Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Scienze e Tecnologie dell'Informazione

Tesi di Laurea

Modellistica e simulazione numerica applicata allo studio della malattia parkinsoniana

Numerical modelling and simulation as a tool to study the parkinsonian disease

Samuele Carli

Relatore:
Prof. Luigi Brugnano

Correlatori:
Ing. Daniele Caligiore
Dr. Sebastian Luca D'Addario

Anno Accademico 2021-2022

Anyone who has never made a mistake
has never tried anything new.

Albert Einstein

An immense “thank you” to all who believed in me and kept me sane.
You know who you are.
I couldn’t have made it without you.

Contents

Outline

Parkinson's disease is a debilitating degenerative brain disorder that expresses with motor symptoms, such as slow movement, tremor, rigidity, imbalance, and a wide variety of non-motor complications like cognitive impairment, mental health disorders, sleep disorders, pain and other sensory disturbances. Symptoms usually begin gradually and worsen over time. As the disease progresses, the development of motor impairments such as dyskinesias (involuntary movements) and dystonias (painful involuntary muscle contractions) result in speech impairment, mobility limitations and consequently restrictions in many life areas. Many people with PD also develop dementia during the course of their disease. The World Health Organization reports that globally, disability and death due to PD are increasing faster than for any other neurological disorder. While the most prominent symptoms of PD occur when nerve cells in the basal ganglia (which produce dopamine) become impaired or die, there are also alterations in the sympathetic nervous system which cause a change in the production of noradrenaline and serotonine. This work is focused on studying some of the circuits that emerge from the direct and monoamine-mediated interactions of the brain areas in the basal ganglia and in the brain stem, and therefore the effects of such circuits on the overall behaviour of the brain as a system of interacting areas. Chapter 1 provides an overview of the brain: a brief description of how neurons work, how their electrical behaviour can be modeled and which are the most important complications and limitations of this approach, and motivates the following higher-level description of the brain as a composition of functional areas and their interactions. Chapter 2 introduces a dynamical system that models the interaction of the areas on which this work is focused, the data which is used as a reference, and how the model is used to reproduce the available data. In chapter 3 the descriptive and predictive performances of the model are analyzed; the compatibility of the model's predictions with literature data is assessed, and finally the model is applied to predict the expected effects of an hypothetical treatment.

Chapter 1

Neurons, brain and Parkinson's disease: an overview

> And here I am, finally standing on the shoulder of giants, and *of course* I forgot my glasses.
>
> Sam's diary

1.1 The neuron

A neuron is a cell that specializes in the processing of electrical signals. Figure 1.1 shows the structure of a typical neuron: the cell body (or *soma*), which contains the nucleus, extends several short *dendrites* in multiple directions where it receives inputs from other neurons or cells through *synapses* [1]. Dendrites are also branching out to form multiple connections; a single neuron can have as many as 10^4 *dendrite spines* which extend to form as many as 10^5 synapses [2], hence can receive and integrate signals from that many inputs at once. Dendrites are usually a few micrometers long [3] (Figure A.4 summarizes typical dendrite dimensions).
The cell body also originates a tubular *axon* that carries signals to other neurons. An axon can convey electrical signals to distances ranging from a few micrometers to a few meters; the electrical signal, called *action potential*, is a spike in the electrical potential around the axon due to a controlled exchange of ions through the axon's membrane [4] (shown in Figure 1.2). A single axon in vertebrate cortex can connect to more than 10^4 post-synaptic neurons.
Aside the neurons lives other cells, called *glial cells* [1], which in fact outnumber the neurons by a factor 2 to 10, differentiate in many kinds and cover

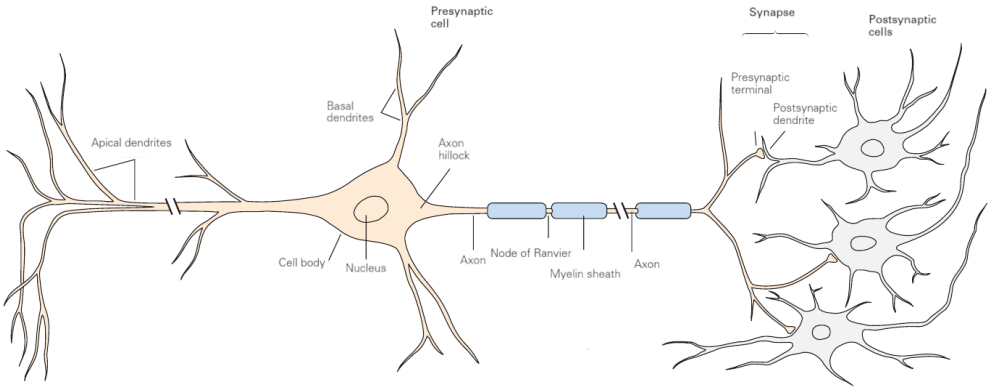


Figure 1.1: A neuron [1].

important roles in support of the neurons: other than functioning as interface between blood vessels and neurons for the delivery of nutrients (Figure 1.3), they form lipidic insulating sheaths called *myelin sheaths*; this insulation layer prevents the action potential spike from snowballing slowly down the axon via ionic exchange but forces it to propagate as electric potential just inside the axon and manifest only at the nodes (called *saltatory propagation*). Saltatory propagation is much faster, raising signal propagation speeds from 0.5–10 m/s for unmyelinated axons to up to 150 m/s [5]. The myelination of the axon also serves as electrical insulation, limiting the interaction between the potentials of axons which happen to be bundled together. Near its end the axon divides in branches that form synapses with other dendrites or somas of other neurons.
The neuron is therefore a typically asymmetric (*polarized*) cell, structured in a way such that electric potential spikes tend to flow from the dendrites and cell body to the synapses at the end of the axons. Depending on their shape, neurons can be classified as *unipolar*, *bipolar* or *multipolar* (Figure 1.4).
Most neurons, regardless of their type, have four distinct regions each of which fulfill a specific function (Figure 1.5): the *input* region gathers signals; the *integrative* region covers the crucial component of the neuron's computation role by deciding whether there will be an excitation state or not; the *conductive* region carries the electrical output signal to the *output* region which in turn converts the electrical signal in a chemical one for it to pass through the synapse.

1.1.1 Spikes and spike trains

Neuronal signals consist of short electrical pulses which can be measured by placing an electrode on the soma or axon of a neuron. This pulses, usually called *action potentials* or *spikes*, usually peak around 100 mV and last 1–2 ms. Figure 1.6 shows an example of a spike: if the combined post-synaptic

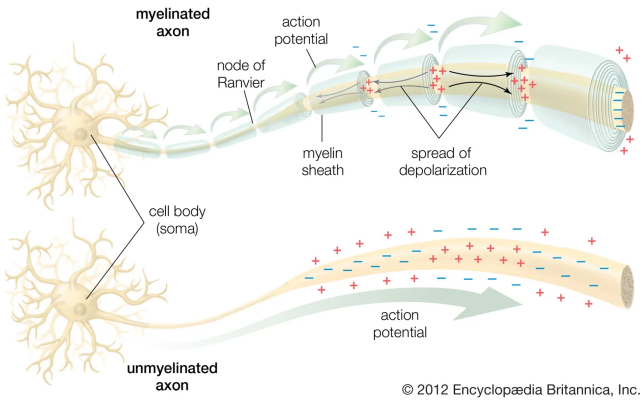


Figure 1.2: Action potential flow for myelinated and unmyelinated axons [1].

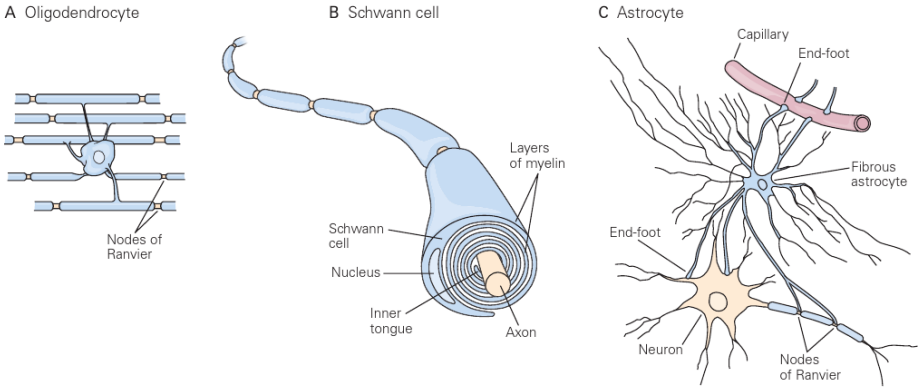


Figure 1.3: Glial cells [1].

stimulus entering a neuron surpasses its activation threshold, the neuron fires a spike signal which travels through the axon. The spike is followed by a short refractory period during which the resting voltage is lower (effectively raising the neuron's activation threshold) until the neuron's ionic balance is restored and the neuron is ready to fire again with the same input potential as before.
In fact, there is an *absolute refractory* period after a spike, during which it's impossible for the neuron to be excited again; this is followed by a *relative refractoriness* period during which exitation is difficult but not impossible.
Isolated spikes of a single neuron look alike and may get attenuated and deformed during their travel through the axon. It is therefore not the shape of the signal that carries the information, but the count and timing of spikes that matter [6].
A burst of spikes emitted by a neuron, with a maximum frequency dictated by its refractory period, is called a *spike train*. Specialized neurons have particular

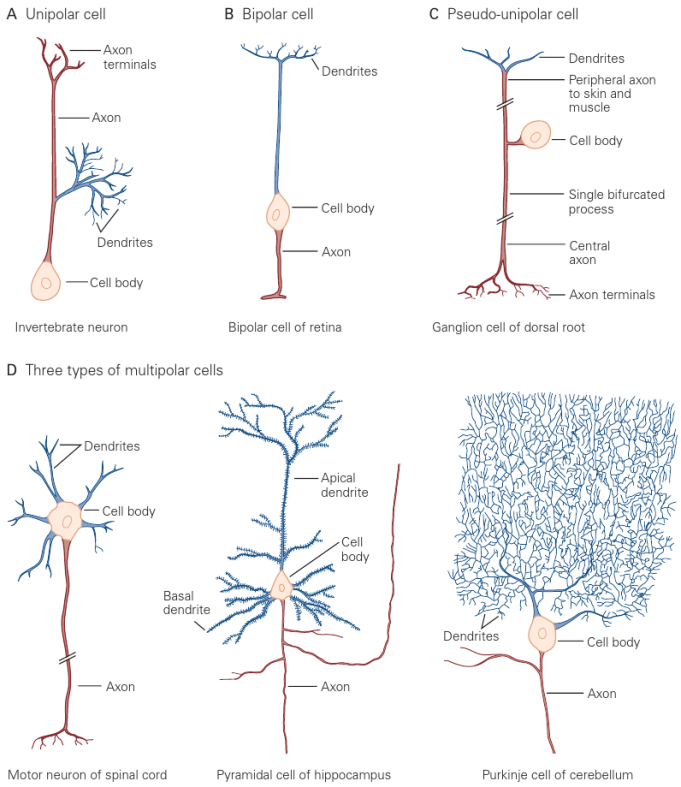


Figure 1.4: Unpolar, bipolar, multipolar neurons [1].

behaviors and emit different kind of spike trains; for example there are neurons that spike regularly with a frequency dependent on its inputs and neurons that spike regularly but emit high frequency burst when excited [7] (Figure 1.7).

1.1.2 The Synapse

The *synapse* is a structure that a neuron uses to communicate, electrically or chemically, with other neurons or other target cells (for example, muscle fibres). Electrical synapses are characterized by a pre- to post-synaptic cell membrane distance of 4 nm and obtain cytoplasmic continuity via numerous *gap-junction* channels that allow the action potential ionic current to flow from the pre-synaptic neuron to the post-synaptic one. Electrical synapses can transmit signals virtually without delay, and the transmission can be bidirectional. Electrical synapses allow for rapid, synchronous firing of interconnected cells. Chemical synapses instead maintain a greater junction distance of 20–40 nm. There is no cytoplasmic continuity; pre-synaptic electric signals are converted into chemical transmitters that are diffused in the *synaptic cleft* and captured by specific receptor channel on the post-synaptic cell. This synapses have a

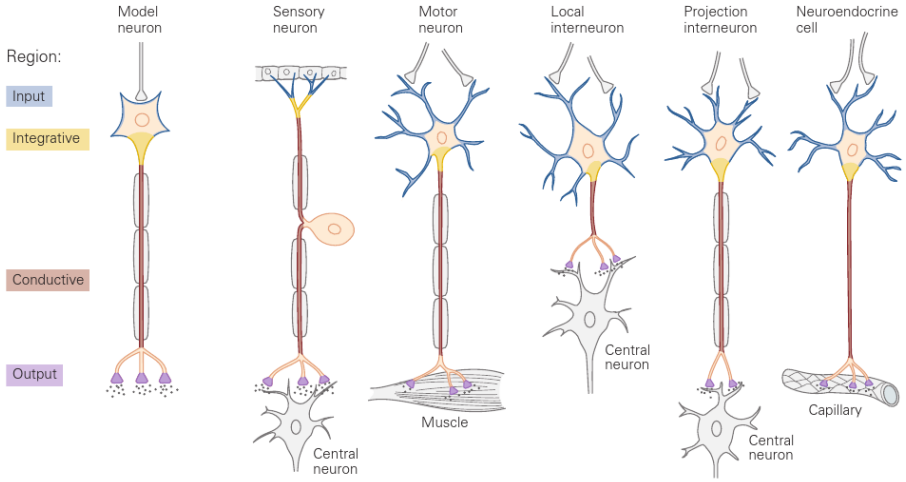


Figure 1.5: Functional regions of neurons [1].

significant signal transmission delay of 0.3–5 ms or longer, and are unidirectional. Chemical synapses have the important property of being able to amplify signals: through a chemical synapse, a small pre-synaptic nerve terminal which generates only a weak electrical current can depolarize a large post-synaptic cell. Chemical synapses can be classified according to the neurotransmitter released: glutamatergic (often excitatory), GABAergic (often inhibitory), cholinergic (e.g. vertebrate neuromuscular junction), and adrenergic (releasing norepinephrine). Because of the complexity of receptor signal transduction, chemical synapses can have complex effects on the post-synaptic cell [1, ch. 8], [9]. Both electrical and chemical synapses co-exist in the adult brain, although their ratio changes with age and regions of the brain [10].

1.1.3 Integrate and fire model

The basic behaviour of a neuron can be approximated using an *integrate and fire* model. This model is extremely simplified and neglects many aspects of neuronal dynamic, but can nonetheless be useful for understanding the underlying basing principles.[6].
Let the post-synaptic potential of neuron i at the time t be $u_i(t)$. At rest, we have $u_i(t) = u_{rest}$. Let also $\varepsilon_{ij}(t)$ be the the post-synaptic potential effect on neuron i of neuron j firing at $t = 0$. A typical spike can be modeled as an exponential rise and decay (Figure 1.8):

$$\varepsilon(t) = V(e^{-\lambda_{fall}t} - e^{-\lambda_{rise}t}), \quad t \geq 0, \tag{1.1.1}$$

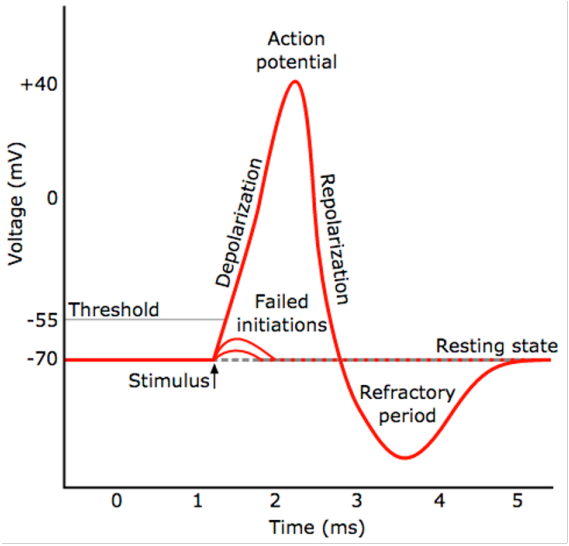


Figure 1.6: Example of neuronal membrane voltage spike [8].

where V scales the peak value and the time constants $\lambda_{rise}, \lambda_{fall}$ define the shape of the spike. We could therefore model the effects of neurons j firing at times t_j^f on the post-synaptic potential of neuron i as the sum (Figure 1.9):

$$u_i(t) = \sum_j \sum_f w_{ij}\varepsilon_{ij}(t - t_j^f) + u_{rest}, \tag{1.1.2}$$

where j iterates over the afferent neurons, f over the firing episodes of neuron j, and w_{ij} represents the strength of the synapse, hence scales the effect that neuron j has on neuron i.

However, this linear integration behaviour breaks down as soon as the membrane potential reaches a threshold value ϑ_i: as Figure 1.6 shows, once the threshold is passed, the neuron exhibits a spike-like excursion followed by an overcompensation that brings the potential lower than the neuron's typical resting potential u_{rest}.
The post-synaptic spike potential (PSP) $w_{ij}\varepsilon_{ij}(t)$ can be positive (excitatory) or negative (inhibitory); typically PSPs have amplitudes of about 1 mV, thus in reality about 20–50 spikes have to arrive to a neuron in a short time window for it to be excited.

Before discussing the non-linear firing behaviour, it is worth noticing that in first approximation equation (1.1.2), being a summation process, implies that the shape of the input spikes does not carry important information; a neuron is triggered when its input potential reaches its threshold ϑ_i from below, regardless of the input signal shape. Therefore, in the context of modelling neural networks, where the focus is on the response of the network rather than on the

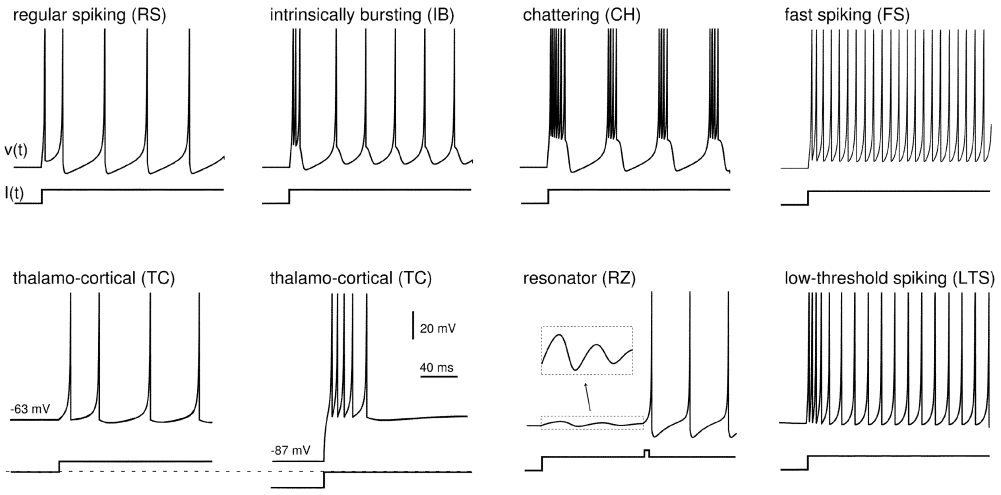


Figure 1.7: Some known types of spike patterns [7].

exact signals being carried around, it is useful to approximate the response of a neuron with a function that models its general behaviour, even if some nuances of its output signal are not entirely reproduced. The leaky current-base integrate and fire model is an example of such simplification.

Leaky current-based integrate and fire model Instead of taking into account the shape of the spikes, we can address only the effects induced in the receiving neuron due to changes in their frequency and amplitude. We can thus model a synapse as a time-dependent, potential-independent electrical current flow to the receiving neuron; the input of the receiving neuron i would then be:

$$I_i(t) = \sum_j I_j^{ps}(t), \tag{1.1.3}$$

where $I_j^{ps}(t)$ indicates the post-synaptic current neuron i is receiving from neuron j at time t.
Since the neuron is a cell enclosed by a dielectric membrane, we can model a neuron at rest as a capacitor which holds a charge $Q_i(t)$ (and has capacitance C). We also know that the cellular membrane is not an ideal dielectric, hence there is a leakage current that we can model with a resistor of value R in parallel with the capacitor. Finally, the neuron has a well-defined resting potential which can be modeled by adding a battery in series with the resistor to obtain the model circuit shown in Figure 1.10. We can therefore express the input current with the equation:

$$I_i(t) = I_R(t) + I_C(t) = \frac{u_i(t) - u_{rest}}{R} + \frac{dQ(t)}{dt} = \frac{u_i(t) - u_{rest}}{R} + C\frac{du_i(t)}{dt}. \tag{1.1.4}$$

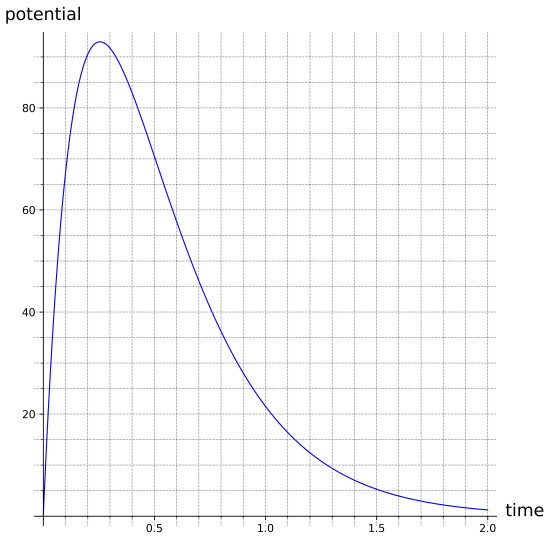


Figure 1.8: Exponential spike signal approximation example, $y = 500(e^{-3t} - e^{-5t})$.

Multiplying both sides of the equation by R we obtain:

$$RI_i(t) = u_i(t) - u_{rest} + RC\frac{du_i(t)}{dt} \quad \longrightarrow \quad \tau_i\frac{du_i(t)}{dt} = -(u_i(t) - u_{rest}) + RI_i(t), \tag{1.1.5}$$

where $\tau_i = RC$ is called the *membrane time constant* of the neuron.
When the neuron does not receive inputs ($I_i(t) = 0, t > 0$), (1.1.5) is a linear differential equation of the first order. In particular we have:

$$u_i'(t) + \frac{1}{\tau_i}u_i(t) = \frac{1}{\tau_i}u_{rest}, \tag{1.1.6}$$

which can be solved by multiplying both sides by the integrating factor $e^{\frac{1}{\tau_i}t}$:

$$e^{\frac{1}{\tau_i}t}\ u_i'(t) + \frac{1}{\tau_i}e^{\frac{1}{\tau_i}t}u_i(t) = e^{\frac{1}{\tau_i}t}\ \frac{1}{\tau_i}u_{rest} \quad \longrightarrow \quad \frac{d}{dt}e^{\frac{1}{\tau_i}t}\ u_i(t) = e^{\frac{1}{\tau_i}t}\ \frac{1}{\tau_i}u_{rest}, \tag{1.1.7}$$

whose solution can be obtained integrating both sides:

$$e^{\frac{1}{\tau_i}t}\ u_i(t) = \int e^{\frac{1}{\tau_i}t}\ \frac{1}{\tau_i}u_{rest}\ dt = \tau_i e^{\frac{1}{\tau_i}t}\ \frac{1}{\tau_i}u_{rest} + c \tag{1.1.8}$$

$$u_i(t) = u_{rest} + ce^{-\frac{1}{\tau_i}t}. \tag{1.1.9}$$

We can now impose an initial condition $u(0) = u_0$ and obtain the final solution:

$$u_i(t) = u_{rest} + (u_0 - u_{rest})\, e^{-\frac{1}{\tau_i}t}, \tag{1.1.10}$$

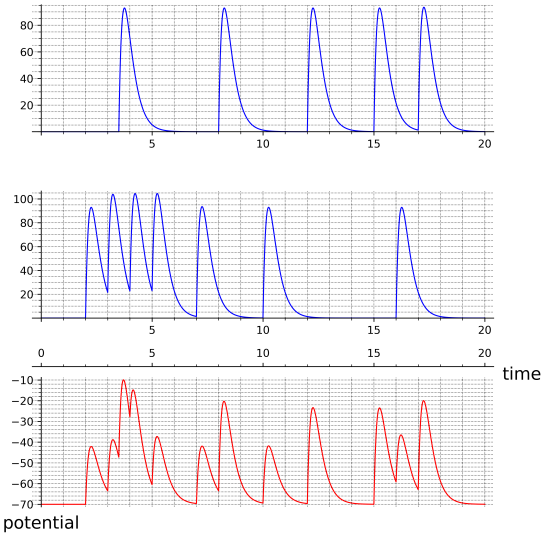


Figure 1.9: Example of application of equation (1.1.2) [p.16]: two input neurons (top and middle) spike randomly to contribute to the input potential of a neuron with a resting potential of -70 mV (bottom graph).

from which is evident that the solution has initial value $u_i(0) = u_0$ and tends to u_{rest} when $t \to \infty$, as we expected.
Similarly, assuming the neuron is excited with a constant current $I_i(t) = I_0, t > 0$, and was initially at rest ($u_0 = u_{rest}$), using the same technique we obtain the solution:

$$u_i(t) = u_{rest} + RI_0 - RI_0 e^{-\frac{1}{\tau_i}t}, \tag{1.1.11}$$

hence, were the input current to stay constant and the neuron not to fire, the membrane potential would asymptotically rise to $u_{rest} + RI_0$.

When the neuron's input current is not a constant but a time-dependent function $I_i(t)$, equation (1.1.5) [p.18] is a linear first-order differential equation:

$$\frac{du_i(t)}{dt} + \frac{1}{\tau_i}u_i(t) = \frac{u_{rest} + RI_i(t)}{\tau_i}. \tag{1.1.12}$$

Multiplying both sides by the integrating factor $e^{\int \frac{1}{\tau_i}dt}$ we obtain:

$$\frac{d}{dt}e^{\int \frac{1}{\tau_i}dt}u_i(t) = \frac{u_{rest} + RI_i(t)}{\tau_i}e^{\int \frac{1}{\tau_i}dt}, \tag{1.1.13}$$

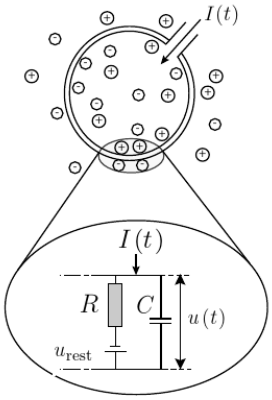


Figure 1.10: Input-current behavioural approximation of a neuron [6].

and therefore, integrating both sides:

$$u_i(t) = e^{-\int \frac{1}{\tau_i} dt} \int \frac{1}{\tau_i} u_{rest} e^{\int \frac{1}{\tau_i} dt} + \frac{R}{\tau_i} I_i(t) e^{\int \frac{t}{\tau_i} dt} \, dt \tag{1.1.14}$$

$$= u_{rest} + e^{-\int \frac{1}{\tau_i} dt} \, \frac{R}{\tau_i} \int I_i(t) e^{\int \frac{1}{\tau_i} dt} \, dt. \tag{1.1.15}$$

This formulation of $u_i(t)$ does not yet account for the ability of the neuron to fire. As described in Figure 1.6, a spike is initiated when the threshold potential is reached; the potential quickly rises and descends to a value lower than the rest threshold, where it slowly recovers from the refractory period before the neuron's sensitivity is completely restored. This more accurate spike shape can be modeled with a damped oscillation which is set to happen at the firing time t_f:

$$s(t_f, t) = \begin{cases} a e^{-b(t-t_f)} \sin(b(t-t_f)) & t \geq t_f \\ 0 & \text{otherwise,} \end{cases} \tag{1.1.16}$$

where parameter a scales the amplitude and b modulates the frequency response. Figure 1.11 shows an example of this function.
Assuming the firing times T_f of neuron i are known, equation (1.1.14) can be completed:

$$u_i(t) = u_{rest} + e^{-\int \frac{1}{\tau_i} dt} \, \frac{R}{\tau_i} \int I_i(t) e^{\int \frac{1}{\tau_i} dt} \, dt \; + \sum_{t_f \in T_f} s(t_f, t). \tag{1.1.17}$$

Figure 1.12 shows an example of the application of (1.1.17), obtained by numerically integrating its derivative (1.1.12) [p.19], to which we now must

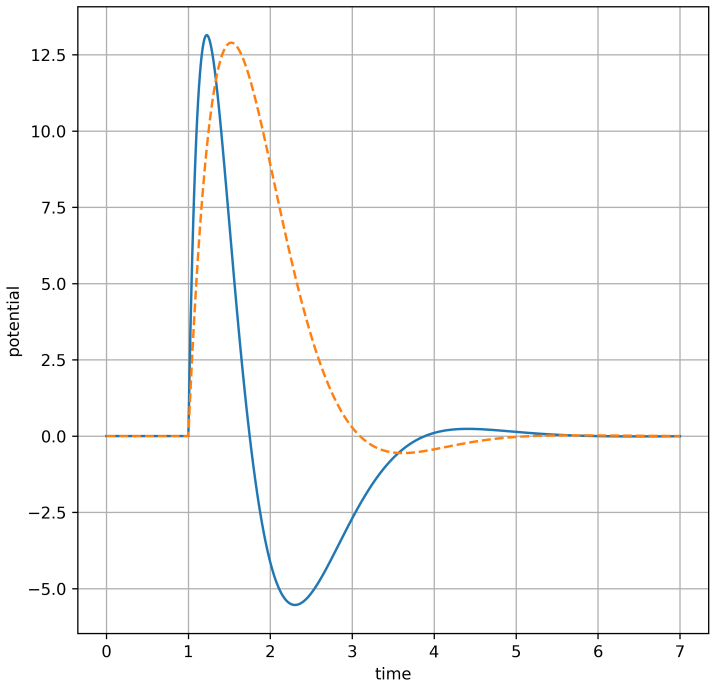


Figure 1.11: Example of damped oscillation: the orange dashed line shows the damped oscillation modeled by equation (1.1.16) [p.20] with a spike happening at $t_f = 1$. The blue solid line shows how adding the damped oscillation term (equation (1.1.17) [p.20]) affects the leaky neuron response; in this example it is the solution of: $\frac{du_i(t)}{dt} = -\frac{1}{\tau_i}u_i(t) + \frac{ds(t_f,t)}{dt}$

also add the corresponding derivatives of the spiking terms:

$$\frac{du_i(t)}{dt} = -\frac{1}{\tau_i}u_i(t) + \frac{u_{rest} + RI_i(t)}{\tau_i} + \sum_{t_f \in T_f} \frac{ds(t_f,t)}{dt}, \tag{1.1.18}$$

where

$$\frac{ds(t_f,t)}{dt} = \begin{cases} abe^{-b(t-t_f)}(\cos(b(t-t_f)) - \sin(b(t-t_f))) & t \geq t_f \\ 0 & \text{otherwise,} \end{cases} \tag{1.1.19}$$

More refined formulations of the above definitions of neuron's potentials behaviour, using both this linear differential equation approach and alternatively

linear filters, are described in detail in [6]. It should however be clear by now, as it is also indicated in the literature, that this approach presents several limitations: first and foremost, it is a highly simplified model that neglects many aspects of the behaviour of neurons and the underling physical and chemical phenomena. For example, the pre-synaptic input currents are most certainly not integrated linearly but the integration depends on the state of the post-synaptic neurons; furthermore, this model has no memory of previous spikes while it has been proven that spiking frequency and neuron sensitivity are related, and activation thresholds can change in time with a phenomenon called adaptation. Additionally, this particular analytical formulation requires external knowledge on the firing times, which should instead be determined solely by the neuron's potential.
However, this kind of leaky integrate and fire models, when completed by taking into account adaptation, bursting and inhibitory rebound, have been shown to be able to reliably predict spike times and firing patterns [6], and can be used to simulate large populations of connected neurons; they are therefore useful to understand the working principles of large neural networks, which, despite neglecting many aspects, is still an important step towards understanding how the whole brain really works.
This low-level approach to neuron simulation can be extended to also include chemical and physical phenomena and hence model all known aspects of a neuron's and a neural network's behaviour; for example the Blue Brain Project [11] successfully modeled a part of the neocortical tissue [12] using a supercomputer. However, exactly because of its complexity and associated computational costs, this kind of models may not be yet employable on a larger scale.
A complementary approach (which is the one employed in this work) is to model higher level phenomena (for example, the average activation frequency across many neurons of a particular brain region) instead; this latter approach may not ultimately yield results as accurate as a full simulation, but is nonetheless useful to understand and predict how complex systems behave.

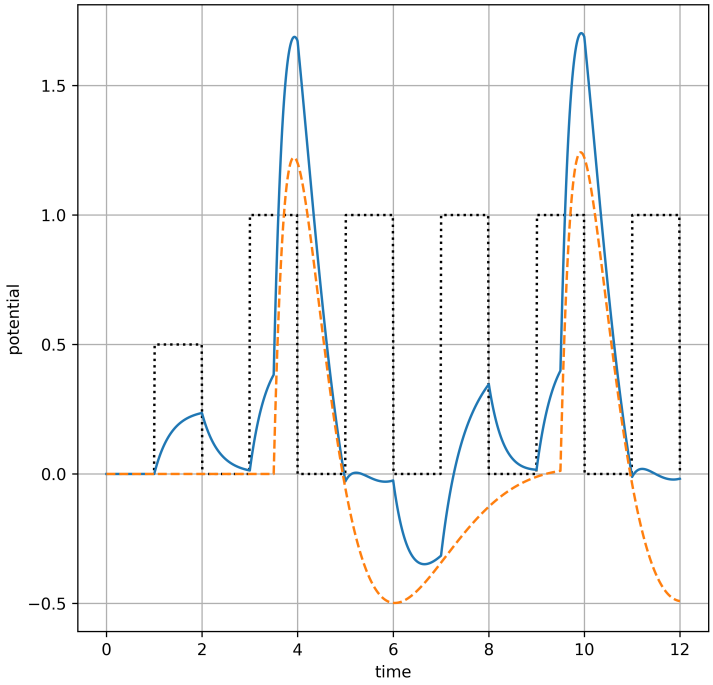


Figure 1.12: Example of neuron spikes happening at $t_1 = 3.5$ and $t_2 = 9.5$. The black dotted line represent an hypothetical square wave neuron input current $I(t)$. The first cycle of the current is not strong enough to drive the neuron's potential (solid blue line) past its activation threshold; when the input ceases, the leaky neuron discharges according to its time constant. The second cycle is strong enough to trigger a spike. The third and fourth current input cycles happen during the refractory period after the spike, and despite being identical to the second one in magnitude, they are not strong enough to trigger the neuron again. At last, the fifth cycle comes after the refractory period and the neuron is triggered once again. The orange dashed line shows what the behaviour of the potential would be if the neuron would spike naturally without any input current contribution.

1.2 Morphological regions

Without any presumption of being exhaustive, this section provides an overview of the most relevant structures that compose the brain. More detailed descriptions can be found in [1; 16; 17; 18].

There are several main parts that can be identified as composing the *Central Nervous System* (CNS) from the structural point of view:

The *Cerebrum* (Figure 1.13 A.7 and B) comprises of two hemispheres, each consisting of a wrinkled outer layer (*cerebral cortex*) in turn divided into four lobes and three internal structures: the *basal ganglia*, the *hippocampus* and the *amygdala* (Figure 1.17). Physically, the cerebrum comprises of *grey matter*, which consists mainly of neuronal cell bodies and glial cells, and *white matter*, which consist mainly of myelinated axons.

The *Diencephalon* comprises of four main structures: the *thalamus*, which processes most of the information reaching the cerebral cortex from the rest of the central nervous system, the *hypothalamus*, which regulates autonomic, endocrine, and visceral functions. the *epithalamus*, which participates in the regulation of the body's circadian rhythm, and the *subthalamus* which is involved in somatic motor functions (Figure 1.15, Figure 1.17). The diencephalon is attached to the *optic nerve*, an afferent sensory nerve responsible for vision and sight which runs from the eye through the optic canal in the skull.

The *Cerebellum* is a structure attached to the bottom of the brain. Like the cerebrum it has a cortical surface, but is structured as finely spaced parallel grooves instead of broad irregular convolutions; in fact, the cerebellar cortex is a tightly folded but continuous layer of tissue somewhat reminiscent of an accordion's bellow. This layer consists of several types of neurons regularly arranged. Almost all of the output from the cerebellar cortex passes through a set of small clusters of neurons called *deep nuclei* (Figure 1.16).

The *Midbrain* is the forward-most portion of the *brain stem*, effectively located in the middle of the brain. It is composed of multiple structures, most notably the *cerebral aqueduct* which is part of the ventricular system that circulates the *cerebrospinal fluid* (Figure 1.14).

The *Pons* is in the brain stem, between the midbrain and the medulla oblongata, and in front of the cerebellum. It includes direct *neural pathways* and *tracts* (bundles of axons) which connect the brain to the cerebellum and medulla, as well as tracts carry sensory information up to the thalamus (Figure 1.14). The *Medulla oblongata* is a long stem-like structure that follows the pons in the path from the cerebrum to the *spinal cord* which in turn connects the central nervous system (CNS) to the body (Figure 1.15).

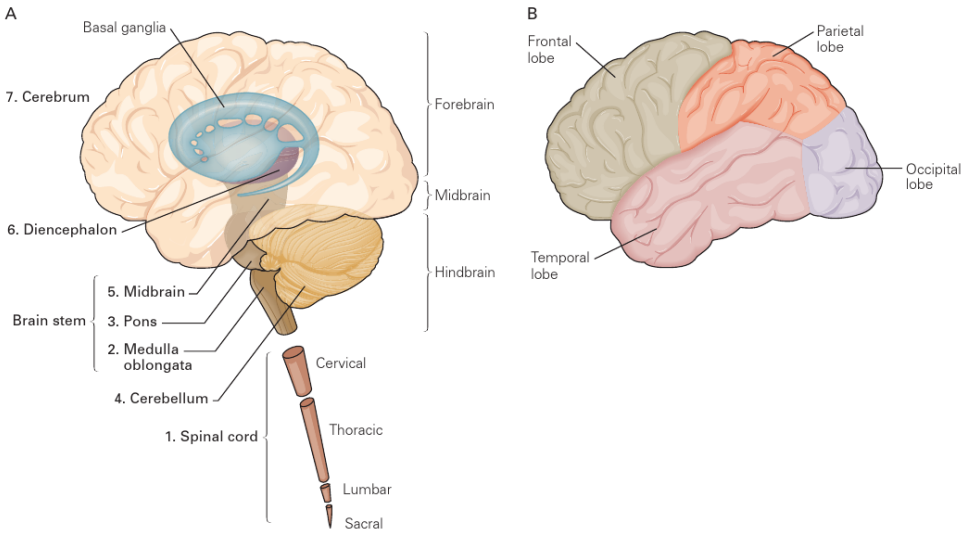


Figure 1.13: Brain structure: Lateral view and cortical lobes [1]

The *Medulla oblongata* is also part of the brain stem, and is responsible for autonomic functions, such as vomiting, sneezing, breathing, regulation of the heart rate and blood pressure (Figure 1.14).

Finally, the *Spinal cord* is a long structure composed of nervous tissue that connects the motor cortex to the body: receives and processes sensory information from the skin, joints, and muscles of the limbs and trunk and controls the contractions of muscles, and therefore the movement, of limbs and trunk. It also contains reflex arcs, which are neuronal pathways that can independently control reflexes without the explicit intervention of the motor cortex.

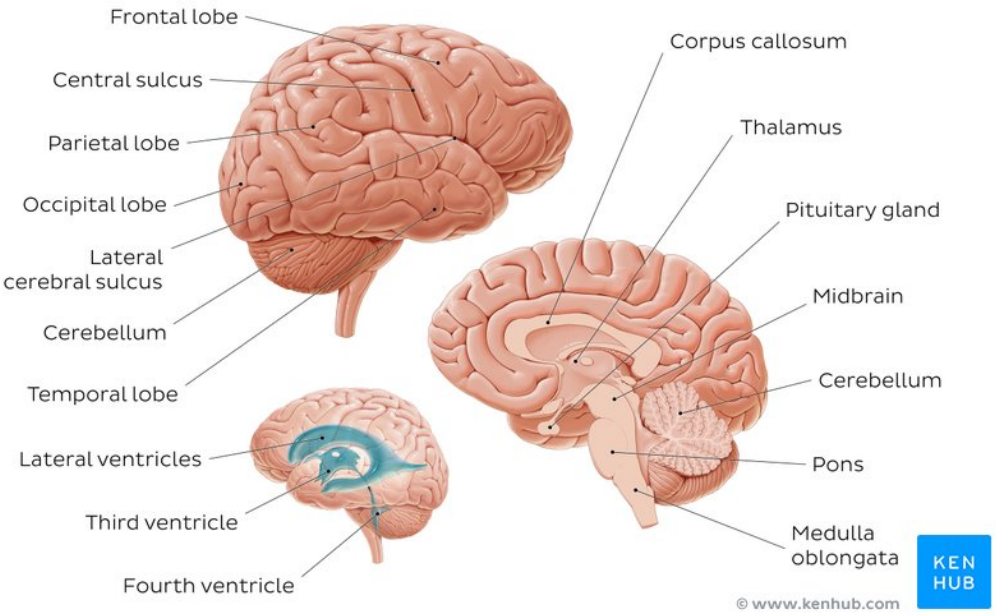


Figure 1.14: Brain structure: overview [13]

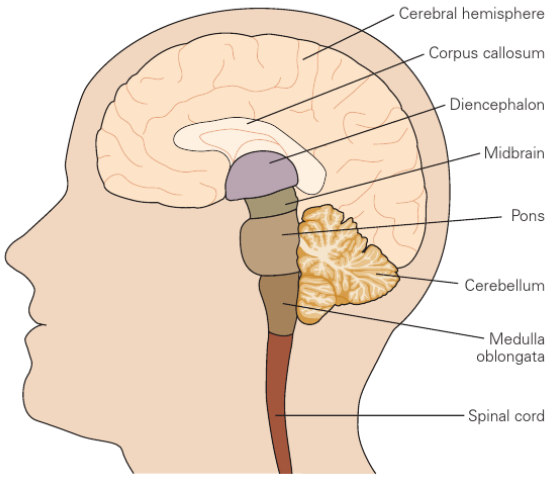


Figure 1.15: Brain structure: midbrain median section [1]

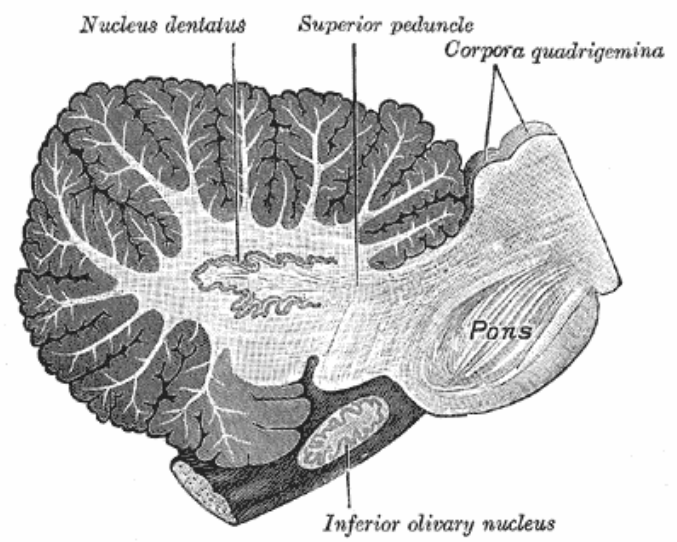


Figure 1.16: The cerebellum [14]

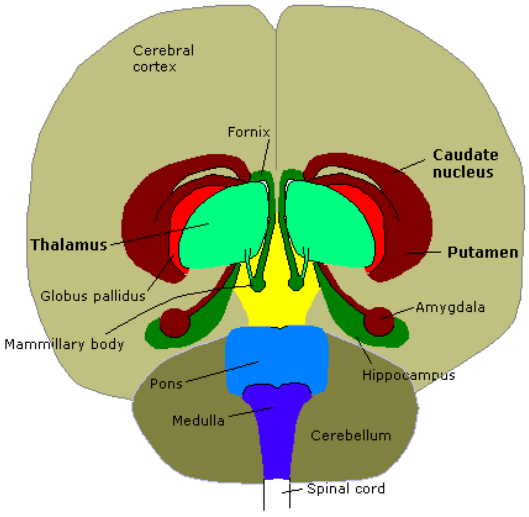


Figure 1.17: Brain structure: frontal section [15]

1.3 Functional regions

Areas of the brain can also be distinguished by the function they perform. Most functional regions and morphological structures coincide, although morphological structures that perform multiple (even opposing) functions do exist. Figure 1.18 illustrates some of the functions performed by the cerebral cortex, while also providing an example of morphological regions which perform multiple functions.
The exhaustive enunciation and description of all the functional areas is also an open research subject and is out of the scope of this work; detailed information about the current understanding of functional regions can be found in [1; 16; 17; 18]. Some of the functional areas located in the *basal ganglia* and the brain stem are of central importance for this work and we therefore provide a brief introduction to their functions.

The *basal ganglia* (Figure 1.13) are a group of subcortical nuclei located centrally in the brain. This nuclei are functionally distinct, and some of them are part of the neurotransmitter signal loops which are involved in the onset of Parkinson's disease and are the focus of this study. In particular:

The striatum (Str) is a critical component of the motor and reward systems. It coordinates multiple aspects of cognition, motor and action planning, motivation, reinforcement, reward and decision-making. The striatum is composed of neurons of two characteristic types: D1 and D2 which are both dopamine receptors but perform respectively excitatory and inhibitory functions. Because of the distinct behaviour and functions of the two neuron types, in this work we consider the striatum as two distinct areas, StrD1 and StrD2.

The globus pallidus (GP), located approximately at the center of the basal ganglia, is involved in the regulation of voluntary movement. When the globus pallidus is damaged or disregulated, it can cause movement disorders as it fails to exert its inhibitory action that normally balances the excitatory action of the cerebellum. The striatum has projections to the globus pallidus which exhibit inhibitory effects.

The substantia nigra is another basal ganglia structure located in the midbrain. Anatomical studies have found that the substantia nigra is in fact composed of two parts with very different connections and functions: the substantia nigra pars reticulata and the substantia nigra pars compacta (SNc). The latter serves mainly as a dopaminergic projection to basal ganglia structures. Parkinson's disease is characterized by the loss of dopaminergic neurons in the substantia nigra pars compacta.

