define the meaning of said messages;

$[(T + enter\ n.P) \mid Q] \mid [(T' + accept\ n.R) \mid S] \rightarrow [\ [P \mid Q] \mid R \mid S\ ]$	Red In
$[[(T + exit\ n.P) \mid Q] \mid (T' + expel\ n.R) \mid S] \rightarrow [P \mid Q] \mid [R \mid S]$	Red Out
$[(T + merge{+}\ \ n.P) \mid Q] \mid [(T' + merge{-}\ \ n.R) \mid S] \rightarrow [P \mid Q \mid R \mid S]$	Red Merge
$(T + local\ n!\{m\}.P) \mid (local\ n?\{p\}.Q + T') \rightarrow P \mid Q\{p \leftarrow m\}$	Red Local
$(T + p2c\ n!\{m\}.P) \mid [(c2p\ n?\{p\}.Q + T') \mid R] \rightarrow P \mid [Q\{p \leftarrow m\} \mid R]$	Red Parent Output
$[R \mid (T + c2p\ n!\{m\}.P)] \mid (p2c\ n?\{p\}.Q + T') \rightarrow [R \mid P] \mid Q\{p \leftarrow m\}$	Red Parent Input
$[R \mid (T + s2s\ n!\{m\}.P)] \mid [(s2s\ n?\{p\}.Q + T') \mid S] \rightarrow [R \mid P] \mid [Q\{p \leftarrow m\} \mid S]$	Red Sibling
$P \rightarrow Q \ \Rightarrow\ (\mathsf{new}\ n)P \rightarrow (\mathsf{new}\ n)Q$	Red Res
$P \rightarrow Q \ \Rightarrow\ [P] \rightarrow [Q]$	Red Amb
$P \rightarrow Q \ \Rightarrow\ P \mid R \rightarrow Q \mid R$	Red Par
$P \equiv P',\ P \rightarrow Q,\ Q \equiv Q' \ \Rightarrow\ P' \rightarrow Q'$	Red $\equiv$ (Struct)

- P_o, also called **policy**, are the rules that define how the agents interpret and use the received messages.

2.2 Definition of agents

In this paper, we will describe multiple agents. Most of them are what we call **reactive agents**, while the neuron is a **adaptive agent**. Let's start by defining reactive agents

Definition 11. A **reactive agent** is an agent that can be defined through the following 6-tuple

$$< E, Per, Ac, see, do, action > .$$

The elements E and Ac are already defined in Section 2.1, the other ones are:

- Per (perception) is a partition of the environment E that represents the agent's perception of the environment;
- see is a function such that

$$see : E \to Per$$

  that maps the environment into the perceived section;

- $action$ is a function such that

$$action : Per \to Ac$$

  that maps the perceived environment into an action;

- do is a function such

$$do : Ac \times E \to E$$

  That maps an action on the environment and updates said environment.

Basically, this type of agents observe the environment (see), they find the most appropriate $action$ and then the act on the environment (do).

Definition 12. An **adaptive agent** is an agent that can be defined through the following 9-tuple

$$< I, E, Per, Ac, i_0, see, next, do, action > .$$

The elements E and Ac are already defined in Section 2.1, the other ones are:

- E, Per, Ac, see and do are the same already defined in Definition 11;
- $I = \{i_0, i_1, ..., i_s\}$ is the set of the internal states of the agent;
- i_0 is the initial internal state;

- *action* is a function such that

$$action : I \times Per \to Ac$$

  that maps the perceived environment and the internal states into an action;

- *next* is a function such that

$$next : I \times Per \to I.$$

Chapter 3

Environments, interfaces, agents and parameters in this system

3.1 Environments

The environment in which Parkinson develops is the **substantia nigra**. This is the region of the brain that is responsible for the production of dopamine. It's here that the so called **dopaminergic neurons** are situated. The death of those neurons is acknowledged as one of the reasons why Parkinson's disease happens. This environment can be treated by using deep brain stimulation and by using dopamine replacement drugs such as **levodopa** paired with **carbidopa**. Those treatments can slow the infenction process, thus making the environment more ostile to the desease.

Another environment is the **peripheral immune system** situated in the **gut**. In this environment the real autoimmune reaction happens, since **T-Cells** are infected by **antigens** released by the dying neurons in the brain.

By going deeper inside the previously mentioned sections of the brain, we observe that an even smaller environment for Parkinson is the neuron itself. It's in the nerve cells the infection process actually takes place, and it is through them that the desease is spread inside the substantia nigra. In section 3.2 we will also describe

them as adaptive agents, because they react to the external environment while also having inside states.

3.2 Agents

3.2.1 Neuron as an agent inside the Substantia nigra

The substantia nigra is the area of the brain in which the desease develops. The observed agents are the following

- The **neuron** is an agent inside the substantia nigra. In fact, they are the reason why $\alpha-$synuclein proteins and cellular debris are passed from one neuron to the other. This happens in two ways when the neuron dies:

  (1) Antigens and cellular debris are released into the brain, infecting nearby microglias and astrocytes. Antigens also travel through the blood brain barrier and

  (2) $\alpha-$synuclein are transferred to the next neurons through the axon terminals (the extremities of the neuron that act like a bridge between cells).

- **Microglia** and **Astrocytes** release cytokines and infect the neuron from the outside;

- **Cytokines**: When released from astrocytes and microglia they become neurotoxic and infect other neurons.

- **Antigens**: They are released by the neurons upon their death. They travel across the blood brain barrier in order to reach the peripheral immune system and they infect nearby T-Cells once there.

3.2.2 Neuron as an environment

In this section, we will observe the agents inside the neuron. We observed the following agents

- The main agent described by the automa in Section 4.2 is the $\alpha-$**synuclein**. The $\alpha-$synuclein is the protein responsible for the birth of the desease. This happens when it starts to reproduce in an abnormal way, thus increasing exponentially its quantity inside the neuron;
- **Lewy's bodies**: They are formed by fibrils and oligomers. Those two are the result of the aggregation of $\alpha-$synucleins;
- **Mythocondria**: They are infected by $\alpha-$synucleins and impair the correct function of the autophagosome, causing a fault in the communication between neurons;
- **Autophagosome**: They clean the neuron from $\alpha-$synucleins and infected mythocondria.

3.2.3 Agents in the peripheral immune system

As we've already said, we are also studying the agents situated in the gut and in the peripheral immune system that communicate with the brain. Those agents are:

- **T-Cells**: Those are the agents that produce the various helpers. The helpers have the role to both help or harm the system, depending on the helper's type;
- **T-Helpers 1**: Those are pro-inflammatory helpers, meaning that they will contribute to the spreading of the disease;
- **T-Helpers 2**: Those are good helpers, meaning that they will contribute to the stop of the disease;
- **Levodopa**: they are introduced in the system via oral acquisition. It's a medicine that needs to travel the blood brain barrier in order to reach the brain and help the system.

3.2.4 Parameters of the simulation

The simulation contains not only agents, but also parameters. Parameters are modifiers of the simulation thanks to which the behaviour of the simulation changes accordigly. In this case, the two modifiers that are the essential for the correct functioning of the solution are

- Dopamine effectiveness: describes how effective the dopamine in the brain is:
- Carbidopa effectiveness: describes how effective the carbidopa in the gut is.

3.3 Interface: The Blood Brain Barrier

The blood brain barrier[4], that is the barrier situated between the brain and the peripheral immune system, is implemented in our simulation as an interface. It uses the set of semantics and syntaxes that were previously defined in Section 2.1 and it's implemented as a **monitor**. Being a monitor means that the interface is constantly listening to the two environments while waiting for the right conditions to be met in order to execute an action. In this case, the only action that the barrier can perform is the transportation of cells from one environment to the other.

Chapter 4

Describing Parkinson's disease

4.1 Finite state automata

A finite-state automaton (FSA) is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSA can change from one state to another in response to some inputs; the change from one state to another is called a transition.

A finite state automaton (FSA) is described by a five-element tuple $(E, \Sigma, \delta, S_0, F)$, where

- E is a finite set of states in which the system can be;
- Σ is a finite nonempty input alphabet that contains the actions the system can perform;
- $Ac \in \Sigma$ is the finite set of actions that the system performs to go from one state to the next;
- e_0 is the initial state;
- F is the set of the final states. In this case, since the automa loops on itself, there are none.

4.2 Description of the states with neuron as an environment

4.2.1 Describing the states using the automata

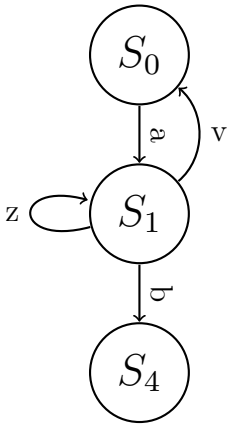
In this section we are gonna describe the states that the neurons go through during parkinson. We assume an initial state of the neuron in which the protein is present in normal quantity and then we went through every step of the process starting from the moment in which the α-synuclein overwhelmes the neuron and activates the process that leads to the death of the neuron. Then, we also describe how the α-synuclein are able to move from one neuron to the next, thus spreading the desease.