The *dorsal raphe nucleus* (DRN) is located in the brain stem, and is the largest serotonergic nucleus to provide innervation to many other areas, including the

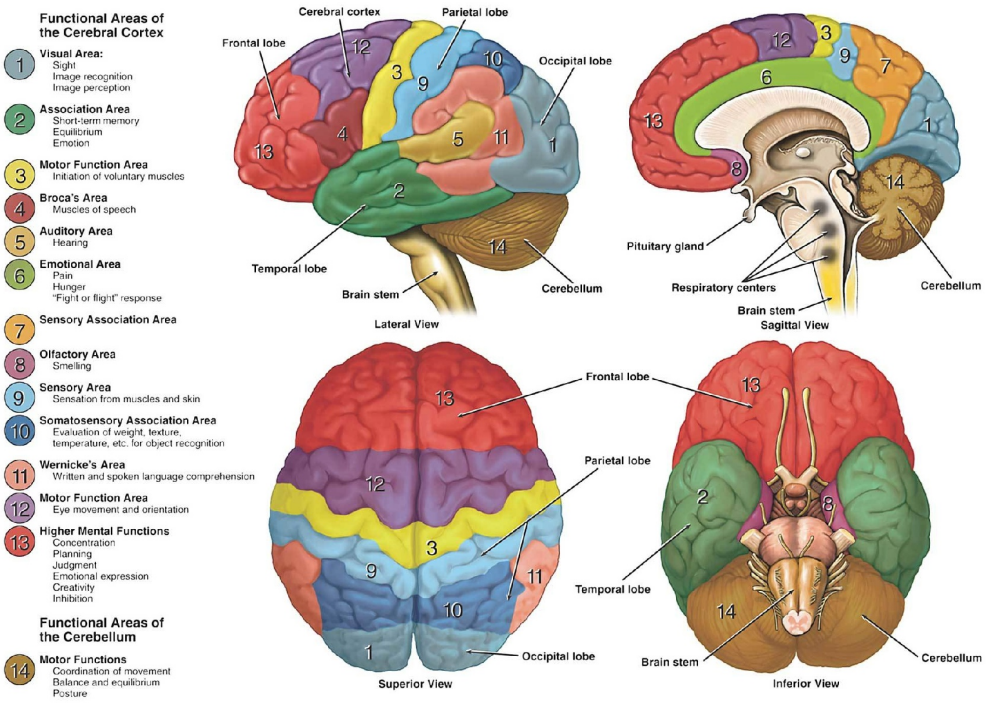


Figure 1.18: Brain Functional overview [19])

basal ganglia and the locus coeruleus.

The *locus coeruleus* (LC) is a nucleus located in the pons, and is involved with physiological responses to stress and panic. It is the area most involved in the production of noradrenaline, which is projected to many areas of the brain including the basal ganglia and the dorsal raphe nucleus.

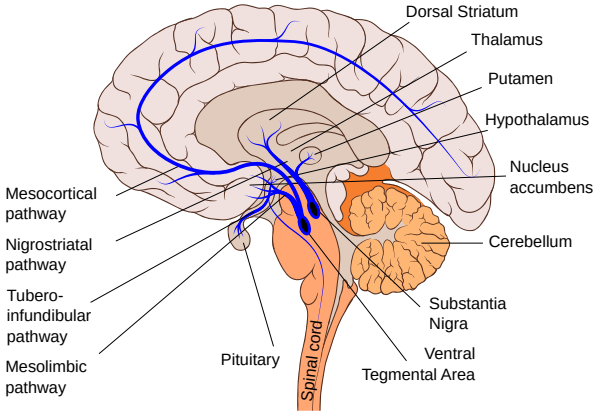


Figure 1.19: Dopamine pathways overview. [20]

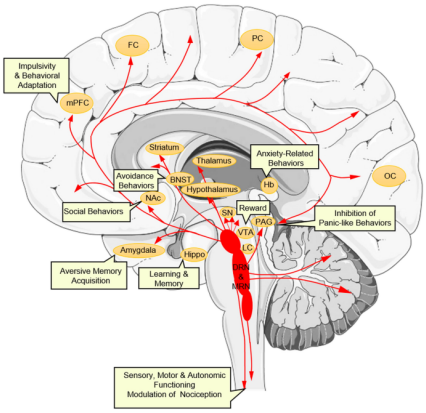


Figure 1.20: Serotonine pathways overview. [21]

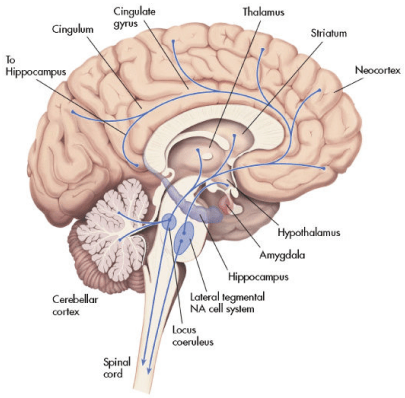


Figure 1.21: Noradrenaline pathways overview. [22]

1.4 Parkinson's disease

Parkinson's disease is a degenerative disorder of the central nervous system. While the most important symptoms involve the motor system with tremors, rigidity and slowness of movement, there are also cognitive and behavioural problems that develop with the progression of the disease like depression, anxiety, apathy and ultimately dementia. Another clinical aspect of the disease is a loss of automaticity of movement with a consequent increased need for voluntary control, which manifests as a growing difficulty in carrying out simultaneous movements. The disruption of well-learned movements is believed to reflect a dysfunction of the basal ganglia's role in procedural learning. The disease typically occurs in people over the age of 60, with a total incidence of about 1–2% slightly skewed towards the male population, which account for 6 cases every 10. PD's diagnosis is mainly based on symptoms; neuroimaging can be used to rule out other diseases. The salient feature of the disease is the degeneration of dopaminergic cells in the substantia nigra pars compacta that project dopamine to the striatum and other ganglia nuclei; this loss in dopamine is considered to be the cause of most of the movement abnormalities, since they respond to dopamine replacement therapy. Nonmotor symptoms (depression, anxiety etc.) instead do not respond to dopamine replacement therapy very well and must therefore be caused by other imbalances in the usual brain circuits equilibrium. According to recent studies these features may be caused by pathological changes affecting some lower brain stem nuclei such as the locus coeruleus and the dorsal raphe. Direct evidence for the reduction of dopaminergic inputs to the striatum comes from postmortem chemical analyses and from PET studies in humans, which demonstrate that the dopamine reduction is most severe in the caudal putamen, a portion of the striatum involved with the motor circuitry. Post-mortem studies have also assessed that motor signs of the disease occur when more than 70% of the striatal dopamine is lost, hence also demonstrating a significant capacity of the basal ganglia network to compensate for changes in dopamine levels. No cure for PD's is currently known, but there are medications, surgeries and physical treatments that may provide relief and improve a person's quality of life. Levodopa is a precursor of dopamine that can successfully increase dopamine production in the brain and consequently diminish motor symptoms. It is however not free from important side effects and long-term complications which may render the medication ineffective while leaving the patient dependent, where withdraw can also develop life-threatening side effects. Dopamine agonists are a family of drugs that can bind to dopamine receptors in the brain and have similar effects to levodopa, although they are usually less effective. They are usually preferred in the treatment of younger-onset of PD as they can provide a period of efficacy with milder side effects compared to levodopa, and may allow a better quality of life

before the treatment with levodopa becomes necessary. Surgery has also proved to be effective, in both deep brain stimulation and lesional form. Deep brain stimulation consists in the installation of electrodes in relevant brain areas to provide a controlled electrical stimulation, and is mainly used in subjects which do not respond to medications. Lesional surgery, which consist in the deliberate formation of lesions to suppress the overactivity of some areas, unlike deep brain stimulation, is not reversible and is therefore left as a last resort. As for the previous sections, this summary is just a very brief overview. More detailed descriptions can be found in [1; 23; 24; 25; 20] and many of the works cited in the bibliography.

1.5 Modelling based neurological research

The brain, especially the human one, is a really complex system with over $120 \cdot 10^9$ neurons which are estimated to form upwards of 10^{15} synapses. Moreover, neurons and synapses come both in hundreds of different kinds, and interact with a plethora of other cells in the brain, which contribution to the brain's activity and pathologies is not yet completely understood. The interactions between neurons can be of electrical, chemical, and electro-chemical nature; some aspects of this interactions can be monitored in live subjects, sometimes with greatly invasive procedures, usually with limited spacial and temporal resolution. Some other aspects (like fore example local or diffuse chemical concentrations) can only be analyzed post-mortem if at all, sometimes with limited precision and time-sensitivity due to the natural decay of the chemicals themselves. The challenge of acquiring an as-complete-as-possible picture of the state of the brain of a patient is a greatly important active field of research. Until that challenge is overcome, however, neurological research will have to deal with incomplete, imprecise and sometimes erroneous information; nevertheless the understanding of the brain, how it works, how pathologies develop and if and how they can be cured is an important endeavour for both mankind and for the unfortunate individuals who happens to develop such problems and deserve hope in a cure. In this context, modelling-based research is one of the most successful approaches in guiding progress in the field.
Building models is an approach that has several advantages. The most important is simplification: a model is by definition a simplified representation of a system. Nonetheless, a simple model that is able to correctly predict the phenomenon it represents within useful precision and is at the same time comprehensible has a great value, since it can unveil what are the most relevant and basic phenomena at play and what features are instead not as relevant. An understandable model can be a great base to develop more comprehensive theories about the system under study. A great example are Newton's laws

of motion: they are in fact a simplified model, since they don't take into account relativistic effects as instead relativistic mechanics models do. However, Newton's laws of motion are much easier to understand, and usefully predict with great precision the behaviour of moving bodies if relativistic speeds are not involved. It is therefore important to know what are the limits of applicability of the model.
A model can also provide isolation by deliberately ignoring some aspects of a phenomenon, and focus only on a particular aspect of a system. Such a model can still be useful to prove (or disprove) the relevance of the ignored aspects, and can still help in understanding many aspects of nature.
Many models can be made to compete for the representation of a phenomenon; in fact, this statement could almost be taken as the definition of the scientific method. It is nonetheless a very important aspect: a simple model may provide an understanding of the basic principles that regulate a system, while a complementary complex model, although maybe not really humanly comprehensible, could instead provide very accurate predictions without contradicting the simpler one.
Since the human brain cannot yet be measured without interfering for the most part, and we don't have control over many aspects of it for obvious ethical reasons, in the field of neurological research exploratory modeling is especially important. Numerical models can provide great insights on many aspects of the brain's functions. Low level models, like the previously mentioned blue brain project [11] can be used to understand the role of eletrical and chemical interactions between neurons and emergent behaviour of big biological neural networks; high level models like the one presented in this study can acquire insights on how entire brain areas interact and affect each other at the systemic level. Since common treatments such as drugs and electrical stimulation do in fact have measurable effects at the systemic levels, such high level models can be also useful in understanding the expected effects of such treatments. Numerical models do however need to be validated against real data. Model validation unfortunately require a degree of control over some aspects of the system being measured; for example, to model how dopamine levels change in the brain of parkinsonian subjects, it is necessary to measure both a healthy control group and a parkinsonian group. For this reason it also common practice to use biological models: some animals, for example rats, do exhibit an outstanding similarity with humans in many clinical aspects, including the brain. This animals, being complex biological systems, can be seen as a way more complete and complex model, over which it is possible to exert some control. For example, it is possible to breed specific strains of mice that naturally develop early onset parskinsonism, or to physically/chemically induce pathologies in a subset of the population, compare them with healthy subjects, and use the collected data to validate numerical models. Animal models do of course present great

ethical challenges as humans do, and should be used as sparingly as possible. For this reason, numerical models acquire even more importance: for example, many concurrent models can be developed for the same pathology to test as many theories as possible. The models themselves will define the data that is necessary to validate or dispute the model. An animal model population can then be used to measure all the necessary aspects at once, instead of having to use an entire animal model population to test each theory independently.

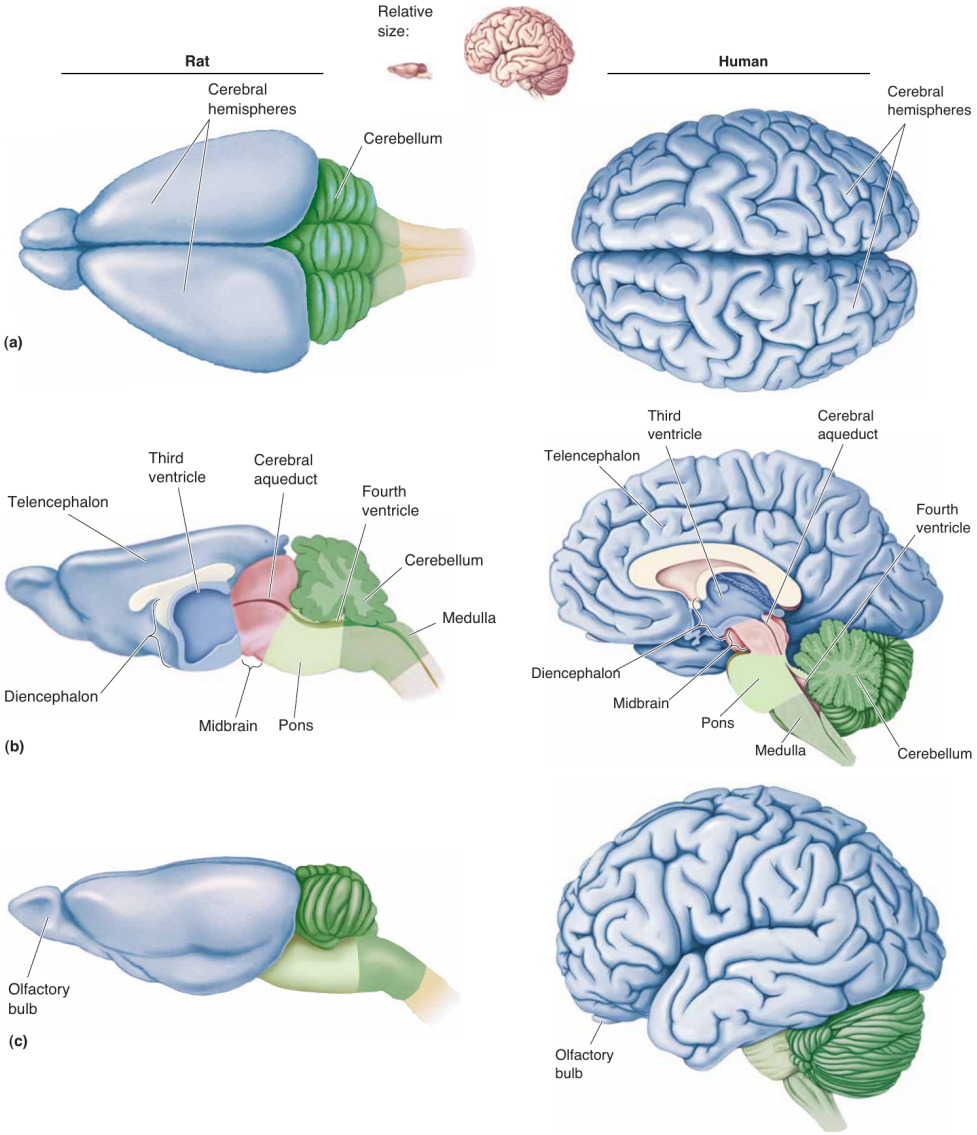


Figure 1.22: Comparison of rat and human brain [16]. Rats are often used as animal models to test theories about the human brain since they exhibit many biological resemblances to humans and are easy to breed.

Chapter 2

Model of brain areas interactions

> Science is what we understand well enough to explain to a computer; art is everything else.
>
> Donald E. Knuth

2.1 Background

Common theoretical and empirical approaches to studying Parkinson's Disease (PD) mainly focus on dysfunctions in dopamine-producing cells (DA) in the substantia nigra pars compacta (SNc). The substantia nigra in turn projects to the striatum (composed of two distinct parts, StrD1 and StrD2 [26]), which is the principal input gate of the basal ganglia, the subcortical nuclei which is critical to managing motor behaviour [27; 28].
A consistent reduction of striatal dopamine levels causes malfunctioning of the basal ganglia circuits that, in turn, may contribute to the emergence of different PD symptoms [29; 30; 31]. The main motor symptoms include: resting tremor, bradykinesia, rigidity, and freezing of gait [32; 33; 34]. Cognitive impairments might be evident at the time of diagnosis, even though they significantly manifest in the later stage of the disease progression [35; 36].
However, several recent studies suggest that psychiatric disorders, such as depression or anxiety, often develop several years before typical motor symptoms [37; 38]; in particular motivational system dysfunctions manifest early on [39; 40; 41].
Based on the evidence supporting dopaminergic malfunctioning, drug therapies for PD often aim at recovering dopamine levels [32]. While these approaches

seem to produce amelioration for most motor dysfunctions, they generate variable responsiveness for others (e.g., resting tremor [42; 43; 25]). In addition, long-term use of dopamine therapy may cause adverse effects like dystonic movements and impulse control disorders. The lack of consistency in dopamine-based therapies could be explained by considering that dysfunctional mechanisms leading to PD involve a network of areas and circuits interacting dynamically and influencing each other, rather than specific regions and molecular mechanisms working in isolation [29; 44; 45; 46; 33; 47]. In this respect, literature suggest that aside from the dopaminergic system, PD could also involve dysfunctions of noradrenergic and serotonergic neuronal populations [48; 49; 50; 51].
In PD, impairments of locus coeruleus (LC), the dorsal pontine nucleus that synthesizes noradrenaline (NE), begins before nigral pathology and appears to be more severe [52; 53; 54; 55]. Similarly, the dorsal raphe nucleus (DRN), which is critical for serotonine (5-HT) release, could show impairments earlier than the dopaminergic system and is involved with the development of both non-motor and motor symptoms [45; 56; 57; 58; 59; 60].
Starting from this system-level perspective, this work proposes a bio-constrained computational model that, for the first time, explicitly investigates the neural mechanisms underlying interactions between dopamine, noradrenaline, and serotonine in PD. The model is able to reproduce real data showing the effects of noradrenaline and serotonine depletions in 6OHDA-induced parkinsonian animal models [53], suggesting possible causal dynamical interactions between the basal ganglia regions and the areas involved in the neuromodulators release. In addition, the model gives some predictions on how the activity in other brain areas not investigated in the target experiments of [53] could change, and also on possible alternative treatments acting on LC and DRN activity. A stability analysis that confirms the soundness of the model is also performed and in fact used directly during the parameters search phase to filter out candidates which lack the desired stability properties. This point could be critical to validate the effectiveness of the model [61; 62].
The understanding of PD as a multifactorial disease which affects the noradrenergic and serotonergic systems beyond the dopaminergic one could support the development of more effective drugs, possibly with fewer side effects. Moreover, the understanding of the system's dynamical behaviour could allow the development of new tools for early diagnosis based on the interaction of the different monoaminergic systems [50; 52; 45].

2.2 Scope and methodology

This work proposes a simplified model of the interaction of brain areas which are involved in the onset of Parkinson's disease. In particular, the focus is on

the circuits given rise to by neuromodulators interactions, namely serotonine, dopamine and noradrenaline. The aim is to identify a model that explains a broad range of observed interactions within this system and can potentially hint at which are the most important effects at play. This interactions are not yet completely understood at the systemic level and there is only limited experimental data available.

The work is organized in the following steps:

1. **Data aggregation**: available experimental data on Sprague-Dawley rats is aggregated from multiple articles. The data about brain areas activation and neuromodulator concentration had to be carefully chosen to be meaningful in the context of this meta-study. In particular, it is important to select only measurements which have been performed with similar and compatible measurement methods and techniques: electrical and chemical measurements of brain areas are extremely sensitive to hard-to-control variables. For example, electrodes placement in-vivo can only be roughly estimated during the procedure and have to be assessed post-mortem, with relatively low spatial precision; chemical concentrations can oftentimes also only be measured post-mortem and are sensitive to the timing and extraction techniques.

   Values from different studies, despite being obtained consistently in each case, are likely to be measured using different techniques and are therefore hard (if not impossible) to compare in terms of absolute values. Relative change trends have instead been shown to be more consistently reproducible across studies, and are therefore more informative in this meta-study context.

2. **Data generation**: aggregated data about the activation frequency of brain areas in different states (healthy, affected by induced parkinson or other monoaminergic imbalances) is condensed as random variables with an associated distribution, which will be used later on to generate synthetic data compatible with experimental studies. In particular, a synthetic population of virtual mice is generated according to the identified distributions; each virtual mice therefore identifies an instance of the constraints that a model must be able to reproduce.

3. **Model hypothesis**: A model architecture hypothesis is formulated according to the structures and interactions suggested by available data and literature; model parameters that happen to represent measurable quantities that are present in the data are set to values derived from literature instead of being left free for optimization. Fitness measures and evaluation criterions suitable for the model hypothesis are defined.

4. **Model validation**: The free parameters of the model are optimized for each individual of the synthetic population, resulting in a population of models which reproduce the synthetic data with the desired accuracy. If instead the model would not able to reproduce the data (hence it would not be possible to optimize the model's parameters), the architecture is evidently wrong and the model must be reformulated.
5. **Direct predictions**: The population of models is used to extract predictions of known and unknown variables. If the predictions on the behaviour of known variables are in accordance with experimental data, the model is considered valid; predictions of yet unknown variables can be extracted and subject to experimental verification in future studies.
6. **Indirect predictions** The validated model can now be used to make predictions on the effects of changing some parameters of interest (for example, to predict the effects of an hypothetical treatment which stimulates one of the brain areas.)

Steps 3 and 4 effectively define a model exploration cycle. Various model hypothesis have been explored; the journey of this work started with a linear model which included only the most important connections identified in literature, and was enriched in successive steps by adding missing connections and introducing non-linear effects as suggested by the literature until we identified the simplest model that could:

- agree with the available literature in terms of connections and effects between the areas
- be able to reproduce the data with the desired accuracy.

This work only presents the details of the last iteration of the model which is indeed able to reproduce the phenomena under study.

2.3 Available data

This section summarizes the data that have been collected across a large number of published and peer-reviewed articles. The articles, and therefore the data, have been carefully selected for consistency: whenever possible, measurements done with similar or comparable procedures were preferred to data obtained by different means. The chosen data-source articles all referred to measurements of mice of the same specie, sex and other relevant characteristics. The data presented in the following tables is used to impose constraints, parameters and expected values in healthy and lesioned subjects. In the available literature, the data is usually reported in the form of an average value, always accompanied

by a confidence interval and sometimes by distribution characteristics. Since data about single subjects is oftentimes not directly available (and when it is available, it may not be complete), we have no choice but to define what average values and distribution should look like, and generate synthetic individuals that we can then reproduce and analyze.

2.3.1 Average brain area activation in healthy subjects

This data is obtained by measuring and averaging the activity on many neurons from the same area; despite being independent, neurons of the same area tend to exhibit similar average behaviours like firing patterns and rates. The averaged firing frequency is therefore a useful indicator of the overall activity of that particular cluster of neurons.

Area	Value	Details
GP	22.0Hz	Globus pallidus (average, internal end external) [52; 63; 64]
StrD1	10.0Hz	Striatum, Medium spiny neurons type D_1 [65; 66]
StrD2	9.0Hz	Striatum, Medium spiny neurons type D_2 [65; 66]
SNc	4.47Hz	Substantia nigra pars compacta [67]
DRN	1.41Hz	Dorsal raphe nucleus [67]
LC	2.3Hz	Locus coeruleus [68]

For this values, we assume a normal distribution around the average value, with a normalized maximum excursion of $\pm 50\%$.

2.3.2 Time constants in healthy subjects

The time constant is the time it takes for a neuron to go back to its baseline activation frequency after a stimulation. Different kind of neurons, which are usually clustered in brain areas, exhibit similar time constants. Since in this work we are focusing on the asymptotic behaviour of the system, we can reduce the complexity of the model by imposing that an entire brain area is characterized by its time constant, which in turn we assume not to be altered by lesions or other factors.

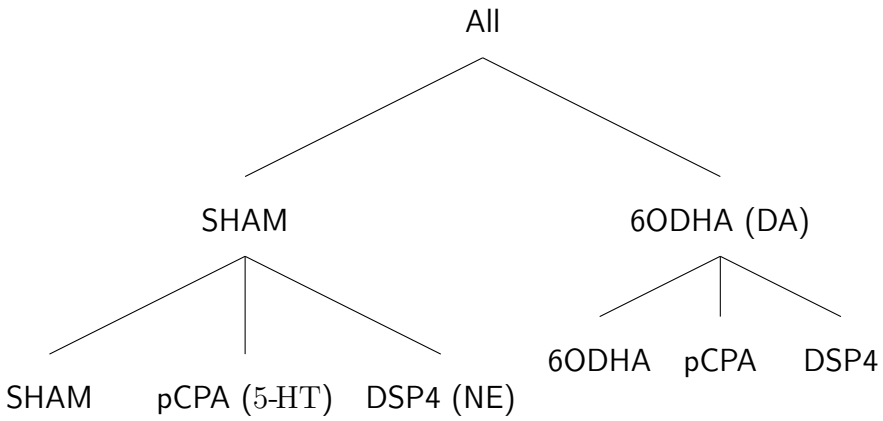
Parameter	Value	Details
τ_{DRN}	3.3 ± 0.3 ms	[69]
τ_{SNc}	1.5 ± 0.3 ms	[70]
τ_{LC}	0.8 ± 0.3 ms	[71]
τ_{GP}	18 ± 0.3 ms	[72]
τ_{StrD1}	2 ± 0.3 ms	[65; 66]
τ_{StrD2}	2 ± 0.3 ms	[65; 66]

2.3.3 Reference data for areas interaction

In [52] a population of Adult male Sprague–Dawley rats are subdivided in six groups using the following procedure:

1. Half of the rats are injected with 6OHDA, the other half with a saline solution (control group, also called SHAM)
2. Three weeks later, half of each population is injected with either a saline solution, pCPA or DSP4

There are therefore six populations:



where:

- SHAM: the rats were injected a saline solution which is expected to have no effect, hence this represents the control group

- 6ODHA: this drug selectively targets dopaminergic neurons and induces a parkinsonian state
- DSP4: this drug selectively lesions the noradrenerginc neurons of the Locus Coeruleus
- pCPA: is a selective inhibitor of serotonin synthesis which induces a 50-80% serotonin depletion effect, reversible 4 days after the injection

The experiment identified the following changes in the activity of the Globus Pallidus:

Group	Frequency	Comment
SHAM	22Hz	[52; 64]
SHAM + 6ODHA	22Hz	= SHAM [64]
SHAM + DSP4	22Hz	= SHAM [52]
SHAM + pCPA	15Hz	(= 0.65 SHAM) [52]
6ODHA + DSP4	11–22Hz	(0.5 SHAM <= x <= SHAM) [52]
6ODHA + pCPA	11–22Hz	(0.5 SHAM <= x <= SHAM) [52]

The lesions also have effects on other areas:

Group	Effect
SHAM+6OHDA	SNc drops by at least 90% wrt SHAM [52]
	LC drops by at least 20% wrt SHAM [68]
SHAM+DSP4	LC drops by at least 80% wrt SHAM [52]
SHAM+pCPA	DRN drops by at least 70% wrt SHAM [52]

The synthetic data that is generated according to this table used to fit the model instances is discussed in details in section 2.6.5.

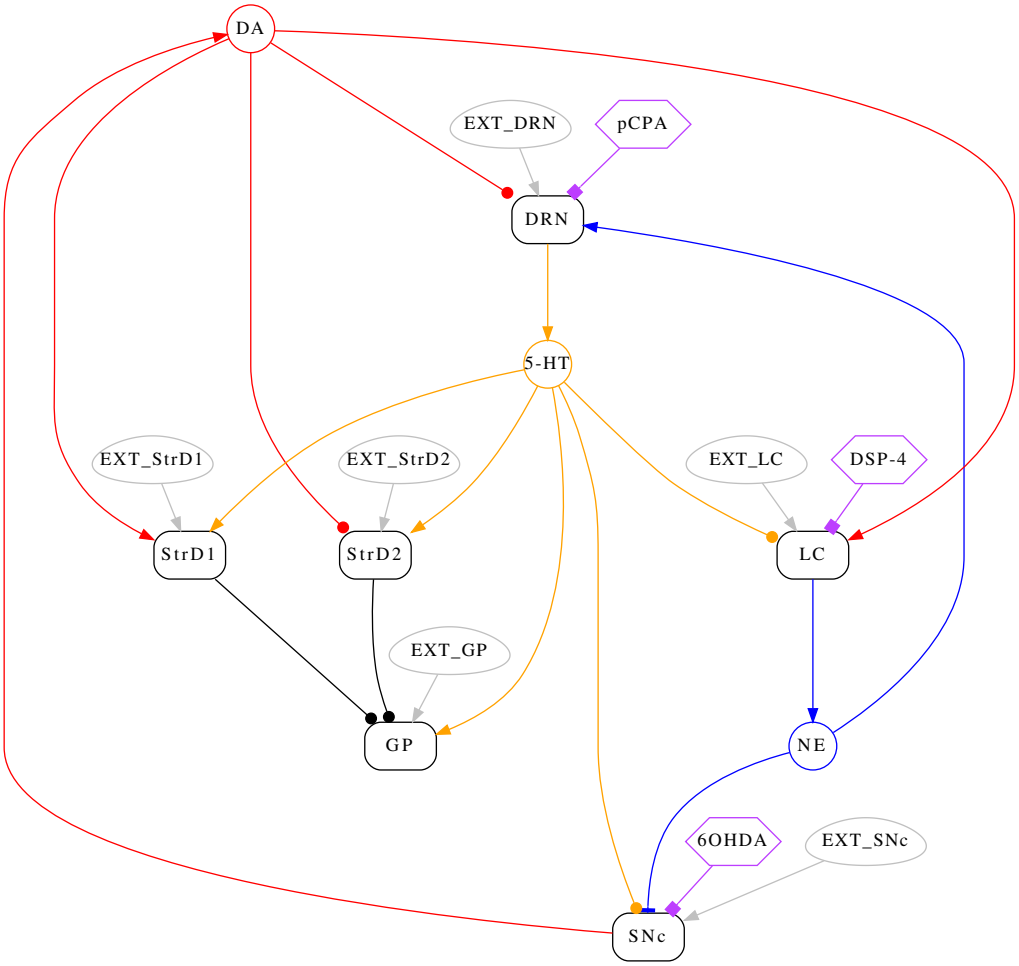


Figure 2.1: Conceptual model schema. The average activation frequencies of six brain areas are modelled (rounded rectangles); some interactions are modulated by monoamines (circles). Arrows represent positive (excitatory) effects while circles represent negative (inhibitory) effects. Noradrenaline has a nonlinear (both excitatory and inhibitory) effect on SNc which is indicated by a bar. Each area has a corresponding stimulus (ovals) which represents self-activation as well as any other stimulus the area might receive from the rest of the brain, which is not explicitly modelled. Finally, hexagons indicate which areas are affected by the administration of which drugs.

2.4 The model

2.4.1 Assumptions and simplifications

Figure 2.1 summarizes the conceptual model. The average activation frequency of six brain areas is represented by the corresponding rounded boxes: the *locus coeruleus* (LC), the *dorsal raphe nucleus* (DRN), the *substantia nigra pars compacta* (SNc), the *striatum* (StrD1, StrD2), and the the *globus pallidus* (GP). The striatum is comprised of D1- and D2-type neurons which react differently to dopamine and are therefore represented separately. In this model, brain areas release a fixed amount of monoamines (*noradrenaline* (NE) from LC, *serotonin* (5-HT) from DRN and *dopamine* (DA) from SNc) which is then projected in different amounts to other areas, with excitatory (arrow), inhibitory (dot) or non-linear (bar) effects. Each area has a characteristic rest activation frequency which is due to internal and external (the rest of the unmodelled brain) factors; this factors are assumed to be constant and are modelled as an excitatory stimulus from the oval blobs. Finally, the lesions are applied by means of administering a drug which interferes with the normal functioning of the affected brain area; in particular we assume that the sensitivity of the affected area, with respect to all the modelled stimuli, changes when the lesion is applied. The drugs are represented with hexagonal blocks.
The model is developed on the base of a few main assumptions: first of all, we assume that a brain area produces an amount of monoamine that is directly proportional to its average activation frequency. Moreover, we assume that the produced monoamine is distributed to the targeted areas with constant ratios which do not depend on the activation frequency. The weight of the connection between a monoamine and its targeted area in Figure 2.1 therefore represents at the same time the fraction of monoamine that is projected to the targeted area and the area's sensitivity to the molecule. Under the aforementioned assumptions, it is not strictly necessary to represent the monoamine concentration explicitly and the schema of Figure 2.1 can be simplified to the one of Figure 2.2 which remains conceptually equivalent: each area has an influence on other areas which is proportional to its activation frequency, to the strength of the monoamine projection and the sensitivity of the receiver. External stimuli and drugs are also hidden to avoid clutter and leave the focus on the modelled brain circuit. We assume that every area responds linearly to the monoamine projection it receives. The sole exception is the substantia nigra pars compacta, for which instead we assume a non-linear reaction to the noradrenaline projected from the locus coeroleus which can either be inhibitory or excitatory in character. Moreover, we do not model explicitly the specific mean of action of each drug (and hence the different nature of each lesion), but we assume that the drug alters the sensitivity of the affected area to all its afferent inputs.

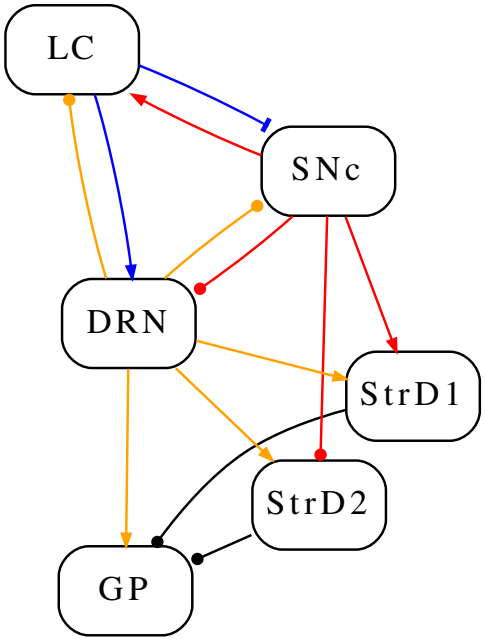


Figure 2.2: Simplified model schema: the same circuits defined in Figure 2.1 can be represented as direct connections under the assumption that the projected monoamine amount is directly proportional to the average activation frequency of an area. External stimuli and drugs are also not represented here for the sake of simplicity.

2.4.2 Dynamic model

The schema from Figure 2.2 can be represented using the following system of equations:

$$\dot{\mathrm{GP}} = -\frac{1}{\tau_{\mathrm{GP}}}\mathrm{GP} - \alpha_{\mathrm{GP}}^{\mathrm{StrD1}}\mathrm{StrD1} - \alpha_{\mathrm{GP}}^{\mathrm{StrD2}}\mathrm{StrD2} + \alpha_{\mathrm{GP}}^{\mathrm{DRN}}\mathrm{DRN} + \alpha_{\mathrm{GP}}^{\mathrm{ext}} \tag{2.4.1}$$

$$\dot{\mathrm{StrD1}} = -\frac{1}{\tau_{\mathrm{StrD1}}}\mathrm{StrD1} + \alpha_{\mathrm{StrD1}}^{\mathrm{SNc}}\mathrm{SNc} + \alpha_{\mathrm{StrD1}}^{\mathrm{DRN}}\mathrm{DRN} + \alpha_{\mathrm{StrD1}}^{\mathrm{ext}} \tag{2.4.2}$$

$$\dot{\mathrm{StrD2}} = -\frac{1}{\tau_{\mathrm{StrD2}}}\mathrm{StrD2} - \alpha_{\mathrm{StrD2}}^{\mathrm{SNc}}\mathrm{SNc} + \alpha_{\mathrm{StrD2}}^{\mathrm{DRN}}\mathrm{DRN} + \alpha_{\mathrm{StrD2}}^{\mathrm{ext}} \tag{2.4.3}$$

$$\dot{\mathrm{SNc}} = -\frac{1}{\tau_{\mathrm{SNc}}}\mathrm{SNc} - \alpha_{\mathrm{SNc}}^{\mathrm{DRN}}\mathrm{DRN} - \alpha_{\mathrm{SNc}}^{\mathrm{LC}}\mathrm{LC} + \beta_{\mathrm{SNc}}^{\mathrm{LC}}\mathrm{LC}^2 + \alpha_{\mathrm{SNc}}^{\mathrm{ext}} \tag{2.4.4}$$

$$\dot{\mathrm{DRN}} = -\frac{1}{\tau_{\mathrm{DRN}}}\mathrm{DRN} - \alpha_{\mathrm{DRN}}^{\mathrm{SNc}}\mathrm{SNc} + \alpha_{\mathrm{DRN}}^{\mathrm{LC}}\mathrm{LC} + \alpha_{\mathrm{DRN}}^{\mathrm{ext}} \tag{2.4.5}$$

$$\dot{\mathrm{LC}} = -\frac{1}{\tau_{\mathrm{LC}}}\mathrm{LC} + \alpha_{\mathrm{LC}}^{\mathrm{SNc}}\mathrm{SNc} - \alpha_{\mathrm{LC}}^{\mathrm{DRN}}\mathrm{DRN} + \alpha_{\mathrm{LC}}^{\mathrm{ext}} \tag{2.4.6}$$

where the abbreviated notation $\dot{x}$ stands for $\frac{dx}{dt}$ and:

- the time constants τ_x are all positive and refer to a dampening term which brings back the activity of each area to its resting activation level in the absence of external stimulation (see section sec:taus)
- the parameters α represent the linear components of the system, are all positive and follow the notation: α_{to}^{from};
- the parameters α_x^{ext} are synthetic terms that implicitly account for the rest activation of each area and other external stimuli which are not part of the modelled circuit.
- the parameter β is also positive and follows the same notation β_{to}^{from}, but account for non-linear effects.

2.4.3 Formalization

Let **y** be the status vector of the system of equations (2.4.1) – (2.4.6); we also define s to be the size of **y**, hence the number of equations in the system. We therefore have:

$$\mathbf{y} = (\mathrm{GP}, \mathrm{StrD1}, \mathrm{StrD2}, \mathrm{SNc}, \mathrm{DRN}, \mathrm{LC})^T \in \mathbb{R}^s \tag{2.4.7}$$

The system can be represented in the general form:

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)) \tag{2.4.8}$$

where each component of the function $\mathbf{f} : (\mathbb{R} \times \mathbb{R}^s) \to \mathbb{R}^s$ is defined by the corresponding equation in (2.4.1) – (2.4.6) [p.47].
In particular, none of the equations depend on the independent variable t; the system is therefore autonomous or time-independent:

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t)) \tag{2.4.9}$$

and we assume the initial state $\mathbf{y}(t_0) = \mathbf{y}_0$ to be known.

The linear case When the β parameter is zero, system (2.4.1) – (2.4.6) [p.47] is linear and can therefore be represented in matrix form as:

$$\dot{\mathbf{y}}(t) = A\mathbf{y}(t) + \mathbf{b}, \qquad a_{ij} = \alpha_i^j, \quad \mathbf{b} = \begin{pmatrix} \alpha_1^{ext} \\ \vdots \\ \alpha_s^{ext} \end{pmatrix} \tag{2.4.10}$$

where, $a_{ij} = 0$ if the corresponding α_i^j is not defined and likewise $b_i = 0$ if α_i^{ext} is not defined.

Specialization Considering system (2.4.1) – (2.4.6) [p.47], the non-linear part can also be easily represented in matrix form. In particular we have:

$$\dot{\mathbf{y}}(t) = A\mathbf{y}(t) + C(\mathbf{y}(t) \circ \mathbf{y}(t)) + \mathbf{b} \tag{2.4.11}$$

were α_{ij} and b_i are defined as in (2.4.10) and also $c_{ij} = \beta_i^j$ or $c_{ij} = 0$ where again the corresponding β_i^j coefficient is not defined; "$\circ$" indicates the element-wise vector product. We will, from now on, refer to the explicit notation from (2.4.1) – (2.4.6) [p.47] or to this compact notation interchangeably, in effort to keep the exposition as clear as possible and focused on the mathematical or biological aspects as necessary.

Equation (2.4.11) [p.48] therefore represents the system:

$$\dot{y_1} = -\frac{1}{\tau_1}y_1 - a_{12}y_2 - a_{13}y_3 + a_{15}y_5 + b_1 \tag{2.4.12}$$

$$\dot{y_2} = -\frac{1}{\tau_2}y_2 + a_{24}y_4 + a_{25}y_5 + b_2 \tag{2.4.13}$$

$$\dot{y_3} = -\frac{1}{\tau_3}y_3 - a_{34}y_4 + a_{35}y_5 + b_3 \tag{2.4.14}$$

$$\dot{y_4} = -\frac{1}{\tau_4}y_4 - a_{45}y_5 - a_{46}y_6 + c_{46}y_6^2 + b_4 \tag{2.4.15}$$

$$\dot{y_5} = -a_{54}y_4 - \frac{1}{\tau_5}y_5 + a_{56}y_6 + b_5 \tag{2.4.16}$$

$$\dot{y_6} = a_{64}y_4 - a_{65}y_5 - \frac{1}{\tau_6}y_6 + b_6 \tag{2.4.17}$$

2.4.4 Modelling lesions

Each component of the status vector (2.4.7) [p.48] represent the average activation frequency of the corresponding brain area, which in turn indirectly represents respectively how much monoamine is produced and projected to the affected areas.
As explained in [52], three kinds of monoamine depletion are chemically induced by administering the corresponding drug in the brain region of interest:

Drug	Effect	Affected area
6ODHA	Dopamine depletion	SNc
pCPA	Serotonine depletion	DRN
DSP4	Noradrenaline depletion	LC

With this model we assume a monoaminic depletion to be caused by the death (or temporary incapacitation) of a fraction of an area's neurons, which in turn we assume to be directly reflected by a fall in the average activation frequency of the area.

Each equation of the model is composed by three conceptual blocks:

- a damping term

- a constant stimulus
- reaction to projections from other areas

The constant stimulus represents external and internal activation sources that are not directly accounted for in this model. Together with the damping term, the constant stimulus accounts for the resting behaviour of the area: the area will stabilize to its rest activation frequency. In absence of reaction terms each equation has an equilibrium point:

$$y'(t) = -\frac{1}{\tau}y(t) + k, \quad y'(t) \equiv 0 \Rightarrow y(t) = k\tau. \tag{2.4.18}$$

The time constants τ are derived from literature (see values from section 2.3.2) and we assume them to be typical values for the specific kind of neuron found in an area; we therefore assume they are not altered by the lesion.

It is however reasonable to expect that the sensitivity of lesioned areas to internal and external stimuli will change in such a way that the average activation frequency changes to the levels which have been experimentally measured.

We can therefore define multiple versions of the same model, which differ from the healthy model only for the constant and reaction coefficients of the leasioned area.
For example, suppose a healthy subject *Bob* is modelled using model (2.4.11) [p.48] by the coefficients held in $A, C, \mathbf{b}$. Having received a dopaminergic lesion (hence, SNc neurons have been affected by 6ODHA), sick Bob will now be modelled by the same equations of model (2.4.11) [p.48] but this time with coefficients $A_{6OHDA}, C_{6OHDA}, \mathbf{b}_{6OHDA}$, which differ by $A, C, \mathbf{b}$ only by the values corresponding to the parameters of the equation for SNc, namely $\alpha^{\mathrm{DRN}}_{\mathrm{SNc}}$, $\alpha^{\mathrm{LC}}_{\mathrm{SNc}}$, $\beta^{\mathrm{LC}}_{\mathrm{SNc}}$, $\alpha^{\mathrm{ext}}_{\mathrm{SNc}}$.
Likewise, when Bob also receives a serotonergic lesion by being adminstered pCPA, there will be a third set of Bob's parameters $A_{6OHDA+pCPA}$, $C_{6OHDA+pCPA}$, $\mathbf{b}_{6OHDA+pCPA}$ which again differ from $A_{6OHDA}, C_{6OHDA}, \mathbf{b}_{6OHDA}$ only by the parameters corresponding to the equation for DRN, and so on.
A single subject is therefore represented by multiple versions of the parameters matrices $A, C, \mathbf{b}$, each set corresponding to one particular state: Healthy (also called SHAM), 6ODHA, pCPA, DSP4 when only one of the lesions is applied, 6ODHA+pCPA, 6ODHA+DSP4 when lesions are combined, and so on.

2.4.5 Parameters dimensionality analysis

Each component of the status vector $\mathbf{y}$ directly represents the average activation frequency of a brain area, and is therefore expressed in Hz.
The derivative term in each equation of system (2.4.1) – (2.4.6) [p.47] are all

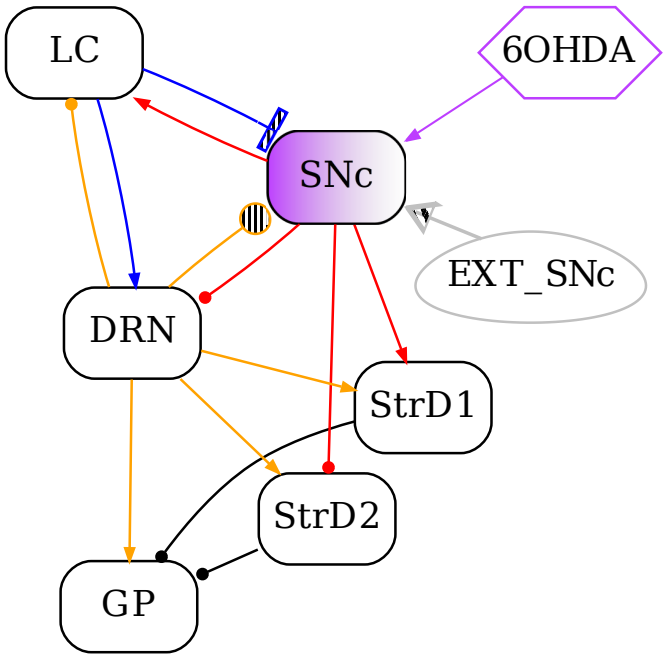


Figure 2.3: Representation of how the modelled lesion affects the system: the drug-induced lesion influences the behaviour of an area (in this case SNc) by modifying its sensitivity to the stimuli it receives.

derivatives with respect to time of a frequency, hence they are all expressed in Hz/s (or $1/s^2$).
Consequently, the external stimulus parameters α^{ext} must also be expressed in Hz/s, while the remaining α parameters must be 1/s, hence Hz.
The second order term parameters β are instead pure numbers, since $Hz^2=1/s^2=Hz/s$.
Finally, all time constants τ are naturally expressed in seconds.