The states are the following:

- (S_0) Stable state. The neuron is in the homeostasic state, so it's regulating and correctly producing α-synucleins;
- (S_1) Alpha state. The neuron transitions out of the homeostasic state, thus starting to produce massive quantities of α-synucleins. Those proteins also start to misfold and overwhelms the neuron;
- (S_2) Dead state. The accumulation of misfolded α-synucleins causes the death of the neuron;

The actions between states are the following:

- (z) the α-synuclein production increases exponentially and starts to misfold, thus creating Lewy's bodies;
- (a) the neuron exists his homeostatic state;
- (b) the aggregation of misfolded α-synucleins causes the death of the neuron;
- (v) The autophagosome is able to clean the neuron and bring it back to the stable state or the dopamine injected in the brain by *levodopa* is able to turn it back.



4.3 Description of the states with neuron as an agent

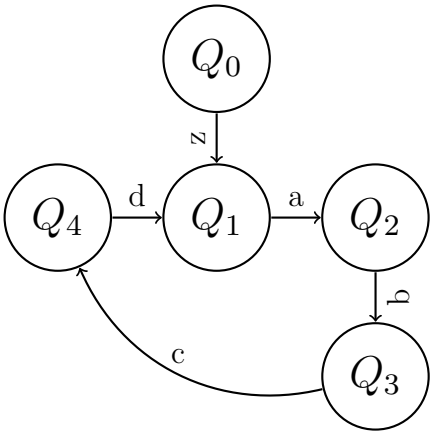
In this case, the neuron stops being the environment. The studied environment is the substantia nigra, the region of the brain that produces dopamine. The states (in this case described by Q) are the following:

- (Q_0) All of the neurons inside the substantia nigra are healthy;
- (Q_1) The neuron dies;
- (Q_2) Nearby Microglia are infected;
- (Q_3) Nearby Astrocytes are infected;
- (Q_4) The neuron is infected.

The actions between states are the following:

- (z) The infected neuron goes through the internal processes described by the automata in Section 4.2 and dies;
- (a) The dead neuron releases cellular debris and α-synucleins, infecting nearby microglia;
- (b) The dead microglia releases cytokines;

- (c) The dead astrocyte significantly increments the release of cytokines, attacking nearby neurons;
- (d) The infection caused by the cytokines kills the neuron.



4.4 Finite State Automata for the peripheral immune system

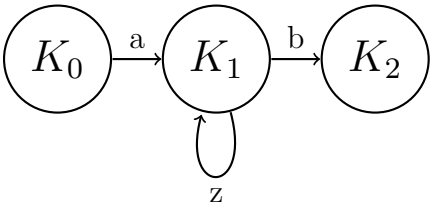
The steps of the peripheral system are the following

- (K_0) All of the T-Cells in the environment are healthy;
- (K_1) One or more T-Cells are infected;
- (K_2) Movement of TH1s towards the brain starts.

and the actions are

- (a) T-Cells are infected by either antigens or dopamine;
- (b) $TH1/TH2 > 1$
- (z) Production of TH1s

The following is the finite state automata


K_0
a
K_1
b
K_2
z

Chapter 5

Rules of the system and implementation

In this section we will describe the rules of the system. We will describe the perception of the agents, the actions they perform and how they update the environment.

5.1 The environments

The two environments we will be looking at are the substantia nigra and the gut, implemented on two discrete grids of dimensions 40×40.

5.2 Substantia nigra

5.2.1 The neuron

The neuron, in our grid, will have a perceived environment of its 1th-neighbourhood, meaning that it will be able to observe the agents in the cells in its closest proximity. So, the *Per* of the neuron is described in the following grid:

O	O	O
O	$\mathcal{S}$	O
O	O	O

In this case, the 'o' represents the perceived environment of the neuron.

The 9-tuple of the neuron as an adaptive agent is

(1) $See : E = \{$40x40 grid$\} \rightarrow Per = \{$3x3 grid, excluding the center$\}$;

(2) $I = \{$'stable', 'dead', 'alpha'$\}$. $i_0 =$ stable;

(3) $Ac = \{$'release cellular debris', 'release antigens'$\}$;

(4) $Action : Per \times I \rightarrow Ac$;

(5) $do : Ac \times E \rightarrow E$;

(6) $next : I \times Per \rightarrow I$

The stable state

In its stable state the neuron can either

(1) Transition into the α state on its own, via the process described in Section 4.2. This transition doesn't need any external input to happen and it will look like this

O	O	O
O	α	O
O	O	O

Another way to transition into the α state is if the stable neuron has a dead neuron that previously was in the alpha state in its close proximity. So, from this

o	o	o
o	n	d_α
o	o	o

It goes to this

o	o	o
o	α	d_α
o	o	o

In this case, the label d_α represents a dead neuron that previously was in the alpha state. When this happens, all of the α-synucleins and misfolded α-synucleins travel to the nearby neuron;

(2) Transition into the d state, if it's infected by nearby cytokines. So, if we have a configuration with at least two cytokines adiacent to the neuron like the following

o	o	o
o	n	ci
o	ci	o

we end up with this

o	o	o
o	d_s	ci
o	ci	o

In this case, the label d_s means that the dead neuron previously was in the stable state.

What can also happen is the death of the neuron caused by 2 or more *th1s* in its close proximity. So from this

o	o	o
o	n	$th1$
o	$th1$	o

to this

o	o	o
o	d_s	$th1$
o	$th1$	o

(3) Remain into the stable state. This happens if neither of the previous conditions are met.

The α state

In the α state the neuron can

(1) Transition back to the stable state n, if the autophagosome inside the neuron are able to clean the misfolded α proteins before the neuron can degenerate. So, from this

o	o	o
o	α	o
o	o	o

it can transition back to this

o	o	o
o	s	o
o	o	o

Although it is very rare that a neuron is able to heal on its own. What can also happen is that the neuron is able to heal thanks to the *dopamine* in the

brain created by the *levodopa*, which is more likely to happen;

(2) Transition into the dead state d through its internal process or if cytokines or TH1s are in its close proximity. When an α neuron dies it spreads misfolded α-synucleins mixed with cellular debris (we will call that br) and antigens (called an) in the close proximity. So, we go from this

o	o	o
o	α	o
o	o	o

to this

br	br	an
br	d_α	an
br	br	br

If a neuron dies while not being in the α state it won't release those proteins. Note that the neuron has a 70% chance of producing *debris* and a 30% chance of producing *antigens*.

The dead state

A neuron in the dead state d cannot change state and will stay dead.

Inside the neuron

During each and every tick of the simulation, dopaminergic neurons have random chance to transition into the α state. When the number of α-synucleins inside the neuron goes above a certain threshold, lewy's bodies will form. A large accumulation of Lewy's bodies will kill the neuron.

The other path of the inside states is death through the infection of mitochondria. At each tick, a number of misfolded α-synucleins will attack mitochondria and

they have a chance to infect it. After the percentage of infected mitochondria reaches a certain threshold, the neuron will die.

Lastly, during each and every tick of the simulation autophagosome will clean the neuron from a random number of misfolded α-synucleins. Auphagosomes won't be able to clean already formed Lewy's bodies or infected mitochondria, so they are there to stay.

Finally, dopamine will help the neuron go back to the stable state through a parameter that we will discuss in the next chapter.

5.2.2 Microglias

The microglial cells perform an impressive number of tasks in the central nervous system (CNS). They are primarily associated with the immune response and maintenance of homeostasis[1].

Microglias only have two states: alive (m_i) and dead (m_d). .

.

This is the 6-tuple that describes it

(1) $See : E = \{40\text{x}40 \text{ grid}\} \rightarrow Per = \{3\text{x}3 \text{ grid, excluding the center}\}$;

(2) $Ac = \{\text{'release cytokines'}\}$;

(3) $Action : Per \rightarrow Ac$;

(4) $do : Ac \times E \rightarrow E$.

The alive state

While in the alive state, a microglia can transition to the dead state if at least two cellular debris spread by dead α neurons are in its close proximity. So, from this

[1]Homeostasis refers to the body's ability to maintain a stable internal environment despite external changes.

br	o	o
o	mi	br
o	o	o

it transitions to this

br	o	o
o	m_d	br
o	o	o

Once a microglia is dead, it spreads cytokines in its close proximity like this

br	ci	ci
ci	m_d	br
ci	ci	ci

The dead state

A dead microglia will remain dead.