2.4.6 Stability conditions

Let us consider a non-linear system $\dot{\mathbf{y}} = f(t, \mathbf{y})$, with initial conditions $\mathbf{y}(t_0) = \mathbf{y}_0$ and which is assumed to admit an equilibrium point in $\bar{\mathbf{y}} = \mathbf{0}$[†] The null equilibrium point is *stable* if:

$$\forall \epsilon > 0 \ \exists \delta = \delta(\epsilon, t_0) \text{ such that } \|\mathbf{y}_0\| < \delta \Rightarrow \|\mathbf{y}(t)\| < \epsilon, \quad \forall t \geq t_0 \quad (2.4.19)$$

or in words, a neighborhood of the equilibrium point exists such that any initial point from that neighbourhood remains arbitrarily close to the equilibrium point. More stringently, we can ask for the neighborhood to be constant with respect to time: if $\delta = \delta(\epsilon)$, the solution is *uniformly stable*.
If also $\lim_{t\to\infty} \mathbf{y}(t) = \mathbf{0}$ holds, the solution is *asymptotically stable*.
An even more stringent requirement defines an *exponentially asymptotically stable* solution:

$$\exists \alpha \geq 1, \ \beta, \delta > 0, \text{ such that } \|\mathbf{y}_0\| \leq \delta \Rightarrow \|\mathbf{y}(t)\| \leq \alpha\delta e^{-\beta(t-t_0)}, \ \forall t \geq t_0. \quad (2.4.20)$$

System (2.4.11) [p.48]:

$$\dot{\mathbf{y}}(t) = A\mathbf{y}(t) + C(\mathbf{y}(t) \circ \mathbf{y}(t)) + \mathbf{b} = f(t, \mathbf{y}) \quad (2.4.21)$$

can be translated to have an equilibrium point at the origin. Let $\bar{\mathbf{y}}$ be the equilibrium point, such that:

$$\mathbf{0} = A\bar{\mathbf{y}} + C(\bar{\mathbf{y}} \circ \bar{\mathbf{y}}) + \mathbf{b} = f(t, \bar{\mathbf{y}}) \quad (2.4.22)$$

We obtain:

$$(\mathbf{y} + \bar{\mathbf{y}})' = \dot{\mathbf{y}} = A(\mathbf{y} + \bar{\mathbf{y}}) + C((\mathbf{y} + \bar{\mathbf{y}}) \circ (\mathbf{y} + \bar{\mathbf{y}})) + \mathbf{b} \quad (2.4.23)$$

$$= A\mathbf{y} + A\bar{\mathbf{y}} + C(\mathbf{y} \circ \mathbf{y}) + 2C(\mathbf{y} \circ \bar{\mathbf{y}}) + C(\bar{\mathbf{y}} \circ \bar{\mathbf{y}}) + \mathbf{b} \quad (2.4.24)$$

$$= A\mathbf{y} + C(\mathbf{y} \circ \mathbf{y}) + 2C(\mathbf{y} \circ \bar{\mathbf{y}}) \quad (2.4.25)$$

[†] a point such that $\dot{\mathbf{y}} = f(t, \mathbf{0}) = \mathbf{0}$

since (2.4.22) [p.52] holds. Now $\mathbf{y} = \mathbf{0}$ is clearly an equilibrium point, since A and C are linear combinations and an element-wise product by the zero vector is zero.
Let $D_{\bar{\mathbf{y}}} = \mathrm{diag}(\bar{\mathbf{y}})$; we can now rewrite $\mathbf{y} \circ \bar{\mathbf{y}}$ as $D_{\bar{\mathbf{y}}}\mathbf{y}$. We therefore have:

$$\dot{\mathbf{y}} = A\mathbf{y} + C(\mathbf{y} \circ \mathbf{y}) + 2CD_{\bar{\mathbf{y}}}\mathbf{y} = (A + 2CD_{\bar{\mathbf{y}}})\mathbf{y} + C(\mathbf{y} \circ \mathbf{y}) \tag{2.4.26}$$

which can be represented as a sum of a linear and a non-linear term by setting $\tilde{A} = (A + 2CD_{\bar{\mathbf{y}}})$:

$$\dot{\mathbf{y}}(t) = \tilde{A}\mathbf{y}(t) + \mathbf{g}(\mathbf{y}(t)) \tag{2.4.27}$$

since matrices $A, C, D_{\bar{\mathbf{y}}}$ are constants with respect to time in (2.4.26).

It is now possible to check the applicability of Perron's theorem ([73, p.132]), which states that given a system in the form $\dot{\mathbf{y}}(t) = A\mathbf{y}(t) + \mathbf{g}(t, \mathbf{y}(t))$, if $\sigma(A) \subset \mathbb{C}^-$ and:

$$\lim_{\|\mathbf{y}\| \to 0} \frac{\|\mathbf{g}(t, \mathbf{y})\|}{\|\mathbf{y}\|} = 0 \tag{2.4.28}$$

uniformly with respect to t, then $\mathbf{y} = \mathbf{0}$ is exponentially asymptotically stable.

Since:

$$\|\mathbf{g}(\mathbf{y})\| = \|C(\mathbf{y} \circ \mathbf{y})\| \tag{2.4.29}$$

condition (2.4.28) is clearly satisfied. Hence, if also $\sigma(\tilde{A}) \subset \mathbb{C}^-$ holds, the system is exponentially asymptotically stable.

Assuming A, C and $\mathbf{b}$ to be known for a particular instance of system (2.4.11) [p.48], it is therefore necessary to compute an approximation of the equilibrium point $\bar{\mathbf{y}}$ and consequently $\tilde{A}$:

$$\tilde{A} = A + 2CD_{\bar{\mathbf{y}}}, \quad D_{\bar{\mathbf{y}}} = \mathrm{diag}(\bar{\mathbf{y}}) \tag{2.4.30}$$

before the stability condition can be verified.
In the particular case of system (2.4.1) – (2.4.6) [p.47], C has only one non-zero element, and therefore:

$$2CD_{\bar{\mathbf{y}}} = 2(c_{46}\mathbf{e}_4)(\bar{y}_6\mathbf{e}_6^T) = 2\beta_{\mathrm{SNc}}^{\mathrm{LC}}\bar{y}_6\mathbf{e}_4\mathbf{e}_6^T \tag{2.4.31}$$

where $\mathbf{e}_k$ is as usual the k-th versor of the canonical base.
The equilibrium point $\bar{\mathbf{y}}$ can be approximated by applying the Newton method to find the root of:

$$f(\mathbf{y}) = A\mathbf{y} + C(\mathbf{y} \circ \mathbf{y}) + \mathbf{b} \tag{2.4.32}$$

using the iteration:

$$\bar{\mathbf{y}}^{l+1} = \bar{\mathbf{y}}^{l} - \frac{f(\bar{\mathbf{y}}^{l})}{f'(\bar{\mathbf{y}}^{l})} \tag{2.4.33}$$

$$= \bar{\mathbf{y}}^{l} - (A + 2CD_{\bar{\mathbf{y}}^{l}})^{-1} f(\bar{\mathbf{y}}^{l}) \tag{2.4.34}$$

and taking as the starting point $\bar{\mathbf{y}}^0$ the solution of the linear part $A\bar{\mathbf{y}} + \mathbf{b} = \mathbf{0}$. The system is therefore exponentially asymptotically stable if both $\sigma(A) \subset \mathbb{C}^-$ and $\sigma(\tilde{A}) \subset \mathbb{C}^-$ are verified.
The iteration to compute the equilibrium point can be stopped when:

$$\max_i |\bar{y}_i^{l+1} - \bar{y}_i^{l}| < \text{tol} \tag{2.4.35}$$

hence when the maximum error on each component of $\bar{\mathbf{y}}^{l+1}$ is smaller than a set tolerance, or after an arbitrary limit of maximum allowed iterations is reached.

2.5 Defining fitness measures

A fitness measure of a model is a single figure of merit, usually normalized to $[0, 1]$, that summarises how close the model is to achieving a set of aims.

Defining an appropriate fitness measure is the first necessary step for optimizing the parameters of the model and evaluating its performance. In this section we identify a fitness measure suitable for system (2.4.11) [p.48] and the corresponding lesioned variations described in section 2.4.4, by gradually refining a general formulation of a fitness measure until it encompasses all the aspects that are required.

2.5.1 Fitness from distance

Let N be the number of integration steps in the interval t_0, T. The step h is therefore:

$$h = \frac{T - t_0}{N} \tag{2.5.1}$$

and the integration is done over the discrete set $J = \{t_i = t_0 + hi\}$, $i = 0, ..., N$. The solution $\mathbf{y}(t)$ to (2.4.10) [p.48] on J is represented by the matrix:

$$Y(J) = \begin{pmatrix} y_1(t_0) & \cdots & y_1(t_N) \\ \vdots & & \vdots \\ y_s(t_0) & \cdots & y_s(t_N) \end{pmatrix} \in \mathbb{R}^{s \times (N+1)} \tag{2.5.2}$$

Similarly, given a reference solution vector of expected states $\mathbf{y}_T$, we can represent the reference solution with the corresponding matrix $Y_T(J)$:

$$\mathbf{y_T}(t) = \begin{pmatrix} y_{T1}(t) \\ \vdots \\ y_{Ts}(t) \end{pmatrix}, \quad Y_T(J) = \begin{pmatrix} y_{T1}(t_0) & \cdots & y_{T1}(t_N) \\ \vdots & & \vdots \\ y_{Ts}(t_0) & \cdots & y_{Ts}(t_N) \end{pmatrix} \in \mathbb{R}^{s\times(N+1)} \tag{2.5.3}$$

where y_{Ti} is the reference solution of equation y_i.

We can now define the error matrix:

$$E = (e)_{ij} = Y(J) - Y_T(J) \tag{2.5.4}$$

that can be used to compute the mean square error mse:

$$\mathrm{mse} = \frac{\mathrm{Tr}\big(E^T E\big)}{(N+1)s} = \frac{\sum_i \sum_j e_{ij}^2}{(N+1)s} \tag{2.5.5}$$

using which we can finally define a fitness $f \in (0, 1]$ as:

$$f = \frac{1}{1+\mathrm{mse}} \tag{2.5.6}$$

It is straightforward to see that $f \to 1$ when mse approaches zero and conversely, $f \to 0$ when mse grows towards infinity; a model is therefore perfectly fit when $f = 1$.

2.5.2 Fitness tolerance as mean square error

The "closeness to one" (or tolerance) of the fitness measure is of course inversely related to the magnitude of the mean square error. It is intuitively easy to impose a fitness requirement in terms of "number of nines after the comma" to indicate a wanted precision.

More formally, we can require f to be arbitrarily close to one by defining a tolerance y and imposing $f \geq 1 - 10^{-y}$. The tolerance y can also be read as the the maximum mse allowed; in fact, if we approximate the mse with a

negative power of ten, from the fitness definition we obtain:

$$f = 1 - 10^{-y} = \frac{1}{1 + 10^{-x}} \tag{2.5.7}$$

$$10^{-y} = \frac{10^{-x}}{1 + 10^{-x}} \tag{2.5.8}$$

$$-y = \log_{10}\left(\frac{10^{-x}}{1 + 10^{-x}}\right) \tag{2.5.9}$$

$$y = \log_{10}\left(\frac{1}{10^{-x}} + 1\right) = \log_{10}\left(1 + 10^{x}\right) \approx x \tag{2.5.10}$$

hence the fitness tolerance y can be used as a direct and intuitive representation of the (negative) order of magnitude we are requiring the mse to have, especially if y is greater than one.

2.5.3 Accounting for variable steps

Some integration methods dynamically change the step size: when the properties of the dynamical system allow for it (i.e. the problem is not stiff) this approach can greatly reduce the number of function evaluations necessary to solve the equations within the required error tolerances.

The number and size of each step is therefore generally not know in advance but becomes part of the solution which is now composed of two parts, namely the discrete set of integration points Q and the solution vectors at each point:

$$Q = \{t_0, t_1, ..., t_{N-1}, T\}, \quad Y(Q) = \begin{pmatrix} y_1(t_0) & y_1(t_1) & \cdots & y_1(T) \\ \vdots & & & \vdots \\ y_s(t_0) & y_s(t_1) & \cdots & y_s(T) \end{pmatrix} \tag{2.5.11}$$

where $t_i < t_j$ if $i < j$.

The definition of mse from (2.5.5) [p.55] needs to be extended to account for the variability of the step size. Assuming that the reference solution $\mathbf{y_T}$ is known at each point in Q, we weight the error at each point using the width of the interval it spans. In particular, we define:

$$\mathrm{mse} = \frac{1}{s}\sum_{i=1}^{N}\frac{(t_i - t_{i-1})\|\mathbf{e}_i\|_2^2}{T - t_0}, \quad \mathbf{e}_i = \mathbf{y}(t_i) - \mathbf{y_T}(t_i) \tag{2.5.12}$$

which remains conceptually equivalent to the one defined previously in (2.5.5) [p.55]; we can therefore keep the same definition of fitness as in (2.5.6) [p.55].

2.5.4 Accounting for early termination

Early termination can be set to happen on predefined triggers. For example, an integration method can be stopped as soon as one component of the solution changes sign or reaches a threshold. The brain models in this work express a meaningful representation of reality only if all the modelled physical quantities are positive: should a simulation ever reach a negative value in any component of the solution, despite being numerically correct it is physically impossible, so we can save time and computing power by stopping the integration early and discarding that particular model.

Early termination makes it necessary to be able to compare the fitness of early-terminated solutions among themselves and against any other solution. One meaningful way of doing this is to scale the weighted variable-step fitness proportionally to the time span that was actually integrated; in particular, let N_e be the last integration step. We therefore define:

$$f = \frac{t_{N_e} - t_0}{T - t_0} \frac{1}{1 + \mathrm{mse}}, \quad \mathrm{mse} = \frac{1}{s} \sum_{i=1}^{N_e} \frac{(t_i - t_{i-1}) \|\mathbf{e}_i\|_2^2}{t_{N_e} - t_0} \tag{2.5.13}$$

2.5.5 Partial fitness

When we are not interested in the fitness of the solution with respect to all its components but only some of them, the error matrix (2.5.4) [p.55] can naturally be reduced by zeroing (or removing altogether) the rows corresponding to the components one wants to ignore, having care to also set the size s to the number of components being considered.
Likewise, in the variable integration step case one should remove the components to be ignored when computing the error vectors $\mathbf{e}_i$ as defined in (2.5.12) [p.56].

One may also want to ignore a portion of the simulation in the time domain, for example, to disregard transient effects before the solution stabilizes (supposing it does, indeed, eventually become stable). This can be achieved by averaging only over the wanted time span; suppose the first useful time interval starts at time t_k, (2.5.13) becomes:

$$f = \frac{t_{N_e} - t_k}{T - t_k} \frac{1}{1 + \mathrm{mse}}, \quad \mathrm{mse} = \frac{1}{s} \sum_{i=k+1}^{N_e} \frac{(t_i - t_{i-1}) \|\mathbf{e}_i\|_2^2}{T - t_k} \tag{2.5.14}$$

2.5.6 Barrier (and range) fitness

There also cases in which the value of a desired solution is not explicitly known (hence the error as defined previously is not conceptually significant), but there is a relaxed constraint such as being smaller or bigger than a reference solution, or may need to lay between two reference values.

This case can be handled in a sound way by considering errors with the appropriate sign only. In particular, we can define a filtering function $sieve : \mathbb{R}^s \to \mathbb{R}^s$ which, applied to the error vectors, selects only the components relevant to the constraint.

For example, a sieve that considers only positive errors (and hence allows the model to err on the negative side without constraints) on the first component would be defined as follows:

$$sieve(\mathbf{e}) = \begin{pmatrix} \max(0, e_1) \\ e_2 \\ \vdots \\ e_s \end{pmatrix} \tag{2.5.15}$$

The sieve can then be applied before the norm of the error vectors when calculating the mse:

$$\mathrm{mse} = \frac{1}{s} \sum_{i=1}^{N_e} \frac{(t_i - t_{i-1}) \|sieve(\mathbf{e}_i)\|_2^2}{T - t_0} \tag{2.5.16}$$

2.5.7 Composed fitness measures

It is oftentimes necessary to define a fitness measure that is a composition of other measures. For example, as will be discussed in detail later, the fitness of a model with lesions can require a combination of fitness measures of different instances of the model with some parameters tuned against different target solutions.

The most straightforward way of combining fitness values is to geometrically average them. Supposing to have n distinct fitness measures $f_1, ..., f_n$, the combined fitness is therefore:

$$f = \frac{1}{n} \sum_{i=1}^{n} f_i \tag{2.5.17}$$

This average of fitness values, other than being straightforward, has the important advantage that it allows to combine, in a intuitively meaningful way, values which are not necessarily derived from a mean square error as described earlier (for example, a synthetic score which evaluates the stability of the system, or some other wanted property of a system, or a computational cost score; in general any property that can significantly be mapped to $[0, 1]$).

It is nonetheless useful to understand the relationship between this synthetic fitness values and the respective mean square errors they are derived from, since it is important to know how an average fitness requirement translates to the actual distance from a reference solution.

In the particular case where all fitness measures can be derived from a mean square error (as in (2.5.6) [p.55]), averaging the mean square errors before computing the final fitness:

$$f = \frac{1}{1 + \frac{1}{n}\sum_{i=1}^{n} \mathrm{mse}_i} \tag{2.5.18}$$

exhibits a more intuitive behaviour of the fitness measure since it is a linear combination.

The bidimensional example in Figure 2.5 and Figure 2.4 can help in understanding the difference between (2.5.17) [p.58] and (2.5.18). Note that when there are only two errors to combine, the two expressions respectively simplify to:

$$\begin{aligned} f_{avg} &= \frac{1}{2}\left(\frac{1}{1+a} + \frac{1}{1+b}\right) = \frac{a+b+2}{2(a+1)(b+1)}, \\ f_{mse} &= \frac{1}{1 + \dfrac{a+b}{2}} = \frac{2}{a+b+2} \end{aligned} \tag{2.5.19}$$

which happen to be equivalent if $a = b$; it is evident that the average of fitness values is not a linear function with respect to the mse.

It is therefore important to remember, when interpreting composed fitness values, that the averaged fitness measure do not translate directly to the mean square errors unless the values being averaged are sufficiently close to each other.

2.5.7.1 The vanished-in-the-average problem

The efficacy of fitness (2.5.6) [p.55] as a figure of merit unfortunately decreases with the size s of the system under consideration: there are in fact infinite

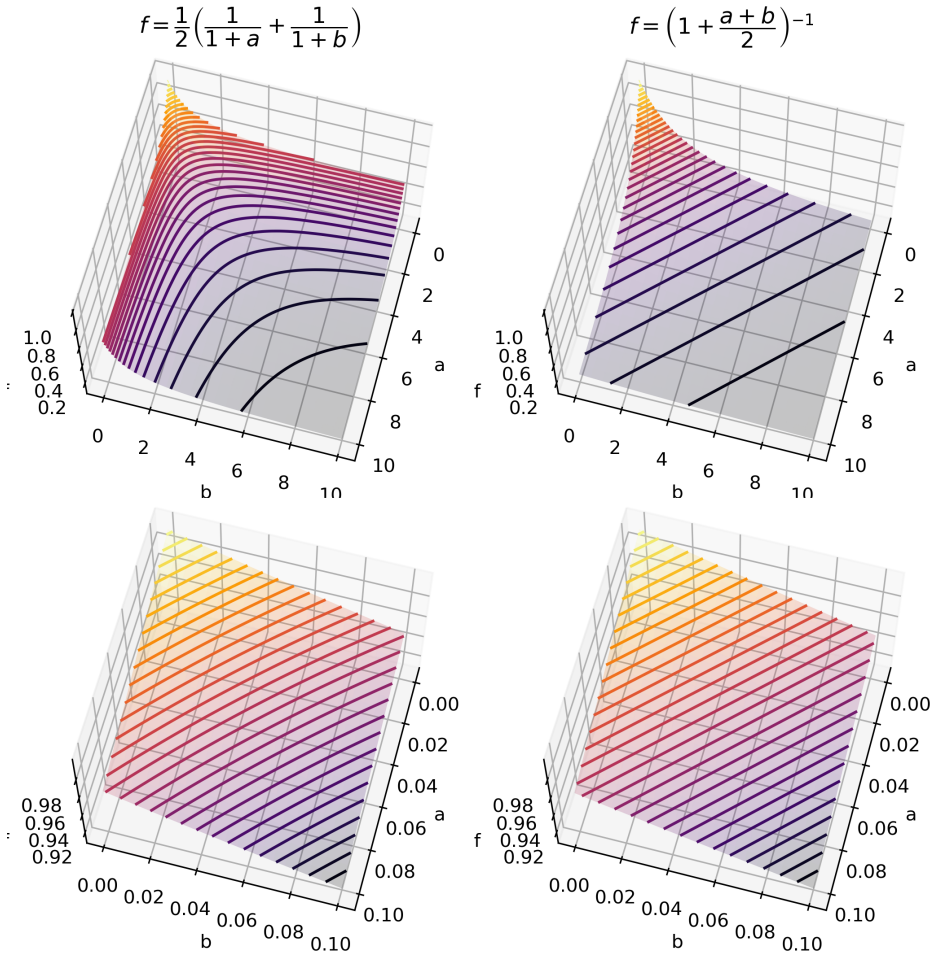


Figure 2.4: Comparison of the behaviour of averaging fitness measures ((2.5.17) [p.58], left) with respect to averaging the mean square errors ((2.5.18) [p.59], right) in function of the mean square errors a and b. The non-linear behaviour of the former is more evident when the mean square errors have greater magnitude.
It is also evident in the top left figure that the equal-fitness lines can span whole ranges of one parameter when the other is near zero, which means that in this condition the fitness function has become insensitive to one of its components.

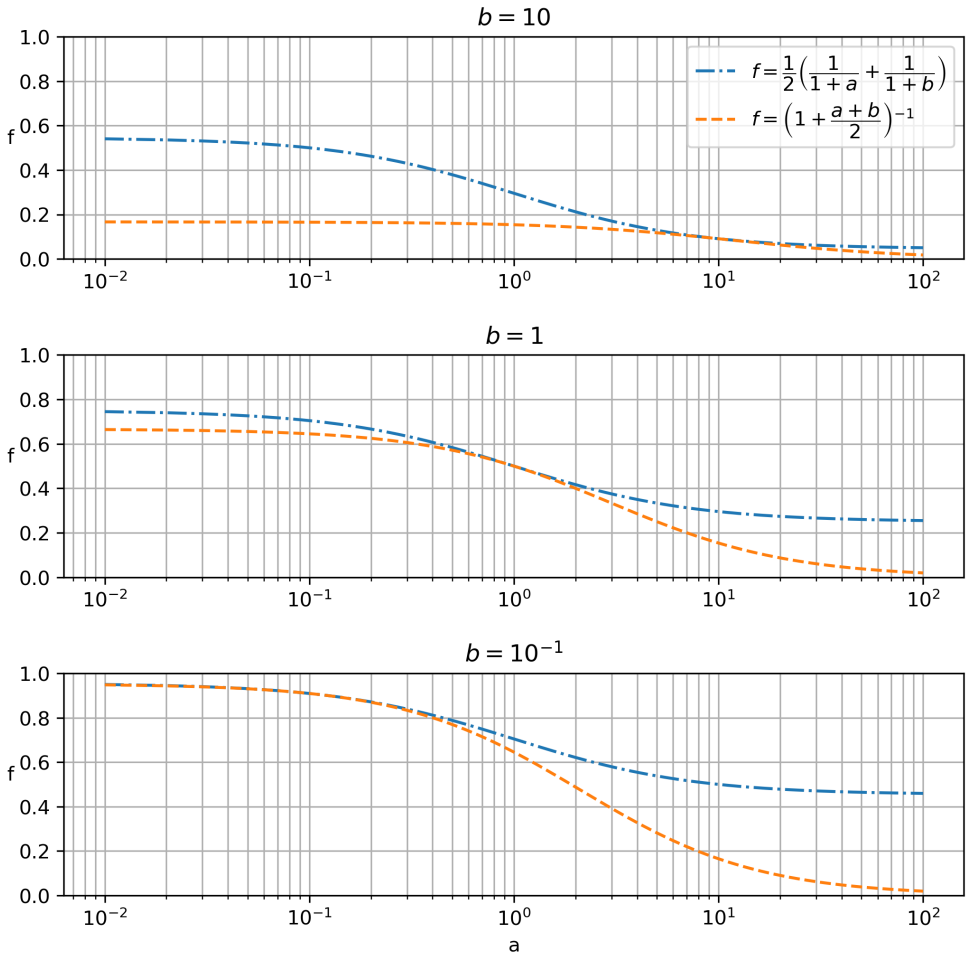


Figure 2.5: Cross section of Figure 2.4, comparison of the behaviour of averaging fitness measures ((2.5.17) [p.58], blue) with respect to averaging mean square errors ((2.5.18) [p.59], orange), in function of the mean square errors a and b, in the case of only two components. Their equivalence is evident where $a = b$. The averaging of mean square errors is more sensitive to the biggest error, hence provides lower fitness value if one of the errors is big irrespectively of the value of the smaller error.

combinations of two or more mean square errors that can compute to the same fitness value.

The formula represented on right side of Figure 2.4 is equivalent to the above-mentioned fitness formulation in the case of a system of two equations; when averaging mean square errors, it is indeed evident that having fixed a fitness value as requirement, any optimization algorithm would not have any means to distinguish a solution that fits both equations equally well over one that fits one equation much better that the other.

This problem is unfortunately amplified by the average of fitness values as defined in (2.5.17) [p.58]: the left side of Figure 2.4 clearly shows that with this formulation, when one of the errors being averaged becomes really small (hence, there is good fitness), the combined fitness measure also loses sensitivity with respect to the other parameters: the equal-fitness lines become almost orthogonal to the axis with the lower error.

Equation (2.5.17) [p.58] can be reformulated to mitigate this effect. In particular, we define:

$$f = \sqrt{\min_i(f_i)\frac{1}{n}\sum_{i=1}^{n} f_i} \tag{2.5.20}$$

This formulation reduces the loss of sensitivity when some of the combined fitness values are much closer to 1 than others; Figure 2.7 shows how this combined fitness grows much slower compared to the simple average, but still behaves similarly when the averaged values are close enough.
In fact, there is an important difference: this combined fitness grows slower than both the other measures when the two averaged values are very similar and only one starts to grow (see the neighborhood of the points for which $a = b$ in Figure 2.7 and Figure 2.6).
This can be seen as a ridge (right side of Figure 2.6) in the combined fitness function which assigns greater fitness values to errors similar in magnitude, and penalizes diverging errors magnitudes (provided their distance is small enough). The extreme cases where some components have maximum fitness (hence zero error) can still be problematic as we have discussed previously for (2.5.17) [p.58], although the sensitivity loss is reduced: the $\min_i(f_i)$ term firmly keeps the combined fitness low even in the extreme cases where many components have maximum fitness and only one of them does not fit well.
The mitigated composed fitness (2.5.20) is therefore advantageous since, within appropriate conditions, it can guide an optimization algorithm to prefer solutions which have similar fitness values for all its fitness components.

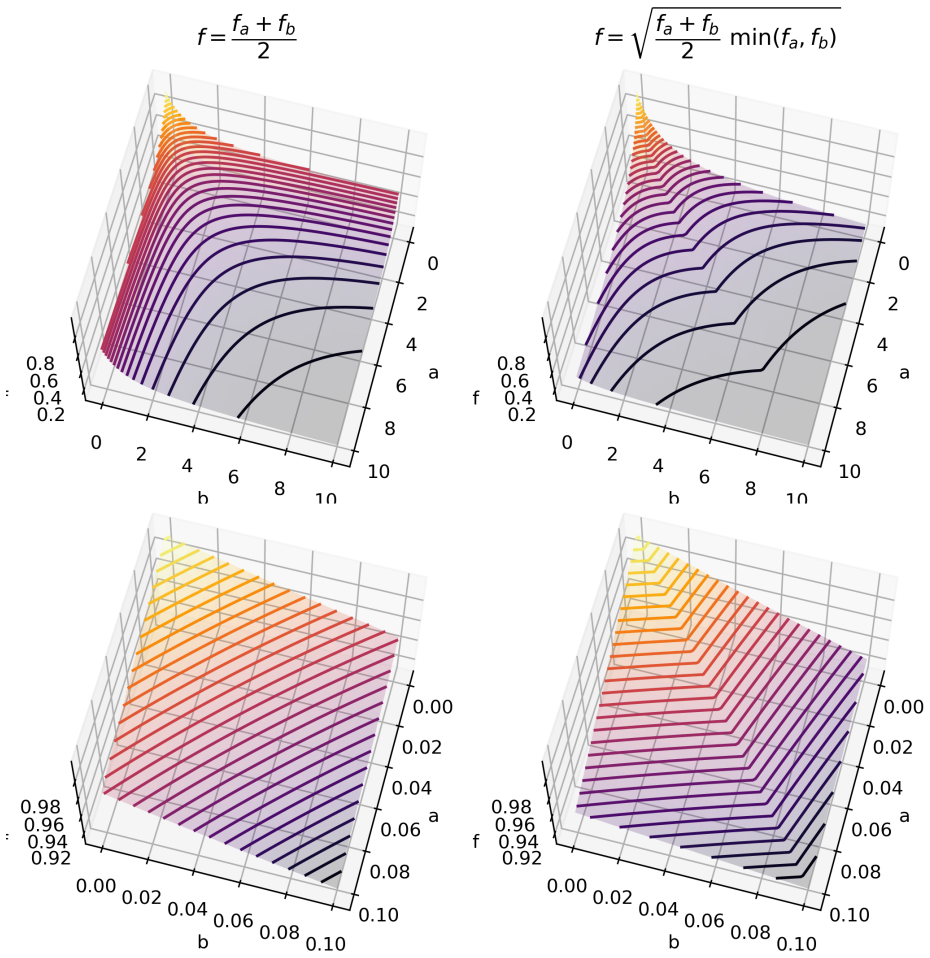


Figure 2.6: Comparison of the non-linear mitigation strategy (right) with the previously discussed simple fitness average. The function's preference (in terms of fitness) for parameters with similar magnitude is visible as a ridge along the $a = b$ line. The lack of sensitivity at the extremes (whene $a = 0$ or $b = 0$) is still present, but mitigated.

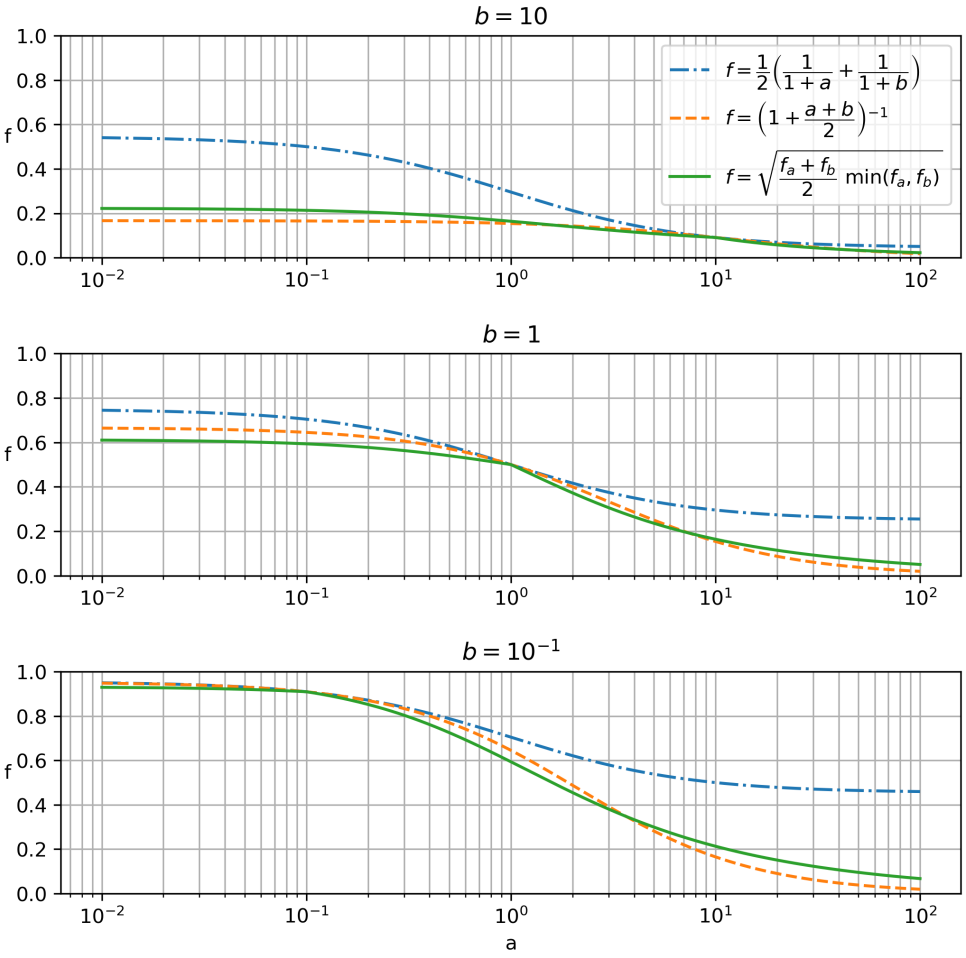


Figure 2.7: Cross section of Figure 2.6, comparison of the three fitness averaging methods, in function of the mean square errors a and b. The mitigated average (solid line) behaves like the intuitively sound average of the mean square errors, but has the desirable property of growing slower than the alternatives in an appropriate neighbourhood of the optimum.

2.5.7.2 Simplifications and optimizations

We have seen that formulation of fitness composition in (2.5.20) [p.62] potentially outperforms the averaging of mean square errors, since it can guide an optimizer to a solution with the preferred property of having a similar magnitude of all values which are being averaged.
It is therefore useful to redefine the mse relative to a reference solution in is such a way to take advantage of this formulation. In particular, we can split the error matrix (2.5.4) [p.55] by rows, and combine fitnesses resulting from each mse corresponding to each component of the status vector. We therefore define:

$$f = \sqrt{\min_i(f_i)\frac{1}{s}\sum_{i=1}^{s} f_i}, \quad f_i = \frac{1}{1+\mathrm{mse}_i},$$

$$\mathrm{mse}_i = \sum_{j=1}^{N} \frac{(t_j - t_{j-1})e_{ij}^2}{T - t_0}, \quad i = 1, .., s \tag{2.5.21}$$

where e_{ij} is an element of the error matrix as previously defined in (2.5.4) [p.55].

Since the mean square error and consequently the fitness we just defined are non surjective functions, it also make sense to further simplify the mse definition by removing the scaling factor:

$$\mathrm{mse}_i = \sum_{j=1}^{N}(t_j - t_{j-1})e_{ij}^2 \tag{2.5.22}$$

This formulation is an upper bound for the previous definition of $\mathrm{mse_i}$ that is not only less computationally expensive, but can also prevent numerical issues when the time intervals and the errors shrink towards the machine's numerical precision μ.

2.6 Simulation setup

2.6.1 Free parameters and constants

As previously hinted is section 2.6.5 and 2.4.4, we require a subject's model to be able to reproduce its target data in four different states at the same time: healthy (SHAM), dopaminergic lesion (6OHDA), noradrinergic lesion (DSP4) and serotonergic lesion (pCPA). The combinations 6OHDA+DSP4 and 6OHDA+pCPA are instead constrained only to a target range, to be able to

also serve as a prediction (and hence as a measure of the agreement of the model with experimental data).

Let S_i be the set of parameters that define the model representing test subject i. S_i contains:

- 6 time constants: τ_{GP}, τ_{StrD1}, τ_{StrD2}, τ_{SNc}, $\tau_{\mathrm{DRN}}, \tau_{\mathrm{LC}}$. As discussed in 2.4.4, the time constants are derived from literature and are not optimized.
- SHAM: The healthy model instance has 20 free parameters: all α and β parameters defined in system (2.4.1) – (2.4.6) [p.47]
- 6OHDA: The dopaminergic lesion instance has 4 free parameters: $\alpha_{\mathrm{SNc}}^{\mathrm{DRN}}$, $\alpha_{\mathrm{SNc}}^{\mathrm{LC}}$, $\beta_{\mathrm{SNc}}^{\mathrm{LC}}$, $\alpha_{\mathrm{SNc}}^{\mathrm{ext}}$. Those are all the parameters of the SNc equation. All the other parameters are considered constants and stay the same as in SHAM.
- pCPA: The serotonergic lesion instance has 3 free parameters: $\alpha_{\mathrm{DRN}}^{\mathrm{SNc}}$, $\alpha_{\mathrm{DRN}}^{\mathrm{LC}}$, $\alpha_{\mathrm{DRN}}^{\mathrm{ext}}$, All the other parameters are considered constants and stay the same as in SHAM.
- DSP4: The Noradrinergic lesion has 3 free parameters: $\alpha_{\mathrm{LC}}^{\mathrm{SNc}}$, $\alpha_{\mathrm{LC}}^{\mathrm{DRN}}$, $\alpha_{\mathrm{LC}}^{\mathrm{ext}}$, All the other parameters are considered constants and stay the same as in SHAM.
- 6OHDA+pCPA, 6OHDA+DSP4: the combination of lesions do not have any free parameters but are constructed by applying to the SHAM values, in order, the appropriate parameters from each lesion.

The set S_i therefore contains a total of 36 parameters, 30 of which must be optimized at the same time to fit the available data. Appropriate subsets of the parameters in S_i are then used to build the corresponding matrices $A, C, \mathbf{b}$ to completely define system (2.4.11) [p.48], and hence compute its solution and properties.

We will from now on refer to S_i as the complete model for subject i, since it is the set of parameters that completely define it. The variations S_i^{kind}, like $S_i^{\mathrm{SHAM}}, S_i^{\mathrm{6OHDA}}$ and so on, will refer instead to the subset of parameters which are currently being applied to actually simulate the model.
We will denote one solution as:

$$S_i^{\mathrm{SHAM}}(\mathbf{y}_0, t_0, T) = Y = \begin{pmatrix} y_1(t_0) & \cdots & y_1(t_N) \\ \vdots & & \vdots \\ y_s(t_0) & \cdots & y_s(t_N) \end{pmatrix} \tag{2.6.1}$$

Y is therefore the solution obtained by integrating the model in the interval $[t_0, T]$, with the starting vector $\mathbf{y}_0$, and using the SHAM subset of parameters. The number N of integration steps, as well as their size, is usually variable and chosen by the integration method case-by-case, hence it can potentially be different for each subset of parameters.

2.6.2 Fitness measure

The fitness measure for subject S_i is a composition of many fitness terms, combined using (2.5.20) [p.62]. To completely describe the model of a subject, other than its set of parameters S_i, we need to represent its corresponding set of target values T_i: T_i contains the reference solutions that the model is supposed to reproduce when using the parameters in S_i. T_i must therefore have one reference solution for each of the states (healthy and lesions) that we are modelling.
In particular, we assume that we can address a particular reference solution from T_i in a similar way to the solution corresponding to particular subsets of parameters in S_i:

$$T_i^{SHAM}(J) = Y_T(J) = \begin{pmatrix} y_{T1}(t_0) & \cdots & y_{T1}(t_N) \\ \vdots & & \vdots \\ y_{Ts}(t_0) & \cdots & y_{Ts}(t_N) \end{pmatrix} \tag{2.6.2}$$

under the assumption that T_i can provide the reference solution for any discrete set of times $J = \{t_0, ..., t_N\}$ which is decided by the integration algorithm during the computation of the solution $S_i^{\text{SHAM}}(\mathbf{y}_0, t_0, T)$. The same notation of course applies for the other cases, T_i^{6OHDA}, T_i^{pCPA} and so on.

The subject index will intentionally be left out in the following sections to lighten the notation further, since it's not relevant in the context: the fitness is of course computed independently for each subject in the same way.

2.6.2.1 SHAM fitness

The fitness of the healthy instance is divided in one fitness measure for each equation.
To simplify the notation, we define:

- $Y_T = T^{SHAM}(J)$, the corresponding target solution
- $\mathbf{y}_0 = T^{SHAM}(t_0)$ the starting vector

- $Y = S^{\mathrm{SHAM}}(\mathbf{y}_0, t_0, T)$, the simulation of the model using the appropriate subset of parameters
- $J = \{t_0, ..., t_N \leq T\}$, the time base chosen by the integration method

For each of the s equations in the status vector we can compute the corresponding mse_i according to (2.5.22) [p.65]:

$$\mathrm{mse}_i = \sum_{j=1}^{N} (t_j - t_{j-1}) e_{ij}^2, \quad (e)_{ij} = Y_T - Y, \quad i = 1, .., s \tag{2.6.3}$$

and finally the simulation-time-weighted fitness according to (2.5.13) [p.57]:

$$f_i^{SHAM} = \frac{t_N - t_0}{T - t_0} \frac{1}{1 + mse_i} \tag{2.6.4}$$

The set of measures for the SHAM instance is therefore:

$$F^{SHAM} = \left\{ f_i^{SHAM} | i = 1, ..., s \right\} \tag{2.6.5}$$

2.6.2.2 6OHDA fitness

Similarly to the SHAM case, we define:

- $Y_T = T^{6OHDA}(J)$, the corresponding target solution
- $\mathbf{y}_{h0} = T^{SHAM}(t_0)$ the starting vector for the healthy case
- $\mathbf{y}_{l0} = T^{6OHDA}(t_0)$ the starting vector for the 6OHDA case
- $Y = S^{\mathrm{6OHDA}}(\mathbf{y}_0, t_0, T)$, the simulation of the model using the appropriate subset of parameters
- $J = \{t_0, ..., t_N \leq T\}$, the time base chosen by the integration method
- $t_c = \frac{T - t_0}{2}$, the time before which we ignore the solution's fitness

In this case, from the available data we have only three reference solutions to consider: GP, SNc and LC; in particular, the former is to be fitted exactly from experimental data. The SNc fitness is a conceptual requirement since we don't have the corresponding exact experimental data: 6OHDA is a lesion of neurons in SNc that in turn lowers the levels of dopamine. We therefore require that the average activation frequency of SNc has to become at least as low as indicated in the reference solution, but can be free to become even lower. Similarly, available data suggests a lowered activity in LC to be at most 80% of the healthy value.
We also require the solution to be stable with two different initial conditions:

the solution should obviously be stable near the equilibrium point (the 6OHDA solution $\mathbf{y}_{l0}$), but perhaps more importantly, a healthy subject (hence starting with $\mathbf{y}_{h0}$) must be able to transition to the lesioned state (as it naturally occurs in-vivo during the experiment) without incurring in instabilities. This also implies that the first transient phase should be ignored in the computation of the fitness; for that reason we defined t_c as a threshold time before which we ignore the solution. Particular care should be used in choosing t_0, T and consequently t_c big enough with respect to the time constants of the system.
We therefore define two fitness measures for GP:

$$f_{\mathrm{GP}}^{6OHDA}(\mathbf{y}_{l0}, t0, T, t_c), \quad f_{\mathrm{GP}}^{6OHDA}(\mathbf{y}_{h0}, t0, T, t_c) \tag{2.6.6}$$

both are computed as in (2.6.5) [p.68]:

$$f_i^{6OHDA}(\mathbf{y}, t_0, T, t_c) = \frac{t_N - t_0}{T - t_0} \frac{1}{1 + mse_i}, \tag{2.6.7}$$

$$\mathrm{mse}_i = \sum_{j=c}^{N} (t_j - t_{j-1}) e_{ij}^2, \quad (e)_{ij} = Y_T - Y, \tag{2.6.8}$$

where Y, Y_T and hence J are of course computed accordingly to the selected $\mathbf{y}$, c is the index of the first $t \geq t_c$ in J, and the index of GP in the status vector happens to be 1, hence $i = 1$.

The fitness f_{SNc}^{6OHDA} for SNc is computed in a similar way, but also applying a sieve function (as explained in section (2.5.6) [p.58]) that ignores negative errors:

$$f_i^{6OHDA}(\mathbf{y}, t_0, T, t_c) = \frac{t_N - t_0}{T - t_0} \frac{1}{1 + mse_i}, \tag{2.6.9}$$

$$\mathrm{mse}_i = \sum_{j=c}^{N} (t_j - t_{j-1}) \max(0, e_{ij})^2, \quad (e)_{ij} = Y_T - Y, \tag{2.6.10}$$

where i is the index of the SNc equation in the status vector.

Following the same principles, we define the two fitness measures for LC:

$$f_i^{6OHDA}(\mathbf{y}, t_0, T, t_c) = \frac{t_N - t_0}{T - t_0} \frac{1}{1 + mse_i}, \tag{2.6.11}$$

$$\mathrm{mse}_i = \sum_{j=c}^{N} (t_j - t_{j-1}) \max(0, e_{ij})^2, \quad (e)_{ij} = Y_T - Y, \tag{2.6.12}$$

where this time i is the index of the LC equation in the status vector.

The set of measures for the 6OHDA case therefore has four elements:

$$F^{6OHDA} = \big\{ f_{\mathrm{GP}}^{6OHDA}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{GP}}^{6OHDA}(\mathbf{y}_{h0}, t0, T, t_c), \tag{2.6.13}$$

$$f_{\mathrm{SNc}}^{6OHDA}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{SNc}}^{6OHDA}(\mathbf{y}_{h0}, t0, T, t_c), \tag{2.6.14}$$

$$f_{\mathrm{LC}}^{6OHDA}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{LC}}^{6OHDA}(\mathbf{y}_{h0}, t0, T, t_c) \big\} \tag{2.6.15}$$

2.6.2.3 pCPA and DSP4 fitness

The fitness measures for this two instances are conceptually identical to the 6OHDA case; the fitness for GP is therefore computed according to (2.6.7) [p.69], and the fitness for the lesioned areas use the the same sieve as in (2.6.9) [p.69] (but of course selecting the correct lesioned area, respectively DRN and LC). The sets of measures for this two instances are defined as:

$$F^{pCPA} = \Big\{ f_{\mathrm{GP}}^{pCPA}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{GP}}^{pCPA}(\mathbf{y}_{h0}, t0, T, t_c), \tag{2.6.16}$$

$$f_{\mathrm{DRN}}^{pCPA}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{DRN}}^{pCPA}(\mathbf{y}_{h0}, t0, T, t_c) \Big\} \tag{2.6.17}$$

$$F^{DSP4} = \big\{ f_{\mathrm{GP}}^{DSP4}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{GP}}^{DSP4}(\mathbf{y}_{h0}, t0, T, t_c), \tag{2.6.18}$$

$$f_{\mathrm{LC}}^{DSP4}(\mathbf{y}_{l0}, t0, T, t_c), f_{\mathrm{LC}}^{DSP4}(\mathbf{y}_{h0}, t0, T, t_c) \big\} \tag{2.6.19}$$

2.6.2.4 Lesion combination fitness

The combination of lesions is left as unconstrained as possible to be able to serve as a prediction; we do however at least require the corresponding simulation not to diverge and to lie within an acceptable range. In the experiment lesions are applied in succession, 6OHDA always first. It makes sense to require the solution to be stable with both $\mathbf{y}_{h0} = T^{SHAM}(t_0)$ and $\mathbf{y}_{l0} = T^{6OHDA}(t_0)$ as initial conditions. We define a fitness which only considers the temporal span of the solution to penalize early divergence:

$$f^{6OHDA+DSP4}(\mathbf{y}, t_0, T) = \frac{t_N - t_0}{T - t_0}, \quad f^{6OHDA+pCPA}(\mathbf{y}, t_0, T) = \frac{t_N - t_0}{T - t_0} \tag{2.6.20}$$

where J and consequently t_N, t_0 are computed from the corresponding simulation
$S^{6OHDA+\mathit{lesion}}(\mathbf{y}, t_0, T)$

Additionally, we require the GP value of 6OHDA+pCPA to be within reasonable limits. In particular, we define specific limit fitness functions similar to

(2.6.9) [p.69], for example the lower bound:

$$f_{i,\min}^{6OHDA+pCPA}(\mathbf{y}, t_0, T, t_c) = \frac{t_N - t_0}{T - t_0} \frac{1}{1 + mse_i}, \tag{2.6.21}$$

$$\text{mse}_i = \sum_{j=c}^{N} (t_j - t_{j-1}) \min(0, e_{ij})^2, \quad (e)_{ij} = Y_T - Y, \tag{2.6.22}$$

$$Y_T = T^{6OHDA+pCPA-\min}(J) \tag{2.6.23}$$

where i is again the index corresponding to GP and $T^{6OHDA+pCPA-\min}(J)$ is the reference solution containing the lower bound; likewise the upper bound will be defined similarly but using $\max$ as an error sieve against the upper bound reference solution $T^{6OHDA+pCPA-\max}(J)$. This limits are necessary to guide the optimization towards a solution which lies within the experimentally determined range and exclude instead solutions which may exhibit better fitness scores but are outside of the physiological range.

The set of measures for the combination of lesions is therefore:

$$F^{COMB} = \Big\{ f^{6OHDA+DSP4}(\mathbf{y}_{h0}, t_0, T), f^{6OHDA+DSP4}(\mathbf{y}_{l0}, t_0, T), \tag{2.6.24}$$

$$f^{6OHDA+pCPA}(\mathbf{y}_{h0}, t_0, T), f^{6OHDA+pCPA}(\mathbf{y}_{l0}, t_0, T), \tag{2.6.25}$$

$$f_{\text{GP,min}}^{6OHDA+DSP4}(\mathbf{y}_{h0}, t_0, T), f_{\text{GP,min}}^{6OHDA+DSP4}(\mathbf{y}_{l0}, t_0, T), \tag{2.6.26}$$

$$f_{\text{GP,max}}^{6OHDA+DSP4}(\mathbf{y}_{h0}, t_0, T), f_{\text{GP,min}}^{6OHDA+DSP4}(\mathbf{y}_{l0}, t_0, T), \tag{2.6.27}$$

$$f_{\text{GP,max}}^{6OHDA+pCPA}(\mathbf{y}_{h0}, t_0, T), f_{\text{GP,min}}^{6OHDA+pCPA}(\mathbf{y}_{l0}, t_0, T), \tag{2.6.28}$$

$$f_{\text{GP,max}}^{6OHDA+pCPA}(\mathbf{y}_{h0}, t_0, T), f_{\text{GP,max}}^{6OHDA+pCPA}(\mathbf{y}_{l0}, t_0, T) \Big\} \tag{2.6.29}$$

2.6.2.5 Parameters constraints

The fitness function can also be useful to impose soft, dynamic constraints on the parameters. In this case, it makes sense to require the α^{ext} parameters of a lesion to be less or equal than its counterpart in the SHAM instance: that particular brain area have been damaged, and it makes sense to assume it would lower its average activation frequency in absence of other stimuli.
We define the fitness measure:

$$f_l^{PAR} = \frac{1}{1 + max(0, S^l - S^{SHAM})} \tag{2.6.30}$$

where l is one of the three lesions (6OHDA, DSP4, pCPA) and $S^l - S^{SHAM}$ represent the difference between the altered α^{ext} parameter in the lesioned subset and its counterpart in the healthy one.

As usual, we define the set:

$$F^{PAR} = \left\{ f^{PAR}_{6OHDA}, f^{PAR}_{DSP4}, f^{PAR}_{pCPA} \right\} \tag{2.6.31}$$

2.6.2.6 Asymptotic stability constraints

Every subject state considered in this study is supposed to be stable in time; it is therefore important to impose that each set of parameters defines an asymptotically stable system which will ultimately never diverge from its equilibrium point.
As shown in section 2.4.6, system (2.4.1) – (2.4.6) [p.47] is exponentially asymptotically stable if $\sigma(\tilde{A}) \subset \mathbb{C}^-$, where $A = A + 2CD_{\bar{\mathbf{y}}}$ (see (2.4.30) [p.53]).
We can therefore envisage a fitness measure:

$$f^{STAB}_l = \frac{1}{1 + \sum_i \max(0, \mathbb{R}(\lambda_i))} \tag{2.6.32}$$

where l is one of the parameter subsets which define the model (SHAM, 6OHDA, etc.), and λ_i is an eigenvalue of the corresponding $\tilde{A}$. This measure will therefore always be 1 when the system is asymptotically stable, but tend to zero as the real part of the eigenvalues grows more positive.

The computation of $\tilde{A}$ requires using an iterative root-finding method to determine the equilibrium point $\bar{\mathbf{y}}$ of each parameters set $A, C, \mathbf{b}$. It is advantageous to use multiple stopping conditions for this method to avoid unnecessary computation, in particular:

- A tolerance on the precision of $\bar{\mathbf{y}}^l$ as defined in (2.4.35) [p.54]; this tolerance should be set to be compatible with the precision obtained with the parameters optimization algorithm. For example, if the optimization fitness required translates to an mse of 10^{-8}, it makes sense to require $\mathrm{tol} = 10^{-9}$.
- A arbitrary guard on the maximum number of allowed iterations; since precision is not of paramount importance in this context, the number of iterations can be kept rather small (≤ 25).
- A guard on the value of the components of $\bar{\mathbf{y}}$. If the method is converging to an equilibrium point which has some components which are too big or negative, the stability of the system is ultimately meaningless in the context of this study, therefore it is not worth investing computing power in obtaining it with high precision.

We finally define the set:

$$F^{STAB} = \{ f^{STAB}_{SHAM}, f^{STAB}_{6OHDA}, f^{STAB}_{DSP4}, f^{STAB}_{pCPA}, \tag{2.6.33}$$

$$f^{STAB}_{6OHDA+DSP4}, f^{STAB}_{6OHDA+pCPA} \} \tag{2.6.34}$$

2.6.2.7 The fitness, at last

We now have all the elements needed to compute the fitness of subject S_i. Let F be the union of all the sets of measures we have defined:

$$F = F^{SHAM} \cup F^{6OHDA} \cup F^{pCPA} \cup F^{DSP4} \cup F^{COMB} \cup F^{PAR} \cup F^{STAB} \tag{2.6.35}$$

we can now combine all the fitness measures we identified according to (2.5.20) [p.62], to obtain the total (or final) fitness figure:

$$f = \sqrt{\min_i(f_i) \frac{1}{n} \sum_{i=1}^{n} f_i}, \quad f_i \in F,\ n = |F| \tag{2.6.36}$$

2.6.3 Integration method

Given a set of parameters S and an initial value $\mathbf{y}_0$, system (2.4.1) – (2.4.6) [p.47] is completely defined; we however need to choose an appropriate integration method to compute the solution of the initial value problem, to then in turn compute its fitness or examine the behaviour of the system in time.

The Backward Differentiation Formula (BDF) is a family of implicit linear multistep methods for numerical integration of ordinary differential equations [73]. In particular, the initial value problem:

$$\mathbf{y}'(t) = f(t, \mathbf{y}(t)), \quad t \in [t_0, T], \quad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{2.6.37}$$

is approximated on a mesh $J = \{t_0 + ih | i = 0, ..., N\}$ by:

$$\sum_{i=0}^{k} \alpha_i \mathbf{y}_{n+i} = h\beta \mathbf{f}_{n+k} \tag{2.6.38}$$

where k is the order of the method, $\mathbf{y}_n \approx \mathbf{y}(t_n)$, $\mathbf{f}_n = f(t_n, \mathbf{y}_n)$, h is the integration time step and the coefficients α_i, β are chosen such that the method achieves the maximum order† k. Note that the method is implicit, so at each step it also requires the numeric solution of the generally non-linear equation:

$$g(\mathbf{y_{n+k}}) := \mathbf{y_{n+k}} - h\beta f(t_{n+k}, \mathbf{y_{n+k}}) = 0 \tag{2.6.39}$$

†The order conditions for a method impose a limit to the approximation error at each integration step which is proportional to the step size ($O(h^{p+1})$) [73; 74]

This integration schema can be extended to also provide automatic and dynamic selection of the step size and the order of the method on a integration step basis; after each iteration, error estimations and computational cost predictions are used to modify order and step size if possible and favourable [75; 76].

The implementation used in this work, provided by python's `scipy` package [77] is a variable order (1 to 5), variable step scheme; it also supports external conditions checks at each iteration, so the integration can be stopped early, for example when one component of the solution becomes negative or grows in magnitude over a set threshold.
In the context of this work, both stopping conditions (on negative or too high solution component values) are being applied to avoid wasting computation time on obviously not acceptable solutions during the optimization phase. Unfortunately scipy's implementation does not provide a handle to stop the algorithm early in case the computed step size becomes smaller than a set threshold (and hence the integration would perform too many integration steps). It was therefore necessary to externally impose a time limit to intercept and stop early in such cases.

2.6.4 Optimization strategy

The fitness function we defined in (2.6.36) [p.73] is not differentiable in at least some points (since it composes functions which are not differentiable on the whole domain, like $\min$) and is not uni-modal, i.e. it has more than one maximum (since system (2.4.12) [p.49] has infinite solutions for $\mathbf{y}' = \mathbf{0}$ with respect to the free parameters $(a)_{ij}$ and $\mathbf{b}$). We have no indication of where a good solution may lie in the parameter space, hence we need a global optimizer which will not be affected by the initial conditions. Furthermore, we want to restrict the range of the free parameters (in sign and magnitude): we are therefore in need for a non-derivative based and constrained global optimization algorithm.
According to [78], *differential evolution* (DE) is a modern and reliable algorithm that fulfills all this requirements; in particular we use the implementation provided by python's `scipy` package [79].

DE is a population-based optimizer. The objective function is sampled at multiple, randomly chosen initial points (within the domain constraints) which form the initial population of N_p vectors. Similarly to other population-based methods (like Nelder-Mead for example), the population is replaced by new vectors which are perturbations and combinations of the existing ones in successive stages.

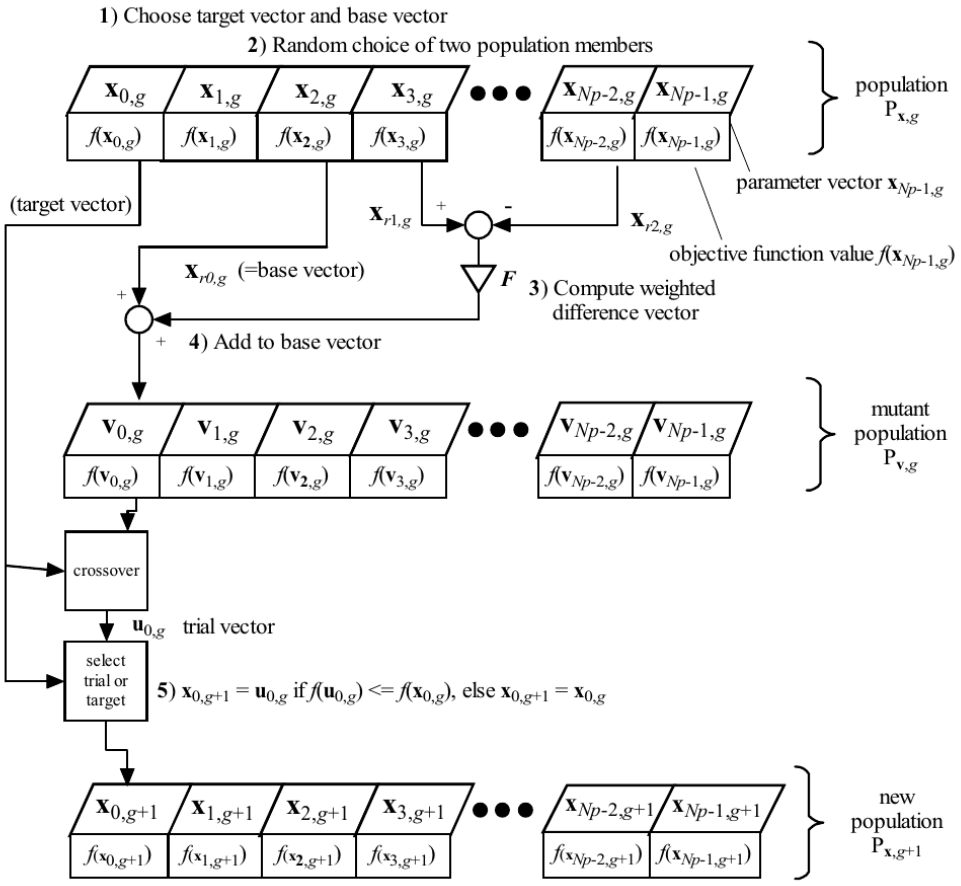


Figure 2.8: A flowchart representation of the Diffirential Evolution optimization algorithm [78].

The most basic implementation of DE is described by the following pseudocode, in which $\mathbf{x_i}$ is one of the N_p vectors of the population of one generation; this population will be replaced by the vectors being defined as $\mathbf{y_i}$ during the next generation. f is the fitness we want to maximize, and $F \in (0, 1)$ is called the *scale factor*:

```
while (convergence criterion not met):
    for i in range(N_p):
        r1 = random_int(0, N_p)
        r2 = random_int(0, N_p)
        r3 = random_int(0, N_p)
        u_i = x_r3 + F * (x_r1 - x_r2)
        if f(u_i) >= f(x_i):
            y_i = u_i
        else:
            y_i = x_i
```

This strategy of generating new vectors, called *mutation*, is usually paired with another one, called *crossover*, which acts on the individual components of the vectors. In particular, an intermediate candidate vector $\mathbf{v_i}$ is generated by mutation from three randomly selected members of the population $\mathbf{x_{r1}}, \mathbf{x_{r2}}, \mathbf{x_{r3}}$ as shown before in the pseudocode:

$$\mathbf{v_i} = \mathbf{x_{r3}} + F(\mathbf{x_{r1}} - \mathbf{x_{r2}}) \tag{2.6.40}$$

but additionally, the candidate vector for replacing $\mathbf{x_i}$, $\mathbf{u_i}$, has its components randomly selected from either $\mathbf{x_i}$ or $\mathbf{v_i}$:

$$(\mathbf{u_i})_j = \begin{cases} (\mathbf{v_i})_j, & \text{if } rand(0,1) \leq C_r \\ (\mathbf{x_i})_j, & \text{otherwise} \end{cases} \tag{2.6.41}$$

where $C_r \in [0, 1]$ defines the *crossover probability*. $\mathbf{u_i}$ is then selected instead of $\mathbf{x_i}$ if $f(\mathbf{u_i}) >= f(\mathbf{x_i})$ as before. Evidently many details are being omitted for the sake of simplicity: at the very least, it is important to ensure that the three vectors involved in the mutation are in fact distinct from each other and that the new candidate vector has all its components within the allowed bounds.
The combination of mutation and crossover allows DE to perform acceptably well on functions that are either decomposable[†] or non-decomposable, while maintaining rotational invariance[‡] [78]. Figure 2.8 illustrates DE with a

[†]A decomposable function can be written as the sum of D one-dimensional functions: $f(\mathbf{x}) = \sum_{i=1}^{D} f_i(x_i)$.

[‡]An algorithm is rotationally invariant if its performances do not depend on the objective function being aligned with a privileged coordinate system.

flowchart.

There are also a plethora of strategies to choose the three candidate vectors for the mutation, which can have dramatic effects on the convergence speed on some problems. For example, the base of the mutation ($\mathbf{x_{r3}}$ in the previous examples) could be the best candidate from the previous generation instead of a random one, to make the algorithm more greedy. The distribution that is used to check the crossover condition C_r can also be changed, as well as the number of vectors used to generate the mutation.
In general, strategies are indicated with triplets, for example `DE/rand/1/bin` indicates the "standard" DE (which we just described), where a random vector is used as the base for the mutation against one difference of two other vectors, and crossover is checked against a binomial distribution; `DE/best/1/exp` instead is a more greedy variant that uses the best candidate from the previous generation as base for the mutation, and the crossover is checked using an exponential distribution.

To summarize, there are five basic parameters which define the behaviour of DE:

- Population size N_p
- Mutation scale F
- Crossover threshold C_r
- Selection strategy: `DE/rand/1/bin`, `DE/best/1/exp`, `DE/best/2/bin`, `DE/rand-to-best/1/bin`, ...
- Initial distribution of the N_p vectors in the allowed space: random, uniform grid, halton, sobol, ...

Following recommendations and performance evaluations on similar problems [80; 78], together with empirical tests, it was determined that populations strongly in excess of $2D$ vectors (where D is the number of free parameters, hence the size of the vectors) do not provide higher probability of convergence while rising the computational cost significatively instead. The strategy `DE/best/1/exp`, with $C_r = F = 0.95$ and a uniform halton distribution, together with $N_p = 3D$ proved to offer good performances on this particular problem.

2.6.5 Synthetic target data

The experimental data we have collected, summarized in sections 2.3.1, 2.3.2 and 2.3.3, ultimately consists of normal distributions around their respective

Figure 2.9: Generated target values distribution for the SHAM case. Values for each individual are generated according to 2.6.5: each area follows a normal distribution around a center value with a maximum spread of $\pm 50\%$ ($4\sigma = 0.5\mu$). Lesioned activation values of an area, when defined, are scaled according to the table presented in section 2.6.5. The dots also directly show the datapoints, the left-right scattering of the dots is random, applied only to help visualization and do not carry any meaning. The placement of the mean line, box, whiskers and fliers is illustrated in Figure A.1.

Figure 2.10: Generated target values distribution for the all cases: in black, the SHAM case. The 6OHDA dopaminergic lesion target (red) differs from SHAM only for SNc and LC. The serotonergic pCPA lesion target (yellow) differs from SHAM only in for GP and DRN. Finally, the noradrenergic DSP4 lesion target (blue) differs from SHAM only for LC.

center values which are to be considered constant; we do not have explicit information about the dynamic behaviour of the system or of the transition from a state to the other. Of course, each subject must go through a dynamic transition from a healthy state to a lesioned state, but both states must be asymptotically stable solutions for the model.
Since there is no single study in literature that lists all the required brain areas activation values for a particular subject at the same time, we have no choice but to generate a synthetic population of virtual subjects with area activation values which lie within the distributions identified across the literature.

In particular, we will generate a number of subjects S_i and each of them will have associated a set of target values T_i which are defined as follows:

Area	Value
GP	$\leftarrow \mathcal{N}(22, 22 \cdot \frac{1}{8})$
GP^{6OHDA}	$= \mathrm{GP}$
GP^{pCPA}	$= \mathrm{GP} \cdot 0.65$
GP^{DSP4}	$= \mathrm{GP}$
$\mathrm{GP}^{6OHDA+pCPA-\max}$	$= \mathrm{GP} \cdot 0.75$
$\mathrm{GP}^{6OHDA+pCPA-\min}$	$= \mathrm{GP} \cdot 0.65$
$\mathrm{GP}^{6OHDA+DSP4-\max}$	$= \mathrm{GP}$
$\mathrm{GP}^{6OHDA+DSP4-\min}$	$= \mathrm{GP} \cdot 0.65$
$\mathrm{StrD1}$	$\leftarrow \mathcal{N}(10, 10 \cdot \frac{1}{8})$
$\mathrm{StrD2}$	$\leftarrow \mathcal{N}(9, 9 \cdot \frac{1}{8})$
SNc	$\leftarrow \mathcal{N}(4.47, 4.47 \cdot \frac{1}{8})$
SNc^{6OHDA}	$= \mathrm{SNc} \cdot 0.1$
DRN	$\leftarrow \mathcal{N}(1.41, 1.41 \cdot \frac{1}{8})$

$$
\begin{aligned}
\mathrm{DRN}^{pCPA} &= \mathrm{DRN} \cdot 0.3 \\
\mathrm{LC} &\leftarrow \mathcal{N}(2.3, 2.3 \cdot \tfrac{1}{8}) \\
\mathrm{LC}^{6OHDA} &= \mathrm{LC} \cdot 0.8 \\
\mathrm{LC}^{DSP4} &= \mathrm{LC} \cdot 0.2
\end{aligned}
$$

where $x \leftarrow \mathcal{N}(\mu, \sigma)$ indicates that x is a random number drawn from a normal distribution using the provided parameters.
Given the completely synthetic nature of this data, it is reasonable to assume activities of every area to follow a normal distribution; we impose every value to lie within $\pm 50\%$ of the center value by putting $4\sigma = \frac{1}{2}\mu$. Since data from literature can be interpreted as an average percentage change in activity for lesioned areas, we decided to treat the generated healthy value for an area as the reference level of an individual and use that as a base to generate the lesioned values where needed. In this way, a subject that has a higher than average value for an area in healthy conditions, will also have an higher than average level in the same area when lesioned, although the value will change by the required proportional amount.

Reference solutions for subject i are therefore composed of constant values. Given a vector of times $J = [t_0, ..., t_n]$, we can define the reference solution matrices:

$$
T_i^L(J) = \begin{pmatrix} GP^L & \cdots & GP^L \\ \vdots & & \vdots \\ LC^L & \cdots & LC^L \end{pmatrix} \in \mathbb{R}^{s \times n+1} \tag{2.6.42}
$$

where L is one of SHAM, 6OHDA, pCPA, DSP4, 6OHDA+pCPA, 6OHDA+DSP4; the appropriate value of each area for the respective lesion is chosen according to L when available, otherwise defaults to the SHAM (not labeled) value. Figure 2.9 shows the distribution of target values for the whole population in the SHAM case, while Figure 2.10 offers an overview of the target values for all cases.

2.6.6 Choosing the simulation time span

The simulation time is arbitrarily set to 0.5s under the assumption that the basic behaviour of each equation in the system will resemble (2.4.18) [p.50], hence the transition time between any state to a stable solution will be dominated by the slowest time constant (which is derived from literature). In fact, since the solution to:

$$y'(t) = -\frac{1}{\tau}y(t) + k \tag{2.6.43}$$

assuming $y(0) = 0$, is

$$y(t) = -k\tau e^{-\frac{t}{\tau}} + k\tau \tag{2.6.44}$$

we can compute the time it takes for the solution to grow past 99% of its limit value:

$$0.99k\tau = -k\tau e^{-\frac{t}{\tau}} + k\tau \Rightarrow t = \log(0.01)\tau \approx 5\tau \tag{2.6.45}$$

(which in engineering contexts is broadly known as the "rule of the five taus").

In this specific case, the slowest $\tau \leq 20$ms, therefore we can assume the transient phase to be finished after $5 \cdot 0.02 = 0.1$s, and a time 5 times longer, 0.5s, should be adequate to see a long stable steady state, and we would expect the dynamic behaviour to have stabilized already around 0.1s. The GP equation in the right graph of Figure 3.10 gives a good example of the five taus rule.

Chapter 3

Results

> What I love about science is that as you learn, you don't really get answers. You just get better questions.
>
> John Green

3.1 Performance and computational costs

A population of 240 target individuals has been generated from the specification described in section 2.6.5. Figure 2.9 and Figure 2.10 illustrate the distribution of the generated data for each brain area. Each individual's target parameters are optimized using the *differential evolution* algorithm, as described in section 2.6.4, applied to the combined fitness function (2.6.36) [p.73].
The parameters for the optimization algorithm have been empirically determined following available rules-of-thumb and performance measurements on other relevant problems ([78; 80; 79]); in particular the strategy that showed the highest probability of convergence on this particular problem is `best1exp` with mutation and recombination both set at 0.95 with a pool of 90 competitors (three times the number of free parameters). Bigger pool sizes dramatically slow down convergence without significantly increasing the probability of converging on a fit solution, while smaller pool sizes would usually not manage to get past some local minima.
Figure 3.1 (in red) also shows that where there is convergence, the fitness is logarithmically approaching its upper limit of 1; the optimization is stopped at the arbitrary measure of fitness greater or equal to $1 - 10^{-8} = 0.99999999$, which is abundantly sufficient for the purpose of this work. The graph also shows that at this fitness, there is no experimental evidence that the fit could not be

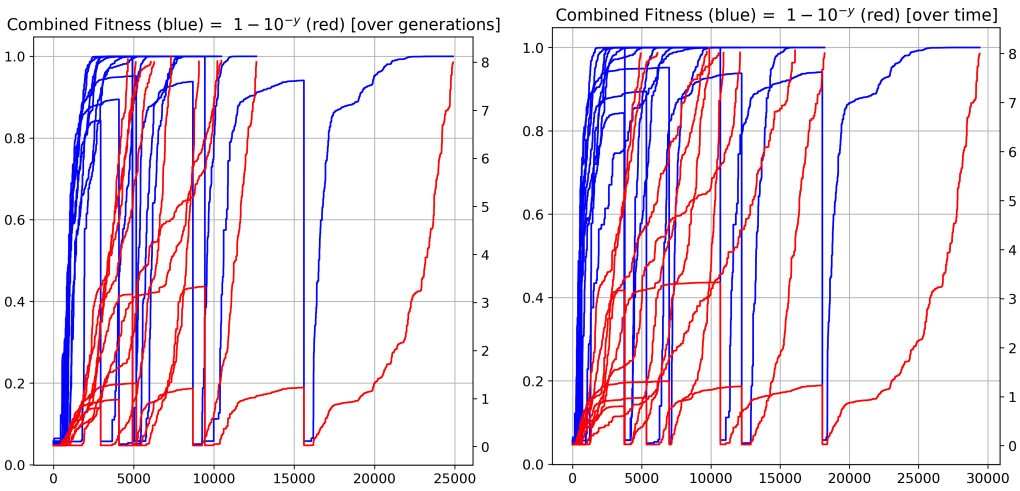


Figure 3.1: Fitness history of a few individuals over generations (left) and time (right), as absolute value (blue) or logarithmic tolerance (red, see section 2.5.2). Not all generations take the same amout of time (and hence have broadly different computational costs); this can be due to the integration method having to select a higher order or finer integration step with some parameters combinations, or to particularly unlucky CPU load patterns. Some outer optimization cycles are also clearly visibe (section 3.1.1)

improved more if given more processing time (when a "definitive" local fitness maximum is reached, the fitness grownth in logarithmical scale also stagnates, as can be seen in the failed attempts from figure Figure 3.1).
With the chosen optimization parameters, an individual takes an average 13859 generations to converge to the desired precision. With a pool of 90 competitors, we can assume a slightly pessimistic estimate of about one and half a million fitness evaluations per individual. Each fitness evaluation involves the integration of 6 distinct systems of equations, for which we require a minimum of 50 time steps. The computational cost, ignoring the implementation details of the BDF algorithm used to integrate, the asymptotic stability checks and the mean square error computation, is in the order of 10^9 matrix multiplications per individual. As shown in the graph, the machines that were used for fitting the models (a 16-core `AMD Ryzen 9 5950X`, a 14-core `intel i9 1200H`, and a 6-core `intel i7 3930k`) take on average 5.87 hours to fit each individual. It is to be noted that, despite each generation being embarrassingly-parallelizable, the computation of 90 fitness measures distributed over sixteen 4 GHz cores is still very fast, and the CPU occupancy can be low because of the synchronization required in between two subsequent generations; the same time per

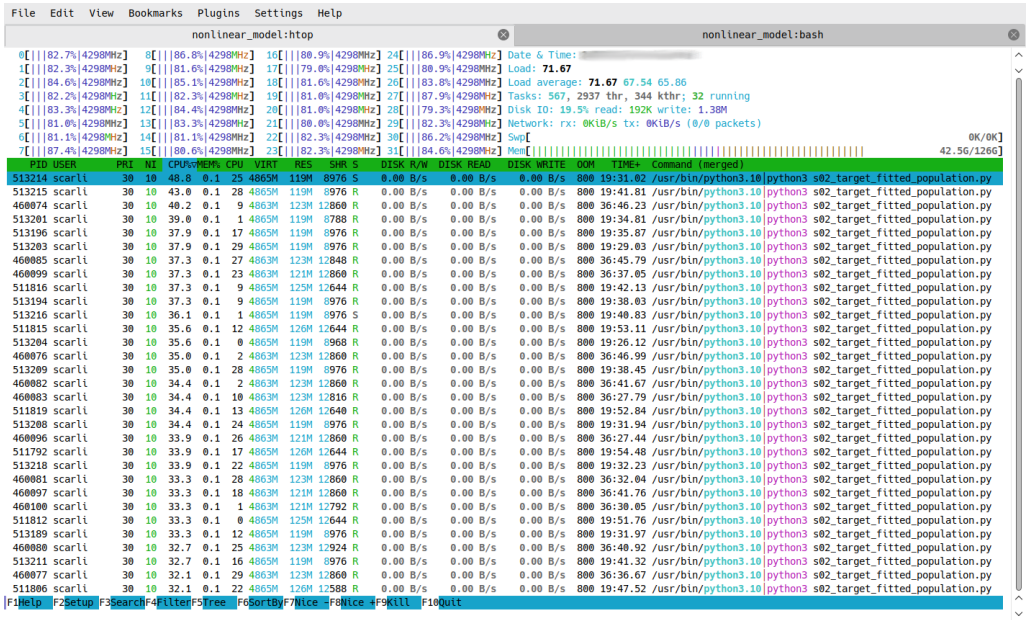


Figure 3.2: Example of machine load while optimizing three subjects in parallel.

individual therefore can still hold when optimizing more individuals in parallel on the same machine, which in turn can bring the CPU occupancy up to about 100% constant (Figure 3.2). Optimizing more individuals in parallel may of course penalize the optimization time of some individuals, but yields a better throughput overall.

Conceptually, each individual's optimization could be made faster by parallelizing the six integrations which compose the fitness computation. However, since the computational cost of a single integration is relatively small, the process creation overhead takes over as the most computationally expensive part of the task, effectively making that approach ineffective; parallelization at two higher levels (single competitor within an individual and individuals) proved to be the most efficient approach. Arguably, parallelizing only at the highest (individual) level would be the most computationally efficient approach; given the exploratory nature of this work, however, also minimizing waiting times for each individual's optimization was a priority: partial results could be examined with much shorter waiting times while testing and evaluating multiple models and configurations.

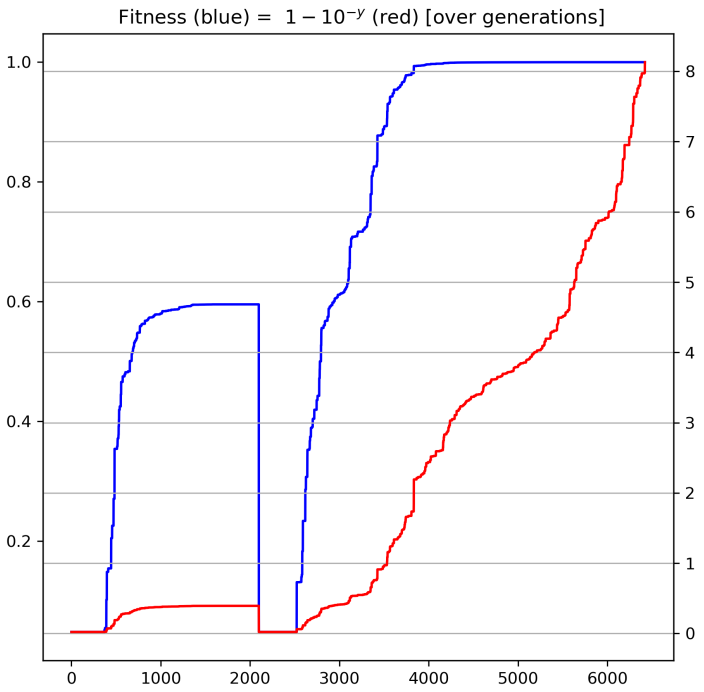


Figure 3.3: An example of the optimizer's sensitivity to the random number generator seed. This particular individual's optimization terminated after 2098 iterations for lack of variance among the competitors, stuck in a low local minima. The second cycle reset the initial point to be the same as the first cycle but used a different seed: this time, there is convergence.

3.1.1 Reproducibility and the need for outer optimization cycles

Reproducibility is essential. In the case of this work, which involves a fair amount of random processes, reproducibility have been ensured by using a fixed seed for every pseudo-random number generator: the target values (generated as described in section 2.6.5) as well as all optimization results can be consistently reproduced (provided one uses exactly the same seeded random number generation algorithm, of course).
Unfortunately, Differential Evolution has empirically proven to be sensitive to the seed used: the optimization of the same individual's parameters may succeed with one particular seed, but converge really slowly or completely fail to converge with another. This effect have been mitigated by exploiting the early stop

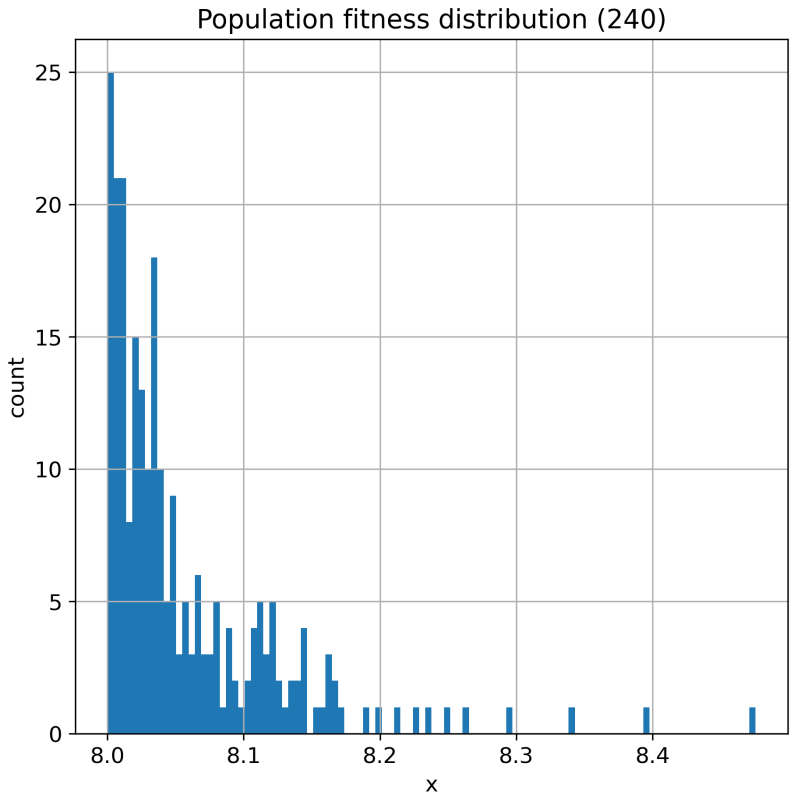


Figure 3.4: The final fitness distribution of the models, expressed as $f = 1 - 10^{-x}$. The stopping criterion of $f \geq 1 - 10^{-8}$ was often exceeded within the last generation.

condition on the competitors distance: when the average distance between all competitors becomes smaller than a set tolerance during an optimization generation, differential evolution can be stopped early independently of the fitness value reached; in fact, when the entire population of competitors have converged to similar vectors the algorithm is stuck in a minimum, either local or, hopefully, global. In this particular case, it has proven overall advantageous to stop the optimization early, with a relatively big distance tolerance (hence not wasting too much time without any significant fitness improvement), and instead restart it with different random seeds, de-facto implementing an outer optimization cycle around the original differential evolution algorithm. An example of the efficacy of this strategy is shown in Figure 3.3. The outer cycles, when present, can clearly be seen in the graphs of Figure 3.1: the growth of the fitness slows down to a stop. The competitors variance diminishes until it reaches the set threshold, at which point the optimization restarts hence the fitness also drops back to the initial value, and subsequently starts rising again as the algorithm progresses. It has experimentally proven advantageous to com-

pletely abandon a badly fitting solution stuck in a local minima and restart the optimization from the original initial point instead of keeping the local minima solution as a starting point for the new optimization cycle.
Finally, 240 models have been optimized to fit the corresponding subjects target values with the desired fitness value $f \geq 1 - 10^{-8}$, as summarized in Figure 3.4. All optimized models are exponentially asymptotically stable in all six lesion conditions, since the eigenvalues of the $\tilde{A}$ matrix ((2.4.30) [p.53]) all have strictly negative real parts.

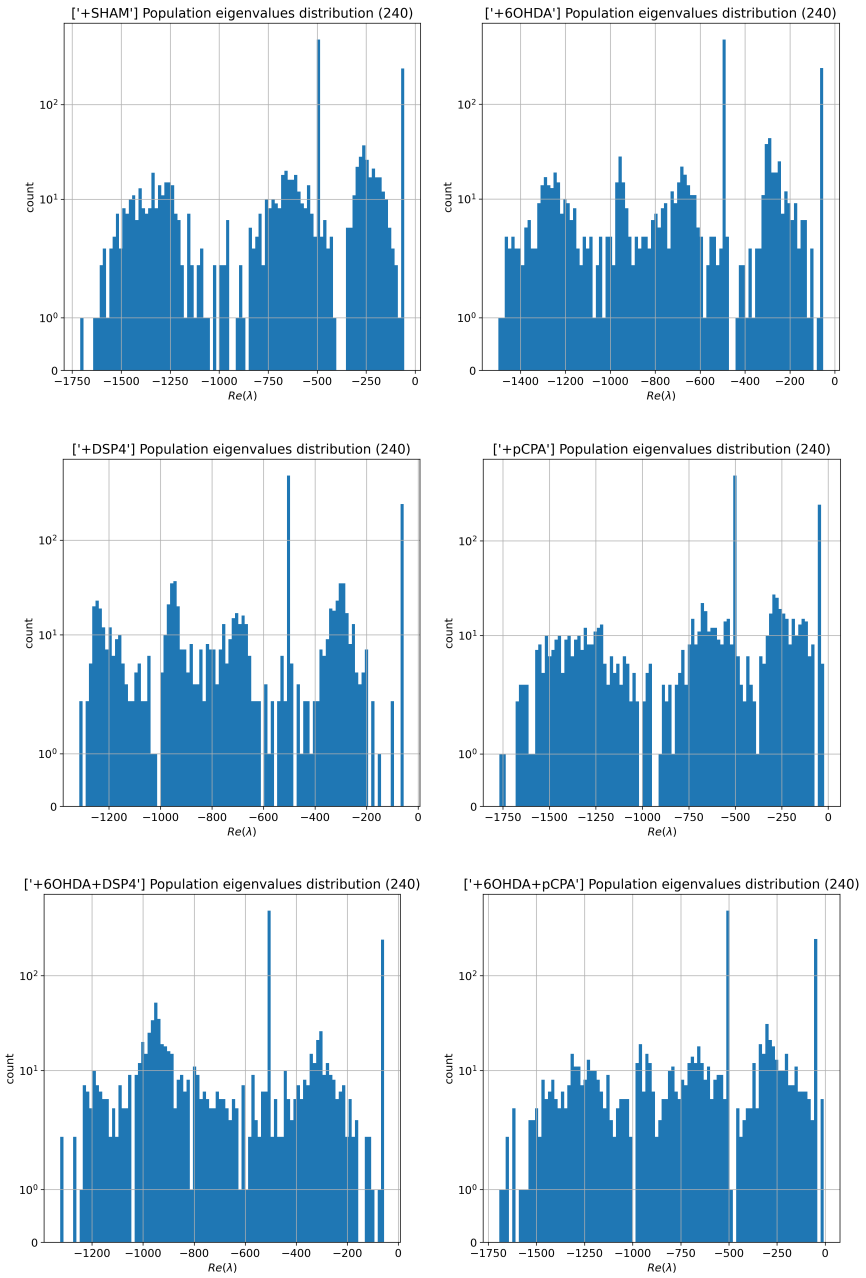


Figure 3.5: Distribution of the real parts of the eigenvalues of $\tilde{A}$ (2.4.30) [p.53] for all the optimized subjects and the six lesion groups. All the eigenvalues have negative real part, hence all the subjects are exponentially asymptotically stable in all cases, as enforced by the fitness measure component (2.6.32) [p.72].

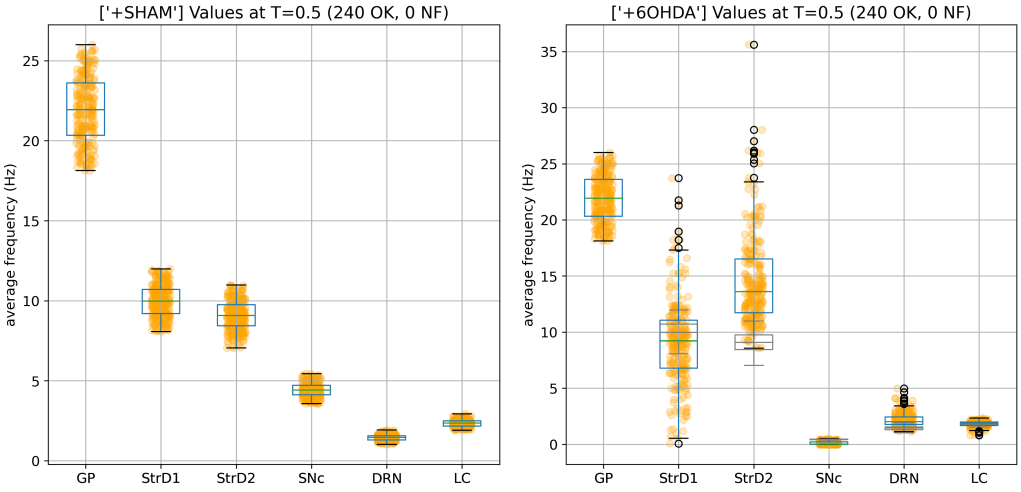


Figure 3.6: Comparison of the distribution of the simulation final value of each area for SHAM (left) and lesion 6OHDA (right) over the whole population. The grey boxes represent the reference solution values distribution while the blue ones describe simulated results. All areas overlap in the SHAM case as required by the fitness measure; in the 6OHDA case only GP and SNc overlap as they are exactly part of the fitness measure, LC is subject to an upper constraint, and the other areas are all predictions.

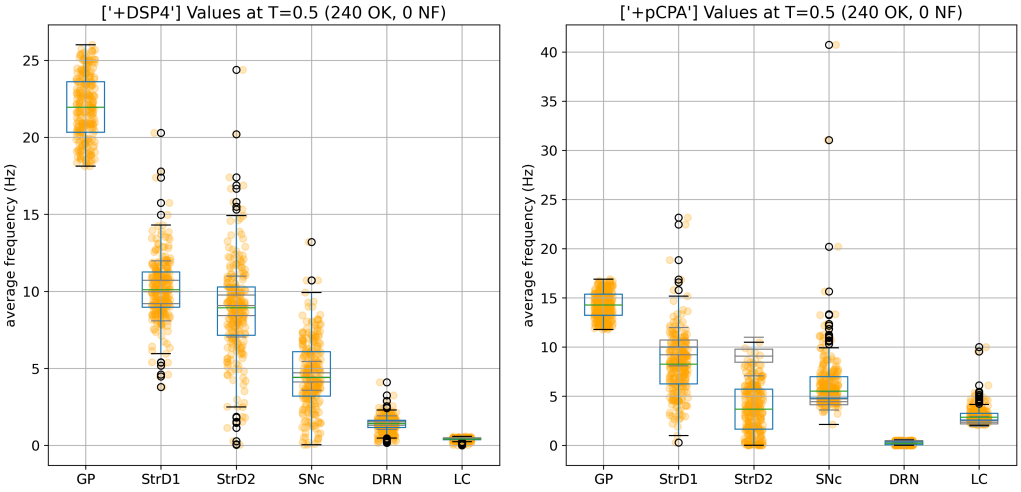


Figure 3.7: Comparison of the distribution of the simulation final values of each area for lesions DSP4 (left) and pCPA (right).

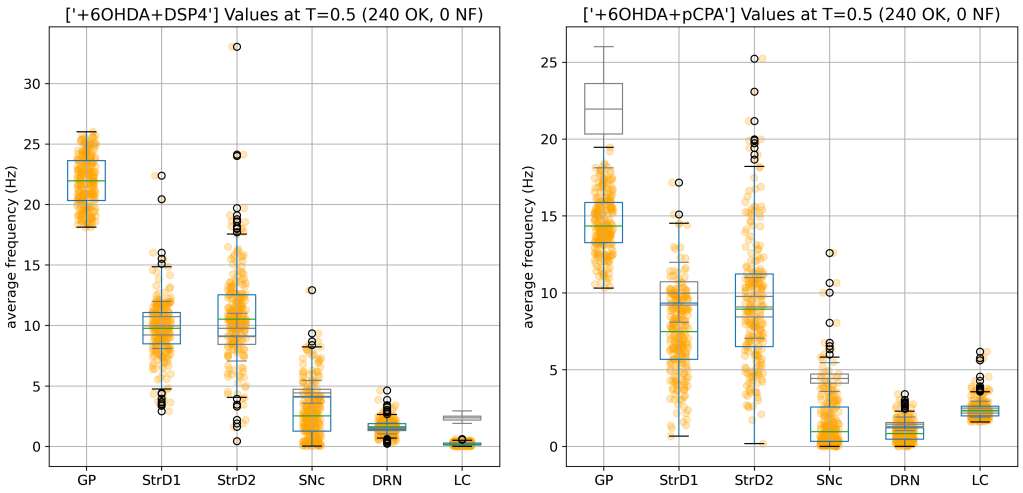


Figure 3.8: Comparison of the distribution of the simulation final values of each area for the lesion combinations 6OHDA+DSP4 (left) and 6OHDA+pCPA (right).

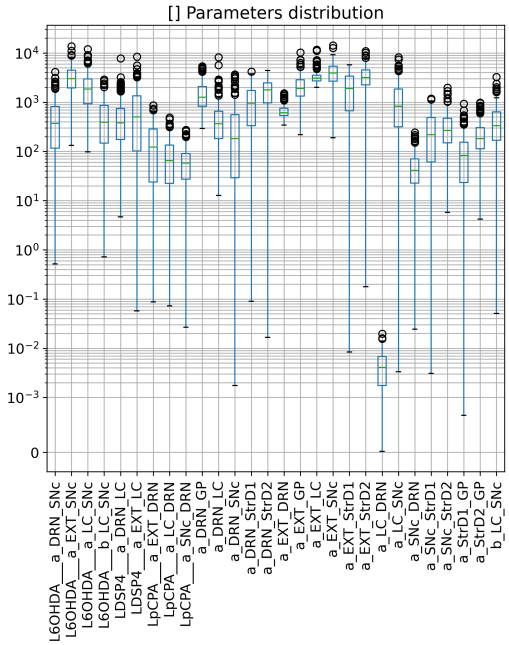


Figure 3.9: Distribution of parameters over the fitted population. The parameters starting with a lesion name are the ones that replace the parameter with the same name when lesions are applied.

3.2 Target data reproduced by the simulation

As described by Figure 3.4, all models fit the corresponding target values with a fitness $f \geq 1 - 10^{-8}$ measured using (2.6.36) [p.73]; since the measure is dominated by the smallest fitness value being combined, the mean square error (as defined in (2.5.22) [p.65]) of each component of the solution from its reference value can certainly be estimated to be smaller than 10^{-6}, which is a suitable precision for the purposes of this work. Figure 3.6, Figure 3.7 and Figure 3.8 show the final values of the whole population for all areas and all six groups. In particular, the grey box plots represent the target distribution of solution values while the colored ones represent the distribution of the simulated solution values for each group. The simulated values for the SHAM case overlap the target values (with the error mardin described earlier) for all six areas, since the values for all areas are part of the fitness measure. In the lesion groups, the limit constraints on the lesioned areas are clearly visible; all other areas are instead purely predictions. Figure 3.5 show the distribution of the eigenvalues of $\tilde{A}$ in the six cases; the fitness measure successfully guided the optimization algorithm towards an asymptotically stable solution.