5.2.3 Astrocytes

Astrocytes are the major glial cells within the central nervous system (CNS) and have a number of important physiological properties related to CNS homeostasis. Their numbers exceed those of neurons in most brain areas.

Inside the grid they are represented with the following symbol

o	o	o
o	a_s	o
o	o	o

Being a glia cell, its description is mostly the same as Section 5.2.2, there are only

two differences. The first difference is that they are present in larger numbers. The formula is the following

$$num_{astrocytes} = (num_{neurons} + num_{microglias}) * 1.5.$$

The second difference is that they can be infected by both cellular debris *br* and cytokines *ci*. As for the microglias, they need to have at least two cytokines in its close proximity in order to become infected. So, from this

o	o	o
o	a_s	ci
ci	o	o

they transition into their dead state

o	o	o
o	a_d	ci
ci	o	o

thus releasing other cytokines

ci	ci	ci
ci	a_d	ci
ci	ci	ci

Because they are present in large numbers they have a major role in the spread of cytokines inside the brain, causing the death of more and more neurons.

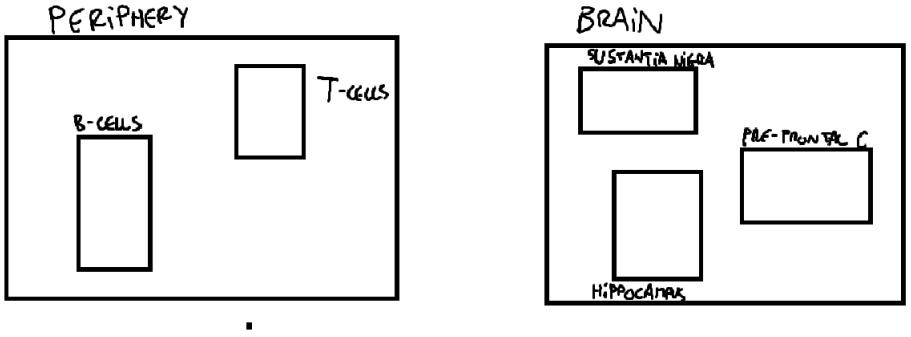
5.2.4 Antigens

The *antiges* are released by the neuron and their objective in the brain is to cross the blood brain barrier. This happens when the antigen reaches the proximity of the BBB and it's described by the use of the following semantics

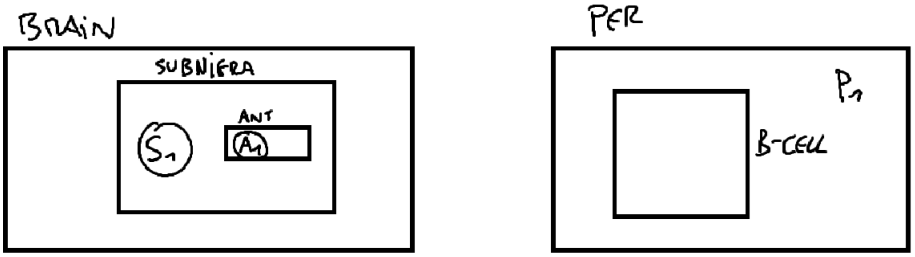
Trying to describe a communication

The communication we are trying to describe is the antigen that travels to the b cell

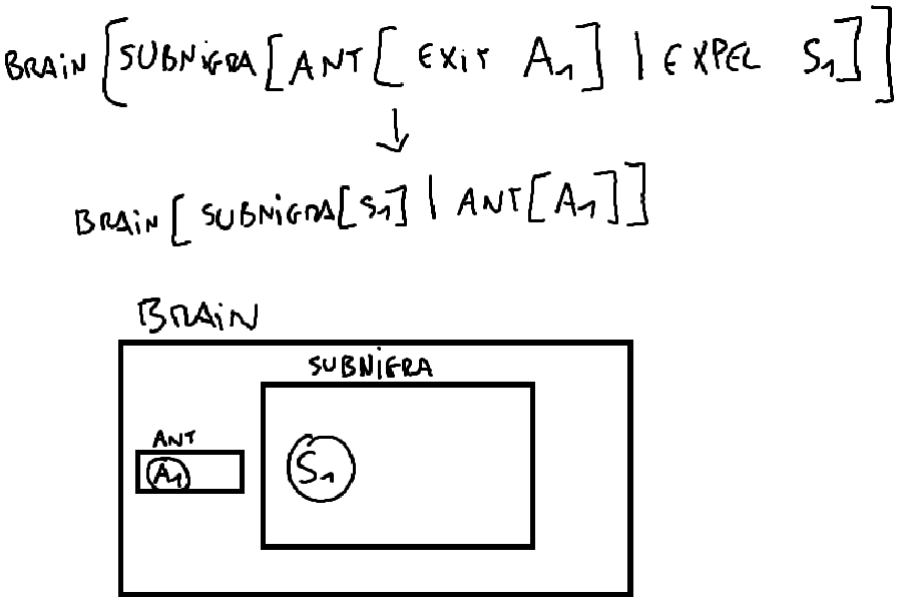
Starting point:



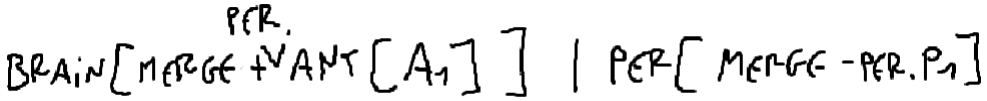
Focus on the environments that we want to study:



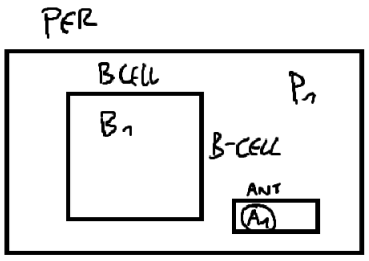
The antigen exits the substantia nigra:



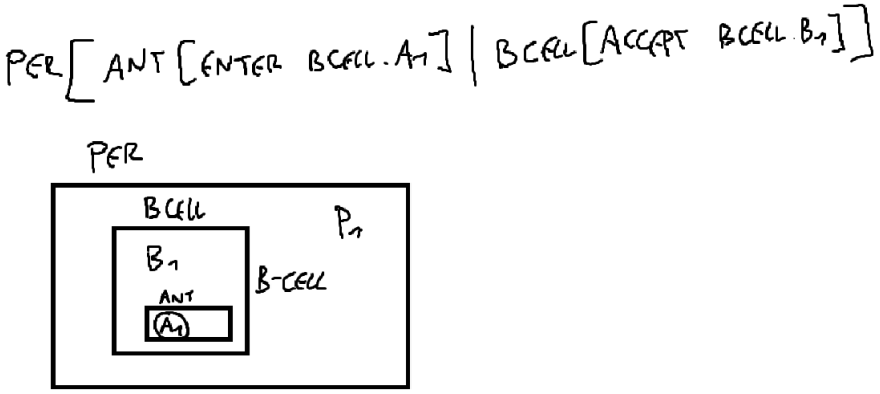
Using this



we exit the brain and enter the peripheral system



Finally, the antigen enters the T-Cell domain



This is an example of how an antigen can travel from the substantia nigra all the way to a T-Cell.

5.3 The peripheral immune system in the gut

5.3.1 T-Cells

T-Cells have two state: stable and active.

Stable state

In the stable state the T-Cells do not perform any action, but it can transition into the active state when it meets a *dopamine* cell or an *antigen*. So if this

o	o	o
o	Ts	an
o	o	o

or this

o	o	o
o	Ts	o
do	o	o

happens, then it turns into active

o	o	o
o	Ta	an
do	o	o

Active state

During the active state, the production rate of TH1s increases (we should really say that is starts). The rate is different depending on what activated it, since a *T-Cell* activated by an *antigen* will produce *TH1s* at an higher rate.

5.3.2 TH1s and TH2s

TH1s and *TH2s* are two cells produced by *T-Cells*. *TH1s* are pro-inflammatory, meaning that they help with the inflammation process, while *TH2s* are not. The role of *TH2s* is to keep the *TH1s* in the gut and to not make them travel across the BBB and through the brain.

TH1s, on the other hand, will try to reach the brain by travelling towards it. This will happen only when [1]

$$\frac{num(th1)}{num(th2)} > 1.$$

Once in the brain, it will move in order to find and infect *neurons*.

5.3.3 Levodopa

Levodopa is the agent that tries to slow the infection process of Parkinson. This agent tries to move towards the brain like the *TH1s*, but it can turn into dopamine at any time during the process [2]. The base probability of that happening is 10%, but thanks to *carbidopa* it can get lower.

Carbidopa

Carbidopa is a **parameter** of the simulation that is a percentage that is able to reduce the probability of *levodopa* turning into *dopamine* with this formula

$$new\_perc = old\_perc * carbidopa\_effectiveness/100.$$

Once the *levodopa* is in the brain, it helps the *neurons* mantain the stable state and it's also considered a parameter as we've said in Section 3.2

Chapter 6

Implementation in Repast4Py

In the realm of complex systems, where interactions between autonomous entities govern behavior, Multi-Agent Systems (MAS) stand as a powerful paradigm. MAS models the dynamics of various agents, each possessing individual goals, knowledge, and capabilities, interacting within a shared environment. These systems find applications across diverse domains, from social sciences to robotics, economics to epidemiology, offering insights into emergent phenomena and facilitating decision-making in complex scenarios.

Implementing MAS involves handling intricacies of agent behaviors, environmental dynamics, and system-level interactions. In this context, Repast4Py emerges as a robust toolkit, providing a comprehensive framework for developing MAS in Python. Repast4Py inherits its philosophy and design principles from the Repast Suite, a widely acclaimed platform for building agent-based models.

At its core, Repast4Py offers a set of tools and libraries tailored for creating, simulating, and analyzing complex agent systems. Leveraging Python's versatility and simplicity, Repast4Py empowers researchers and developers to translate conceptual models into executable simulations efficiently.

In this introductory guide, we delve into the fundamental concepts of multi-agent systems and explore how Repast4Py facilitates their implementation. We'll cover

the key components of Repast4Py, its core features, and demonstrate its usage through practical examples. By the end, readers will gain a solid understanding of both MAS principles and the practical application of Repast4Py for modeling and simulating complex systems.

Let's embark on this journey to harness the power of Repast4Py in building sophisticated multi-agent systems, unraveling the mysteries of emergent phenomena and complex interactions along the way.

6.1 Project structure

The project was structured in a very classical python way, with the following folder structure

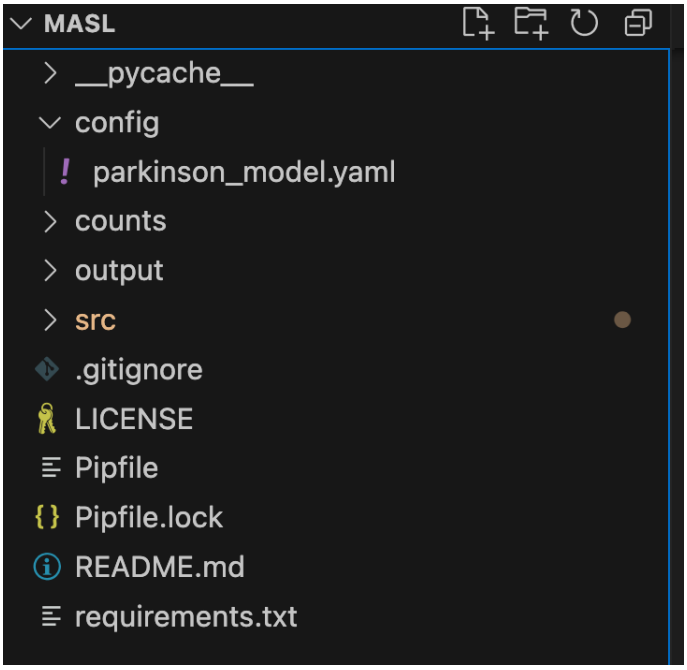


Figure 6.1: Directory Tree

Here, we have:

- The *config* folder: this contains the parameters of the simulation;
- The *counts* folder: this contains the csv files with the number of alpha and

dead neuron for each and every tick. This is used in to plot the graphs of the different simulations;

- the *output* folder: this contains the csv files that describe the movements of the agents. It contains csvs with the x and y coordinates for every tick of the simulation;
- the *src* folder: this contains the python files with the code of the GUI, the agents and the plotter;
- Finally, we have various files for versions control and and packages control.

To start the simulation, we use the following command

```
1    mpirun -n 4 python src/parkinson.py config/parkinson_model.yaml
```

Here, MPI is a python package that allows the user to develop distributed systems such as Repast4Py simulations. In this case, we start the simulation using 4 ranks. This means that Repast4Py will be distributed across 4 threads of the processor.

6.2 Model implementation

To implement a model as complex as this one, we must go through various steps. Here, we list and describe the most important ones.

6.2.1 Simulation initialization

Contexts definition

Firstly, we have to define contexts. In this simulation we have two, the brain context and the peripheral immune system context. We also

```
1    self.params = params
2    self.comm = comm
```

```
3       self.contexts["brain"] = ctx.SharedContext(comm)
4       self.contexts["peripheral"] = ctx.SharedContext(comm)
```

As we can see, we also define the MPI *comm* variable that allows the code to communicate across different ranks of the simulation. Then, using this comm variable, we define two *SharedContext*. Those contexts are Shared, meaning that a single context is distributed across multiple threads.

Scheduler setup

In order to the simulation to be able to run we have to initialize and setup the scheduler.

```
1       self.runner = schedule.init_schedule_runner(comm)
2       self.runner.schedule_repeating_event(1, 1, self.step)
3       self.runner.schedule_stop(params['stop.at'])
4       self.runner.schedule_end_event(self.at_end)
```

In this case we start from tick 1, we proceed 1 tick at a time and we stop the simulation using the *stop.at* parameter in the *.yaml* file.

Grid definition

The following code is used to define the *box* and the *grids* of the two environments.

```
1       box = space.BoundingBox(0, params['world.width'], 0, ...
            params['world.height'], 0, 0)
2
3       self.brain_grid = space.SharedGrid('grid', bounds=box,
4                                          borders=BorderType.Sticky,
5                                          occupancy=OccupancyType.Multiple,
6                                          buffer_size=2, comm=comm)
```

```
7
8       self.periphery_grid = space.SharedGrid('grid', bounds=box,
9                                              borders=BorderType.Sticky,
10                                             occupancy=OccupancyType.Multiple,
11                                             buffer_size=2, comm=comm)
12
13      self.contexts["brain"].add_projection(self.brain_grid)
14      self.contexts["peripheral"].add_projection(self.periphery_grid)
```

As we can see, the box is shared. This decision was made because the two grids are of the same dimensions, so making two different boxes would have been pointless. We then use multiple parameters to define the grids, such as:

- *bounds*: defines the perimeter of the grid using the box;
- *borders*: the type of border. In this case *BorderType.Sticky* means that the agents will stick to the border if they try to go over it;
- *occupancy*: Defines if multiple agents can occupy the same coordinate. In this case, using the Multiple parameter, we are saying that more than one agent can exist in the same coordinates;
- *buffer_size*: the size of the buffer. Here, more info about the buffer can be found;
- *comm*: the MPI comm variable to be used.

Loggers definition

In this section we define and explain how the loggers work. The following is the code in which the loggers are first defined

```
1       self.neurons = Neurons()
2       # The names argument is not specified, so the Counts field names ...
            will be used as column headers.
```

```
3       loggers = logging.create_loggers(self.neurons, op=MPI.SUM, ...
            rank=self.rank)
4       # Create a logging.ReducingDataSet from the list of loggers. ...
            params['counts_file'] is the name of the file to log to.
5       self.data_set = logging.ReducingDataSet(loggers, self.comm, ...
            params['counts_file'])
6       self.brain_data_set = logging.TabularLogger(self.comm, ...
            params['brain_file'], ["tick", "agent_id", "agent_type", "x", ...
            "y", "rank"], delimiter=",")
7       self.periphery_data_set = logging.TabularLogger(self.comm, ...
            params['periphery_file'], ["tick", "agent_id", "agent_type", ...
            "x", "y", "rank"], delimiter=",")
8       self.neuron_status_data_set = logging.TabularLogger(self.comm, ...
            params['neuron_status_file'], ["tick", "agent_id", "is_alive", ...
            "is_alpha", "num_alpha", "num_misfolded", "alpha_ticks", ...
            "rank"], delimiter=",")
9       self.microglia_status_data_set = logging.TabularLogger(self.comm, ...
            params['microglia_status_file'], ["tick", "agent_id", ...
            "is_activated", "rank"], delimiter=",")
10      self.astrocyte_status_data_set = logging.TabularLogger(self.comm, ...
            params['astrocyte_status_file'], ["tick", "agent_id", ...
            "is_activated", "rank"], delimiter=",")
```

We have two different types of loggers

- *ReducingDataSet*: This type of logger uses MPI to aggregate data across the ranks of the simulation;
- *TabularLogger*: This type of logger logs a row in a csv file every tick of the simulation. This doesn't aggregate the ranks, se every rank is logged separately.