3.3 Model parameters distribution and typical solution behaviour

The search space for all parameters has been equally restricted to the range $[0, 10^5]$; all parameters have been defined to be positive numbers in the formulation of the model (hence the lower limit is zero) while the upper limit is arbitrary (although much higher figures would not carry physiological meaning since a brain area certainly can not change its average activation frequency infinitely fast, excluding the case of a sudden catastrophically traumatic event which we are not considering in this study). Figure 3.9 illustrates how the parameters of each model distribute among the entire fitted population. Figure 3.10, Figure 3.11 and Figure 3.12 show instead the dynamic behaviour of one of the simulated subjects in all conditions: every graph starts with the model in the healthy state, then a lesion is applied. In this model, the lesion is instantaneous, and the transients show the transition of each area to its new equilibrium point determined by its time constants and strength of interaction with the other areas. As expected from the simulation time choice motivated in 2.6.6, 0.1s is usually enough to see a stabilization of the dynamic behaviour.

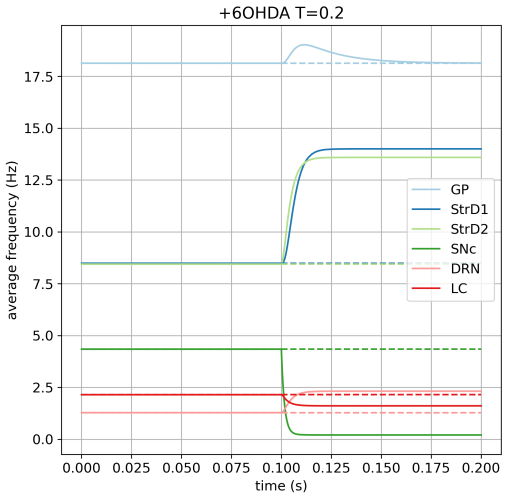


Figure 3.10: Example of the behaviour of a healthy subject's brain areas, when a dopaminergic lesion is applied at T=0.1.

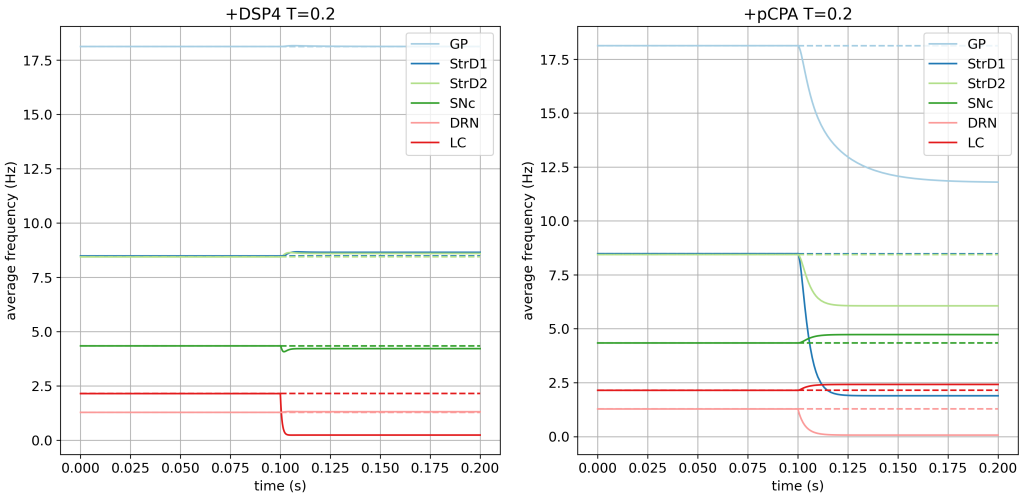


Figure 3.11: Example of the behaviour of a healthy subject's brain areas, when a noradrenergic (left) or serotonergic (right) lesion is applied at T=0.1.

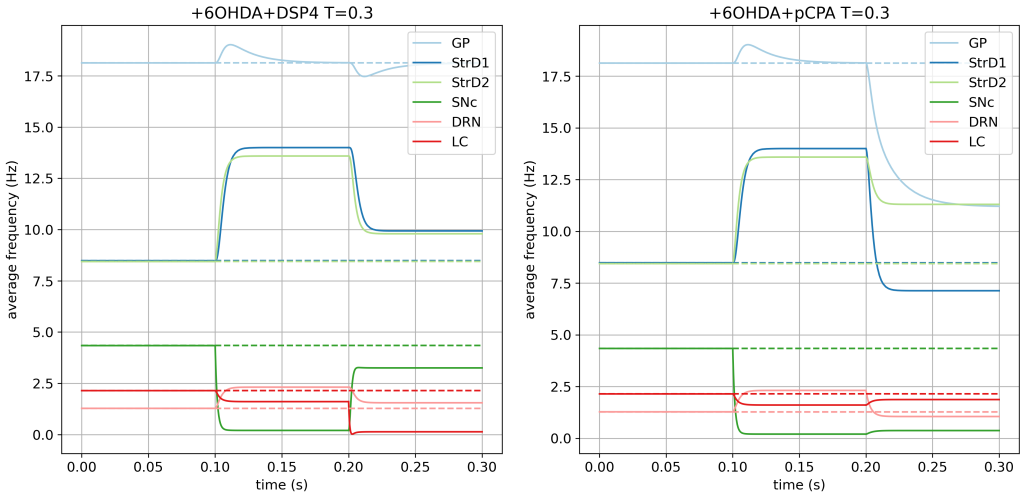


Figure 3.12: Example of the behaviour of a healthy subject's brain areas, when a dopaminergic lesion (applied at T=0.1) is combined with a noradrenergic (left) or serotonergic (right) lesion is applied at T=0.2.

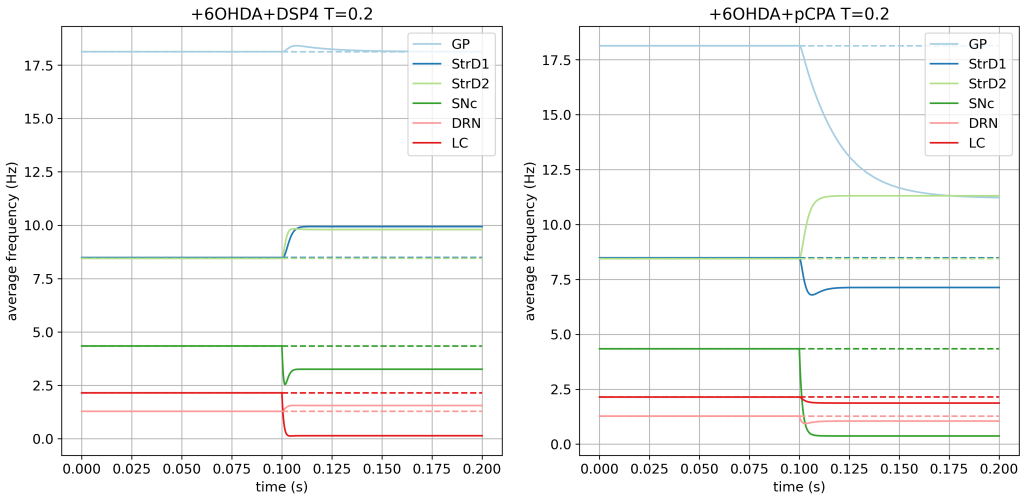


Figure 3.13: Example of directly applying a combined lesion. An equilibrium point is reached as in Figure 3.12, as explicitly required by the fitness measure (in particular, the rules in (2.6.24) [p.71]).

3.4 Comparing simulated results with experimental data

Figure 3.14 illustrates the distribution of the simulated values for GP in the six groups. When considering the whole population, the distributions for SHAM, 6OHDA, pCPA and DSP4 are identical but for center value and scale by definition, as imposed by the target data (section 2.6.5) and the fitness measure (2.6.36) [p.73]; the distributions for the combinations of lesions are instead more predictive in nature, since they are constrained by a range limit.
While it is possible to independently apply and measure different lesions on a simulated subject, this is not possible in experimental conditions: different lesions cannot be applied independently at different times since they are permanent, and some measurements unfortunately imply the death of the subjects being examined. Therefore, in real studies, the same subject can be measured only once and can belong to only one group.
To reproduce this conditions, we subdivide the simulated population in groups and use each virtual subject only once. The right side of Figure 3.14 shows an example of how using smaller, independent groups to measure each area affects the final distribution. Figure 3.15 to Figure 3.19 illustrate the distributions for all the other areas; the effects of each lesion and their combinations on the interested areas (SNc, DRN and LC) is clearly visible. Figure 3.20 gives a less quantitative and more qualitative overview of the same data, this time presented as histograms with an associated standard error and statistical significance. Finally, Figure 3.21 (left) presents the same data as Figure 3.14 (right), but offers a direct comparison with the experimental measurements presented in [52] (right).

All the error bars of histograms in this work are set to represent the standard (mean) error ([81; 82]):

$$E = \frac{S}{\sqrt{n}} \tag{3.4.1}$$

where S is the estimated standard deviation:

$$S = \sqrt{\frac{\sum (X - \bar{x})^2}{n-1}}, \quad X \in \{\text{samples}\}, \quad |\{\text{samples}\}| = n \tag{3.4.2}$$

3.4.1 Statistical significance

As mentioned in the previous section, to have a statistic that is comparable with the ones found in literature, each synthetic subject must be treated as if they were real and can therefore only belong to one group and be measured only once.

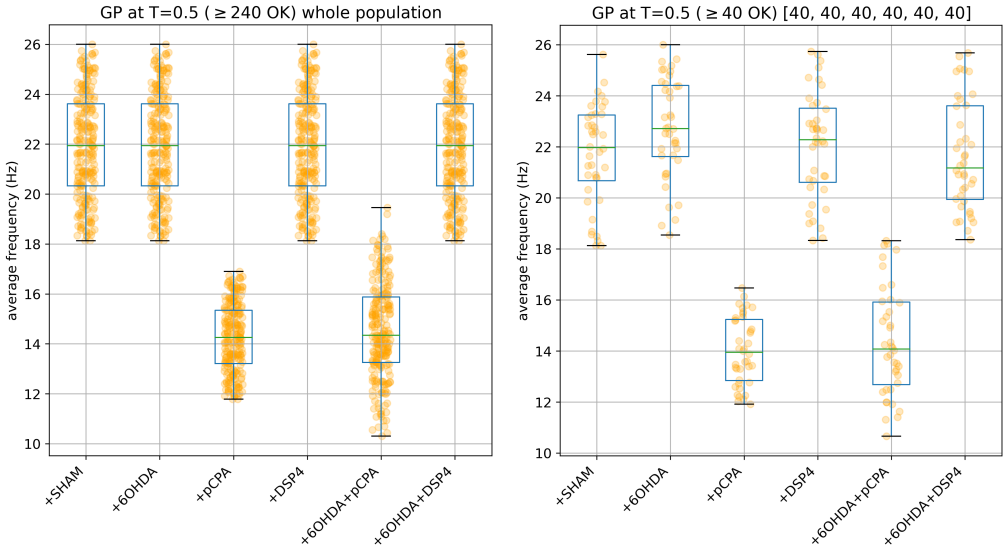


Figure 3.14: Distribution of the simulated equilibrium points for the GP area. Values for SHAM, 6OHDA, pCPA and DSP4 are fitted exactly on the required value, the combinations are instead predictions, with the values being only range-constrained with the rules F^{COMB} ((2.6.24) [p.71]). On the left the distribution is over the whole population (synthetic result), on the right, the population is sliced to have each subject in only one group (which reproduces real laboratory conditions, one subject can only be measured once, and belongs to one group only (section 3.4.1)

It can be insightful, however, to sometimes look at a broader picture where all subjects are measured at the same time; each graph title in this work, when relevant, declares if it is representing values of the whole population, of if the population was split in even groups (in which case, it reports the number of subjects in each group).
Statistical significances are always computed on split groups.

Statistical significance is assessed through the analysis of variance (ANOVA) tests, which determines if the means of two or more sample groups are statistically significantly different through a synthetic index called F-value. Without delving deep into the details, this test is valid under the assumptions that:

- The samples are independent
- Each sample is from a normal distribution
- The groups have equal standard deviations.

Let G be the set of groups and $x_{g,i}$ the i-th sample in group g and $\bar{x}_g$ the mean

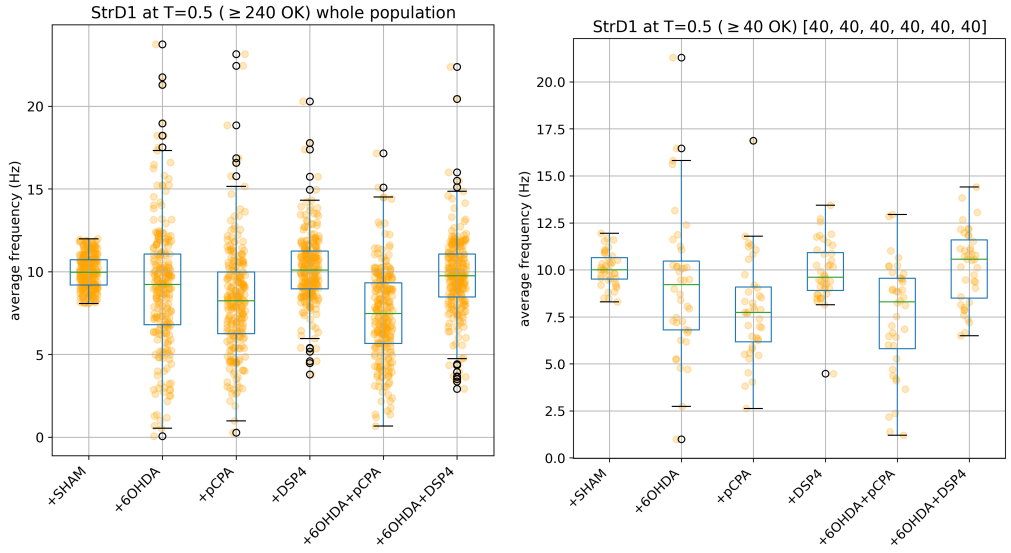


Figure 3.15: Distribution of the simulated equilibrium points for the StrD1 area, as in Figure 3.14.

value of group g. The sum of squares within groups (SSW) is defined as:

$$SSW = \sum_{g \in G} \sum_{i} (x_{g,i} - \bar{x}_g)^2 \tag{3.4.3}$$

therefore, it is the sum of the square differences of the samples of each group compared to the mean value of the group. The sum of squares between groups (SSB) is defined as:

$$SSB = \sum_{g \in G} (\bar{x}_g - \bar{x})^2, \quad \bar{x} = \frac{\sum_{g \in G} \bar{x}_g}{|G|} \tag{3.4.4}$$

therefore, it is the sum of the square differences between the mean of each group and the grand mean. We now define the degrees of freedom between (dof_B) and within (dof_W) groups:

$$dof_B = m - 1, \quad dof_W = n - m, \quad m = |G|, \; n = \sum_{G} |g| \tag{3.4.5}$$

hence n is the number of groups and n is the total number of samples. We can finally compose the F-value:

$$F = \frac{SSB/dof_B}{SSW/dof_W} \tag{3.4.6}$$

The statistical significance is then assessed by comparing the found F-value to a critical value of the Fisher density distribution function, which is different for

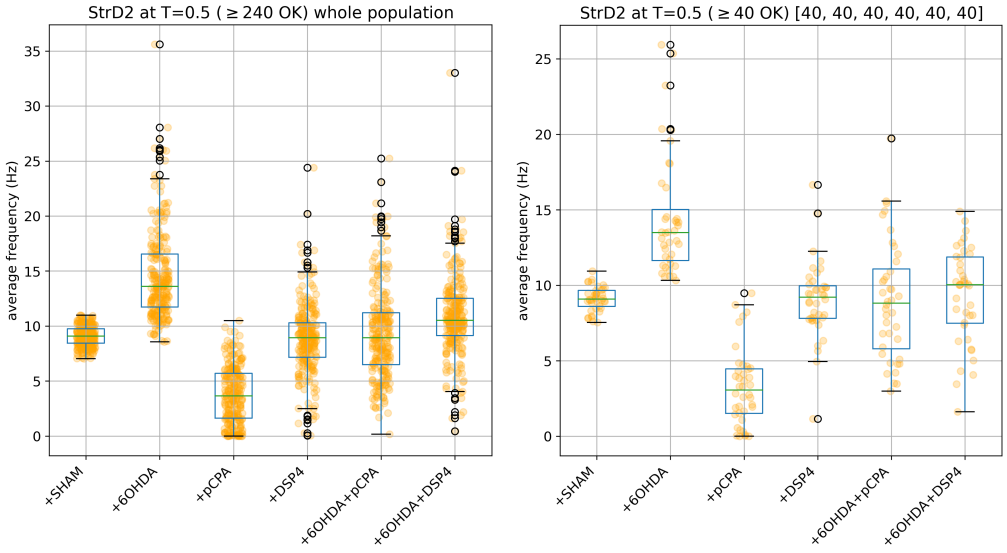


Figure 3.16: Distribution of the simulated equilibrium points for the StrD2 area.

each set of degrees of freedom and wanted confidence. When the F-value indicates a significative result, a p-value must also be computed ($P(F_0 < X < \infty)$) to assess the reliability of the result (Figure A.2 shows the relationship between F- and p-values).
If the ANOVA test is significative, at least one of the groups is significantly different from the others. All group combinations must be now tested in pairs to infer which group is distinguishable by which other; in particular we can apply Tukey's test, which performs pair-wise tests, conceptually similar to t-tests or ANOVA, but scales the results to take into account the existence of all the other groups.
Fisher's functions and the associated p-values may result fairly complex to compute, but there are many available open-source tools specialized in that. In this work we relied on python's `scipy` implementation of one-way-ANOVA and Tukey's tests ([83; 84]). Detailed analysis and discussions on the analysis of variance here just briefly described are in [81; 85; 82; 86; 87; 88; 89].
Histograms in this work have been marked with statistical significance according to the following table:

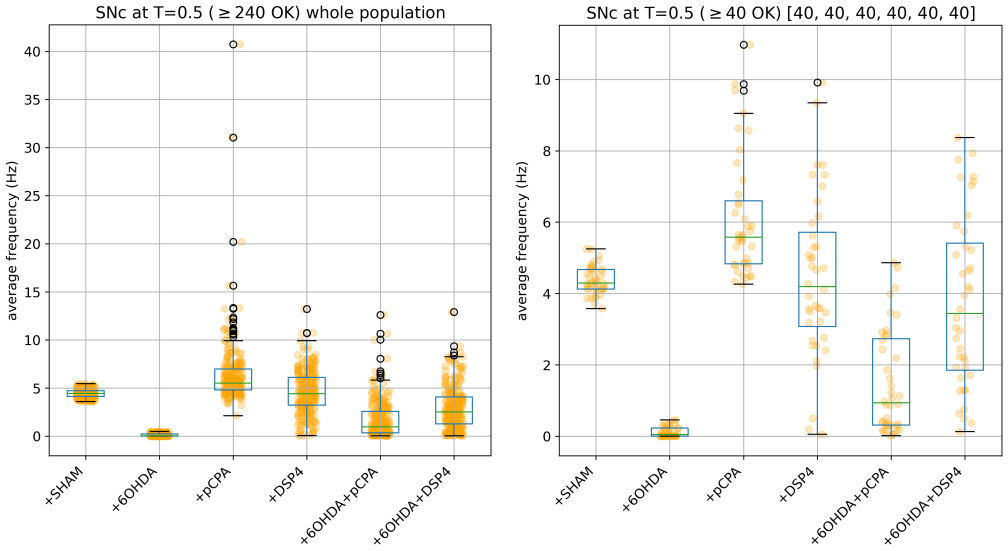


Figure 3.17: Distribution of the simulated equilibrium points for the SNc area, which is affected by the 6OHDA lesion.

mark	meaning
****	p-value ≤ 0.0001
***	p-value ≤ 0.001
**	p-value ≤ 0.01
*	p-value ≤ 0.05
no mark	p-value > 0.05

where the leftmost column, which usually refers to the SHAM case, is used as the comparison base.

3.4.2 Empirical sensitivity analysis

The sensitivity of each simulated brain area with respect to each parameter of the model can be empirically estimated: Each parameter can be varied independently (within some acceptable range) to record its effects on the simulated brain areas; similar observations can be replicated for all individuals of the available population, and the average excursion of each area can then be compared with the average excursion of the parameter to infer a sort of "parameter im-

```
Groups: 0:+SHAM 1:+6OHDA 2:+pCPA 3:+DSP4 4:+6OHDA+pCPA
↪  5:+6OHDA+DSP4
ANOVA: F=1.734e+02, p=1.403e-76, dofB=5, dofW=234
Tukey's HSD Pairwise Group Comparisons (95.0% Confidence
↪  Interval)
Comparison  Statistic   p-value     Lower  CI   Upper CI
 (0 - 1)   -1.008e+00   1.969e-01  -2.267e+00   2.497e-01
 (0 - 2)    7.705e+00   0.000e+00   6.447e+00   8.963e+00
 (0 - 3)   -3.572e-01   9.644e-01  -1.615e+00   9.009e-01
 (0 - 4)    7.365e+00   0.000e+00   6.107e+00   8.623e+00
 (0 - 5)    8.243e-02   1.000e+00  -1.176e+00   1.341e+00
 (1 - 2)    8.714e+00   0.000e+00   7.456e+00   9.972e+00
 (1 - 3)    6.512e-01   6.727e-01  -6.069e-01   1.909e+00
 (1 - 4)    8.374e+00   0.000e+00   7.116e+00   9.632e+00
 (1 - 5)    1.091e+00   1.308e-01  -1.672e-01   2.349e+00
 (2 - 3)   -8.062e+00   0.000e+00  -9.321e+00  -6.804e+00
 (2 - 4)   -3.400e-01   9.712e-01  -1.598e+00   9.181e-01
 (2 - 5)   -7.623e+00   0.000e+00  -8.881e+00  -6.365e+00
 (3 - 4)    7.722e+00   0.000e+00   6.464e+00   8.981e+00
 (3 - 5)    4.396e-01   9.162e-01  -8.185e-01   1.698e+00
 (4 - 5)   -7.283e+00   0.000e+00  -8.541e+00  -6.025e+00
```

Table 3.1: Example of full ANOVA and Tukey's test applied to the GP value of the six lesion groups (on the same data as Figure 3.21 and Figure 3.14 (right)).

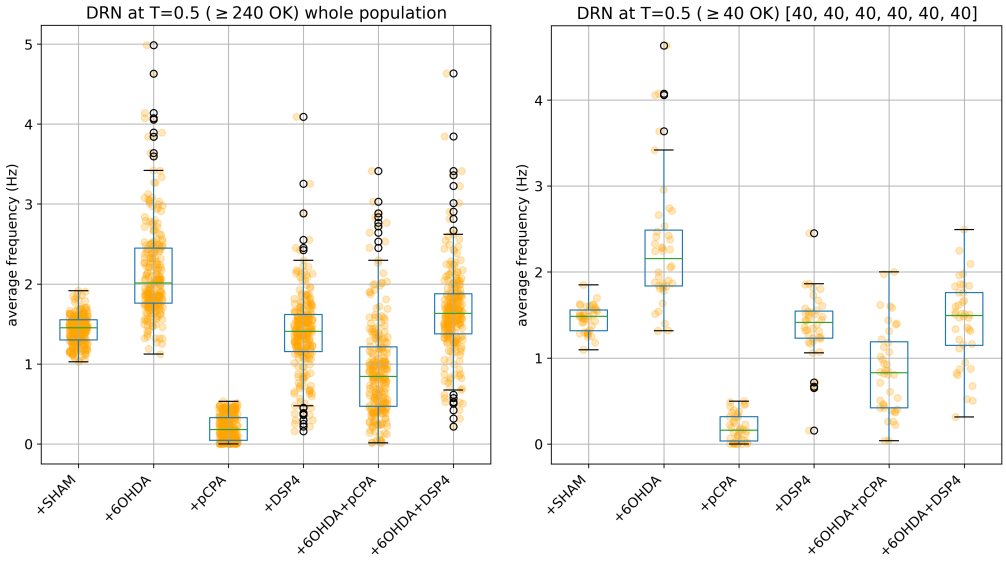


Figure 3.18: Distribution of the simulated equilibrium points for the DRN area, which is affected by the pCPA lesion.

portance".

In particular, for each individual of the population and for each parameter, we:

- Vary the parameter around its original value $\pm 50\%$ in 100 uniform steps: if v is this parameter's value for individual S_i^{SHAM}, we produce the set $V_{param,i} = \{(\frac{2i+97}{198})v\} \subseteq [0.5v, 1.5v]$, $i = 1, ..., 100$. We therefore have a $V_{param,i}$ set for each parameter of each individual.
- Simulate the model using each value in $V_{param,i}$ and save the final value for each brain area. If the simulation stops early (hence the simulation diverged or reached physically impossible states), the result is discarded and the parameter's value removed from $V_{param,i}$. For each parameter and each individual we therefore obtain six sets, one for each brain area, which we call $A^{area}_{param,i}$

The sets are then joined across the population:

$$A^{area}_{param} = \bigcup_i A^{area}_{param,i}, \quad V_{param} = \bigcup_i V_{param,i} \tag{3.4.7}$$

where $param \in S^{SHAM}$ is one of the free parameters as defined in section 2.6.1, $area$ is one of the six brain areas $\{GP, StrD1, StrD2, SNc, DRN, LC\}$, and i points to the $i-$th subject in the population.

A sensitivity index is then computed for each area by scaling both V and A by

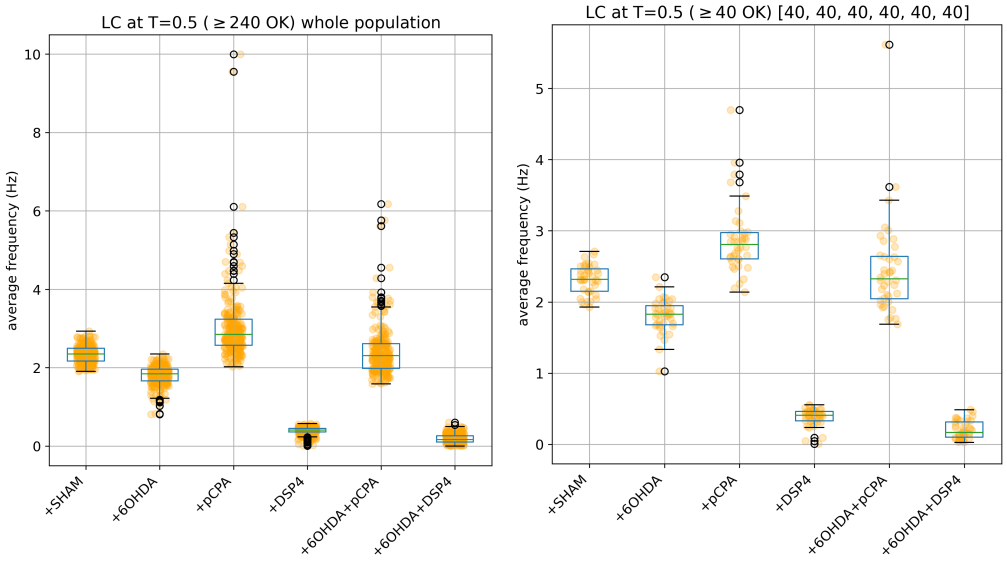


Figure 3.19: Distribution of the simulated equilibrium points for the LC area, which is affected by the DSP4 lesion.

their respective median values and dividing the standard deviations:

$$I_{param,area} = \frac{\text{std}(A^{area}_{param}/\text{median}(A^{area}_{param}))}{\text{std}(V_{param}/\text{median}(V_{param}))} \tag{3.4.8}$$

$I_{param,area}$ can naturally be seen as a *sensitivity matrix*, with one column per area and one row per free parameter.
As a last step, $I_{param,area}$ is normalized with respect to its maximum value.
Figure 3.23 shows the computed sensitivity matrix for the entire fitted population in the SHAM case; a value of 1 indicates the maximum measured sensibility, while a value of 0 would mean that a particular parameter has no effect on that area. It is evident from the matrix that all the areas are relatively sensitive to changes in noradrenalinergic balance (external activation of LC), and also to changes in the serotonergic balance (external activation of DRN). It is therefore reasonable to expect the stimulation of LC and/or DRN to produce changes in the activation of all areas.

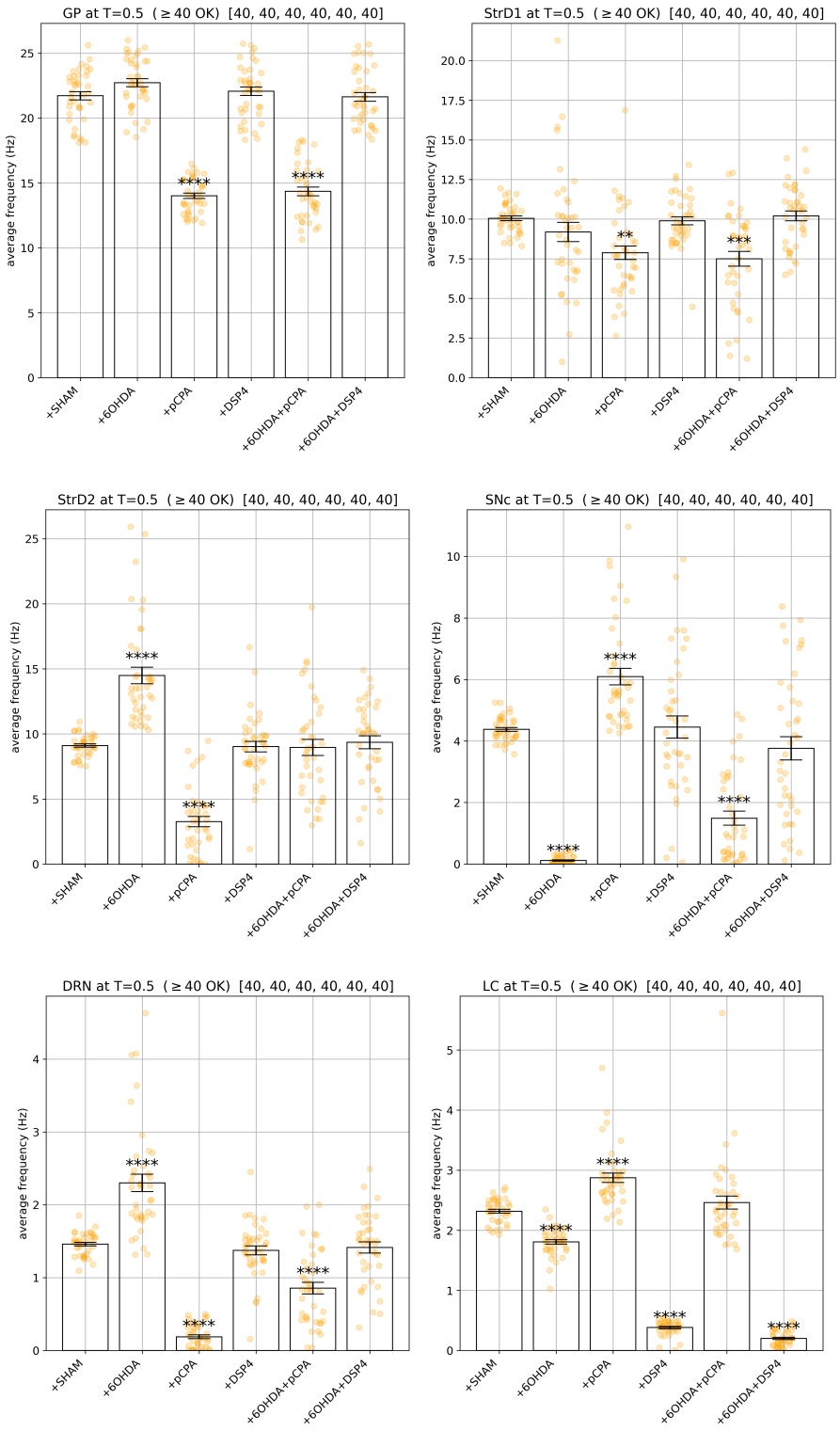


Figure 3.20: Overview of the behaviour of every area in all six groups, this time with the associated standard deviation and statistical significance of each group with respect to SHAM as described in section 3.4.1.

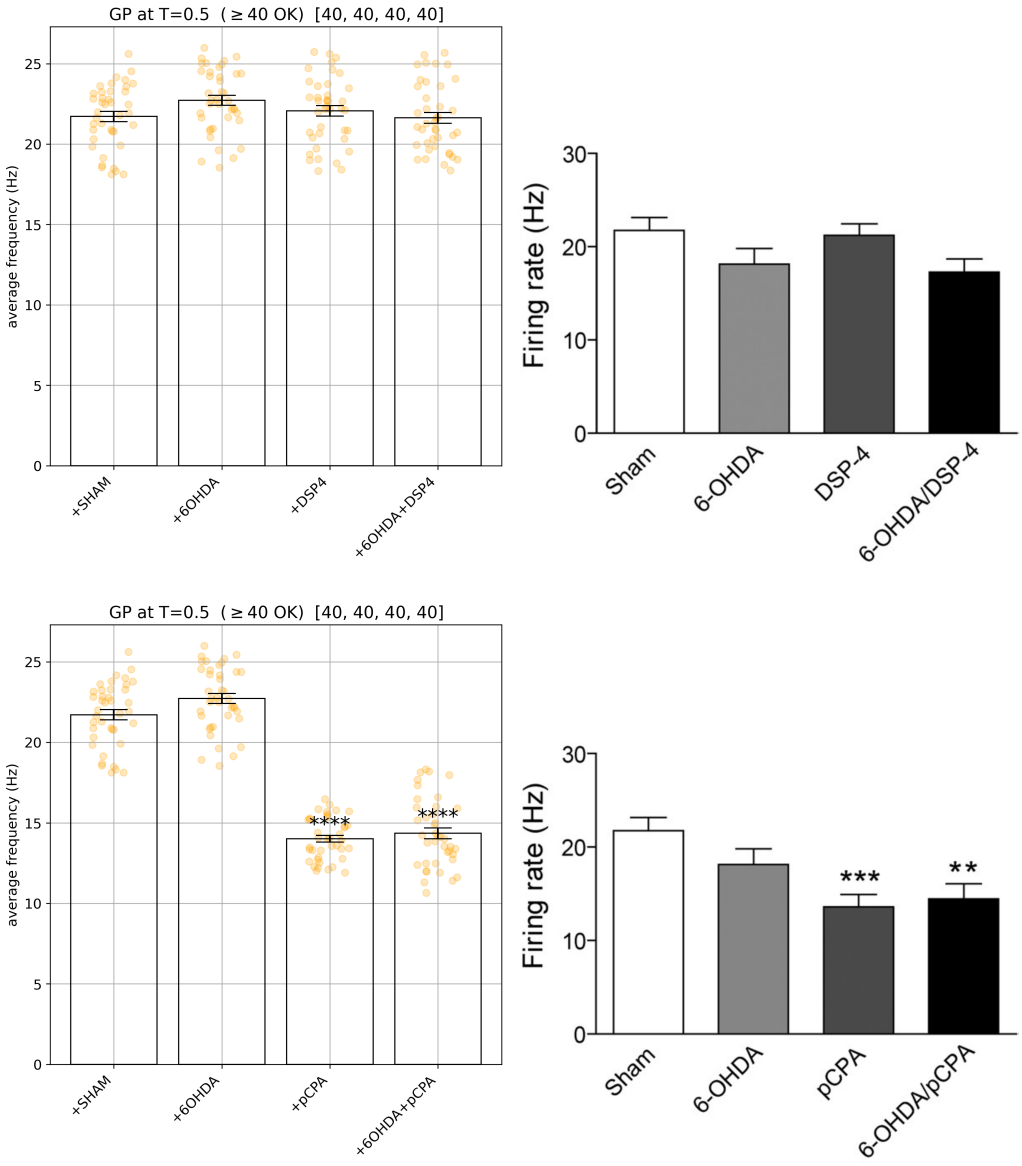


Figure 3.21: Results and predictions of the model (left) for the GP area in all six groups, directly compared to the measurements presented in [52] (right). Behaviour and statistical significance of the simulated groups are compatible with the measurements.

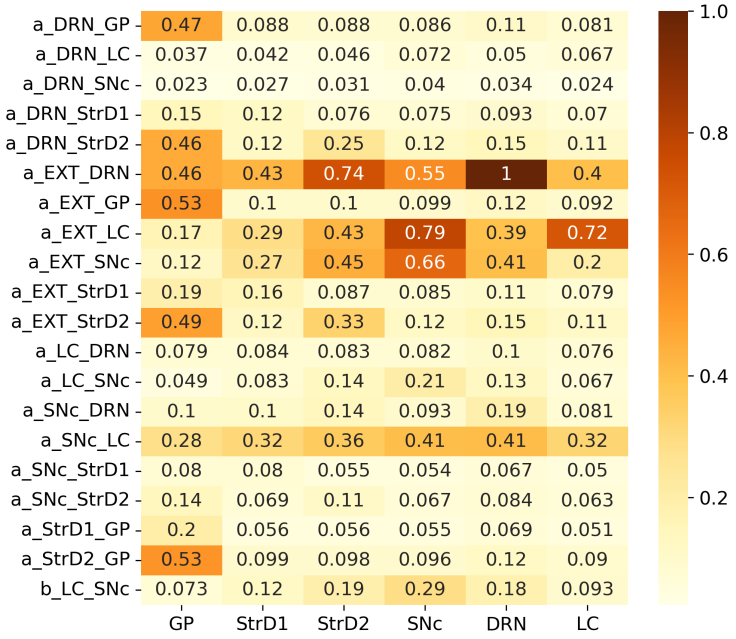


Figure 3.22: Relative sensitivity of each area with respect to every parameter in healthy subjects.

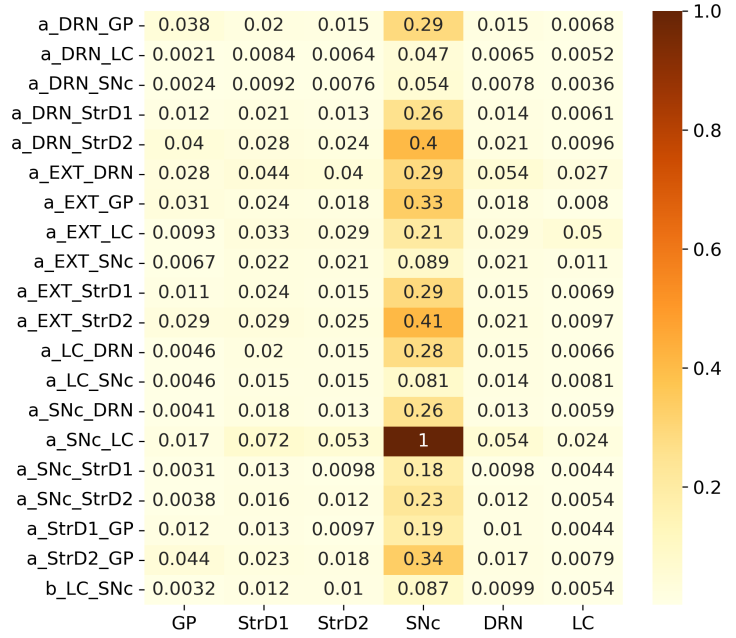


Figure 3.23: Relative sensitivity of each area with respect to every parameter in 6OHDA-induced parkinsonian subjects.

3.5 Possible road to a treatment?

Now that we have a model that reproduces available data, we can use it to make predictions of what can be expected to happen in the cases which have not been experimentally measured yet.

Comparing the brain areas activation levels distributions in the healthy SHAM to the 6OHDA-induced parkinsonism (Figure 3.24 and Figure 3.25) it is evident that the dopaminergic depletion also inhibits DRN and hence provokes a statistically significant serotonergic depletion. This behaviour is compatible with serotonin measurements reported in [52]. The administration of 6OHDA also inhibits the activation of LC (and hence noradrenaline production).
Once there is a state of parkinsonims due to a dopaminergic deficiency (in this case, due to a lesion of the substantia nigra pars compacta, SNc, caused by 6OHDA), could the be restored by acting on the other monoaminic circuits?
According to the model schema in Figure 2.2, excluding the SNc area directly affected by this drug, dopaminergic levels can potentially be altered in two ways:

1. Externally stimulate the locus ceruelus (LC) to change its production of noradrenaline
2. Externally stimulate the dorsal raphe nucleus (DRN) change its production of serotonine

The stimulation could either be chemical, by providing the area of the precursors needed to generate monoamines, or electrical, to artificially alter the average firing rate of the neurons from that area (and hence producing and projecting more monoamines to the areas which receive projections from the stimulated one).
According to the sensitivity matrix in Figure 3.23, although, it is reasonable to expect LC stimulation to be strongly influential on dopamine levels, but DRN stimulation should have a smaller effect on the activation of the SNc and strong side effects instead, which would not be compatible with a successful treatment.

3.5.1 Treatment optimization

Whether the stimulation of LC, DRN or both could potentially restore healthy levels of brain areas in parkinsonian subjects can be verified through the optimization of a subset of the parameters of our model. In particular, we can try to optimize the external stimulation parameter of LC, DRN or both in the 6OHDA version of our subjects.

First of all, we need to extend a subject's set of parameters S_i, as previously defined in section 2.6.1, with three new subsets of parameters, namely:

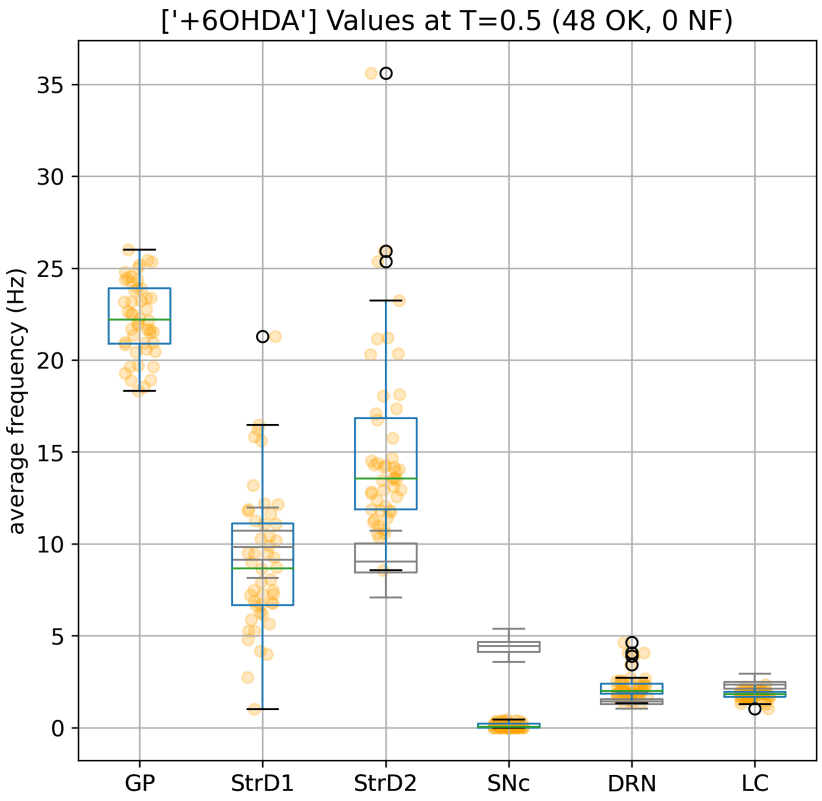


Figure 3.24: Effects of the dopaminergic (parkinsonian) lesion to SNc and LC induced by 6OHDA: SNc activation, and consequently the production of dopamine, are drastically lowered compared to the healthy (grey) levels. LC activity is also lowered to 80% of its SHAM value.

name	free parameters	corresponding fitness measure
6OHDA+cLC	$\alpha^{\text{ext}}_{\text{LC}}$	F^{cLC}
6OHDA+cDRN	$\alpha^{\text{ext}}_{\text{DRV}}$	F^{cDRN}
6OHDA+cCOMB	$\alpha^{\text{ext}}_{\text{LC}}$,$\alpha^{\text{ext}}_{\text{DRV}}$	F^{cCOMB}

As implied by the names, the corresponding model matrices $A, C, \mathbf{b}$ are constructed by using as base the 6OHDA (hence, parkinsonian) set of parameters for a subject and leaves as the only free parameters the external stimulation of the areas being tested.

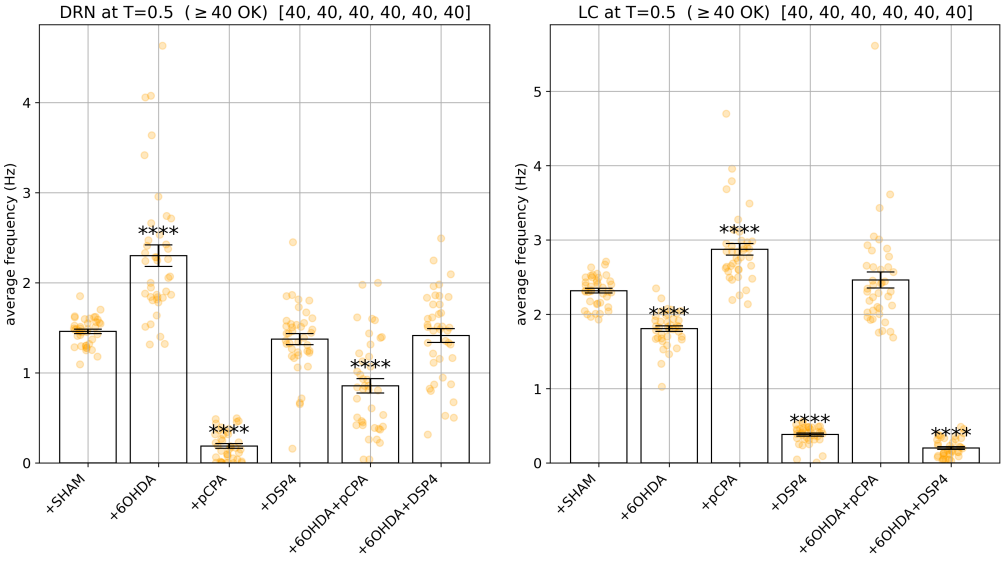


Figure 3.25: Effects of lesions on the dorsal raphe nucleus (DRN) and locus coeruleus (LC). Lesions to the substantia nigra pars compacta (SNc) by administering 6OHDA, which induces a parkinsonian condition, have a significant effect on the average activation levels of both DRN and LC.

The three populations need to have different fitness measures because they each stimulate a different area to simulate the treatment; the stimulated area must of course be ignored by the respective fitness measure.
Let us examine in detail the measure F^{cLC}. Similarly to the composed fitness measure previously described in section 2.6.2 for the base model, this measure is defined as the composition (using (2.5.20) [p.62]) of the following measures:

- The mean square error of the areas activation value, one measure per area, as defined for the SHAM case in (2.6.4) [p.68], but excluding the area being stimulated (in this case, excluding LC).
- A parameter constraint similar to the one defined in (2.6.30) [p.71], but this time used to enforce the external stimulus parameter to be equal or greater than the original once (hence forcing the optimization to choose a stimulation rather than an inhibition). In particular, the component is defined as:

$$f_{cLC}^{PAR} = \frac{1}{1 + max(0, S^{SHAM} - S^{cLC})}. \tag{3.5.1}$$

so that the fitness decreases if the area gets inhibited instead of stimulated.