In our case, the *ReducingDataSet* is used to count neuron in either alpha or the dead state. To do so, we use a dataclass defined as follows

```
    @dataclass
    class Neurons:
        """Dataclass used by repast4py aggregate logging to record
        the number of Humans and Zombies after each tick.
        """
        num_deads: int = 0
        num_alphas: int = 0
```

while the other type logs the positions and the status of each agent during every tick of the simulation. The following is the code used to log those

```
def log_counts(self, tick):
    #num_agents = self.context.size([Neuron.TYPE])#, Cytokine.TYPE])
    #self.counts.zombies = num_agents[Cytokine.TYPE]
    #print(num_agents)

    num_deads = 0
    num_alphas = 0

    for ag in self.contexts["brain"].agents():
        pt = self.brain_grid.get_location(ag)
        saved = ag.save()
        self.brain_data_set.log_row(tick, saved[0][0], saved[0][1], ...
            pt.x, pt.y, self.rank)
        if saved[0][1] == Neuron.TYPE:
            if not saved[1]:
                num_deads += 1
            if saved[2]:
                num_alphas += 1
            self.neuron_status_data_set.log_row(tick, saved[0][0], ...
                saved[1], saved[2], saved[3], saved[4], saved[5], ...
                self.rank)
```

```
19          elif saved[0][1] == Astrocyte.TYPE:
20                  self.astrocyte_status_data_set.log_row(tick, ...
                        saved[0][0], saved[1], self.rank)
21          elif saved[0][1] == Microglia.TYPE:
22                  self.microglia_status_data_set.log_row(tick, ...
                        saved[0][0], saved[1], self.rank)
23
24      for ag in self.contexts["peripheral"].agents():
25          pt = self.periphery_grid.get_location(ag)
26          self.periphery_data_set.log_row(tick, ag.save()[0][0], ...
                ag.save()[0][1], pt.x, pt.y, self.rank)
27
28      self.neurons.num_deads = num_deads
29      self.neurons.num_alphas = num_alphas
30      self.data_set.log(tick)
```

We use the save method to retrieve the parameters of the agents classes.

Environment initialization

Since we don't want more than 1 agent to spawn in the same grid space, two lists to keep track of the agents positions are initialized. We also initialize the seed of the random function so that each rank has a different one.

```
1       self.world_size = comm.Get_size()
2       self.occupied_brain_coords = []
3       self.occupied_periphery_coords = []
4       random.init(self.rank)
```

Then, we spawn the starting agents

```
1   # Adding Neurons to the environment
```

```
2  self.setup(Neuron.TYPE, params, "neuron.perc", "BRAIN")
3  self.num_neurons = int((params['world.width'] * ...
       params['world.height']) * params["neuron.perc"] / 100)
4
5  # Adding Microglia to the environment
6  self.setup(Microglia.TYPE, params, "microglia.perc", "BRAIN")
7
8  # Adding Astrocytes to the environment
9  self.setup(Astrocyte.TYPE, params, "astrocyte.perc", "BRAIN")
10
11 # Adding TCells to the environment
12 self.setup(TCell.TYPE, params, "tcells.perc", "PERIPHERY")
13
14 # Adding TH1s to the environment
15 self.setup(TH1.TYPE, params, "th1.perc", "PERIPHERY")
16
17 # Adding TH2s to the environment
18 self.setup(TH2.TYPE, params, "th2.perc", "PERIPHERY")
19
20 if self.rank == 1 or self.rank == 3:
21     # Adding Levodopa to the environment
22     self.setup(Levodopa.TYPE, params, "levodopa.perc", "PERIPHERY")
```

To spawn the agents we use the *setup* function that takes as parameters

- The agent type;
- The parameters of the simulation;
- The percentage of the grid to be populated with that particular type of agent;
- The environment in which the agents need to be spawned.

For the Levodopa, we make it spawn only in the 1st or 3rd rank that is equals to the bottom of the periphery environment. This decision was taken in order to

better simulate the movement of the Levodopa towards the brain.

The *setup* function works as follows:

```
def setup(self, agent_type, params, param, env):
    total_count = int((params['world.width'] * ...
        params['world.height']) * params[param] / 100)
    pp_count = int(total_count / self.world_size)
    if self.rank < total_count % self.world_size:
        pp_count += 1
    self.add_agents(pp_count, agent_type, env)
```

This function balances the agents across the ranks, so that each rank has the same number of agents. It also uses the *add_agents* function to spawn them, that is defined as follows

```
def add_agents(self, pp_count, type, context_id):
    local_bounds = self.brain_grid.get_local_bounds() if context_id ...
        == "BRAIN" else self.periphery_grid.get_local_bounds()
    ag = None
    for _ in range(pp_count):
        if type == Neuron.TYPE:
            ag = Neuron(((self.rank + 1) * 10000000) + ...
                self.id_counter, self.rank)
        elif type == Microglia.TYPE:
            ag = Microglia(((self.rank + 1) * 10000000) + ...
                self.id_counter, self.rank)
        elif type == Astrocyte.TYPE:
            ag = Astrocyte(((self.rank + 1) * 10000000) + ...
                self.id_counter, self.rank)
        elif type == TCell.TYPE:
            ag = TCell(((self.rank + 1) * 10000000) + ...
```

```
                self.id_counter, self.rank)
13          elif type == TH1.TYPE:
14              ag = TH1(((self.rank + 1) * 10000000) + self.id_counter, ...
                    self.rank)
15          elif type == TH2.TYPE:
16              ag = TH2(((self.rank + 1) * 10000000) + self.id_counter, ...
                    self.rank)
17          elif type == Levodopa.TYPE:
18              ag = Levodopa(((self.rank + 1) * 10000000) + ...
                    self.id_counter, self.rank, self.get_carbidopa_perc())
19
20          x, y = self.generate_coords(context_id, local_bounds.xmin, ...
                local_bounds.xmin + local_bounds.xextent, ...
                local_bounds.ymin, local_bounds.ymin + local_bounds.yextent)
21
22          if context_id == "BRAIN":
23              self.contexts["brain"].add(ag)
24              self.move(ag, x, y, "BRAIN")
25              self.occupied_brain_coords.append((x,y))
26          elif context_id == "PERIPHERY":
27              self.contexts["peripheral"].add(ag)
28              self.move(ag, x, y, "PERIPHERY")
29              self.occupied_periphery_coords.append((x,y))
30
31          self.id_counter += 1
```

This function creates the agent based on the given TYPE, then it calculates the ID using the following formula

$$agent\_id = ((rank + 1) * 10000000) + id\_counter,$$

where *id_counter* is a counter variable that increments each and every time an agent is added to the simulation. This function also generates the coordinates,

checks if they already exist (inside the *generate_coords* function, adds the agent to the environment and updates the coords list. The following is the *generate_coords* function

```
def generate_coords(self, context_id, x_min, x_max, y_min, y_max):

    x = random.default_rng.integers(x_min, x_max)
    y = random.default_rng.integers(y_min, y_max)
    if context_id == "BRAIN":
        while (x,y) in self.occupied_brain_coords:
            x = random.default_rng.integers(x_min, x_max)
            y = random.default_rng.integers(y_min, y_max)
    else:
        while (x,y) in self.occupied_periphery_coords:
            x = random.default_rng.integers(x_min, x_max)
            y = random.default_rng.integers(y_min, y_max)

    return x, y
```

6.2.2 Model run implementation

To run the model, we use a function called *step*. This function is a default of Repast4Py that is called automatically upon the start of the simulation. Let's dive into it.

Run initialization

The following four lines of code are needed to make the simulation work

```
tick = self.runner.schedule.tick
self.contexts["brain"].synchronize(restore_brain_agent)
self.contexts["peripheral"].synchronize(restore_periphery_agent)
```

```
4 self.log_counts(tick)
```

First, we obtain the tick of the simulation. Then we syncronize the agents using the *restore_brain_agent* and *restore_periphery_agent*. The following are the two restore functions:

```
1  agent_periphery_cache = {}
2  def restore_periphery_agent(agent_data: Tuple):
3      """Creates an agent from the specified agent_data.
4
5      This is used to re-create agents when they have moved from one ...
           MPI rank to another.
6      The tuple returned by the agent's save() method is moved between ...
           ranks, and restore_agent
7      is called for each tuple in order to create the agent on that ...
           rank. Here we also use
8      a cache to cache any agents already created on this rank, and ...
           only update their state
9      rather than creating from scratch.
10
11     Args:
12         agent_data: the data to create the agent from. This is the ...
               tuple returned from the agent's save() method
13                     where the first element is the agent id tuple, ...
                           and any remaining arguments encapsulate
14                     agent state.
15     """
16     uid = agent_data[0]
17     # 0 is id, 1 is type, 2 is rank
18     if uid[1] == TCell.TYPE:
19         if uid in agent_periphery_cache:
20             return agent_periphery_cache[uid]
```

```
21          else:
22              tc = TCell(uid[0], uid[2])
23              agent_periphery_cache[uid] = tc
24              return tc
25      elif uid[1] == Antigen.TYPE:
26          if uid in agent_periphery_cache:
27              return agent_periphery_cache[uid]
28          else:
29              ag = Antigen(uid[0], uid[2])
30              agent_periphery_cache[uid] = ag
31
32          ag.to_move = agent_data[1]
33          return ag
34      elif uid[1] == TH1.TYPE:
35          if uid in agent_periphery_cache:
36              return agent_periphery_cache[uid]
37          else:
38              ag = TH1(uid[0], uid[2])
39              agent_periphery_cache[uid] = ag
40              return ag
41      elif uid[1] == TH2.TYPE:
42          if uid in agent_periphery_cache:
43              return agent_periphery_cache[uid]
44          else:
45              ag = TH2(uid[0], uid[2])
46              agent_periphery_cache[uid] = ag
47              return ag
48      elif uid[1] == Levodopa.TYPE:
49          if uid in agent_periphery_cache:
50              return agent_periphery_cache[uid]
51          else:
52              ag = Levodopa(uid[0], uid[2], model.get_carbidopa_perc())
53              agent_periphery_cache[uid] = ag
```

```
54              return ag
55      elif uid[1] == Dopamine.TYPE:
56          if uid in agent_periphery_cache:
57              return agent_periphery_cache[uid]
58          else:
59              ag = Dopamine(uid[0], uid[2])
60              agent_periphery_cache[uid] = ag
61              return ag
```

```
1  agent_brain_cache = {}
2  def restore_brain_agent(agent_data: Tuple):
3      """Creates an agent from the specified agent_data.
4
5      This is used to re-create agents when they have moved from one ...
           MPI rank to another.
6      The tuple returned by the agent's save() method is moved between ...
           ranks, and restore_agent
7      is called for each tuple in order to create the agent on that ...
           rank. Here we also use
8      a cache to cache any agents already created on this rank, and ...
           only update their state
9      rather than creating from scratch.
10
11     Args:
12         agent_data: the data to create the agent from. This is the ...
               tuple returned from the agent's save() method
13                     where the first element is the agent id tuple, ...
                           and any remaining arguments encapsulate
14                     agent state.
15     """
16     uid = agent_data[0]
17     # 0 is id, 1 is type, 2 is rank
```

```
18      if uid[1] == Neuron.TYPE:
19          if uid in agent_brain_cache:
20              n = agent_brain_cache[uid]
21          else:
22              n = Neuron(uid[0], uid[2])
23              agent_brain_cache[uid] = n
24
25          # restore the agent state from the agent_data tuple
26          n.is_alive = agent_data[1]
27          n.is_alpha = agent_data[2]
28          n.num_alpha = agent_data[3]
29          n.num_misfolded = agent_data[4]
30          n.alpha_ticks = agent_data[5]
31          return n
32      elif uid[1] == Microglia.TYPE:
33          if uid in agent_brain_cache:
34              m = agent_brain_cache[uid]
35          else:
36              m = Microglia(uid[0], uid[2])
37              agent_brain_cache[uid] = m
38
39          m.is_activated = agent_data[1]
40          return m
41      elif uid[1] == Astrocyte.TYPE:
42          if uid in agent_brain_cache:
43              a = agent_brain_cache[uid]
44          else:
45              a = Astrocyte(uid[0], uid[2])
46              agent_brain_cache[uid] = a
47
48          a.is_activated = agent_data[1]
49          return a
50      elif uid[1] == Cytokine.TYPE:
```

```
51          if uid in agent_brain_cache:
52              return agent_brain_cache[uid]
53          else:
54              c = Cytokine(uid[0], uid[2])
55              agent_brain_cache[uid] = c
56              return c
57      elif uid[1] == Antigen.TYPE:
58          if uid in agent_brain_cache:
59              return agent_brain_cache[uid]
60          else:
61              an = Antigen(uid[0], uid[2])
62              agent_brain_cache[uid] = an
63
64          an.to_move = agent_data[1]
65          return an
66      elif uid[1] == Debris.TYPE:
67          if uid in agent_brain_cache:
68              return agent_brain_cache[uid]
69          else:
70              d = Debris(uid[0], uid[2])
71              agent_brain_cache[uid] = d
72              return d
73      elif uid[1] == TH1.TYPE:
74          if uid in agent_brain_cache:
75              return agent_brain_cache[uid]
76          else:
77              t = TH1(uid[0], uid[2])
78              agent_brain_cache[uid] = t
79
80          t.to_move = agent_data[1]
81          return t
82      elif uid[1] == Dopamine.TYPE:
83          if uid in agent_brain_cache:
```

```
84              return agent_brain_cache[uid]
85          else:
86              d = Dopamine(uid[0], uid[2])
87              agent_brain_cache[uid] = d
88              return d
89      elif uid[1] == Levodopa.TYPE:
90          if uid in agent_brain_cache:
91              return agent_brain_cache[uid]
92          else:
93              l = Levodopa(uid[0], uid[2])
94              agent_brain_cache[uid] = l
95              return l
```

As we can see, if the agents have some parameters we have to restore them. This is not always the case.

Going back to the run, we call the logging function for the current tick.

Blood brain barrier implementation

The first thing we check is the presence of agents in the vicinity of the BBB.

This is the code for antigens:

```
1 for ag in self.contexts["brain"].agents(Antigen.TYPE):
2     pt = self.brain_grid.get_location(ag)
3     if pt.y == y_extent:
4         self.contexts["brain"].remove(ag)
5         self.BBB.retain(ag, pt, Antigen.TYPE)
```

The blood brain barrier meets the brain on the lower part of the grid. For this reason, we capture the agent if its y position is equals to the *y_extent* of the grid.

The following is the one for TH1s:

```
nums = self.contexts["peripheral"].size([TH1.TYPE, TH2.TYPE])
if (nums[TH1.TYPE] / nums[TH2.TYPE]) > 1:
    dict = list(self.contexts["peripheral"].agents(TH1.TYPE)).copy()
    for ag in dict:
        if ag.uid[1] == TH1.TYPE:
            pt = self.periphery_grid.get_location(ag)
            # print(ag.save(), pt, self.rank)
            if pt.y == 0:
                    self.contexts["peripheral"].remove(ag)
                    self.BBB.retain(ag, pt, TH1.TYPE)
            else:
                    self.move(ag, pt.x, pt.y - ...
                        int(self.params["world.height"]/20), "PERIPHERY")
```

Since TH1s start to move if and only if the following condition is met

$$\frac{num(TH1)}{num(TH2)} > 1$$

we execute the code snippet only in this particural case. We then make a copy of the agents list. This is done because the TH1s are countinously generated and the list may change during the execution, thus causing the program to fail. In this case, the BBB meets the peripheral immune system for $y = 0$.

The last one is for the Levodopa:

```
to_turn = []
dict2 = list(self.contexts["peripheral"].agents(Levodopa.TYPE)).copy()
for ag in dict2:
    pt = self.periphery_grid.get_location(ag)
    turn = ag.step(model, pt, self.params["world.height"])
```

```
6       if turn:
7           to_turn.append(ag)
8       elif pt.y == 0:
9           self.remove_agent(ag, "PERIPHERY")
10          self.BBB.retain(ag, pt, Levodopa.TYPE)
11          self.brain_dopamine += 2.5
12
13
14  for agent in to_turn:
15      pt = self.periphery_grid.get_location(agent)
16      self.remove_agent(agent, "PERIPHERY")
17      self.spawn_dopamine(pt)
```

In this case this is a two step process, since the Levodopa can turn into dopamine inside the peripheral immune system. Also, when levodopa is able to cross the BBB it turns into dopamine automatically. Hence, we increment a variable that represent the effectiveness of the dopamine in the brain. The higher, the more effective.

We also make the agents spawn in the correct environment using this code snippet

```
1  to_release = self.BBB.release(Antigen.TYPE)
2  for item in to_release:
3      antigen, pt = item
4      self.contexts["peripheral"].add(antigen)
5      self.move(antigen, pt.x, 0, "PERIPHERY")
6
7  to_release = self.BBB.release(TH1.TYPE)
8  for item in to_release:
9      th1, pt = item
10     self.contexts["brain"].add(th1)
11     self.move(th1, pt.x, 39, "BRAIN")
```

Finally, this is the actual implementation of the interface

```
class BloodBrainBarrier():

    def __init__(self):
        self.retained_antigens = []
        self.retained_th1s = []
        self.retained_levodopa = []

    def retain(self, ag, pt, type):
        if type == Antigen.TYPE:
            self.retained_antigens.append((ag, pt))
        elif type == TH1.TYPE:
            self.retained_th1s.append((ag, pt))
        elif type == Levodopa.TYPE:
            self.retained_levodopa.append((ag, pt))

    def release(self, agent_type: int):
        to_release = []
        if agent_type == Antigen.TYPE:
            to_release = self.retained_antigens.copy()
            self.retained_antigens = []
        elif agent_type == TH1.TYPE:
            to_release = self.retained_th1s.copy()
            self.retained_th1s = []
        elif agent_type == Levodopa.TYPE:
            to_release = self.retained_levodopa.copy()
            self.retained_levodopa = []

        return to_release
```

This is characterized by two functions

- *retain*: this function captures the agents and stores them in a list;
- *release*: this function does the opposite. It sends out the list of agents to be released and cleans the internal list.