- An asymptotic stability constraint as defined in (2.6.32) [p.72], where of course $\tilde{A}$ is constructed using the current parameters subset $S^{6OHDA+cLC}$.

The fitness measures F^{cDRN} and F^{cCOMB} are of course constructed in an analogous way; in the latter case, mean square errors for both stimulated areas are ignored in the measure.

The optimization is finally performed independently on all subjects of the three groups using the same algorithm described in 2.6.4, including the outer optimization cycles as described in section 3.1.1.

3.5.2 Treatment efficacy

Figure 3.26 shows that the optimizer could successfully restore healthy levels of the measured areas in the vast majority of subjects by stimulating LC or both LC and DRN, but it never succeeded by only stimulating DRN. It is also evident that a small proportion of subjects (less than 10%) could not be successfully treated in any of the cases; we will consider a fitness greater than 5 a success. Figure 3.27 shows that in the combined treatment, which obtained very similar results to the stimulation of LC alone, the relative increment to the external stimulation parameter of DRN is in fact several orders of magnitude smaller than the one applied to the corresponding parameter for LC; we can therefore assume that while the combined stimulation may have resulted in a slightly better fitness from the purely numerical perspective, DRN stimulation is indeed not useful as a treatment also in combination to LC stimulation.
Figure 3.28 clearly show that a statistically significant stimulation of LC is able to restore the healthy balance of serotonin and dopamine (the activation levels of DRN and SNc respectively) in 6OHDA-induced parkinsonian subjects. Figure 3.29 and Figure 3.31 illustrate the changes of distributions in the parameters space induced by the 6OHDA lesion and the subsequent treatment. The right side of Figure 3.31 highlights the differences in parameter distributions between the subjects that have been successfully treated (in green), and the ones whose levels could not be successfully restored (in red). None of the parameters of the curable subjects are significantly different from the one of the non-curable ones; the only parameter that shows a small significance difference is the sensitivity of SNc toward noradrenaline arriving from LC, as shown in Figure 3.30; however the spread of the distribution of that parameter is very large, and the value of that specific parameter alone is not useful for predicting if a subject is curable or not. An accurate statistical study of the parameters space would be necessary to determine if a particular combination of parameters could be used for predicting the curability of a subject.

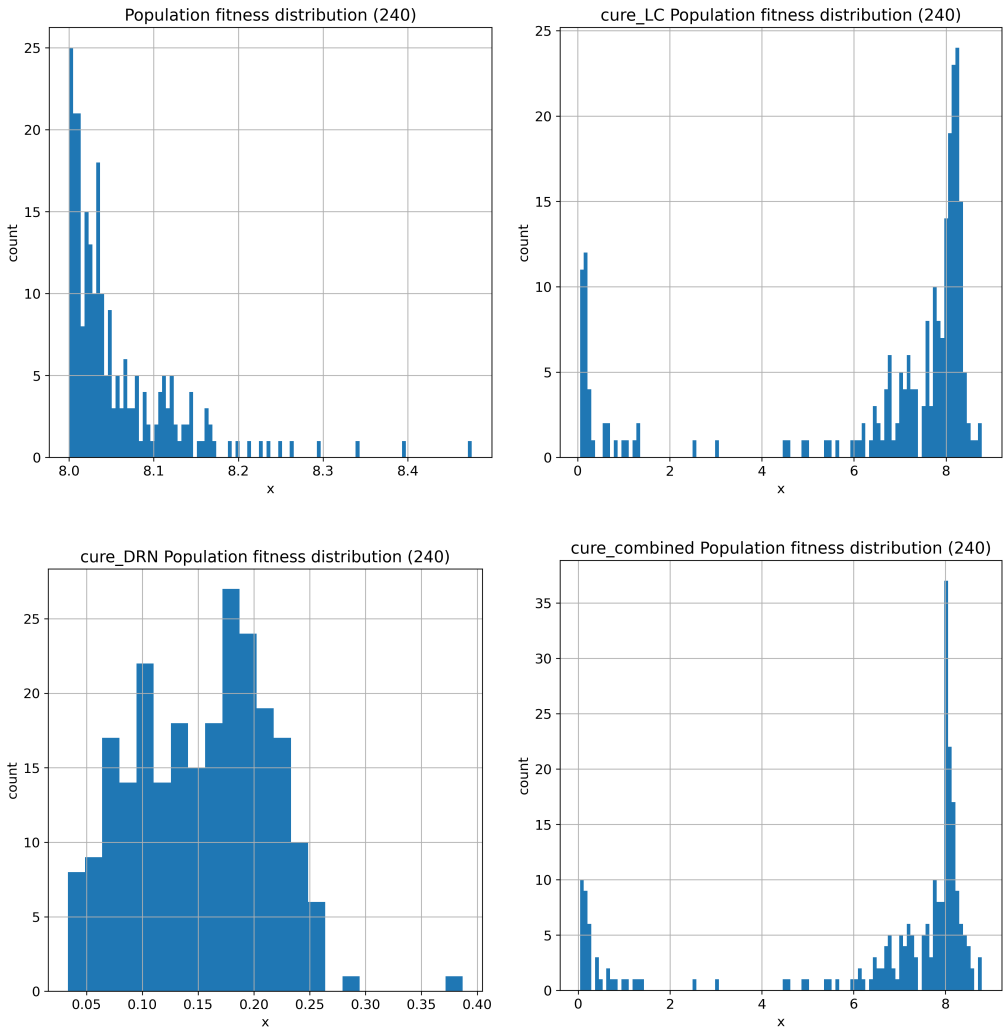


Figure 3.26: Distribution of the fitness of SHAM, cLC, cDRN and cCOMB after the optimization. It is evident that the optimizer never reached a good fitness by only stimulating DRN, while it got similar results when stimulating only LC or both areas.

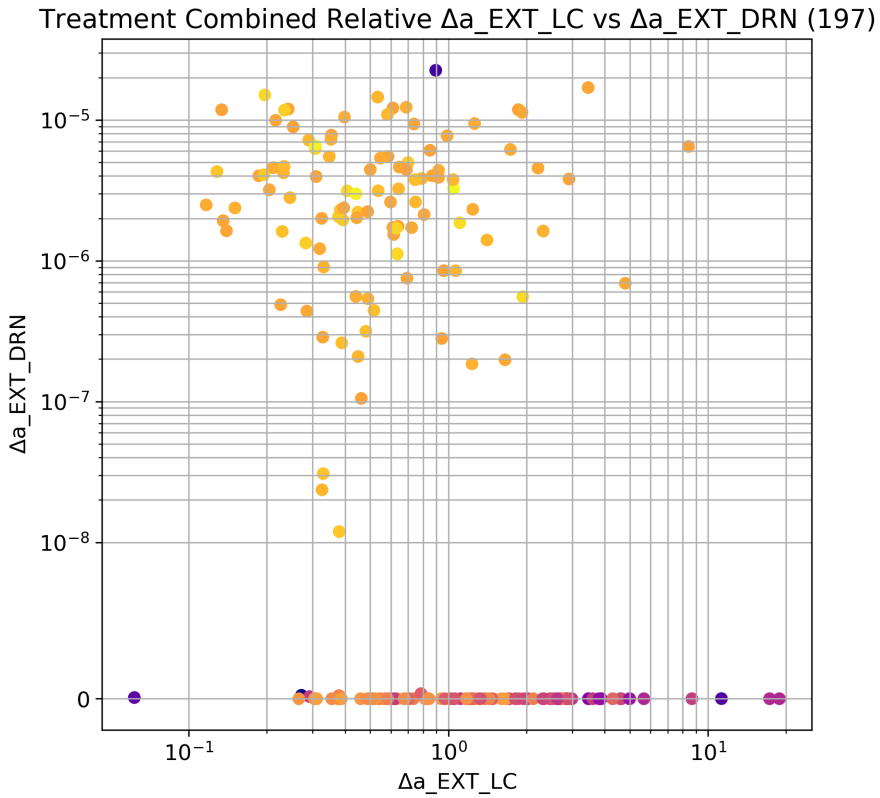


Figure 3.27: Relative increment applied by the optimizer to the external stimulation parameter of LC and DRN in the combined case. The increments to the DRN stimulation are several orders of magnitude smaller than the ones applied to LC; moreover, as shown in Figure 3.26 the sole stimulation of DRN is not a viable treatment. The small changes applied to the DRN stimulation by the optimizer may therefore have contributed to a numerically better solution, which is however not substantially different from the one obtain by the sole stimulation of LC. The subjects which did not reach a fitness of 5 (and hence are not to be considered successfully treated) have been excluded from this plot. The color scale gives an indication of the final fitness reached by the optimizer, blue is the lowest and yellow the highest.

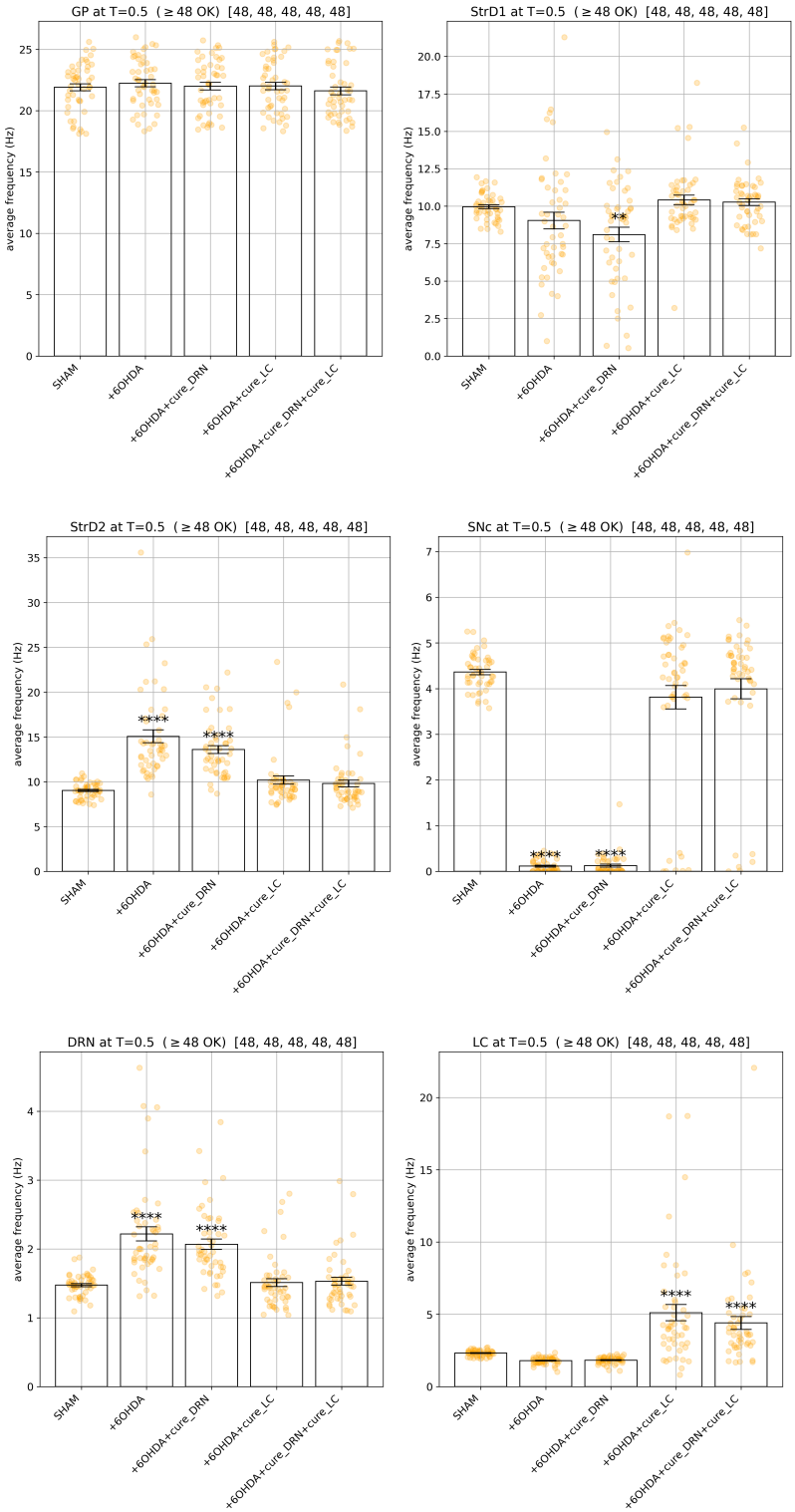


Figure 3.28: Lesion and treated values for all areas. A statistically significant boost of LC average activity (and hence of noradrenaline levels) can restore the activity (and hence monoamine production levels) of all the areas that were significatively impacted by 6OHDA to SHAM levels.

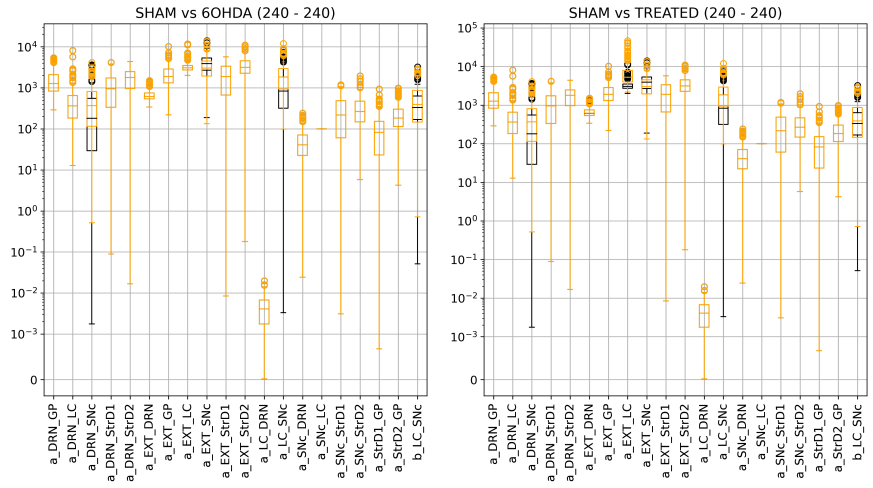


Figure 3.29: Distribution comparison of model parameters, SHAM subjects in black, 6OHDA (left) or treated (right) subjects in orange. As defined in section 2.4.4, only the four parameters which affect the SNc equation change in the 6OHDA case with respect to SHAM. The treatment also modifies the external stimulation to LC and DRN.

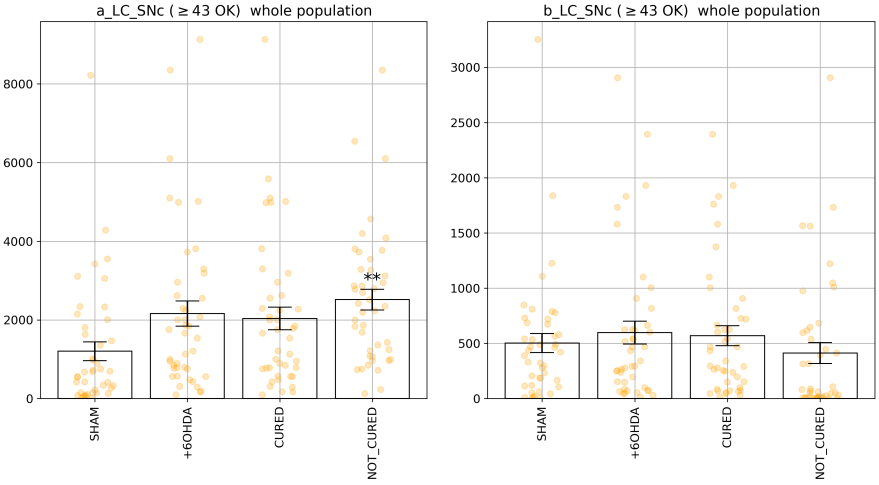


Figure 3.30: The only feature that differentiates, albeit with low significance, individuals for which it was possible to find a treatment stimulating either LC or DRN is the sensitivity of SNc to noradrenaline from LC. Figure 3.31 also shows the parameters-space comparison of 'curable' and 'non curable' individuals for all parameters.

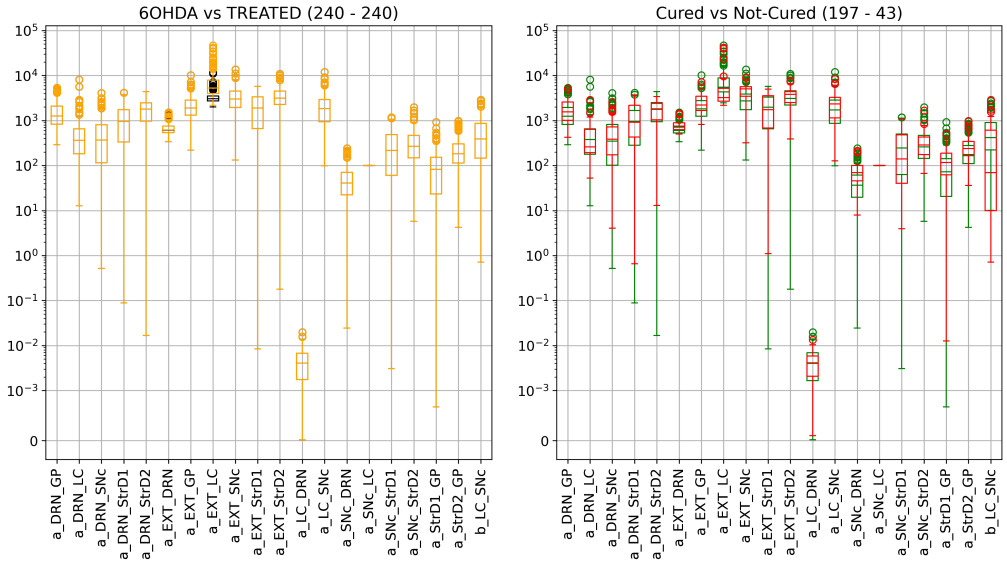


Figure 3.31: Distribution comparison of model parameters in the 6OHDA subjects with respect to treated subjects (left). The treatment modifies the external stimulation to LC and DRN, but is otherwise identical to the 6OHDA case. (Figure 3.27 demonstrates that in fact there is a very small shift in the DRN stimulus distribution which is not appreciable in this plot.)
The right plot shows the distribution of parameters of treated subjects which were successfully cured (green) and which did not reach the desired fitness (red). The subject which did not respond to the treatment happen to be the one whose SNc is not sensitive enough to noradrenaline.

3.6 Discussion

Parkison's Disease affects approximately 10 million people worldwide [90]; we are unfortunately still far from a definitive treatment (and even farther from a cure); research on understanding the underlying phenomena and consequently finding better treatments is still open on many fronts. One important aspect of PD is the involvement of multiple brain circuits, with consequent alteration in the brain neurochemestry [91; 92; 52; 90]. Investigating the relationship between these circuits and how the entire monoamine system reorganizes itself during the development of the pathology is therefore one of the crucial challenges we are called to face.

In this work we proposed a bio-constrained differential equations system that investigates brain regions behaviour after the depletion of serotonin, noradrenaline, or their combination in healthy and 6OHDA induced parkinsonism model. The available data about the average firing rates of brain areas in all depletion states was successfully reproduced by the model on a population of virtual subjects with arbitrary precision, under the assumptions described in section 2.4.1 using fitness measures and optimization strategies described thoroughly in section 2.6.

6-OHDA lesion alone is able to induce change in different brain regions activity: the lesion is simulated by imposing a reduction in SNc and LC activities through a simulated lesion of SNc only [93; 52]; this induced the model to simulate disregulation of other basal regions, in particular DRN showed an increase of firing activity, in accordance with literature [94; 95]. Moreover, the model predicts a tendecy to reduction in striatal D1 activity and increase in striatal D2 activity, which could be referred to the hypokinetc parkinsonian syndrome, that is the result of dysregulation in the activity of the two populatons of medium spiny neurons (MSNs). Dopamine D1 receptor-expressing MSNs (direct), become hypoactive, whereas dopamine D2 receptor-expressing MSNs (indirect) become hyperactive [96; 97; 98; 99], in fact DA activation of direct pathway and inhibition of indirect pathway is necessary for correct motor output.

Also pCPA lesion alone is able to induce change in different brain regions activity: this lesion, similarly to 6OHDA, is simulated by decreasing this time the activity of the DRN, which also causes a decrease of firing rate in the GP according to [52]. The model predicts an increase of SNc activity, which is in accordance with the theory that serotonin could have an inhibitory effect on dopaminergic neurons: in fact, administration of escitalopram (a selective serotonin reuptake inhibitor) strongly decreased the firing rate of dopaminergic neurons in [100], and serotonin-depleted rats show an incresed activity in dopaminergic neurons [101]. Moreover, the model predicts an increase in LC activity which is also in line with literature that suggest a tonic inhibition of LC noradrenergic neurons by serotoninergic afferents [102; 103].

Levodopa is the most common medication used in PD. However, this drug has wide range of adverse effects, most notably motor fluctuations and dyskinesias [104]. The discovery of alternative treatments that not only target dopaminergic system but also noradrenergic or serotonergic systems is a big challenge of our days. Following this path our result suggests that stimulating activity in LC is enough to restore activity in the regions taken in exam. The stimulation of DRN alone seems instead not to be effective, and also when carried out together with LC stimulation it does not play a significant role in the restoration of the neural circuit balance. This result is in line with the theories in which locus coeruleus play a crucial role in development of PD, especially the non motor symptoms at an early stage [105; 106]. It has been shown that restoration of the noradrenergic function using overexpression transcription (Phox2a/2b, DBH, TH) factors directly in the LC can facilitate the recovery of dopaminergic systems [107]. Moreover, there is evidence that in humans LC degeneration can occur much earlier and even to greater extent than in the SN [108; 109; 110; 111].
Taken together these findings suggest that the possibility of acting on both dopaminergic and noradrenergic systems could indeed be an effective strategy for PD treatment in humans.

3.7 Conclusion

In this work we identified a dynamical system which is able to reproduce the available data about one of the neural circuits classically involved in motor and non-motor symptoms of Parkinson's disease. The model offers a high level representation of the neural circuit which is of course based on abstractions; the model therefore does not claim correctness with respect to details but offers a systemic view of the interaction trends at play in between the modelled areas. In particular, the assumed direct proportionality between an area's average activation frequency and its neurotransmitter output, as well as the constant distribution ratios with respect to the projected areas, may be hiding more complex behaviours which are likely to be happening in the real brain. Moreover, the model only includes a small number of areas an does not account for cortical regions (such as the prefrontal cortex and the motor cortex) and other neurotransmitters that might be at play in the studied phenomena.
Despite the aforementioned limitations, the model is nevertheless able to reproduce many aspects of the modelled area's behaviour which have been collected across a broad range of recent scientific literature and its predictions seem to be in accordance with the most recent studies, which results were not taken into account during the optimization of the model; we therefore believe that this model could indeed be useful to improve the current understanding of the interactions between the modelled brain regions in normal and pathological con-

ditions, and offers useful hints on the directions that should be looked towards in the search for a better treatment.
In future work, the model could be expanded to extend its usefulness and concreteness. In particular, some of the abstractions could be concretized, for example by dropping the direct activation-projection proportionality assumption and explicitly representing in the model the individual neuromodulators concentrations at the projected points; additionally, the representation of each area could be split to track the activity of the known different neural populations which compose them.

Appendix A

Additional figures and tables

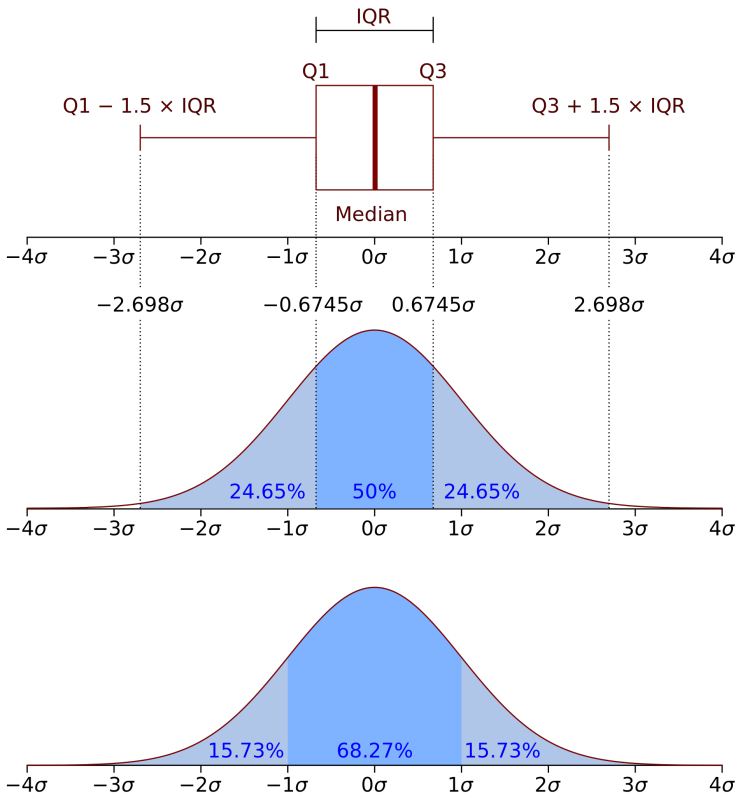


Figure A.1: Boxplot and probability density of a normal distribution. Values outside of the whiskers are usually represented with flares (dots or circles). [112]

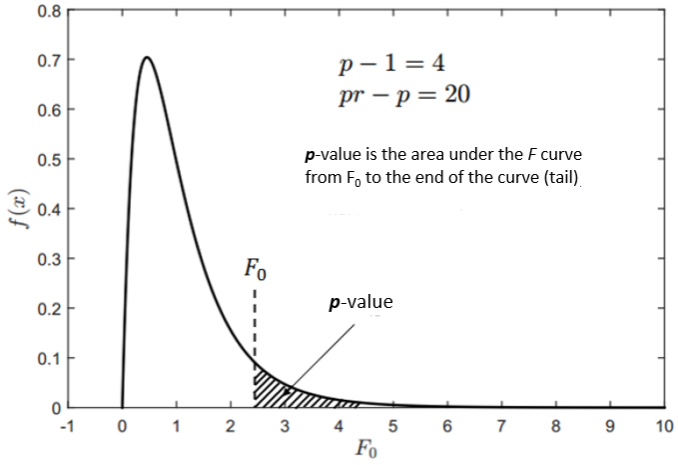


Figure A.2: P-value relationship to F-value curve [113]

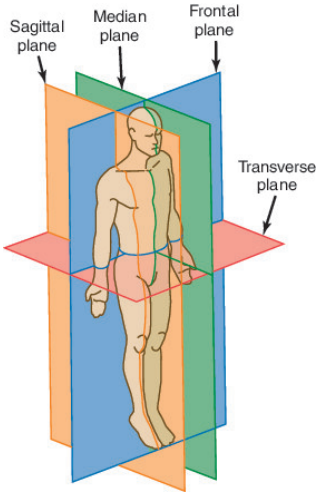


Figure A.3: Reference planes

Some typical dimensions of dendrites for a few types of neurons

(Adapted from Fiala and Harris, 1999.)

Neuron	Average soma diameter (µm)	Number of dendrites at soma	Proximal dendrite diameter (µm)	Number of branch points	Distal dendrite diameter (µm)	Dendrite extent* (µm)	Total dendritic length (µm)
Cerebellar granule cell (cat)	7	4	1	0	0.2–2	15	60
Starburst amacrine cell (rhesus)	9	1	1	40	0.2–2	120	--
Dentate gyrus granule cell (rat)	14	2	3	14	0.5–1	300	3,200
CA1 pyramidal cell (rat)	21						11,900
basal dendrites		5	1	30	0.5–1	130	5,500
s. radiatum		1	3	30	0.25–1	110	4,100
s. lacunosum-moleculare				15	0.25–1	500	2,300
Cerebellar Purkinje cell (guinea pig)	25	1	3	440	0.8–2.2	200	9,100
Principal cell of globus pallidus (human)	33	4	4	12	0.3–0.5	1,000	7,600
Meynert cell of visual cortex (macaque)	35						15,400
basal dendrites		5	3	--	--	250	10,200
apical dendrites		1	4	15	2–3	1,800	5,200
Spinal alpha-motoneuron (cat)	58	11	8	120	0.5–1.5	1,100	52,000

** The average distance from the cell body to the tips of the longest dendrites.*

Figure A.4: Typical dendrite sizes [3]

Appendix B

Source code

"Talk is cheap. Show me the code!"

Linus Torvalds

B.1 Implementation choices

The primary requirements for this implementation were flexibility and readability; the code is therefore structured as a framework to make it as easy as possible to implement and compare different versions of various models, each with their specific structure, set of parameters and fitness functions. Flexibility and readability could be compromised to obtain much faster implementations; for example, the parameters could be represented in a more efficient way directly as vectors or matrices instead of using dictionaries, and the most computationally intensive hotspots could be re-implemented and optimized in a statically typed language such as C and linked in this code as an external library. In this particular context of fast prototyping readability is also extremely important for manual validation of code conformity with specifications and debugging; given the expected short-term lifespan of this code maintainability was not considered an important factor, good engineering practices such as test-driven development and automated testing were therefore not employed and instead we relied on manual testing and validation. However, already with such a small codebase, test and validation proved once again to be rather challenging tasks which explode in complexity with every little feature that is added to the software; it has therefore once more proved true that "one should always employ good engineering practices, no matter how small the project seems to be" [114; 115; 116].

B.2 Model classes

The model base class is implemented as an extension of a dictionary. A model can therefore be accessed as a standard python dictionary to interact with data structures that identify a particular model instance: its *name*, the dictionary of *parameters*, the dictionary of *target values*, and so on. Common useful behaviours are added to *copy* a model (intended as a deep copy which effectively duplicates all the data, so that different copies can be modified without undesired side effects), *save* and *load*, *print* the state in a human-friendly fashion and so on. A model also has an *apply* method which is intended to automatically conform a set of parameters to the particular model. For example, one can initialize a lesioned model with the parameters set of a healthy model, and automatically *apply* the changes which keep the parameter values when necessary, change the set of free parameters according to the lesion definition, change the target values according to the lesion definition and so on. Furthermore, a model is able to *simulate* itself in a given time range with a given starting point, and compute its own *fitness* with respect to the given target values. Since unfortunately scipy's implementation of the initial value problem integrator does not provide a 'minimum step size' or 'maximum number of iterations' condition for early stopping, it was necessary to implement a timeout approach to stop simulations that would run for too long. A model can also *optimize* its own parameters to meet the required fitness; simpler models can be optimized quickly with a local optimization algorithm such as the Nelder-Mead method, however that usually does not converge on a global minimum with more complex models, therefore the default optimization method for a model is the differential evolution algorithm, as previously described in section 2.6.4.
The base model class can then be subclassed to define model-specific properties: the dimensions of the problem, the set of free parameters, the differential equations which define the dynamical behaviour, and the specific fitness measures. Complex models as used in this work, which combine different sets of parameters, provide a standardized *lesion_XXX* method which returns the particular instance one needs. For instance, one can obtain the SHAM or 6OHDA set of parameters of the same model, which represent the same individual in the different states. The base model class provides an automatic mechanisms which allows one to dynamically define the equations one by one and the system of equation is then composed automatically using the 'equations' attribute of the model to keep the correct order. This has proved extremely useful to quickly implement dramatically different systems, but despite having applied memoization to avoid unnecessary dictionary accesses to parameters, this approach still revealed itself to be a bottleneck to the simulation speed. Once the correct form of the model was identified, therefore, this implementation also bypasses that mechanism to instead provide a matrix formulation of the system, which