6.3 Agents implementation

Every agent has been implemented in their own classes. The standard agent implementation requires the class to extend the *repast4py.core.Agent* superclass. Then, inside the *init* of the class we also need to call che superclass constructor like this

```
1 def __init__(self, a_id: int, rank: int):
2     super().__init__(id=a_id, type=Neuron.TYPE, rank=rank)
3     self.is_alive = True
4     self.is_alpha = False
5     self.num_alpha = random.default_rng.integers(800, 1500)
6     self.num_misfolded = 0
7     self.alpha_ticks = 0
```

The previous example is the *init* function of the *Neuron* class. As we can see, we both call the *super().__init__* and we initialize the parameters of the class. Also, every agent has a *TYPE* parameter that uniquely identifies them and a *save()* method defined as follows:

```
1 def save(self) -> Tuple:
2     """Saves the state of this Neuron as a Tuple.
3     Used to move this Neuron from one MPI rank to another.
4     Returns:
5         The saved state of this Neuron.
6     """
```

```
7       return (self.uid, self.is_alive,
8               self.is_alpha, self.num_alpha,
9               self.num_misfolded, self.alpha_ticks)
```

This function returns the parameters and the *uid* of the class and it is called automatically in the *synchronize* method when trying to restore an agent.

6.3.1 Neuron implementation

The neuron ($TYPE = 0$) has five parameters

- *is_alive*: Boolean flag that check if the neuron is in the alive state. True if yes, False if not;
- *is_alpha*: Same as *is_alive*, but this variable checks if the neuron is in the alpha state;
- *num_alpha*: Counts the number of α-synucleins in the neuron;
- *num_misfolded*: Counts the number of misfolded of α-synucleins in the neuron;
- *alpha_ticks*: Counts the number of ticks the neuron has been in the alpha state.

Inside the *step* function, the implementation is divided in two cases. Firstly, we initialize the function with the following code

```
1       def step(self, model):
2           release_antigens = True
3           dopamine_effectiveness = float(model.getDopamineEffectiveness())
4           grid = model.brain_grid
5           pt = grid.get_location(self)
6           at = dpt(0, 0)
7           count_cytos, count_levos, count_deads, count_th1s = 0, 0, 0, 0
8           if self.is_alive:
```

```
9               nghs = model.ngh_finder.find(pt.x, pt.y)
10              for ngh in nghs:
11                  at._reset_from_array(ngh)
12                  for obj in grid.get_agents(at):
13                      if obj.uid[1] == Cytokine.TYPE:
14                          count_cytos += 1
15                      elif obj.uid[1] == Levodopa.TYPE:
16                          count_levos += 1
17                      elif obj.uid[1] == TH1.TYPE:
18                          count_th1s += 1
19                      elif self.is_alive and self.is_alpha == False and ...
                            obj.uid[1] == Neuron.TYPE and obj.is_alive == ...
                            False and obj.is_alpha:
20                          obj.is_alpha = False
21                          self.num_alpha += obj.num_alpha
22                          self.num_misfolded += obj.num_misfolded
23                          obj.num_alpha = 0
24                          obj.num_misfolded = 0
25                          count_deads += 1
```

Here, we either initialize useful variables and we count the number of ther different agents in the neighbourhood of the neuron. Then, we have an execution for when the neuron is not in the alpha state

```
1 if self.is_alpha == False:
2     self.alpha_ticks = 0
3
4     number_sup = 100 * (count_levos + 1)
5     if count_cytos ≥ 2 or count_th1s ≥ 2:
6         self.is_alive = False
7     elif count_levos ≤ 5 and (count_deads ≥ 1 or ...
          random.default_rng.integers(0, number_sup) > number_sup − 4):
```

```
8           self.is_alpha = True
```

We start by resetting the *alpha_ticks* variable, since in this case we are in the non alpha part of the code. Then, a variable named *number_sup* is initialized as follows

$$number\_sup + 100 * (count\_levos) + 1.$$

This is done so that the more levodopa is situated around the neuron, the higher is this number. In the ELIF condition, this is used to tell the neuron when it should enter the alpha state. We note that the neuron cannot enter the alpha state if the number of Levodopa is 6 or above. Finally, if the number of *cytokines* or *th1s* is at least 2 the neuron dies.

The following is the code for the alpha state

```
1  else:
2      if count_cytos ≥ 2 or (self.num_misfolded / self.num_alpha) > ...
           0.90 or count_th1s ≥ 2:
3          self.is_alive = False
4          self.is_alpha = False
5          return (release_antigens, pt)
6      elif (self.alpha_ticks > 12) and ((float(self.num_misfolded) / ...
           float(self.num_alpha) < 0.45) or ...
           (random.default_rng.integers(0, 100) < dopamine_effectiveness ...
           * 100)):
7          self.is_alpha = False
8
9      self.num_alpha += int(self.num_alpha * ...
           random.default_rng.integers(1, 4) / 100)
10
11     new_misfolded = int((self.num_alpha - self.num_misfolded) * ...
           random.default_rng.integers(35, 40) / 100)
```

```
12      if (self.num_misfolded + new_misfolded > self.num_alpha):
13          self.num_misfolded = self.num_alpha
14      else:
15          self.num_misfolded += new_misfolded
16
17      self.num_misfolded -= int(self.num_misfolded * ...
            random.default_rng.integers(0, 25) / 100)
18
19      self.alpha_ticks += 1
```

We start by killing the neuron is 2 or more *cytokines* or *th1s* are around it. We also kill the neuron if the proportion between misfolded α-synucleins and normal α-synucleins is greater than 0,9. In this case, we also tell the model to release *antigens* and *cellular debris*. We proceed by defining a way for the neuron to actually exit the alpha state by either having a low enough percentage of misfolded protein over the total (¡ 0.45) or if a random variable turns out to be lower then the *dopamine.effectiveness* parameter.

Then, we increase the number of alpha proteins, we generate new misfolded ones and, finally, we let the mythocondria clean a percentage of them. At the end, we update the *alpha_ticks* variable.

6.3.2 Microglia implementation

The microglia ($TYPE = 3$) only has 1 parameter, the *is_activated* variable. Its only function is to prevent the agent to activate multiple times after it has been activated once.

This is the code snippet that regulates it

```
1 if count_debris ≥ 2:
2     self.is_activated = True
3     return (release_cytokine, pt)
```

If the number of debris is greater or equals than 2 the microglia is activated, thus releasing cytokines.

6.3.3 Astrocyte implementation

The implementation ($TYPE = 4$) is nearly identical to the *microglia* one. Here, there also is the *is_activated* variable and the code of the agent is the following

```
1 if count_debris ≥ 2 or count_cytos ≥ 2:
2     self.is_activated = True
3     return (release_cytokine, pt)
```

Basically, we activate the agent if either the number of debris or the number of cytokines is greater or equals than 2.

6.3.4 T-Cell implementation

The *T-Cells* ($TYPE = 5$) has two parameter

- *is_activated*: Checks if the *T-Cell* has been activated by an *antigen*;
- *is_dopamine_activated*: Checks if the *T-Cell* has been activated by *dopamine*.

We determine when and how a *T-Cell* is activated via the following

```
1 for obj in grid.get_agents(at):
2     if obj.uid[1] == Antigen.TYPE:
3         self.is_activated = True
4         obj.num_encounters += 1
5     elif obj.uid[1] == Dopamine.TYPE:
6         self.is_dopamine_activated = True
```

Basically, if the *T-Cell* encounters an *antigen* or a *dopamine* is activated. In the case of the *antigen*, we can see that a variable referring to the *antigen* itself

is incremented. This happens because every antigen can infect at most 2 *T-Cells*.

The reason why two variables are needed is the following

```
1 if self.is_activated:
2     if random.default_rng.integers(0, 100) ≥ 90:
3         model.spawn_th1()
4 elif self.is_dopamine_activated:
5     if random.default_rng.integers(0, 100) ≥ 98:
6         model.spawn_th1()
```

If the *T-Cell* is activated by an *antigen*, the infection is more aggressive thus translating into an higher probability of spawning a *th1*.

6.3.5 TH1 implementation

The *TH1* ($TYPE = 9$) agent has only 1 parameter named *to_move*, that tells the agent if it should move or not. It's initially setup as false.

Behavior in the gut

The TH1 agent starts its life-cycle in the peripheral immune system in the gut. During his time here, it follows just one simple rule: When

$$\frac{num(th1s)}{num(th2s)} > 1$$

it starts to move towards the brain, thus setting the *to_move* variable as true. The code is the following

```
1 self.move(ag, pt.x, pt.y − int(self.params["world.height"]/20), ...
      "PERIPHERY")
```

The movement is defined as a velocity

$$v = \frac{\Delta x}{\Delta t},$$

where Δx is equals to the height of the world and Δt to the number of maximum ticks we want the agent to be able to cross the entire environment. If, for example, the world's height is 40, then a th1 will move with a speed of $2\frac{squares}{tick}$

Behavior in the brain

Once the *TH1* is actually able to cross the blood brain barrier and reach the brain, there is a change in behavior. Here, the agent starts to seek neurons in the brain in order to infect them. This is done using a random walker like follows

```
1  OFFSETS_X = np.array([-2, 2, 3, -3, 1, -1])
2  OFFSETS_Y = np.array([1, -4, -5])
3
4  if self.to_move:
5      # choose two elements from the OFFSET array
6      # to select the direction to walk in the
7      # x and y dimensions
8      x_dir = random.default_rng.choice(TH1.OFFSETS_X, size=1)
9      y_dir = random.default_rng.choice(TH1.OFFSETS_Y, size=1)
10     model.move(self, pt.x + x_dir[0], pt.y + y_dir[0], "BRAIN")
```

Basically, once it reaches the brain it moves randomly by choosing one number for each of the OFFSETS arrays.

6.3.6 Antigen implementation

The *antigens* ($TYPE = 6$) are released by the neuron upon their death and follow the exact same rules of the *TH1s* but inverted. Meaning that while in the brain they will try to reach the peripheral system and once there they will move randomly trying to infect *T-Cells*.

The only difference is that the *to_move* parameter is set as True and they have another parameter called *num_encounters* that is used like this

```
if self.num_encounters > 2:
    self.to_move = False
```

Basically, once two *T-Cell* are encountered they stop moving. This is a design choice that was made in order to avoid a single antigen infecting too many *T-Cells*.

6.3.7 Levodopa implementation

Levodopa ($TYPE = 2$) is another agent that is situated in the peripheral immune system and moves towards the brain. The difference is that, in this case, levodopa stops being an agent and turns into a parameter once in the brain. It has only one parameter called *carbidopa_sup* that is defined as

$$carbidopa\_sup = 90 + ((100 - 90) * carbidopa\_perc),$$

where 90 is the base probability of a *levodopa* agent turning into *dopamine* in the gut and *carbidopa_perc* is the percentage of effectiveness of the carbidopa (this is a parameter that we will discuss in the next section).

The following is the code of the step function

```
turn = True

if random.default_rng.integers(0, 100) ≥  self.carbidopa_sup:
    return turn
else:
    model.move(self, pt.x, pt.y — int(height/13), "PERIPHERY")
    return not turn
```

As we can see, at each step of the simulation the agent has a certain probability of turning into dopamine. If it doesn't turn, then it moves towards the brain. If it does turn, then the following code is called

```
1 pt = self.periphery_grid.get_location(agent)
2 self.remove_agent(agent, "PERIPHERY")
3 self.spawn_dopamine(pt)
```

in which the *levodopa* agent is removed from the grid and it's replaced by *dopamine*.

6.3.8 Dopamine, debris, TH2s and Cytokines implementation

Those are what we call **passive agents**, meaning that they just exist in the environment but do not move nor execute actions.

6.4 Parameters of the simulation

The simulation uses many parameters, which we describe here in detail.

- *random.seed*: This is the seed of the randomizer used in the simulation. It's essential in order for the simulation to be deterministic and reproducible;
- *stop.at*: This represents the tick number when the simulation is stopped;
- *neuron.perc*: The percentage of *neurons* in the substantia nigra;
- *microglia.perc*: The percentage of *microglia* in the substantia nigra;
- *astrocyte.perc*: The percentage of *astrocytes* in the substantia nigra;
- *levodopa.perc*: The percentage of *levodopa* in the gut;
- *tcells.perc*: The percentage of *t-cells* in the gut;
- *th2.perc*: The percentage of *th2s* in the gut;

- *th1.perc*: The percentage of *th1s* in the gut;
- *carbidopa.effectiveness*: The effectiveness of *carbidopa* in the gut. This is also a percentage and represents a reduction in the probability of the levodopa turning into dopamine in the gut before reaching the blood brain barrier. This was referred to in Section 6.3.7 with the *carbidopa_perc* name;
- *dopamine.effectiveness*: The effectiveness of *dopamine* in the brain. This is also a percentage that represent the probability of a neuron going back to the stable state. This was referred in Section 6.3.1;
- *world.width*: The width of the grid;
- *world.height*: The height of the grid.

Chapter 7

Conclusions

In this project, we embarked on a journey to explore the intricate dynamics of Parkinson's disease treatment through the lens of multi-agent systems (MAS), leveraging the simulation capabilities of Repast4Py. Our endeavor aimed to investigate the interplay between levodopa and carbidopa, two crucial medications in managing Parkinson's symptoms, within a simulated environment populated by agents representing various biological entities and physiological processes.

Throughout our exploration, several key insights have emerged, shedding light on the efficacy, challenges, and potential avenues for optimizing the administration of levodopa and carbidopa in Parkinson's disease management.

Understanding Parkinson's Disease Dynamics

Our simulation framework provided a valuable platform for comprehending the complex dynamics underlying Parkinson's disease progression. By modeling the interactions between dopaminergic neurons, neurotransmitter levels, and motor symptoms, we gained deeper insights into the nonlinear and emergent properties of the disease.

Impact of Levodopa and Carbidopa Administration

Through our simulations, we observed the significant impact of levodopa and carbidopa administration on mitigating motor symptoms associated with Parkinson's disease. Levodopa, serving as a precursor to dopamine, effectively replenished depleted neurotransmitter levels, leading to symptomatic relief and improved motor function. Furthermore, the addition of carbidopa enhanced the bioavailability of levodopa, optimizing its therapeutic efficacy and minimizing adverse effects.

Challenges and Trade-offs

Despite the therapeutic benefits of levodopa and carbidopa, our simulations also highlighted several challenges and trade-offs inherent in their administration. These included fluctuations in drug levels, the onset of dyskinesias, and the development of tolerance over time. Addressing these challenges necessitates a nuanced understanding of individual patient characteristics, personalized treatment regimens, and innovative therapeutic approaches.

Optimizing Treatment Strategies

Our simulations provided a platform for exploring and evaluating various treatment strategies aimed at optimizing the therapeutic outcomes for Parkinson's disease patients. By adjusting dosing schedules, exploring combination therapies, and incorporating feedback mechanisms, we identified potential avenues for enhancing treatment efficacy while minimizing adverse effects and long-term complications.

Future Directions and Implications

As we conclude our study, it becomes evident that the role of levodopa and carbidopa in Parkinson's disease management extends beyond mere symptomatic relief. These medications represent pivotal components in a broader therapeutic landscape, where personalized treatment strategies, interdisciplinary collaboration, and advanced technologies converge to improve patient outcomes and quality of life.

Moving forward, future research endeavors could delve deeper into the individualized modeling of patient-specific responses to levodopa and carbidopa, leveraging advancements in data-driven approaches, machine learning, and precision medicine. Additionally, the integration of real-time data streams, wearable sensors, and advanced monitoring technologies holds promise for enhancing the accuracy and predictive capabilities of MAS-based simulations in Parkinson's disease research and clinical practice.

In conclusion, our journey through the realms of multi-agent systems, Parkinson's disease dynamics, and therapeutic interventions using levodopa and carbidopa has provided valuable insights and opportunities for innovation. By harnessing the power of simulation, computational modeling, and interdisciplinary collaboration, we are poised to advance our understanding of Parkinson's disease and revolutionize its management in the years to come.

Through our collective efforts, we aspire to pave the way for a future where Parkinson's disease is not merely managed but truly understood, where each patient's journey is characterized by hope, resilience, and meaningful improvements in health and well-being.

As we bid farewell to this chapter of exploration, let us carry forward the lessons learned, the insights gained, and the aspirations ignited, knowing that our journey has only just begun.

Thank you.

Bibliography

[1] D. B. G. Burgaletto C, Munafò A. The immune system on the trail of alzheimer's disease. *Journal of Interesting Things*, Oct 2020.

[2] K. R. Gandhi. Levodopa (l-dopa). Jan 2024.

[3] A. Regev, E. M. Panina, W. Silverman, L. Cardelli, and E. Shapiro. Bioambients: an abstraction for biological compartments. *Theoretical Computer Science*, 325(1):141–167, 2004. Computational Systems Biology.

[4] C. G. Zenaro E, Piacentino G. The blood-brain barrier in alzheimer's disease. Jul 2016.