through the use of the numpy library is delegated to a fast C library which employs automatic vectorization and other harwdware-specific optimizations; the latter implementation is therefore much faster, albeit less flexible and perhaps, less easily human-readable. The individual definition of the equations is anyway left in the model, and have been used to cross-validate the optimized implementation.

```
import copy
import functools
import gc
import math
import pickle
import pprint
import time
from concurrent.futures import ProcessPoolExecutor

import numpy as np
import scipy
import scipy.optimize
import tqdm
from numpy import array as nparray, diagflat, linalg
from numpy.linalg import inv as invert_matrix
from scipy.integrate import solve_ivp
import signal

from plotting import *

np.random.seed(5)

# np.seterr(over='raise', divide='raise', invalid='raise', under='ignore')
# np.seterr(all='raise')

PARAMETERS_SEARCH_HISTORY_FILE_PATH = '/ramtmp/PSH'

nano = 1.  # 10**-9
milli = 10 ** -3

from datetime import datetime

PLOT_OPTIMIZE = False
TIME_LIMIT_SECONDS = 60 * 60 * 12
IVP_TIME_LIMIT_SECONDS = 3


def sieve_pass_positive(x):
    return np.maximum(0, x)


def sieve_pass_negative(x):
    return np.minimum(0, x)


def sieve_pass_all(x):
    return x


class TimeOutException(Exception):
    pass


class Model(dict):
```

```
55
56      def __init__(self):
57          self['name'] = 'Model'
58          self['equations'] = list()
59          self['parameters'] = dict()
60          self['parameters_constraints'] = dict()
61          self['constants'] = dict()
62          self['target'] = dict()
63          self['target_constraints'] = dict()
64          self['applied_lesions'] = list()
65          self['fitness_history'] = list()
66          self.save_parameters_search_history = False
67          self.default_min_param = 0
68          self.default_max_param = 1e5
69
70      def apply(self):
71          pass
72
73      def copy(self, keep_fitness_history=False):
74          fh = self.pop('fitness_history')
75          c = copy.deepcopy(self)
76          self['fitness_history'] = fh
77          if not keep_fitness_history:
78              c['fitness_history'] = list()
79          else:
80              c['fitness_history'] = copy.deepcopy(fh)
81          return c
82
83      def _impose_target(self, other):
84          other['target'] = copy.deepcopy(self['target'])
85          return other
86
87      def print(self):
88          m = self.copy()
89          m.pop('fitness_history')
90          pprint.pprint(m, sort_dicts=True, width=100)
91
92      def save(self, filename):
93          self['timestamp'] = datetime.now().isoformat()
94          with open(filename, 'bw') as f:
95              pickle.dump(self.copy(keep_fitness_history=True), f)
96
97      def _invalidate_caches(self):
98          for cp in ['P', 'parameters_and_constants', 'E', 'y_prime_functions']:
99              if cp in self.__dict__:
100                 del self.__dict__[cp]
101
102     def _clean_constants(self):
103         for k in self['parameters'].keys():
104             try:
105                 self['constants'].pop(k)
106             except KeyError:
107                 pass
108
109     def __setitem__(self, key, value):
110         super(Model, self).__setitem__(key, value)
111         self._invalidate_caches()
112
113     @classmethod
114     def load(self, filename):
115         with open(filename, 'br') as f:
116             new = self()
117             new.update(pickle.load(f))
118             return new
```

```
119
120         @functools.cached_property
121         def parameters_and_constants(self):
122             # parameters can overwrite constants!
123             return self['constants'] | self['parameters']
124
125         @functools.cached_property
126         def P(self):
127             """
128             combined dictionary with all constants and parameters
129             :return:
130             """
131             return self.parameters_and_constants
132
133         @functools.cached_property
134         def E(self):
135             """
136             equation index dictionary
137             :return:
138             """
139             return dict((k, i) for (i, k) in enumerate(self['equations']))
140
141         def with_constants_only(self):
142             self['constants'] = self.P
143             self['parameters'] = {}
144             self._clean_constants()
145             self._invalidate_caches()
146             return self
147
148         @functools.cached_property
149         def y_prime_functions(self):
150             return tuple(self.__getattribute__(f)() for f in self['equations'])
151
152         def y_prime(self, t, y):
153             return nparray([f(t, y) for f in self.y_prime_functions])
154
155         def simulate(self, y0: np.array, t0: float, T: float) -> dict:
156
157             def event_negative(t, y):
158                 return min(y)
159
160             event_negative.terminal = True
161             event_negative.direction = 1
162
163             def event_too_large(t, y):
164                 return max(y) - 100
165
166             event_negative.terminal = True
167             event_negative.direction = -1
168
169             def timeout_handler(num, stack):
170                 raise TimeOutException()
171
172             signal.signal(signal.SIGALRM, timeout_handler)
173             signal.alarm(IVP_TIME_LIMIT_SECONDS)
174             try:
175                 sim = solve_ivp(self.y_prime, (t0, T), y0,
176                                 method='BDF',
177                                 vectorized=True,
178                                 max_step=(T - t0) / 25,
179                                 events=[event_negative, event_too_large]
180                                 )
181                 signal.alarm(0)
182             except TimeOutException:
```

```
183             # The simulation is taking too long, it must be taking too small steps.
184             # Return a fake 'null' solution.
185             sim = {
186                 't': np.zeros(25),
187                 'y': np.zeros((len(self['equations']), 25))
188             }
189             signal.alarm(0)
190         return sim
191 
192     def target_as_y0(self):
193         return np.array([self['target'][eq] if eq in self['target'] else 0.5 for eq in
        ↪  self['equations']])
194 
195     def _fitness_simulation_mse(self, y0, t0, T, limit_to_equations=None,
196                                 simulation=None,
197                                 ignore_before_t=None,
198                                 sieve=sieve_pass_all):
199 
200         # for e in limit_to_equations:
201         #     if e not in self['equations']:
202         #         raise Exception('WRONG limit_to_equation EQ %s!' % e)
203 
204         if ignore_before_t is None:
205             ignore_before_t = t0
206 
207         errors = list()
208 
209         if not simulation:
210             res = self.simulate(y0, t0, T)
211         else:
212             res = simulation
213 
214         solution = res['y']
215         time = res['t']
216 
217         # Ignore the ignore_before_t if the simulation didn't go long enough, or if
        ↪  there aren't enough values
218         # after.
219         if time.max() <= ignore_before_t or (time >= ignore_before_t).sum() <= 3:
220             ignore_before_t = t0
221 
222         solution = solution[:, time >= ignore_before_t]
223         time = time[time >= ignore_before_t]
224 
225         t = time[1:]
226         dt = t - time[:-1]
227 
228         for k, v in self['target'].items():
229             if limit_to_equations is None or (limit_to_equations is not None and k in
            ↪  limit_to_equations):
230                 if not callable(v):
231                     f = lambda t: np.ones(len(t)) * v
232                 else:
233                     f = v
234 
235                 e = sieve(np.array(solution[self.E[k]][1:] - f(t))) ** 2 * dt
236                 errors.append(e)
237 
238         errors = np.array(errors)
239 
240         return errors.sum()
241 
242     def _fitness_time_score(self, y0, t0, T, simulation=None, ignore_before_t=None):
243 
```

```
244             if not simulation:
245                 simulation = self.simulate(y0, t0, T)
246 
247             # 0 if no simulation, 1 if whole interval integrated (account for early
                ↪  stopping)
248             t = simulation['t']
249             if ignore_before_t:
250                 t0 = ignore_before_t
251                 if t[-1] < t0:
252                     return 0
253             return ((t[-1] - t0) / (T - t0))
254 
255         def _fitness_simulation(self, y0, t0, T, limit_to_equations=None, simulation=None,
                ↪  ignore_before_t=None,
256                                 sieve=sieve_pass_all):
257 
258             if not simulation:
259                 res = self.simulate(y0, t0, T)
260             else:
261                 res = simulation
262 
263             # if not res['success']:
264             #     return 0
265 
266             t = res['t']
267 
268             mse = self._fitness_simulation_mse(y0, t0, T,
269                                                limit_to_equations=limit_to_equations,
270                                                simulation=res,
271                                                ignore_before_t=ignore_before_t,
272                                                sieve=sieve).flatten()
273 
274             return float(self._fitness_time_score(y0, t0, T, res) / (1 + mse))
275 
276         def fitness(self, y0, t0, T):
277             return self._fitness_simulation(y0, t0, T)
278 
279         def new_mutated_target_model(self, scale=1.):
280             new_model = self.copy()
281 
282             def mutate(v, check_constraints=(-np.inf, np.inf)):
283                 # If we don't find an acceptable value after some trials, give up mutating
                    ↪  and keep what's there.
284                 for _ in range(100):
285                     new_v = np.random.normal(loc=v, scale=scale * v)
286                     if check_constraints[0] <= new_v <= check_constraints[1]:
287                         return new_v
288                 raise Exception("This should never really happen... (%s <= (%s, %s) <= %s)"
                    ↪  % (
289                     check_constraints[0], new_v, v, check_constraints[1]))
290                 return v
291 
292             for k, v in self['target'].items():
293                 new_model['target'][k] = mutate(v, self['target_constraints'].get(k, (
294                     self.default_min_param, self.default_max_param)))
295 
296             new_model._invalidate_caches()
297             return new_model
298 
299         def _optimize_get_state(self):
300             keys = sorted(self['parameters'].keys())
301             return [self['parameters'][k] for k in keys]
302 
303         def _optimize_get_state_keys(self):
```

```
304             keys = sorted(self['parameters'].keys())
305             return keys
306 
307         def _optimize_get_bounds(self):
308             keys = sorted(self['parameters'].keys())
309             bounds = [self['parameters_constraints'].get(k, (self.default_min_param,
                ↪  self.default_max_param)) for k in keys]
310             return scipy.optimize.Bounds(*zip(*bounds))
311 
312         def _optimize_get_bounds_as_list(self):
313             keys = sorted(self['parameters'].keys())
314             bounds = [self['parameters_constraints'].get(k, (self.default_min_param,
                ↪  self.default_max_param)) for k in keys]
315             return bounds
316 
317         def _optimize_set_state(self, state):
318             keys = sorted(self['parameters'].keys())
319             for i, k in enumerate(keys):
320                 self['parameters'][k] = state[i]
321             self._invalidate_caches()
322 
323         def optimize_local(self,
324                            y0: np.array,
325                            t0: float,
326                            T: float,
327                            N_JOBS=-1,
328                            save_checkpoint_name=False):
329 
330             fitness_history = list()
331 
332             def error(x):
333                 m = self.copy()
334                 m._optimize_set_state(x)
335                 fitness = m.fitness(y0, t0, T)
336                 if self.save_parameters_search_history:
337                     with open(PARAMETERS_SEARCH_HISTORY_FILE_PATH, 'ba') as f:
338                         f.write(pickle.dumps((m.copy(), fitness)))
339                 return (1. - fitness)
340 
341             x0 = self._optimize_get_state()
342 
343             with tqdm.tqdm() as progressbar:
344                 def callback(x):
345                     f = 1. - error(x)
346                     if fitness_history:
347                         conv = (f - fitness_history[-1][1])
348                     else:
349                         conv = 0
350                     fitness_history.append((time.time(), f))
351                     progressbar.update()
352                     progressbar.set_postfix({'fitness': '%e (%s)' % (f, -np.log10(1 - f)),
353                                              'conv'   : '%e' % conv, 'name': self['name']})
354 
355                     m = self.copy()
356                     m._optimize_set_state(x)
357                     m['fitness_history'] = fitness_history
358                     if save_checkpoint_name:
359                         m.save(save_checkpoint_name)
360 
361                 res = scipy.optimize.minimize(error, x0,
362                                               options={
363                                                   'maxfev'  : 1000000,
364                                                   'maxiter' : 2000,
365                                                   'adaptive': True,
```

```
                                            'xatol'   : 1e-6,
                                            'fatol'   : 1e-6,
                                        },
                                        callback=callback,
                                        bounds=self._optimize_get_bounds(),
                                        method='Nelder-Mead'
                                        )

        best = self.copy(keep_fitness_history=True)
        best._optimize_set_state(res.x)
        best['fitness_history'] += fitness_history
        # best.print()
        # print('Target ', str(np.array([self['target'][k] for k in
        ↪  self['equations']])))
        print('Fitness ', best.fitness(y0, t0, T))
        return best, fitness_history

    def _optimize_global_error(self, x):
        m = self.copy()
        m._optimize_set_state(x)
        fitness = m.fitness(self._og_y0, self._og_t0, self._og_T)
        if self.save_parameters_search_history:
            with open(PARAMETERS_SEARCH_HISTORY_FILE_PATH, 'ba') as f:
                f.write(pickle.dumps((m.copy(), fitness)))
        # del m
        return (1 - fitness)

    def optimize_global_DE(self,
                           y0: np.array,
                           t0: float,
                           T: float,
                           N_JOBS=-1,
                           save_checkpoint_name=False,
                           seed=1984,
                           popsize=2,
                           tol=1e-3):

        self._og_y0 = y0
        self._og_t0 = t0
        self._og_T = T

        x0 = self._optimize_get_state()

        start_time = time.time()

        fitness_history = self['fitness_history']
        if not len(fitness_history):
            fitness_history.append((start_time, 1. - self._optimize_global_error(x0)))

        if PLOT_OPTIMIZE:
            fig = plt.figure()
            plt.ion()
            plot_parameters([self], figure=fig)
            plt.show()
            plt.draw()
            plt.pause(0.00001)

        with tqdm.tqdm() as progressbar:
            def callback(x, convergence=0):
                f = 1. - self._optimize_global_error(x)

                m = self.copy()
                m._optimize_set_state(x)
                m['fitness_history'] = fitness_history
```

```
                if save_checkpoint_name:
                    m.save(save_checkpoint_name)

                if fitness_history:
                    conv = (f - fitness_history[-1][1])
                else:
                    conv = 0

                fitness_history.append((time.time(), f))
                progressbar.update()
                progressbar.set_postfix(
                        {'fitness' : '%e (%s)' % (f, -np.log10(1 - f)),
                         'alg_conv': '%e' % convergence,
                         'conv'    : '%e' % conv,
                         'name'    : self['name']})

                if conv != 0 and PLOT_OPTIMIZE:
                    fig.clear()
                    plot_parameters([m], figure=fig)
                    plt.draw()
                    plt.pause(0.00001)

                if time.time() - start_time > TIME_LIMIT_SECONDS:
                    return True

                if f > 1 - 1e-8:
                    return True
                else:
                    return False

            # import ray
            # from ray.util.multiprocessing import Pool as RayPool
            # runtime_env = {"working_dir": "./"}
            # ray.init(runtime_env=runtime_env)
            # ray_remote_args = {"scheduling_strategy": "SPREAD", 'num_cpus': 1}
            # MAP = RayPool(ray_remote_args=ray_remote_args).map

            executor = ProcessPoolExecutor()
            MAP = executor.map

            res = scipy.optimize.differential_evolution(
                    func=self._optimize_global_error,
                    bounds=tuple(self._optimize_get_bounds_as_list()),
                    callback=callback,
                    x0=x0,
                    maxiter=100000,
                    strategy='best1exp',
                    workers=MAP,
                    updating='deferred',
                    polish=False,
                    mutation=0.95,
                    recombination=0.95,
                    init='halton',
                    popsize=popsize,
                    tol=tol,
                    seed=seed,
            )

        best = self.copy(keep_fitness_history=True)
        best._optimize_set_state(res.x)
        best['fitness_history'] = fitness_history
        # best.print()
        # print('Target ', str(np.array([self['target'][k] for k in
        ↪  self['equations']])))
```

```
492             final_fitness = best.fitness(y0, t0, T)
493             print('Fitness ', final_fitness)
494             gc.collect()  # When looping optimizations, if fast, gc may not run frequently
                ↪  enough
495             # print(res)
496             return best, fitness_history, final_fitness
497
498         def optimize(self,
499                      y0: np.array,
500                      t0: float,
501                      T: float,
502                      N_JOBS=-1,
503                      save_checkpoint_name=False,
504                      seed=False,
505                      popsize=4,
506                      tol=1e-3
507                      ):
508
509             best = self
510
511             for i, seed in enumerate([42, 1984, 69, 2013, 126, 500, 86, 31, 71546, 978456]):
512
513                 fresh_start = best.__class__()
514                 fresh_start.apply()
515                 best._optimize_set_state(fresh_start._optimize_get_state())
516
517                 best, fitness_history, final_fitness = best.optimize_global_DE(
518                         y0, t0, T, N_JOBS=N_JOBS,
519                         save_checkpoint_name=save_checkpoint_name,
520                         seed=seed,
521                         popsize=popsize,
522                         tol=tol)
523
524                 if final_fitness >= 1 - 1e-8:
525                     break
526
527             return best, fitness_history
528
529
530     class Healthy(Model):
531
532         def __init__(self):
533             super(Healthy, self).__init__()
534             self['name'] = 'S00'
535             self['equations'] = ['GP', 'StrD1', 'StrD2', 'SNc', 'DRN', 'LC']
536
537             self['parameters'] = {
538             }
539
540             self['constants'] = {
541                 'a_GP_GP'      : 18 * milli,
542                 'a_EXT_GP'     : 100.,  # 22 / (18 * milli),
543                 'a_StrD1_GP'   : 100.,
544                 'a_StrD2_GP'   : 100.,
545                 'a_DRN_GP'     : 100.,
546
547                 'a_StrD1_StrD1': 2 * milli,
548                 'a_EXT_StrD1'  : 100.,  # 8 / (2 * milli),
549                 'a_SNc_StrD1'  : 100.,
550                 'a_DRN_StrD1'  : 100.,
551
552                 'a_StrD2_StrD2': 2 * milli,
553                 'a_EXT_StrD2'  : 100.,  # 9 / (2 * milli),
554                 'a_SNc_StrD2'  : 100.,
```

```
555             'a_DRN_StrD2'  : 100.,
556
557             'a_SNc_SNc'    : 1.5 * milli,
558             'a_EXT_SNc'    : 100.,  # 4.5 / (1.5 * milli),
559             'b_LC_SNc'     : 100,
560             'a_DRN_SNc'    : 100.,
561             'a_LC_SNc'     : 100.,
562
563             'a_DRN_DRN'    : 3.3 * milli,
564             'a_EXT_DRN'    : 100.,  # 1.2 / (3.3 * milli),
565             'a_SNc_DRN'    : 100.,
566             'a_LC_DRN'     : 100.,
567
568             'a_LC_LC'      : 0.8 * milli,
569             'a_EXT_LC'     : 100.,  # 2.5 / (0.8 * milli),
570             'a_DRN_LC'     : 100.,
571             'a_SNc_LC'     : 100.,
572
573         }
574
575         self['parameters_signs'] = {
576             'a_GP_GP'      : -1,
577             'a_EXT_GP'     : 1,
578             'a_StrD1_GP'   : -1,
579             'a_StrD2_GP'   : -1,
580             'a_DRN_GP'     : 1,
581
582             'a_StrD1_StrD1': -1,
583             'a_EXT_StrD1'  : 1,
584             'a_SNc_StrD1'  : 1,
585             'a_DRN_StrD1'  : 1,
586
587             'a_StrD2_StrD2': -1,
588             'a_EXT_StrD2'  : 1,
589             'a_SNc_StrD2'  : -1,
590             'a_DRN_StrD2'  : 1,
591
592             'a_SNc_SNc'    : -1,
593             'a_EXT_SNc'    : 1,
594             'b_LC_SNc'     : 1,
595             'a_DRN_SNc'    : -1,
596             'a_LC_SNc'     : -1,
597
598             'a_DRN_DRN'    : -1,
599             'a_EXT_DRN'    : 1,
600             'a_SNc_DRN'    : -1,
601             'a_LC_DRN'     : 1,
602
603             'a_LC_LC'      : -1,
604             'a_EXT_LC'     : 1,
605             'a_DRN_LC'     : -1,
606             'a_SNc_LC'     : 1,
607
608         }
609
610         self['parameters_constraints'] = {
611             # constraints are [self.default_min_param, self.default_max_param] by
                ↪  default
612         }
613
614         self['target'] = {
615             'GP'   : 22,  # Hz
616             'StrD1': 10,  # Hz
617             'StrD2': 9,  # Hz
```

```
618             'SNc'  : 4.47,  # Hz
619             'DRN'  : 1.41,  # Hz
620             'LC'   : 2.3,  # Hz
621         }
622 
623         self['target_constraints'] = {
624             'GP'   : [18, 26],  # Hz
625             'StrD1': [8, 12],  # Hz
626             'StrD2': [7, 11],  # Hz
627             'SNc'  : [3.5, 5.5],  # Hz
628             'DRN'  : [1, 2],  # Hz
629             'LC'   : [1.9, 3],  # Hz
630         }
631 
632     def apply(self):
633 
634         if 'SHAM' not in self['applied_lesions']:
635             self['applied_lesions'].append('SHAM')
636             self['name'] += ' +SHAM'
637 
638         self._invalidate_caches()
639         self['constants'] = self.P
640 
641         self['parameters'] = {
642 
643             'a_StrD1_GP' : self.P['a_StrD1_GP'],
644             'a_StrD2_GP' : self.P['a_StrD2_GP'],
645             'a_DRN_GP'   : self.P['a_DRN_GP'],
646             'a_EXT_GP'   : self.P['a_EXT_GP'],
647             #
648             'a_SNc_StrD1': self.P['a_SNc_StrD1'],
649             'a_DRN_StrD1': self.P['a_DRN_StrD1'],
650             'a_EXT_StrD1': self.P['a_EXT_StrD1'],
651             #
652             'a_SNc_StrD2': self.P['a_SNc_StrD2'],
653             'a_DRN_StrD2': self.P['a_DRN_StrD2'],
654             'a_EXT_StrD2': self.P['a_EXT_StrD2'],
655             #
656             'a_DRN_SNc'  : self.P['a_DRN_SNc'],
657             'a_LC_SNc'   : self.P['a_LC_SNc'],
658             'b_LC_SNc'   : self.P['b_LC_SNc'],
659             'a_EXT_SNc'  : self.P['a_EXT_SNc'],
660             #
661             'a_SNc_DRN'  : self.P['a_SNc_DRN'],
662             'a_LC_DRN'   : self.P['a_LC_DRN'],
663             'a_EXT_DRN'  : self.P['a_EXT_DRN'],
664             #
665             'a_DRN_LC'   : self.P['a_DRN_LC'],
666             'a_SNc_LC'   : self.P['a_SNc_LC'],
667             'a_EXT_LC'   : self.P['a_EXT_LC'],
668         }
669         self._clean_constants()
670         self['parameters_constraints'] = {
671             # constraints are [self.default_min_param, self.default_max_param] by
                ↪  default
672         }
673 
674     def GP(self):
675         gp_idx = self.E['GP']
676         strd1_idx = self.E['StrD1']
677         strd2_idx = self.E['StrD2']
678         drn_idx = self.E['DRN']
679         T_GP = self.P['a_GP_GP']
680         a_StrD1_GP = self.P['a_StrD1_GP']
```

```
681             a_StrD2_GP = self.P['a_StrD2_GP']
682             a_DRN_GP = self.P['a_DRN_GP']
683             a_EXT_GP = self.P['a_EXT_GP']
684 
685             def _GP(t, y):
686                 return -(1. / T_GP) * y[gp_idx] - a_StrD1_GP * y[strd1_idx] - a_StrD2_GP *
                    ↪  y[strd2_idx] + \
687                     a_DRN_GP * y[drn_idx] + a_EXT_GP
688 
689             return _GP
690 
691         def StrD1(self):
692             strd1_idx = self.E['StrD1']
693             drn_idx = self.E['DRN']
694             snc_idx = self.E['SNc']
695             t_strd1 = self.P['a_StrD1_StrD1']
696             a_drn_strd1 = self.P['a_DRN_StrD1']
697             a_snc_strd1 = self.P['a_SNc_StrD1']
698             a_ext_strd1 = self.P['a_EXT_StrD1']
699 
700             def _StrD1(t, y):
701                 return -(1. / t_strd1) * y[strd1_idx] + a_drn_strd1 * y[drn_idx] +
                    ↪  a_snc_strd1 * y[snc_idx] + a_ext_strd1
702 
703             return _StrD1
704 
705         def StrD2(self):
706             strd2_idx = self.E['StrD2']
707             drn_idx = self.E['DRN']
708             snc_idx = self.E['SNc']
709             t_strd2 = self.P['a_StrD2_StrD2']
710             a_drn_strd2 = self.P['a_DRN_StrD2']
711             a_snc_strd2 = self.P['a_SNc_StrD2']
712             a_ext_strd2 = self.P['a_EXT_StrD2']
713 
714             def _StrD2(t, y):
715                 return -(1. / t_strd2) * y[strd2_idx] + a_drn_strd2 * y[drn_idx] -
                    ↪  a_snc_strd2 * y[snc_idx] + a_ext_strd2
716 
717             return _StrD2
718 
719         def SNc(self):
720             snc_idx = self.E['SNc']
721             drn_idx = self.E['DRN']
722             lc_idx = self.E['LC']
723             t_snc = self.P['a_SNc_SNc']
724             a_drn_snc = self.P['a_DRN_SNc']
725             a_lc_snc = self.P['a_LC_SNc']
726             b_lc_snc = self.P['b_LC_SNc']
727             a_ext_snc = self.P['a_EXT_SNc']
728 
729             def _SNc(t, y):
730                 return - (1. / t_snc) * y[snc_idx] \
731                     - a_drn_snc * y[drn_idx] - a_lc_snc * y[lc_idx] + (b_lc_snc * y[lc_idx]
                        ↪  ** 2) + a_ext_snc
732 
733             return _SNc
734 
735         def DRN(self):
736             drn_idx = self.E['DRN']
737             snc_idx = self.E['SNc']
738             lc_idx = self.E['LC']
739             t_drn = self.P['a_DRN_DRN']
740             a_snc_drn = self.P['a_SNc_DRN']
```

```
741             a_lc_drn = self.P['a_LC_DRN']
742             a_ext_drn = self.P['a_EXT_DRN']
743 
744             def _DRN(t, y):
745                 return - (1. / t_drn) * y[drn_idx] + a_lc_drn * y[lc_idx] - a_snc_drn *
                    ↪  y[snc_idx] + a_ext_drn
746 
747             return _DRN
748 
749         def LC(self):
750             lc_idx = self.E['LC']
751             drn_idx = self.E['DRN']
752             snc_idx = self.E['DRN']
753             t_lc = self.P['a_LC_LC']
754             a_drn_lc = self.P['a_DRN_LC']
755             a_snc_lc = self.P['a_SNc_LC']
756             a_ext_lc = self.P['a_EXT_LC']
757 
758             def _LC(t, y):
759                 return - (1. / t_lc) * y[lc_idx] - a_drn_lc * y[drn_idx] + a_snc_lc *
                    ↪  y[snc_idx] + a_ext_lc
760 
761             return _LC
762 
763         def _invalidate_caches(self):
764             super(Healthy, self)._invalidate_caches()
765             if '_matrices' in self.__dict__:
766                 del self.__dict__['_matrices']
767 
768         @functools.cached_property
769         def _matrices(self):
770             eqs = self['equations']
771             signs = self['parameters_signs']
772             len_eqs = len(eqs)
773             A = np.zeros((len_eqs, len_eqs))
774             C = np.zeros((len_eqs, len_eqs))
775             b = np.zeros((len_eqs, 1))
776 
777             equation_index = dict((e, i) for i, e in enumerate(eqs))
778             for n, v in self.P.items():
779                 if n.startswith('a') or n.startswith('b'):
780                     dest, eq_from, eq_to = n.split('_')
781                     if dest == 'a':
782                         if eq_from == eq_to:
783                             v = 1 / v
784                         if eq_from == 'EXT':
785                             b[equation_index[eq_to]] = signs[n] * v
786                         else:
787                            A[equation_index[eq_to], equation_index[eq_from]] = signs[n] * v
788                     elif dest == 'b':
789                         C[equation_index[eq_to], equation_index[eq_from]] = signs[n] * v
790 
791             return A, C, b
792 
793         def y_prime(self, t, y):
794             A, C, b = self._matrices
795             return A.dot(y) + C.dot(y ** 2) + b
796 
797         def _eigenvalues_real_part(self):
798             A, C, b = self._matrices
799             real_part_of_eigs = np.real(np.linalg.eig(A)[0])
800 
801             def f(y):
802                 return A.dot(y) + C.dot(y * y) + b
```

```
803
804             def fprime(y):
805                 return A + 2 * C.dot(diagflat(y))
806
807             eigsum = sum((max(0, e) for e in real_part_of_eigs))
808             if eigsum <= 0:
809                 tol = 1e-9
810                 y = -invert_matrix(A).dot(b)
811                 itercount = 0
812                 while itercount <= 25:
813                     itercount += 1
814                     y_last = y
815                     y = y_last - invert_matrix(fprime(y_last)).dot(f(y_last))
816                     if abs(y - y_last).max() <= tol:
817                         break
818                     if y.max() > 100 or y.min() < 0:
819                         break
820
821                 A_tilde = fprime(y)
822                 real_part_of_eigs = np.real(np.linalg.eig(A_tilde)[0])
823
824             return real_part_of_eigs
825
826         def asymptotic_stability_score(self):
827             eigsum = sum((max(0, e) for e in self._eigenvalues_real_part()))
828             return 1. / (1 + eigsum)
829
830         def _split_fitness(self, y0, t0, T):
831             res = self.simulate(y0, t0, T)
832             fits = list()
833             for eq in self['equations']:
834                 fits.append(self._fitness_simulation(y0, t0, T, simulation=res,
835                                                      limit_to_equations=[eq],
836                                                      ignore_before_t=None))
837             return fits
838
839         def _combine_split_fitnesses(self, fits):
840             # return np.min(fits)
841             # return np.average(fits)
842             # return min(fits) / np.average(fits)
843             return math.sqrt(min(fits) * np.average(fits))
844             # return np.prod(fits)**(1/len(fits))
845
846         def fitness(self, y0, t0, T):
847             return self._combine_split_fitnesses(self._split_fitness(y0, t0, T))
848
849
850     class L6OHDA(Healthy):
851
852         def __init__(self):
853             super(L6OHDA, self).__init__()
854
855         def apply(self):
856             if '6OHDA' not in self['applied_lesions']:
857                 self['applied_lesions'].append('6OHDA')
858                 self['name'] += ' +6OHDA'
859
860                 # self['target']['GP'] *= 0.90
861                 self['target']['SNc'] *= 0.1
862                 self['target']['LC'] *= 0.8
863
864             self._invalidate_caches()
865             self['constants'] = self.P
866             self['parameters'] = {
```

```
            'b_LC_SNc' : self.P.get('b_LC_SNc', False) or 1.,
            'a_LC_SNc' : self.P.get('a_LC_SNc', False) or 1.,
            'a_DRN_SNc': self.P.get('a_DRN_SNc', False) or 1.,
            'a_EXT_SNc': self.P.get('a_EXT_SNc', False) or 1.,
        }
        self['parameters_constraints']['b_LC_SNc'] = [0, self.default_max_param]
        self['parameters_constraints']['a_EXT_SNc'] = (
            self.default_min_param, self.P.get('a_EXT_SNc', False) or
            ↪  self.default_max_param)

        self._clean_constants()

    def _split_fitness(self, y0, t0, T):
        res = self.simulate(y0, t0, T)

        return [
            self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
                                     simulation=res,
                                     ignore_before_t=(T - t0) / 2),
            self._fitness_simulation(y0, t0, T, limit_to_equations=['SNc'],
                                     simulation=res,
                                     ignore_before_t=(T - t0) / 2,
                                     sieve=sieve_pass_positive
                                     ),
            self._fitness_simulation(y0, t0, T, limit_to_equations=['LC'],
                                     simulation=res,
                                     ignore_before_t=(T - t0) / 2,
                                     sieve=sieve_pass_positive
                                     )

        ]

    def fitness(self, y0, t0, T):
        return self._combine_split_fitnesses(self._split_fitness(y0, t0, T))


class LpCPA(Healthy):

    def __init__(self):
        super(LpCPA, self).__init__()

    def apply(self):
        if 'pCPA' not in self['applied_lesions']:
            self['applied_lesions'].append('pCPA')
            self['name'] += ' +pCPA'

            self['target']['GP'] *= 0.65
            self['target']['DRN'] *= 0.3

        self._invalidate_caches()
        self['constants'] = self.P
        self['parameters'] = {
            'a_EXT_DRN': self.P['a_EXT_DRN'],
            'a_LC_DRN' : self.P['a_LC_DRN'],
            'a_SNc_DRN': self.P['a_SNc_DRN'],
        }
        self['parameters_constraints']['a_EXT_DRN'] = (
            self.default_min_param, self.P.get('a_EXT_DRN', False) or
            ↪  self.default_max_param)
        self._clean_constants()

    def _split_fitness(self, y0, t0, T):
        res = self.simulate(y0, t0, T)

```

```
929         return [
930             self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
931                                      simulation=res,
932                                      ignore_before_t=(T - t0) / 2),
933 
934             self._fitness_simulation(y0, t0, T, limit_to_equations=['DRN'],
935                                      simulation=res,
936                                      ignore_before_t=(T - t0) / 2,
937                                      sieve=sieve_pass_positive),
938         ]
939 
940     def fitness(self, y0, t0, T):
941         return self._combine_split_fitnesses(self._split_fitness(y0, t0, T))
942 
943 
944 class LDSP4(Healthy):
945 
946     def __init__(self):
947         super(LDSP4, self).__init__()
948 
949     def apply(self):
950         if 'DSP4' not in self['applied_lesions']:
951             self['applied_lesions'].append('DSP4')
952             self['name'] += ' +DSP4'
953 
954             self['target']['LC'] *= 0.2
955 
956         self._invalidate_caches()
957         self['constants'] = self.P
958         self['parameters'] = {
959             'a_EXT_LC': self.P['a_EXT_LC'],
960             'a_DRN_LC': self.P['a_DRN_LC']
961         }
962         self['parameters_constraints']['a_EXT_LC'] = (
963             self.default_min_param, self.P.get('a_EXT_LC', False) or
                ↪  self.default_max_param)
964         self._clean_constants()
965 
966     def _split_fitness(self, y0, t0, T):
967         res = self.simulate(y0, t0, T)
968 
969         return [
970             self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
971                                      simulation=res,
972                                      ignore_before_t=(T - t0) / 2),
973             self._fitness_simulation(y0, t0, T, limit_to_equations=['LC'],
974                                      simulation=res,
975                                      ignore_before_t=(T - t0) / 2,
976                                      sieve=sieve_pass_positive),
977         ]
978 
979     def fitness(self, y0, t0, T):
980         return self._combine_split_fitnesses(self._split_fitness(y0, t0, T))
981 
982 
983 class L6OHDA_LDSP4(Healthy):
984 
985     def __init__(self):
986         super(L6OHDA_LDSP4, self).__init__()
987 
988     def apply(self):
989         if '6OHDA+DSP4' not in self['applied_lesions']:
990             self['applied_lesions'].append('6OHDA+DSP4')
991             self['name'] += ' +6OHDA+DSP4'
```

```
            self.original_GP = self['target']['GP']
            self.max_GP = self.original_GP
            self.min_GP = self.original_GP * 0.65

        self._invalidate_caches()
        self['constants'] = self.P
        self['parameters'] = {}
        self._clean_constants()

    def _split_fitness(self, y0, t0, T):
        res = self.simulate(y0, t0, T)

        time_score = self._fitness_time_score(y0, t0, T,
                                              simulation=res)

        self['target']['GP'] = self.min_GP
        min_GP_score = self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
                                                simulation=res,
                                                ignore_before_t=(T - t0) / 2,
                                                sieve=sieve_pass_negative
                                                )
        self['target']['GP'] = self.max_GP
        max_GP_score = self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
                                                simulation=res,
                                                ignore_before_t=(T - t0) / 2,
                                                sieve=sieve_pass_positive
                                                )
        self['target']['GP'] = self.original_GP

        return [
            time_score,
            min_GP_score,
            max_GP_score
        ]

    def fitness(self, y0, t0, T):
        return self._combine_split_fitnesses(self._split_fitness(y0, t0, T))


class L6OHDA_LpCPA(Healthy):

    def __init__(self):
        super(L6OHDA_LpCPA, self).__init__()

    def apply(self):
        if '6OHDA+pCPA' not in self['applied_lesions']:
            self['applied_lesions'].append('6OHDA+pCPA')
            self['name'] += ' +6OHDA+pCPA'

        self.original_GP = self['target']['GP']
        self.max_GP = self.original_GP * 0.75
        self.min_GP = self.original_GP * 0.55

        self._invalidate_caches()
        self['constants'] = self.P
        self['parameters'] = {}
        self._clean_constants()

    def _split_fitness(self, y0, t0, T):
        res = self.simulate(y0, t0, T)

        time_score = self._fitness_time_score(y0, t0, T,
                                              simulation=res)

```

```
1056            self['target']['GP'] = self.min_GP
1057            min_GP_score = self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
1058                                                    simulation=res,
1059                                                    ignore_before_t=(T - t0) / 2,
1060                                                    sieve=sieve_pass_negative
1061                                                    )
1062            self['target']['GP'] = self.max_GP
1063            max_GP_score = self._fitness_simulation(y0, t0, T, limit_to_equations=['GP'],
1064                                                    simulation=res,
1065                                                    ignore_before_t=(T - t0) / 2,
1066                                                    sieve=sieve_pass_positive
1067                                                    )
1068            self['target']['GP'] = self.original_GP
1069
1070            return [
1071                time_score,
1072                min_GP_score,
1073                max_GP_score
1074            ]
1075
1076        def fitness(self, y0, t0, T):
1077            return self._combine_split_fitnesses(self._split_fitness(y0, t0, T))
1078
1079
1080    class Healthy_combined_fit(Healthy):
1081
1082        def __init__(self):
1083            super(Healthy_combined_fit, self).__init__()
1084
1085        def apply(self):
1086            self['parameters_constraints']['L6OHDA___b_LC_SNc'] = [0,
                ↪  self.default_max_param]
1087
1088            self._invalidate_caches()
1089            self['constants'] = self.P
1090
1091            # self['constants'].update(
1092            #         {
1093            #         }
1094            # )
1095
1096            self['parameters'] = {
1097
1098                'a_StrD1_GP'        : self.P['a_StrD1_GP'],
1099                'a_StrD2_GP'        : self.P['a_StrD2_GP'],
1100                'a_DRN_GP'          : self.P['a_DRN_GP'],
1101                'a_EXT_GP'          : self.P['a_EXT_GP'],
1102
1103                'a_SNc_StrD1'       : self.P['a_SNc_StrD1'],
1104                'a_DRN_StrD1'       : self.P['a_DRN_StrD1'],
1105                'a_EXT_StrD1'       : self.P['a_EXT_StrD1'],
1106
1107                'a_SNc_StrD2'       : self.P['a_SNc_StrD2'],
1108                'a_DRN_StrD2'       : self.P['a_DRN_StrD2'],
1109                'a_EXT_StrD2'       : self.P['a_EXT_StrD2'],
1110
1111                'a_DRN_SNc'         : self.P['a_DRN_SNc'],
1112                'a_LC_SNc'          : self.P['a_LC_SNc'],
1113                'b_LC_SNc'          : self.P['b_LC_SNc'],
1114                'a_EXT_SNc'         : self.P['a_EXT_SNc'],
1115
1116                'a_SNc_DRN'         : self.P['a_SNc_DRN'],
1117                'a_LC_DRN'          : self.P['a_LC_DRN'],
1118                'a_EXT_DRN'         : self.P['a_EXT_DRN'],
```

```
1119
1120                'a_DRN_LC'          : self.P['a_DRN_LC'],
1121                'a_EXT_LC'          : self.P['a_EXT_LC'],
1122                'a_SNc_LC'          : self.P['a_SNc_LC'],
1123
1124                'L6OHDA___a_LC_SNc' : self.P.get('L6OHDA___a_LC_SNc', False) or
                    ↪  self.P['a_LC_SNc'],
1125                'L6OHDA___b_LC_SNc' : self.P.get('L6OHDA___b_LC_SNc', False) or
                    ↪  self.P['b_LC_SNc'],
1126                'L6OHDA___a_DRN_SNc': self.P.get('L6OHDA___a_DRN_SNc', False) or
                    ↪  self.P['a_DRN_SNc'],
1127                'L6OHDA___a_EXT_SNc': self.P.get('L6OHDA___a_EXT_SNc', False) or
                    ↪  self.P['a_EXT_SNc'],
1128
1129                'LpCPA___a_LC_DRN'  : self.P.get('LpCPA___a_LC_DRN', False) or
                    ↪  self.P['a_LC_DRN'],
1130                'LpCPA___a_SNc_DRN' : self.P.get('LpCPA___a_SNc_DRN', False) or
                    ↪  self.P['a_SNc_DRN'],
1131                'LpCPA___a_EXT_DRN' : self.P.get('LpCPA___a_EXT_DRN', False) or
                    ↪  self.P['a_EXT_DRN'],
1132
1133                'LDSP4___a_DRN_LC'  : self.P.get('LDSP4___a_DRN_LC', False) or
                    ↪  self.P['a_DRN_LC'],
1134                'LDSP4___a_EXT_LC'  : self.P.get('LDSP4___a_EXT_LC', False) or
                    ↪  self.P['a_EXT_LC'],
1135
1136            }
1137            self['parameters_constraints']['b_LC_SNc'] = [0, self.default_max_param]
1138
1139            self._clean_constants()
1140            self._invalidate_caches()
1141
1142        def lesion_SHAM(self):
1143            healthy = Healthy()
1144            healthy.update(self.copy())
1145            healthy.apply()
1146
1147            healthy._clean_constants()
1148            healthy._invalidate_caches()
1149            return healthy
1150
1151        def lesion_L6OHDA(self):
1152            l6ohda = L6OHDA()
1153            l6ohda.update(self.copy())
1154            l6ohda.apply()
1155            l6ohda['parameters']['a_LC_SNc'] = self.P['L6OHDA___a_LC_SNc']
1156            l6ohda['parameters']['b_LC_SNc'] = self.P['L6OHDA___b_LC_SNc']
1157            l6ohda['parameters']['a_DRN_SNc'] = self.P['L6OHDA___a_DRN_SNc']
1158            l6ohda['parameters']['a_EXT_SNc'] = self.P['L6OHDA___a_EXT_SNc']
1159
1160            l6ohda._clean_constants()
1161            l6ohda._invalidate_caches()
1162            return l6ohda
1163
1164        def lesion_LpCPA(self):
1165            lpcpa = LpCPA()
1166            lpcpa.update(self.copy())
1167            lpcpa.apply()
1168            lpcpa['parameters']['a_LC_DRN'] = self.P['LpCPA___a_LC_DRN']
1169            lpcpa['parameters']['a_SNc_DRN'] = self.P['LpCPA___a_SNc_DRN']
1170            lpcpa['parameters']['a_EXT_DRN'] = self.P['LpCPA___a_EXT_DRN']
1171
1172            lpcpa._clean_constants()
1173            lpcpa._invalidate_caches()
```

```
        return lpcpa

    def lesion_LDSP4(self):
        ldsp4 = LDSP4()
        ldsp4.update(self.copy())
        ldsp4.apply()
        ldsp4['parameters']['a_DRN_LC'] = self.P['LDSP4___a_DRN_LC']
        ldsp4['parameters']['a_EXT_LC'] = self.P['LDSP4___a_EXT_LC']

        ldsp4._clean_constants()
        ldsp4._invalidate_caches()
        return ldsp4

    def lesion_L6OHDA_LpCPA(self):
        lesioned = L6OHDA_LpCPA()
        lesioned.update(self.copy())
        lesioned.apply()

        lesioned['constants'] = lesioned.P
        lesioned['parameters'] = dict()

        lesioned['constants']['a_LC_SNc'] = self.P['L6OHDA___a_LC_SNc']
        lesioned['constants']['b_LC_SNc'] = self.P['L6OHDA___b_LC_SNc']
        lesioned['constants']['a_DRN_SNc'] = self.P['L6OHDA___a_DRN_SNc']
        lesioned['constants']['a_EXT_SNc'] = self.P['L6OHDA___a_EXT_SNc']

        lesioned['constants']['a_LC_DRN'] = self.P['LpCPA___a_LC_DRN']
        lesioned['constants']['a_SNc_DRN'] = self.P['LpCPA___a_SNc_DRN']
        lesioned['constants']['a_EXT_DRN'] = self.P['LpCPA___a_EXT_DRN']

        lesioned._clean_constants()
        lesioned._invalidate_caches()

        return lesioned

    def lesion_L6OHDA_LDSP4(self):
        lesioned = L6OHDA_LDSP4()
        lesioned.update(self.copy())
        lesioned.apply()

        lesioned['constants'] = lesioned.P
        lesioned['parameters'] = dict()

        lesioned['constants']['a_LC_SNc'] = self.P['L6OHDA___a_LC_SNc']
        lesioned['constants']['b_LC_SNc'] = self.P['L6OHDA___b_LC_SNc']
        lesioned['constants']['a_DRN_SNc'] = self.P['L6OHDA___a_DRN_SNc']
        lesioned['constants']['a_EXT_SNc'] = self.P['L6OHDA___a_EXT_SNc']

        lesioned['constants']['a_DRN_LC'] = self.P['LDSP4___a_DRN_LC']
        lesioned['constants']['a_EXT_LC'] = self.P['LDSP4___a_EXT_LC']

        lesioned._clean_constants()
        lesioned._invalidate_caches()

        return lesioned

    def _split_fitness_parameters_limits(self):
        # Lesioned EXT must be smaller than healthy EXT
        f_6OHDA = 1 / (1 + max(0, self.P['L6OHDA___a_EXT_SNc'] - self.P['a_EXT_SNc']))
        f_pCPA = 1 / (1 + max(0, self.P['LpCPA___a_EXT_DRN'] - self.P['a_EXT_DRN']))
        f_DSP4 = 1 / (1 + max(0, self.P['LDSP4___a_EXT_LC'] - self.P['a_EXT_LC']))
        return [f_6OHDA, f_pCPA, f_DSP4]

    def fitness(self, y0, t0, T):
```

```
        healthy = self.lesion_SHAM()
        l6ohda = self.lesion_L6OHDA()
        lpcpa = self.lesion_LpCPA()
        ldsp4 = self.lesion_LDSP4()

        l6ohdapcpa = self.lesion_L6OHDA_LpCPA()
        l6ohdadsp4 = self.lesion_L6OHDA_LDSP4()

        fits = \
            healthy._split_fitness(healthy.target_as_y0(), t0, T) + \
            l6ohda._split_fitness(healthy.target_as_y0(), t0, T) + \
            l6ohda._split_fitness(l6ohda.target_as_y0(), t0, T) + \
            lpcpa._split_fitness(healthy.target_as_y0(), t0, T) + \
            lpcpa._split_fitness(lpcpa.target_as_y0(), t0, T) + \
            ldsp4._split_fitness(healthy.target_as_y0(), t0, T) + \
            ldsp4._split_fitness(ldsp4.target_as_y0(), t0, T) + \
            l6ohdapcpa._split_fitness(healthy.target_as_y0(), t0, T) + \
            l6ohdapcpa._split_fitness(l6ohda.target_as_y0(), t0, T) + \
            l6ohdadsp4._split_fitness(healthy.target_as_y0(), t0, T) + \
            l6ohdadsp4._split_fitness(l6ohda.target_as_y0(), t0, T) + \
            [healthy.asymptotic_stability_score(),
             l6ohda.asymptotic_stability_score(),
             lpcpa.asymptotic_stability_score(),
             ldsp4.asymptotic_stability_score(),
             l6ohdapcpa.asymptotic_stability_score(),
             l6ohdadsp4.asymptotic_stability_score(),
             ] + \
            self._split_fitness_parameters_limits()

        return self._combine_split_fitnesses(fits)


class Cure(Healthy_combined_fit):

    def apply(self):
        self._invalidate_caches()
        self['constants'] = self.P
        self['parameters'] = {}
        self._clean_constants()
        self._invalidate_caches()
        self._param_fitness_penalty = 1

    def lesion_SHAM(self):
        lesioned = self.__class__()
        lesioned.update(self._impose_target(super(Cure, self).lesion_SHAM()))
        return lesioned

    def lesion_L6OHDA(self):
        lesioned = self.__class__()
        lesioned.update(self._impose_target(super(Cure, self).lesion_L6OHDA()))
        return lesioned

    def _split_fitness(self, y0, t0, T):
        res = self.simulate(y0, t0, T)
        fits = list()
        equations = self['equations']
        for eq in equations:
            fits.append(self._fitness_simulation(y0, t0, T, simulation=res,
                                                 limit_to_equations=[eq],
                                                 ignore_before_t=(T - t0) / 2))
        return fits

    def fitness(self, y0, t0, T):
        return self._combine_split_fitnesses(
```

```
                self._split_fitness(y0, t0, T) +
                [self.asymptotic_stability_score()]
        )


class Cure_DRN(Cure):

    def apply(self):
        super(Cure_DRN, self).apply()

        if 'cure_DRN' not in self['applied_lesions']:
            self['applied_lesions'].append('cure_DRN')
            self['name'] += ' +cure_DRN'

        self['parameters'] = {
            'CDRN___a_EXT_DRN': self.P.get('CDRN___a_EXT_DRN', False) or
            ↪  self.P['a_EXT_DRN']
        }
        self._clean_constants()
        self._invalidate_caches()

    def cure_DRN(self):
        cure = Cure_DRN()
        cure.update(self.copy())
        cure.apply()
        cure['parameters']['a_EXT_DRN'] = self.P['CDRN___a_EXT_DRN']
        cure._clean_constants()
        cure._invalidate_caches()
        return cure

    def _split_fitness_parameters_limits(self):
        f = 1 / (1 + self._param_fitness_penalty * max(0, self.P['a_EXT_DRN'] -
        ↪  self.P['CDRN___a_EXT_DRN']))
        return [f]

    def _split_fitness(self, y0, t0, T):
        res = self.simulate(y0, t0, T)
        fits = list()
        equations = self['equations'].copy()
        # equations.pop(equations.index('SNc'))
        equations.pop(equations.index('DRN'))
        # equations.pop(equations.index('LC'))
        for eq in equations:
            fits.append(self._fitness_simulation(y0, t0, T, simulation=res,
                                                 limit_to_equations=[eq],
                                                 ignore_before_t=(T - t0) / 2))
        return fits + [self.asymptotic_stability_score()]

    def fitness(self, y0, t0, T):
        limits = self._split_fitness_parameters_limits()
        cured = self.cure_DRN()
        return cured._combine_split_fitnesses(cured._split_fitness(y0, t0, T) + limits)


class Cure_LC(Cure):
    def apply(self):
        super(Cure_LC, self).apply()

        if 'cure_LC' not in self['applied_lesions']:
            self['applied_lesions'].append('cure_LC')
            self['name'] += ' +cure_LC'

        self['parameters'] = {
            'CLC___a_EXT_LC': self.P.get('CLC___a_EXT_LC', False) or self.P['a_EXT_LC']
```

```
1364            }
1365            self._clean_constants()
1366            self._invalidate_caches()
1367
1368        def cure_LC(self):
1369            cure = Cure_LC()
1370            cure.update(self.copy())
1371            cure.apply()
1372            cure['parameters']['a_EXT_LC'] = self.P['CLC___a_EXT_LC']
1373            cure._clean_constants()
1374            cure._invalidate_caches()
1375            return cure
1376
1377        def _split_fitness_parameters_limits(self):
1378            f = 1 / (1 + self._param_fitness_penalty * max(0, self.P['a_EXT_LC'] -
                ↪  self.P['CLC___a_EXT_LC']))
1379            return [f]
1380
1381        def _split_fitness(self, y0, t0, T):
1382            res = self.simulate(y0, t0, T)
1383            fits = list()
1384            equations = self['equations'].copy()
1385            # equations.pop(equations.index('SNc'))
1386            # equations.pop(equations.index('DRN'))
1387            equations.pop(equations.index('LC'))
1388            for eq in equations:
1389                fits.append(self._fitness_simulation(y0, t0, T, simulation=res,
1390                                                     limit_to_equations=[eq],
1391                                                     ignore_before_t=(T - t0) / 2))
1392            return fits + [self.asymptotic_stability_score()]
1393
1394        def fitness(self, y0, t0, T):
1395            limits = self._split_fitness_parameters_limits()
1396            cured = self.cure_LC()
1397            return cured._combine_split_fitnesses(cured._split_fitness(y0, t0, T) + limits)
1398
1399
1400    class Cure_combined(Cure):
1401        def apply(self):
1402            super(Cure_combined, self).apply()
1403
1404            if 'cure_DRN' not in self['applied_lesions']:
1405                self['applied_lesions'].append('cure_DRN')
1406                self['name'] += ' +cure_DRN'
1407            if 'cure_LC' not in self['applied_lesions']:
1408                self['applied_lesions'].append('cure_LC')
1409                self['name'] += ' +cure_LC'
1410
1411            self['parameters'] = {
1412                'CDRN___a_EXT_DRN': self.P.get('CDRN___a_EXT_DRN', False) or
                    ↪  self.P['a_EXT_DRN'],
1413                'CLC___a_EXT_LC'  : self.P.get('CLC___a_EXT_LC', False) or
                    ↪  self.P['a_EXT_LC']
1414            }
1415            self._clean_constants()
1416            self._invalidate_caches()
1417
1418        def cure_DRN_LC(self):
1419            cure = Cure_combined()
1420            cure.update(self.copy())
1421            cure.apply()
1422            cure['parameters']['a_EXT_DRN'] = self.P['CDRN___a_EXT_DRN']
1423            cure['parameters']['a_EXT_LC'] = self.P['CLC___a_EXT_LC']
1424            cure._clean_constants()
```

```
1425            cure._invalidate_caches()
1426            return cure
1427
1428        def _split_fitness_parameters_limits(self):
1429            fdrn = 1 / (1 + self._param_fitness_penalty * max(0, self.P['a_EXT_LC'] -
                ↪  self.P['CLC___a_EXT_LC']))
1430            flc = 1 / (1 + self._param_fitness_penalty * max(0, self.P['a_EXT_DRN'] -
                ↪  self.P['CDRN___a_EXT_DRN']))
1431            return [fdrn, flc]
1432
1433        def _split_fitness(self, y0, t0, T):
1434            res = self.simulate(y0, t0, T)
1435            fits = list()
1436            equations = self['equations'].copy()
1437            # equations.pop(equations.index('SNc'))
1438            equations.pop(equations.index('DRN'))
1439            equations.pop(equations.index('LC'))
1440            for eq in equations:
1441                fits.append(self._fitness_simulation(y0, t0, T, simulation=res,
1442                                                     limit_to_equations=[eq],
1443                                                     ignore_before_t=(T - t0) / 2))
1444            return fits
1445
1446        def fitness(self, y0, t0, T):
1447            limits = self._split_fitness_parameters_limits()
1448            cured = self.cure_DRN_LC()
1449            return cured._combine_split_fitnesses(
1450                    cured._split_fitness(y0, t0, T) + \
1451                    limits + \
1452                    [self.asymptotic_stability_score()])
```

B.3 Plotting helpers

```
1   import random
2   from random import uniform as random_uniform
3
4   import numpy as np
5   import pandas as pd
6   import pyfiglet
7   from joblib import Parallel, delayed
8   from matplotlib import pyplot as plt
9   from scipy import stats
10
11  plt.rcParams['figure.figsize'] = (7, 7)
12  plt.rc('font', size=12)
13  SUBS = [1, 2, 3, 4, 5, 6, 7, 8, 9]
14  LINTHRESH = 10 ** -4
15
16
17  def pvalue_to_asterisks(pvalue):
18      if pvalue <= 0.0001:
19          return "****"
20      elif pvalue <= 0.001:
21          return "***"
22      elif pvalue <= 0.01:
23          return "**"
24      elif pvalue <= 0.05:
25          return "*"
26      return ""
27
28
```

```
29  def print_title(msg, banner=None):
30      if banner:
31          pyfiglet.print_figlet(banner)
32      print(msg)
33
34
35  def plot_population(model, population, y0, t0, T, plot_target=True,
    ↪  linthresh=LINTHRESH, max_models_in_plot=5):
36      boxplot_figure = plt.figure()
37
38      boxplot_population_targets(population, linthresh=linthresh, figure=boxplot_figure)
39      boxplot_population_last_value(population, figure=boxplot_figure,
        ↪  linthresh=linthresh, t0=t0, T=T)
40
41      plot_figure = plt.figure()
42      plt.grid(which='both')
43
44      step = max(1, int(len(population) / max_models_in_plot))
45
46      for m in population[::step]:
47          plot_model(m, y0 or m.target_as_y0(), t0, T, figure=plot_figure,
            ↪  plot_target=plot_target, linthresh=linthresh)
48
49      return (boxplot_figure, plot_figure)
50
51
52  def plot_model(model, y0, t0, T, figure=None, plot_target=True, linthresh=LINTHRESH):
53      if figure:
54          plt.figure(figure)
55      else:
56          figure = plt.figure()
57          plt.grid(which='both')
58
59      solution = model.simulate(y0, t0, T)
60      cmap = plt.get_cmap('Paired')
61      # neqs = float(len(model['equations']))
62
63      for i, eq in enumerate(model['equations']):
64          plt.plot(solution['t'], solution['y'][i], color=cmap(i))
65
66      plt.legend(model['equations'])
67
68      if plot_target:
69          for i, eq in enumerate(model['equations']):
70              target = model['target'][eq]
71              if not callable(target):
72                  f = lambda t: np.ones(len(solution['t'])) * target
73              else:
74                  f = target
75              plt.plot(solution['t'], f(solution['t']), '--', color=cmap(i))
76      plt.title(''.join(model['name'].split(' ')[1:]) + ' T=%s' % T)
77      plt.xlabel('time (s)')
78      plt.ylabel('average frequency (Hz)')
79
80      if linthresh:
81          plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
82          # plt.xscale('symlog', linthresh=linthresh)
83
84      return figure
85
86
87  def solutions_last_values(equations: list, solutions: list[dict]):
88      a = np.zeros((len(solutions), len(equations)))
89      for i, s in enumerate(solutions):
```

```
90          a[i, :] = s['y'].transpose()[-1]
91      return a
92
93
94  def plot_parameters(models: list, t=0, figure=None, linthresh=LINTHRESH):
95      if figure:
96          plt.figure(figure)
97      else:
98          figure = plt.figure()
99      plt.grid(which='both')
100
101     colorsP = [x['color'] for x in plt.cycler(color=plt.cm.get_cmap('Set1').colors)]
102     colorsC = [x['color'] for x in plt.cycler(color=plt.cm.get_cmap('Accent').colors)]
103
104     for i, model in enumerate(models):
105         columns = sorted(list(model['constants'].keys()))
106         columns = [str(c) for c in columns]
107         values = [model['constants'][k] for k in columns]
108         values = [v(t) if callable(v) else v for v in values]
109        plt.plot(values, columns, 'o', label=model['name'] + ' Const', color=colorsC[i])
110
111         columns = sorted(list(model['parameters'].keys()))
112         columns = [str(c) for c in columns]
113         values = [model['parameters'][k] for k in columns]
114         values = [v(t) if callable(v) else v for v in values]
115         plt.plot(values, columns, 'v', label=model['name'] + ' Params',
            ↪  color=colorsP[i])
116
117     if linthresh:
118         plt.xscale('symlog', linthresh=linthresh, subs=SUBS)
119
120     if len(models) < 5:
121         plt.legend()
122
123     return figure
124
125
126 def boxplot_population_targets(population: list, figure=None, linthresh=LINTHRESH,
    ↪  scatterplot=False, color='grey'):
127     if figure:
128         plt.figure(figure)
129     else:
130         figure = plt.figure()
131     plt.grid(which='both')
132     equations = population[0]['equations']
133     targets = np.array([[m['target'][x] for m in population] for x in
        ↪  equations]).transpose()
134     targets_df = pd.DataFrame(columns=equations, data=targets)
135
136     if scatterplot:
137         targets_df.boxplot()
138         for i, col in enumerate(equations):
139             random.seed(1)
140             points = targets_df[col]
141             x = i + 1
142             width = 0.125
143             L = x - width
144             R = x + width
145             plt.plot([random_uniform(L, R) for _ in range(len(points))], points, 'o',
                ↪  alpha=0.25, color='orange',
146                      zorder=0)
147     else:
148         targets_df.boxplot(color=color)
149
```

```
150     if linthresh:
151         plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
152     plt.title('%s Population target values' % ''.join(population[0]['name'].split('
        ↪  ')[1:]))
153     plt.ylabel('average frequency (Hz)')
154     return figure
155 
156 
157 def boxplot_population_last_value(population: list, figure=None, linthresh=LINTHRESH,
    ↪  t0=0, T=1):
158     if figure:
159         plt.figure(figure)
160     else:
161         figure = plt.figure()
162     plt.grid(which='both')
163     equations = population[0]['equations']
164 
165     data = Parallel(n_jobs=-1)(delayed(m.simulate)(m.target_as_y0(), t0, T) for m in
        ↪  population)
166 
167     targets = [m['y'].transpose()[-1] for m in data if m['t'][-1] >= T]
168     if len(targets):
169         targets_df = pd.DataFrame(columns=equations, data=np.array(targets))
170         targets_df.boxplot()
171 
172         for i, col in enumerate(equations):
173             random.seed(1)
174             points = targets_df[col]
175             x = i + 1
176             width = 0.125
177             L = x - width
178             R = x + width
179             plt.plot([random_uniform(L, R) for _ in range(len(points))], points, 'o',
                ↪  alpha=0.25, color='orange',
180                      zorder=0)
181 
182     missing_targets = [m['y'].transpose()[-1] for m in data if m['t'][-1] < T]
183     if len(missing_targets):
184         targets_df = pd.DataFrame(columns=equations, data=np.array(missing_targets))
185         targets_df.boxplot(color='orange', )
186 
187     if linthresh:
188         plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
189 
190     plt.title('%s Values at T=%s (%s OK, %s NF)' % (
191         str(population[0]['name'].split(' ')[1:]), T, len(targets),
            ↪  len(missing_targets)))
192     plt.ylabel('average frequency (Hz)')
193     return figure
194 
195 
196 def extract_column(data, column):
197     pops = [list() for pop in data]
198     for pindex, pop in enumerate(data):
199         for ind in pop:
200             pops[pindex].append(ind[column])
201     df = pd.DataFrame(pops)
202     return df.T
203 
204 
205 def boxplot_populations_last_value_by_equation(populations: list[list],
    ↪  linthresh=LINTHRESH, t0=0, T=1,
206                                                title_postfix=''):
207     equations = populations[0][0]['equations']
```

```
208     populations_names = [''.join(p[0]['name'].split(' ')[1:]) or 'SHAM' for p in
        ↪  populations]
209
210     figures = dict()
211
212     sim_data = [Parallel(n_jobs=-1)(delayed(m.simulate)(m.target_as_y0(), t0, T) for m
        ↪  in p) for p in populations]
213     data = [[m['y'].transpose()[-1] for m in pop if m['t'][-1] >= T] for pop in
        ↪  sim_data]
214
215     for idx, eq in enumerate(equations):
216         figure = plt.figure()
217         plt.grid(which='both')
218
219         df = extract_column(data, idx)
220         df.columns = populations_names
221         df = df.dropna()
222         df.boxplot()
223
224         for i, col in enumerate(populations_names):
225             random.seed(1)
226             points = df[col]
227             x = i + 1
228             width = 0.125
229             L = x - width
230             R = x + width
231             plt.plot([random_uniform(L, R) for _ in range(len(points))], points, 'o',
                ↪  alpha=0.25, color='orange',
232                      zorder=0)
233
234         if linthresh:
235             plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
236
237         ok_count = min([len(x) for x in data])
238         plt.title(eq + ' at T=%s ' % T + '($\geq$%s OK)' % ok_count + title_postfix)
239
240         plt.setp(figure.axes[0].get_xticklabels(), rotation=45,
            ↪  horizontalalignment='right')
241         plt.ylabel('average frequency (Hz)')
242         figures[str(eq)] = figure
243     return figures
244
245
246 def histplot_populations_last_value_by_equation(populations: list[list],
    ↪  linthresh=LINTHRESH, t0=0, T=1,
247                                                 title_postfix=''):
248     equations = populations[0][0]['equations']
249     populations_names = [''.join(p[0]['name'].split(' ')[1:]) or 'SHAM' for p in
        ↪  populations]
250
251     figures = dict()
252
253     sim_data = [Parallel(n_jobs=-1)(delayed(m.simulate)(m.target_as_y0(), t0, T) for m
        ↪  in p) for p in populations]
254     data = [[m['y'].transpose()[-1] for m in pop if m['t'][-1] >= T] for pop in
        ↪  sim_data]
255
256     for idx, eq in enumerate(equations):
257         figure = plt.figure()
258         plt.grid(which='both')
259
260         df = extract_column(data, idx)
261         df.columns = populations_names
262         df = df.dropna()
```

```
263
264             means = df.mean()
265             sem = df.sem()
266             means[np.isnan(means)] = 0
267             sem[np.isnan(sem)] = 0
268
269             stat = stats.tukey_hsd(*np.array(df).transpose())
270             annotations = [pvalue_to_asterisks(v) for v in stat.pvalue[0]]
271
272            container = plt.bar(means.index, means, yerr=sem, capsize=12, edgecolor='black',
273                                color=plt.cm.binary(range(0, 128, int(128 /
                                ↪  len(equations))), alpha=0), zorder=2)
274             plt.bar_label(container, annotations, size=18)
275
276             for idx, bar in enumerate(container):
277                 random.seed(1)
278                 x = bar.get_x()
279                 width = bar.get_width() / 2.
280                 L = x + width - width / 2
281                 R = x + width + width / 2
282                 points = df[populations_names[idx]]
283                 plt.plot([random_uniform(L, R) for _ in range(len(points))], points, 'o',
                    ↪  alpha=0.25, color='orange',
284                          zorder=1)
285
286             if linthresh:
287                 plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
288
289             ok_count = min([len(x) for x in data])
290             plt.title(eq + ' at T=%s ' % T + ' ($\geq$%s OK) ' % ok_count + title_postfix)
291             plt.setp(figure.axes[0].get_xticklabels(), rotation=45,
                ↪  horizontalalignment='right')
292             plt.ylabel('average frequency (Hz)')
293             figures[str(eq)] = figure
294         return figures
295
296
297     def boxplot_population_parameters(population: list, linthresh=LINTHRESH, figure=None,
        ↪  color=None, alt_title=None):
298         if figure:
299             plt.figure(figure)
300         else:
301             figure = plt.figure()
302
303         plt.grid(which='both')
304         columns = population[0]._optimize_get_state_keys()
305         data = np.array([m._optimize_get_state() for m in population])
306         df = pd.DataFrame(columns=columns, data=data)
307         fp = {'markeredgecolor': color}
308
309         if data.sum() > 0:
310             df.boxplot(vert=True, color=color, rot=90, flierprops=fp, )
311         if linthresh:
312             plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
313
314         if alt_title is None:
315             plt.title('%s Parameters distribution' % str(population[0]['name'].split('
                ↪  ')[1:]))
316         else:
317             plt.title(alt_title)
318
319         return figure
320
321
```

```
322 def histplot_population_parameters(populations: list[list], linthresh=LINTHRESH,
    ↪ title_postfix=''):
323     figures = dict()
324     columns = populations[0][0]._optimize_get_state_keys()
325     populations_names = [''.join(p[0]['name'].split(' ')[1:]) or 'SHAM' for p in
    ↪ populations]
326     for column in columns:
327         figure = plt.figure()
328         plt.grid(which='both')
329         data = [[x.P[column] for x in p] for p in populations]
330         df = pd.DataFrame(data)
331         df = df.transpose()
332         df.columns = populations_names
333         df = df.dropna()
334 
335         means = df.mean()
336         sem = df.sem()
337         means[np.isnan(means)] = 0
338         sem[np.isnan(sem)] = 0
339 
340         stat = stats.tukey_hsd(*np.array(df).transpose())
341         annotations = [pvalue_to_asterisks(v) for v in stat.pvalue[0]]
342 
343        container = plt.bar(means.index, means, yerr=sem, capsize=12, edgecolor='black',
344                            color=plt.cm.binary(range(0, 128, int(128 /
                            ↪ len(populations))), alpha=0), zorder=2)
345         plt.bar_label(container, annotations, size=18)
346 
347         for idx, bar in enumerate(container):
348             random.seed(1)
349             x = bar.get_x()
350             width = bar.get_width() / 2.
351             L = x + width - width / 2
352             R = x + width + width / 2
353             points = df[populations_names[idx]]
354             plt.plot([random_uniform(L, R) for _ in range(len(points))], points, 'o',
                ↪ alpha=0.25, color='orange',
355                      zorder=1)
356 
357         if linthresh:
358             plt.yscale('symlog', linthresh=linthresh, subs=SUBS)
359 
360         ok_count = min([len(x) for x in data])
361         plt.title(column + ' ($\geq$%s OK) ' % ok_count + title_postfix)
362         plt.xticks(rotation='vertical')
363 
364         figures[str(column)] = figure
365     return figures
366 
367 
368 def plot_max_eigenvalue_distribution(population: list, figure=None, title_label=''):
369     if figure:
370         plt.figure(figure)
371     else:
372         figure = plt.figure()
373     eigs = [e for pop in population for e in pop._eigenvalues_real_part()]
374 
375     plt.grid(which='both')
376     plt.hist(eigs, bins=100)  # , range=[0, 1])
377     plt.title('%sPopulation eigenvalues distribution (%s)' % (title_label,
    ↪ str(len(population))))
378     plt.ylabel('count')
379     plt.xlabel('$Re(\lambda)$')
380     plt.yscale('symlog')
```

```
381     return figure
382 
383 
384 def plot_population_fitness_distribution(population: list, figure=None,
    ↪ title_label=''):
385     if figure:
386         plt.figure(figure)
387     else:
388         figure = plt.figure()
389 
390     plt.grid(which='both')
391     fits = -np.log10(1 - np.array([i['fitness_history'][-1][1] for i in population]))
392     plt.hist(fits, bins=int((max(fits) + 2) * 10))  # , range=[0, 1])
393     plt.title('%sPopulation fitness distribution (%s)' % (title_label,
        ↪ str(len(population))))
394     plt.ylabel('count')
395     plt.xlabel('x')
396     return figure
397 
398 
399 def plot_population_fitness_delta_distribution(population: list, figure=None,
    ↪ title_label=''):
400     if figure:
401         plt.figure(figure)
402     else:
403         figure = plt.figure()
404 
405     plt.grid(which='both')
406     fits = np.array([i['fitness_history'][-1][1] for i in population]) - np.array(
407             [i['fitness_history'][0][1] for i in population])
408     plt.hist(fits, bins=int((max(fits) + 2) * 10))  # , range=[0, 1])
409     plt.title('%sPopulation $\Delta$fitness distribution (%s)' % (title_label,
        ↪ str(len(population))))
410     plt.ylabel('count')
411     plt.ylabel('$\Delta$fitness')
412     return figure
413 
414 
415 def plot_population_fitness(population: list, figure=None, color='blue'):
416     if figure:
417         plt.figure(figure)
418     else:
419         figure = plt.figure()
420 
421     fits = -np.log10(1 - np.array([i['fitness_history'][-1][1] for i in population]))
422     plt.barh([i['name'] for i in population], fits, color=color)  # , range=[0, 1])
423     plt.setp(figure.axes[0].get_xticklabels(), rotation=90,
        ↪ horizontalalignment='right')
424     # plt.setp(figure.axes[0].get_yticklabels(), rotation=90,
        ↪ horizontalalignment='right')
425     plt.title('Population fitness by individual')
426     plt.grid(which='both')
427     return figure
428 
429 
430 def plot_fitness(fitness_history: list, model_name, figure=None, base='generations'):
431     if figure:
432         plt.figure(figure)
433         if len(figure.axes) < 2:
434             plt.twinx()
435     else:
436         figure = plt.figure()
437         plt.twinx()
438 
```

```
439     history = np.array(fitness_history).transpose()
440     if not len(history):
441         history = np.array([[0], [0]])
442 
443     if base == 'generations':
444         x = list(range(len(history[0])))
445     else:
446         x = history[0]
447         x -= x[0]
448 
449     plt.sca(figure.axes[0])
450     plt.plot(x, history[1], label='Fitness', color='blue')
451 
452     plt.sca(figure.axes[1])
453     plt.plot(x, -np.log10(1 - history[1]), label='9s', color='red')
454     # plt.yscale('log', subs=SUBS,)
455     plt.grid(which='both')
456     plt.title(model_name + ' Fitness (blue) =  $1-10^{-y}$ (red) ' + '[over ' + base +
        ↪  ']')
457 
458     return figure
```

B.4 Basic exploration of the 'standard' subject

```
1  from CONF import POPULATION_BASE_PATH, SIMULATION_TIME
2  from models import *
3  from plotting import *
4 
5  PLOT_DERIVATIVES = False
6  PLOT_LINTHRESH = False  # 10 ** -4
7  BOXPLOT_LINTHRESH = 1e-4
8 
9 
10 def main(fit=True, plot=False):
11 
12     checkpoint_filename = False  # './HEALTHY_CHECKPOINT'
13     model = Healthy_combined_fit()
14     model.apply()
15     y0 = model.target_as_y0()
16     t0 = 0
17     T = SIMULATION_TIME
18 
19     if fit:
20         model['name'] = 'S_000'
21         model, fitness_history = model.optimize(y0, t0, T,
           ↪  save_checkpoint_name=checkpoint_filename)
22         model.save(POPULATION_BASE_PATH + 'S_000')
23     else:
24         model = model.load(POPULATION_BASE_PATH + 'S_000')
25 
26     if plot:
27         print_title("STEP 01: healthy fit on average target data", 'STEP 01')
28         plot_fitness(model['fitness_history'], model['name'], base='generations')
29         plot_fitness(model['fitness_history'], model['name'], base='time')
30         model.apply()
31         plot_parameters([model])
32         plot_model(model, model.target_as_y0(), t0, T,
33                    linthresh=PLOT_LINTHRESH)
34 
```

```
        lesions = [model.lesion_L6OHDA(), model.lesion_LDSP4(), model.lesion_LpCPA(),
                   model.lesion_L6OHDA_LDSP4(), model.lesion_L6OHDA_LpCPA()]

        for model in lesions:
            plot_parameters([model, model], linthresh=BOXPLOT_LINTHRESH)
            plot_model(model, model.target_as_y0(), t0, T, linthresh=PLOT_LINTHRESH)

        plt.show()


if __name__ == '__main__':
    main(fit=True, plot=False)
```

B.5 Optimization and plots of the whole population

```
import math
import os

from CONF import FIG_DPI, SIMULATION_TIME, POPULATION_BASE_PATH
from models import *
from plotting import *

N_JOBS = 1
PLOT_LINTHRESH = False  # 10 ** -4
BOXPLOT_LINTHRESH = 1e-3


def slice_populations(populations):
    groups = len(populations)
    size = len(populations[0])
    per_group = math.floor(size / groups)
    sliced = list()
    for i, group in enumerate(populations):
        sliced.append(group[i * per_group:(i + 1) * per_group])
    return sliced


def main(fit=True, plot=False):
    t0 = 0
    T = SIMULATION_TIME
    people_in_population = 240
    mutation_scale = 0.5 / 4

    if fit:
        base = Healthy_combined_fit()
        base['name'] = 'S_xxx'
        np.random.seed(1984)
        individuals = [base] + [base.new_mutated_target_model(scale=mutation_scale) for
        ↪  i in
                                range(people_in_population)]
        for i, ind in enumerate(individuals):
            ind['name'] = ind['name'].split('_')[0] + '_' + '%03i' % (i)
            ind.apply()

        for idx, ind in enumerate(individuals[::1]):
            try:
                ind = Healthy_combined_fit.load(POPULATION_BASE_PATH + '%s' %
                ↪  ind['name'])
                individuals[idx] = ind
```

```
43              except FileNotFoundError:
44                  filename = POPULATION_BASE_PATH + '%s' % ind['name']
45                  ind.apply()
46                  ind = ind.optimize(ind.target_as_y0(), t0, T,
                    ↪  save_checkpoint_name=filename)[0]
47                  individuals[idx] = ind
48                  ind.save(filename)
49
50      else:
51          files = sorted(filter(lambda x: x.startswith('S_'),
            ↪  os.listdir(POPULATION_BASE_PATH + '')))
52          individuals = [Healthy_combined_fit.load(POPULATION_BASE_PATH + '%s' % f) for f
            ↪  in files]
53          people_in_population = len(individuals)
54
55      if plot:
56          print_title("STEP 02: fitted population (%s) on target distribution with
            ↪  mutation scale %s" %
57                      (people_in_population, mutation_scale), 'STEP 02')
58
59          figure = plt.figure()
60          plt.grid(which='both')
61          for i in individuals:
62              plot_fitness(i['fitness_history'], 'Combined', figure=figure,
                ↪  base='generations')
63          figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_fitness_generations.png'), dpi=FIG_DPI,
64                         bbox_inches='tight')
65
66          figure = plt.figure()
67          plt.grid(which='both')
68          for i in individuals:
69              plot_fitness(i['fitness_history'], 'Combined', figure=figure, base='time')
70          figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_fitness_time.png'), dpi=FIG_DPI,
71                         bbox_inches='tight')
72
73          figure = plot_population_fitness_distribution(individuals)
74          figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_fitness_distribution.png'), dpi=FIG_DPI,
75                         bbox_inches='tight')
76
77          figure = boxplot_population_targets(individuals, linthresh=False,
            ↪  scatterplot=True)
78          figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_targets_before_fit.png'), dpi=FIG_DPI,
79                         bbox_inches='tight')
80
81          TP = [i.lesion_L6OHDA() for i in individuals]
82          figure = boxplot_population_targets(TP, linthresh=False, color='red')
83          TP = [i.lesion_LpCPA() for i in individuals]
84          figure = boxplot_population_targets(TP, figure=figure, linthresh=False,
            ↪  color='orange')
85          TP = [i.lesion_LDSP4() for i in individuals]
86          figure = boxplot_population_targets(TP, figure=figure, linthresh=False,
            ↪  color='blue')
87          figure = boxplot_population_targets(individuals, figure=figure,
            ↪  linthresh=False, color='black')
88          figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_targets_before_fit_lesions.png'), dpi=FIG_DPI,
89                         bbox_inches='tight')
90
91          reject_threshold = 1 - 2e-8
```

```
92          rejects = [ind for ind in individuals if ind['fitness_history'][-1][1] <=
            ↪  reject_threshold]
93          individuals = [ind for ind in individuals if ind['fitness_history'][-1][1] >
            ↪  reject_threshold]
94
95          figure = plt.figure()
96          plt.grid(which='both')
97          # plot_population_fitness(individuals, figure=figure, color='blue')
98          plot_population_fitness(rejects, figure=figure, color='red')
99
100         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_fitness_individual.png'), dpi=FIG_DPI,
101                        bbox_inches='tight')
102
103         figure = boxplot_population_parameters(individuals,
            ↪  linthresh=BOXPLOT_LINTHRESH)
104         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_parameters_distribution.png'), dpi=FIG_DPI,
105                        bbox_inches='tight')
106
107         populations = list()
108         for kind in ['lesion_SHAM', 'lesion_L6OHDA', 'lesion_LpCPA', 'lesion_LDSP4',
109                      'lesion_L6OHDA_LpCPA',
110                      'lesion_L6OHDA_LDSP4']:
111             # Healthy
112             current_individuals = [i.__getattribute__(kind)() for i in individuals]
113             populations.append(current_individuals)
114             boxplot_population_parameters(current_individuals,
                ↪  linthresh=BOXPLOT_LINTHRESH)
115             (boxplot, plot) = plot_population(individuals[0], current_individuals,
                ↪  None, t0, T,
116                                               linthresh=PLOT_LINTHRESH, plot_target=False)
117             boxplot.savefig(os.path.join(POPULATION_BASE_PATH,
                ↪  'population_target_%s.png' % kind), dpi=FIG_DPI,
118                             bbox_inches='tight')
119
120             stability_plot = plot_max_eigenvalue_distribution(
121                     current_individuals,
122                     title_label=str(current_individuals[0]['name'].split(' ')[
123                                     1:]) + ' ')
124             stability_plot.savefig(os.path.join(POPULATION_BASE_PATH,
                ↪  'population_max_eigenvalues_%s.png' % kind),
125                                    dpi=FIG_DPI,
126                                    bbox_inches='tight')
127
128         sliced_populations = slice_populations(populations)
129         sliced_populations_desc = ' ' + str([len(x) for x in sliced_populations])
130
131         figures_dict = boxplot_populations_last_value_by_equation(populations,
            ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
132                                                                   title_postfix=' whole
                                                                      ↪  population')
133         for eq_name, figure in figures_dict.items():
134             figure.savefig(os.path.join(POPULATION_BASE_PATH,
                ↪  'population_by_equation_%s.png' % eq_name), dpi=FIG_DPI,
135                            bbox_inches='tight')
136
137         figures_dict = boxplot_populations_last_value_by_equation(sliced_populations,
            ↪  linthresh=PLOT_LINTHRESH, t0=t0,
138                                                                   T=T,
                                                                      ↪  title_postfix=sliced_populations_desc)
139         for eq_name, figure in figures_dict.items():
140             figure.savefig(os.path.join(POPULATION_BASE_PATH,
                ↪  'population_by_equation_%s_sliced.png' % eq_name),
```

```
                    dpi=FIG_DPI,
                    bbox_inches='tight')

    figures_dict = histplot_populations_last_value_by_equation(populations,
    ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
                                                               title_postfix=' whole
                                                                ↪  population')
    for eq_name, figure in figures_dict.items():
        figure.savefig(os.path.join(POPULATION_BASE_PATH,
        ↪  'population_by_equation_hist_%s.png' % eq_name),
                    dpi=FIG_DPI,
                    bbox_inches='tight')

    figures_dict = histplot_populations_last_value_by_equation(sliced_populations,
    ↪  linthresh=PLOT_LINTHRESH, t0=t0,
                                                               T=T,
                                                               ↪  title_postfix=sliced_populations_desc)
    for eq_name, figure in figures_dict.items():
        figure.savefig(os.path.join(POPULATION_BASE_PATH,
        ↪  'population_by_equation_hist_%s_sliced.png' % eq_name),
                    dpi=FIG_DPI,
                    bbox_inches='tight')

    # Additional stuff generated for documentation

    # GP plots for comparison
    DSP4_pop = [populations[0], populations[1], populations[3], populations[5]]

    figures_dict = histplot_populations_last_value_by_equation(DSP4_pop,
    ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
                                                               title_postfix=' whole
                                                                ↪  population')
    figures_dict['GP'].savefig(os.path.join(POPULATION_BASE_PATH,
    ↪  'population_by_equation_hist_DSP4_GP.png'),
                               dpi=FIG_DPI,
                               bbox_inches='tight')
    PCPA_pop = [populations[0], populations[1], populations[2], populations[4]]

    figures_dict = histplot_populations_last_value_by_equation(PCPA_pop,
    ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
                                                               title_postfix=' whole
                                                                ↪  population')
    figures_dict['GP'].savefig(os.path.join(POPULATION_BASE_PATH,
    ↪  'population_by_equation_hist_PCPA_GP.png'),
                               dpi=FIG_DPI,
                               bbox_inches='tight')

    DSP4_pop = [sliced_populations[0], sliced_populations[1],
    ↪  sliced_populations[3], sliced_populations[5]]

    figures_dict = histplot_populations_last_value_by_equation(DSP4_pop,
    ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
                                                               title_postfix=' ' +
                                                               ↪  str([len(x) for
                                                               ↪  x in
                                                               ↪  DSP4_pop]))
    figures_dict['GP'].savefig(os.path.join(POPULATION_BASE_PATH,
    ↪  'population_by_equation_hist_DSP4_GP_sliced.png'),
                               dpi=FIG_DPI,
                               bbox_inches='tight')
    PCPA_pop = [sliced_populations[0], sliced_populations[1],
    ↪  sliced_populations[2], sliced_populations[4]]

```

```
185         figures_dict = histplot_populations_last_value_by_equation(PCPA_pop,
            ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
186                                                                    title_postfix=' ' +
                                                                       ↪  str([len(x) for
                                                                       ↪  x in
                                                                       ↪  DSP4_pop]))
187         figures_dict['GP'].savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_by_equation_hist_PCPA_GP_sliced.png'),
188                                    dpi=FIG_DPI,
189                                    bbox_inches='tight')
190
191         # STATS
192
193         with open(os.path.join(POPULATION_BASE_PATH, 'stats_individuals'), 'w') as f:
194             f.write(str(len(individuals)))
195         with open(os.path.join(POPULATION_BASE_PATH, 'stats_rejects'), 'w') as f:
196             f.write(str(len(rejects)))
197         with open(os.path.join(POPULATION_BASE_PATH, 'stats_population'), 'w') as f:
198             f.write(str(len(rejects) + len(individuals)))
199
200         average_generations = np.average([len(i['fitness_history']) for i in
            ↪  individuals])
201         with open(os.path.join(POPULATION_BASE_PATH, 'stats_average_generations'), 'w')
            ↪  as f:
202             f.write(str(int(average_generations)))
203         average_time = np.average([i['fitness_history'][-1][0] -
            ↪  i['fitness_history'][0][0] for i in individuals])
204         with open(os.path.join(POPULATION_BASE_PATH, 'stats_average_time'), 'w') as f:
205             f.write('%.2f' % (average_time / 3600.))
206
207         figure = plt.figure()
208         plt.grid(which='both')
209         for i in individuals[::int(len(individuals) / 10)]:
210             plot_fitness(i['fitness_history'], 'Combined', figure=figure,
                ↪  base='generations')
211         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_fitness_generations_subsample.png'), dpi=FIG_DPI,
212                        bbox_inches='tight')
213
214         figure = plt.figure()
215         plt.grid(which='both')
216         for i in individuals[::int(len(individuals) / 10)]:
217             plot_fitness(i['fitness_history'], 'Combined', figure=figure, base='time')
218         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_fitness_time_subsample.png'), dpi=FIG_DPI,
219                        bbox_inches='tight')
220
221         # Transient plots
222         m = individuals[5]
223
224         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
225         mL6OHDA = m.lesion_L6OHDA()
226         mL6OHDA['target'] = m['target']
227         figure = plot_model(mL6OHDA, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
228         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_L6OHDA.png'), dpi=FIG_DPI,
229                        bbox_inches='tight')
230
231         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
232         mLpCPA = m.lesion_LpCPA()
233         mLpCPA['target'] = m['target']
234         figure = plot_model(mLpCPA, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
```

```
235         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_LpCPA.png'), dpi=FIG_DPI,
236                        bbox_inches='tight')
237
238         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
239         mDSP4 = m.lesion_LDSP4()
240         mDSP4['target'] = m['target']
241         figure = plot_model(mDSP4, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
242         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_LDSP4.png'), dpi=FIG_DPI,
243                        bbox_inches='tight')
244
245         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
246         m6OHDA = m.lesion_L6OHDA()
247         m6OHDA['target'] = m['target']
248         figure = plot_model(m6OHDA, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
249         y0 = m.lesion_L6OHDA().simulate(m.target_as_y0(), 0, 0.1)['y'].transpose()[-1]
250         m6OHDALDSP4 = m.lesion_L6OHDA_LDSP4()
251         m6OHDALDSP4['target'] = m['target']
252         figure = plot_model(m6OHDALDSP4, y0, 0.2, 0.3, figure=figure, linthresh=False)
253         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_6OHDA+DSP4.png'), dpi=FIG_DPI,
254                        bbox_inches='tight')
255
256         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
257         m6OHDA = m.lesion_L6OHDA()
258         m6OHDA['target'] = m['target']
259         figure = plot_model(m6OHDA, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
260         y0 = m.lesion_L6OHDA().simulate(m.target_as_y0(), 0, 0.1)['y'].transpose()[-1]
261         m6OHDALpCPA = m.lesion_L6OHDA_LpCPA()
262         m6OHDALpCPA['target'] = m['target']
263         figure = plot_model(m6OHDALpCPA, y0, 0.2, 0.3, figure=figure, linthresh=False)
264         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_6OHDA+LpCPA.png'), dpi=FIG_DPI,
265                        bbox_inches='tight')
266
267         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
268         mLpCPA = m.lesion_L6OHDA_LpCPA()
269         mLpCPA['target'] = m['target']
270         figure = plot_model(mLpCPA, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
271         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_direct_6OHDA+LpCPA.png'), dpi=FIG_DPI,
272                        bbox_inches='tight')
273
274         figure = plot_model(m, m.target_as_y0(), 0, 0.1, figure=None, linthresh=False)
275         mDSP4 = m.lesion_L6OHDA_LDSP4()
276         mDSP4['target'] = m['target']
277         figure = plot_model(mDSP4, m.target_as_y0(), 0.1, 0.2, figure=figure,
            ↪  linthresh=False)
278         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'transient_example_direct_6OHDA+LDSP4.png'), dpi=FIG_DPI,
279                        bbox_inches='tight')
280
281         figures_dict = histplot_population_parameters(populations, title_postfix='
            ↪  whole population', linthresh=False)
282         for param, figure in figures_dict.items():
283             figure.savefig(os.path.join(POPULATION_BASE_PATH,
284                                         'populations_by_parameter_%s.png' % (param)),
285                            dpi=FIG_DPI,
286                            bbox_inches='tight')
```

```
        figures_dict = histplot_population_parameters(sliced_populations,
        ↪  title_postfix=sliced_populations_desc,
                                                      linthresh=False)
        for param, figure in figures_dict.items():
            figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                        'populations_by_parameter_%s_sliced.png' %
                                        ↪  (param)),
                           dpi=FIG_DPI,
                           bbox_inches='tight')


if __name__ == '__main__':
    main(fit=True, plot=False)
```

B.6 Parameter sensitivity analysis

```
import os
from collections import defaultdict

import matplotlib.cm as cm
import seaborn as sns
from matplotlib.colors import Normalize

from CONF import *
from models import *
from plotting import *

plt.rcParams['figure.figsize'] = (7, 7)

N_JOBS = -1

PLOT_LINTHRESH = False  # 1e-4
BOXPLOT_LINTHRESH = False  # 1e-4


def mutation_state(model, mutation_scale, mutations_number, parameter_index, y0, t0,
↪  T, label='N/A'):
    model_state = model._optimize_get_state()
    value = model_state[parameter_index]
    value_range = np.linspace(value * (1 - mutation_scale), value * (1 +
    ↪  mutation_scale), mutations_number)
    targets = list()
    for v in value_range:
        mutated_model = model.copy()
        new_state = model_state.copy()
        new_state[parameter_index] = v
        mutated_model._optimize_set_state(new_state)
        res = mutated_model.simulate(y0, 0, T)
        targets.append([v, res['y'].transpose()[-1], res['t'][-1]])
    return label, targets


def main(fit=True, plot=False):
    files = sorted(filter(lambda x: x.startswith('S_'),
    ↪  os.listdir(POPULATION_BASE_PATH + '')))
    individuals = [Healthy_combined_fit.load(POPULATION_BASE_PATH + '%s' % f) for f in
    ↪  files]
    reject_threshold = 1 - 2e-8
    individuals = [ind.lesion_SHAM() for ind in individuals if
    ↪  ind['fitness_history'][-1][1] > reject_threshold]

```

```
    t0 = 0
    T = SIMULATION_TIME
    T_mutation = T

    mutation_scale = 0.5
    mutations = 100
    mutations_plot_one_every = 1

    available_params = individuals[0]._optimize_get_state_keys()

    if fit:

        states = defaultdict(list)
        for model in individuals:
            res = model.simulate(model.target_as_y0(), 0, T)
            states['HEALTHY'] += [[0, res['y'].transpose()[-1], res['t'][-1]]]

        mutation_states = Parallel(n_jobs=N_JOBS)(delayed(mutation_state)(model,
                                                                          mutation_scale,
                                                                             mutations,
                                                                         parameter_index,
                                                                    model.target_as_y0(),
                                                                             t0,
                                                                             T_mutation,
                                                                      parameter_name) for
                                                  parameter_index, parameter_name in
                                                  ↪  enumerate(available_params)
                                                  for model in individuals)

        with open('./fitted_models/S_000_SENSITIVITY_STATES', 'bw') as f:
            pickle.dump(mutation_states, f)

        for parameter, results in mutation_states:
            states[parameter] += results

        for key, val in states.items():
            states[key] = sorted(val, key=lambda x: x[0])

        with open('./fitted_models/S_000_SENSITIVITY_STATES', 'bw') as f:
            pickle.dump(states, f)
    else:

        with open('./fitted_models/S_000_SENSITIVITY_STATES', 'br') as f:
            states = pickle.load(f)

    if plot:
        print_title("STEP 03: parameter sensitivity analysis", 'STEP 03')
        model = Healthy_combined_fit().lesion_SHAM()

        healthy_reference = None

        for param, data in list(states.items()):
            continue

            figure = plt.figure()

            values = np.array([x[0] for x in data])
            states_data = np.array([x[1] for x in data])
            end_times = np.array([x[2] for x in data])

            color_normalizer = Normalize(vmin=min(values), vmax=max(values))
            color_map = cm.ScalarMappable(norm=color_normalizer, cmap=cm.spring)
            colors = [color_map.to_rgba(x) for x in values]

```

```
104             df = pd.DataFrame(columns=model["equations"], data=states_data)
105
106             if param == 'HEALTHY':
107                 healthy_reference = df
108                 df.boxplot(color='red')
109             elif healthy_reference is not None:
110                 healthy_reference.boxplot(color='red')
111                 df.boxplot(color='green')
112
113             for i, c in enumerate(df.columns):
114                 y = df[c]
115                 plt.scatter([i + 1] * len(y), y, alpha=0.3, s=25, c=colors)  # , c=y,
                    ↪  s=10)
116                 for n in range(len(y)):
117                     if end_times[n] < T:
118                         plt.scatter([i + 1], y[n], alpha=1, s=200, c='red', marker='x')
                            ↪  # , c=y, s=10)
119
120             if PLOT_LINTHRESH:
121                 plt.yscale('symlog', linthresh=PLOT_LINTHRESH, subs=[1, 2, 3, 4, 5, 6,
                    ↪  7, 8, 9])
122             plt.title(str(param) + '   %s - %s' % (min(values), max(values)))
123
124         # Sensitivity matrix
125         no_param_states = states.pop('HEALTHY')
126
127         sensitivity_matrix = np.zeros(shape=(len(states.keys()),
            ↪  len(no_param_states[0][1])))
128
129         for i, (param, results) in enumerate(states.items()):
130             values, good_results = list(zip(*[(x[0], x[1]) for x in results if x[2] >=
                ↪  T]))
131             values = pd.DataFrame(values)
132             good_results = pd.DataFrame(good_results)
133             G = (good_results / good_results.median()).std()
134             V = (values / values.median()).std()
135             sensitivity_index = G / float(V)
136             sensitivity_matrix[i] = sensitivity_index
137         sensitivity_matrix /= sensitivity_matrix.max()
138         sensitivity_df = pd.DataFrame(sensitivity_matrix, columns=model['equations'],
            ↪  index=states.keys())
139
140         figure = plt.figure()
141         sns.heatmap(sensitivity_df, annot=True, cmap='YlOrBr')
142         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'sensitivity_analysis_matrix.png'), dpi=FIG_DPI,
143                        bbox_inches='tight')
144
145
146 if __name__ == '__main__':
147     main(fit=True, plot=False)
```

B.7 Optimization of candidate treatments

```
1  from CONF import FIG_DPI, SIMULATION_TIME, POPULATION_BASE_PATH
2  from models import *
3  from plotting import *
4  from s02_population import slice_populations
5  import os
6
7  PLOT_DERIVATIVES = False
```

```
8   PLOT_LINTHRESH = False  # 10 ** -4
9   BOXPLOT_LINTHRESH = 1e-3
10
11  BL = {'lesion_SHAM'         : '+SHAM',
12        'lesion_L6OHDA'       : '+6OHDA',
13        'lesion_LpCPA'        : '+pCPA',
14        'lesion_LDSP4'        : '+DSP4',
15        'lesion_L6OHDA_LpCPA': '+6OHDA+pCPA',
16        'lesion_L6OHDA_LDSP4': '+6OHDA+DSP4'}
17
18  POPULATION_BASE_PATH___CURE_DRN = os.path.join(POPULATION_BASE_PATH, 'CURE_DRN')
19  POPULATION_BASE_PATH___CURE_LC = os.path.join(POPULATION_BASE_PATH, 'CURE_LC')
20  POPULATION_BASE_PATH___CURE_COMBINED = os.path.join(POPULATION_BASE_PATH,
    ↪  'CURE_DRN_LC')
21
22  if not os.path.exists(POPULATION_BASE_PATH___CURE_DRN):
23      os.mkdir(POPULATION_BASE_PATH___CURE_DRN)
24  if not os.path.exists(POPULATION_BASE_PATH___CURE_LC):
25      os.mkdir(POPULATION_BASE_PATH___CURE_LC)
26  if not os.path.exists(POPULATION_BASE_PATH___CURE_COMBINED):
27      os.mkdir(POPULATION_BASE_PATH___CURE_COMBINED)
28
29
30  def is_already_optimized(filename):
31      try:
32          Healthy_combined_fit.load(filename)
33          return True
34      except:
35          return False
36
37  def main(fit=True, plot=False):
38      t0 = 0
39      T = SIMULATION_TIME
40      if fit:
41          files = sorted(filter(lambda x: x.startswith('S_'),
            ↪  os.listdir(POPULATION_BASE_PATH + '')))
42          individuals = [Healthy_combined_fit.load(POPULATION_BASE_PATH + '%s' % f) for f
            ↪  in files]
43          reject_threshold = 1 - 2e-8
44          individuals = [ind for ind in individuals if ind['fitness_history'][-1][1] >
            ↪  reject_threshold]
45          individuals_6OHDA = [ind._impose_target(ind.lesion_L6OHDA()) for ind in
            ↪  individuals]
46
47          individuals_cure_DRN = list()
48          individuals_cure_LC = list()
49          individuals_cure_combined = list()
50
51          for i in individuals_6OHDA:
52              cure_individual = Cure_DRN()
53              cure_individual.update(i.copy(keep_fitness_history=False))
54              cure_individual.apply()
55              individuals_cure_DRN.append(cure_individual)
56
57              cure_individual = Cure_LC()
58              cure_individual.update(i.copy(keep_fitness_history=False))
59              cure_individual.apply()
60              individuals_cure_LC.append(cure_individual)
61
62              cure_individual = Cure_combined()
63              cure_individual.update(i.copy(keep_fitness_history=False))
64              cure_individual.apply()
65              individuals_cure_combined.append(cure_individual)
66
```

```
67          optimizer_popsize = 60
68          tol = 1e-6
69          for idx, individual in enumerate(individuals_cure_DRN):
70              individual.apply()
71              filename = (POPULATION_BASE_PATH___CURE_DRN + '/%s' %
                ↪  individual['name']).replace(' ', '_')
72              if not is_already_optimized(filename):
73                  fitted_individual = \
74                      individual.optimize(individual.target_as_y0(), t0, T, seed=1,
                        ↪  popsize=optimizer_popsize, tol=tol)[0]
75                  fh = fitted_individual['fitness_history']
76                  fitted_individual['fitness_history'] = fh
77                  fitted_individual.save(filename)
78                  individuals_cure_DRN[idx] = fitted_individual
79
80          for idx, individual in enumerate(individuals_cure_LC):
81              individual.apply()
82              filename = (POPULATION_BASE_PATH___CURE_LC + '/%s' %
                ↪  individual['name']).replace(' ', '_')
83              if not is_already_optimized(filename):
84                  fitted_individual = \
85                      individual.optimize(individual.target_as_y0(), t0, T, seed=1,
                        ↪  popsize=optimizer_popsize, tol=tol)[0]
86                  fh = fitted_individual['fitness_history']
87                  fitted_individual['fitness_history'] = fh
88                  fitted_individual.save(filename)
89                  individuals_cure_LC[idx] = fitted_individual
90
91          for idx, individual in enumerate(individuals_cure_combined):
92              individual.apply()
93              filename = (POPULATION_BASE_PATH___CURE_COMBINED + '/%s' %
                ↪  individual['name']).replace(' ', '_')
94              if not is_already_optimized(filename):
95                  fitted_individual = \
96                      individual.optimize(individual.target_as_y0(), t0, T, seed=1,
                        ↪  popsize=int(optimizer_popsize / 2),
97                                          tol=tol)[0]
98                  fh = fitted_individual['fitness_history']
99                  fitted_individual['fitness_history'] = fh
100                 fitted_individual.save(filename)
101                 individuals_cure_combined[idx] = fitted_individual
102
103
104     files = sorted(filter(lambda x: x.startswith('S_'),
        ↪  os.listdir(POPULATION_BASE_PATH + '')))
105     individuals = [Healthy_combined_fit.load(POPULATION_BASE_PATH + '%s' % f) for f in
        ↪  files]
106     reject_threshold = 1 - 2e-8
107     individuals = [ind for ind in individuals if ind['fitness_history'][-1][1] >
        ↪  reject_threshold]
108     individuals_6OHDA = [ind._impose_target(ind.lesion_L6OHDA()) for ind in
        ↪  individuals]
109
110     files = sorted(filter(lambda x: x.startswith('S_'),
        ↪  os.listdir(POPULATION_BASE_PATH___CURE_DRN + '')))
111     individuals_cure_DRN = [Cure_DRN.load(POPULATION_BASE_PATH___CURE_DRN + '/%s' % f)
        ↪  for f in files]
112     for i in individuals_cure_DRN:
113         i.apply()
114
115     files = sorted(filter(lambda x: x.startswith('S_'),
        ↪  os.listdir(POPULATION_BASE_PATH___CURE_LC + '')))
116     individuals_cure_LC = [Cure_LC.load(POPULATION_BASE_PATH___CURE_LC + '/%s' % f) for
        ↪  f in files]
```

```
117     for i in individuals_cure_LC:
118         i.apply()
119 
120     files = sorted(filter(lambda x: x.startswith('S_'),
        ↪  os.listdir(POPULATION_BASE_PATH___CURE_COMBINED + '')))
121     individuals_cure_combined =
        ↪  [Cure_combined.load(POPULATION_BASE_PATH___CURE_COMBINED + '/%s' % f) for f in
122                            files]
123     for i in individuals_cure_combined:
124         i.apply()
125 
126     if plot:
127         print_title("STEP 04: treatment", 'STEP 04')
128 
129         # FITNESS DISTRIBUTIONS
130         figure = plot_population_fitness_distribution(individuals, title_label='SHAM ')
131         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_fitness_distribution.png'), dpi=FIG_DPI,
132                        bbox_inches='tight')
133 
134         figure = plot_population_fitness_distribution(individuals_cure_DRN,
            ↪  title_label='cure_DRN ')
135         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_DRN_fitness_distribution.png'), dpi=FIG_DPI,
136                        bbox_inches='tight')
137 
138         figure = plot_population_fitness_distribution(individuals_cure_LC,
            ↪  title_label='cure_LC ')
139         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_LC_fitness_distribution.png'), dpi=FIG_DPI,
140                        bbox_inches='tight')
141 
142         figure = plot_population_fitness_distribution(individuals_cure_combined,
            ↪  title_label='cure_combined ')
143         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_combined_fitness_distribution.png'),
144                        dpi=FIG_DPI,
145                        bbox_inches='tight')
146 
147         # DELTA FITNESS
148         figure = plot_population_fitness_delta_distribution(individuals,
            ↪  title_label='SHAM ')
149         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_fitness_delta_distribution.png'),
150                        dpi=FIG_DPI,
151                        bbox_inches='tight')
152 
153         figure = plot_population_fitness_delta_distribution(individuals_cure_DRN,
            ↪  title_label='cure_DRN ')
154         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_DRN_fitness_delta_distribution.png'),
155                        dpi=FIG_DPI,
156                        bbox_inches='tight')
157 
158         figure = plot_population_fitness_delta_distribution(individuals_cure_LC,
            ↪  title_label='cure_LC ')
159         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_LC_fitness_delta_distribution.png'),
160                        dpi=FIG_DPI,
161                        bbox_inches='tight')
162 
163         figure = plot_population_fitness_delta_distribution(individuals_cure_combined,
            ↪  title_label='cure_combined ')
```

```
164         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_combined_fitness_delta_distribution.png'),
165                        dpi=FIG_DPI,
166                        bbox_inches='tight')
167
168         # STABILITY
169         stability_plot = plot_max_eigenvalue_distribution(individuals,
            ↪  title_label='SHAM ')
170         stability_plot.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'cure_population_max_eigenvalues_%s.png' % "SHAM"),
171                                dpi=FIG_DPI,
172                                bbox_inches='tight')
173
174         stability_plot = plot_max_eigenvalue_distribution(individuals_cure_DRN,
            ↪  title_label='cure_DRN ')
175         stability_plot.savefig(
176                 os.path.join(POPULATION_BASE_PATH,
                        ↪  'cure_population_max_eigenvalues_%s.png' % "cure_DRN"),
177                 dpi=FIG_DPI,
178                 bbox_inches='tight')
179
180         stability_plot = plot_max_eigenvalue_distribution(individuals_cure_LC,
            ↪  title_label='cure_LC ')
181         stability_plot.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'cure_population_max_eigenvalues_%s.png' % "cure_LC"),
182                                dpi=FIG_DPI,
183                                bbox_inches='tight')
184
185         stability_plot = plot_max_eigenvalue_distribution(individuals_cure_combined,
            ↪  title_label='cure_combined ')
186         stability_plot.savefig(
187                 os.path.join(POPULATION_BASE_PATH,
                        ↪  'cure_population_max_eigenvalues_%s.png' % "cure_combined"),
188                 dpi=FIG_DPI,
189                 bbox_inches='tight')
190
191         # PARAMETERS
192         figure = boxplot_population_parameters(individuals,
            ↪  linthresh=BOXPLOT_LINTHRESH)
193         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_sham_parameters_distribution.png'),
194                        dpi=FIG_DPI,
195                        bbox_inches='tight')
196
197         for i in individuals_cure_DRN:
198             i['parameters']['a_EXT_DRN'] = i.P['a_EXT_DRN']
199         figure = boxplot_population_parameters(individuals_cure_DRN,
            ↪  linthresh=BOXPLOT_LINTHRESH)
200         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_DRN_parameters_distribution.png'),
201                        dpi=FIG_DPI,
202                        bbox_inches='tight')
203
204         for i in individuals_cure_LC:
205             i['parameters']['a_EXT_LC'] = i.P['a_EXT_LC']
206         figure = boxplot_population_parameters(individuals_cure_LC,
            ↪  linthresh=BOXPLOT_LINTHRESH)
207         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_LC_parameters_distribution.png'),
208                        dpi=FIG_DPI,
209                        bbox_inches='tight')
210
211         for i in individuals_cure_combined:
212             i['parameters']['a_EXT_DRN'] = i.P['a_EXT_DRN']
```

```
213             i['parameters']['a_EXT_LC'] = i.P['a_EXT_LC']
214         figure = boxplot_population_parameters(individuals_cure_combined,
            ↪  linthresh=BOXPLOT_LINTHRESH)
215         figure.savefig(os.path.join(POPULATION_BASE_PATH,
            ↪  'population_cure_combined_parameters_distribution.png'),
216                        dpi=FIG_DPI,
217                        bbox_inches='tight')
218
219         populations = [individuals, individuals_6OHDA,
220                        [i.cure_DRN() for i in individuals_cure_DRN],
221                        [i.cure_LC() for i in individuals_cure_LC],
222                        [i.cure_DRN_LC() for i in individuals_cure_combined],
223                        ]
224
225         sliced_populations = slice_populations(populations)
226         sliced_populations_desc = ' ' + str([len(x) for x in sliced_populations])
227
228         figures_dict = boxplot_populations_last_value_by_equation(populations,
            ↪  linthresh=PLOT_LINTHRESH, t0=t0, T=T,
229                                                                   title_postfix=' whole
                                                                    ↪  population')
230
231         for eq_name, figure in figures_dict.items():
232             figure.savefig(os.path.join(POPULATION_BASE_PATH,
233                                         'cure_population_6OHDA_by_equation_%s.png' %
                                            ↪  (eq_name)),
234                            dpi=FIG_DPI,
235                            bbox_inches='tight')
236
237         figures_dict = boxplot_populations_last_value_by_equation(sliced_populations,
            ↪  linthresh=PLOT_LINTHRESH,
238                                                                   t0=t0, T=T,
239                                                   title_postfix=sliced_populations_desc)
240         for eq_name, figure in figures_dict.items():
241             figure.savefig(os.path.join(POPULATION_BASE_PATH,
242                                         'cure_population_6OHDA_by_equation_%s_sliced.png'
                                            ↪  % (eq_name)),
243                            dpi=FIG_DPI,
244                            bbox_inches='tight')
245
246         figures_dict = histplot_populations_last_value_by_equation(populations,
            ↪  linthresh=PLOT_LINTHRESH, t0=t0,
247                                                                    T=T, title_postfix=' whole
                                                                     ↪  population')
248         for eq_name, figure in figures_dict.items():
249             figure.savefig(os.path.join(POPULATION_BASE_PATH,
250                                         'cure_population_6OHDA_by_equation_hist_%s.png'
                                            ↪  % (eq_name)),
251                            dpi=FIG_DPI,
252                            bbox_inches='tight')
253
254         figures_dict = histplot_populations_last_value_by_equation(sliced_populations,
            ↪  linthresh=PLOT_LINTHRESH,
255                                                                    t0=t0,
256                                                                    T=T,
                                                                    ↪  title_postfix=sliced_populations_desc)
257
258         for eq_name, figure in figures_dict.items():
259             figure.savefig(os.path.join(POPULATION_BASE_PATH,
260                                         'cure_population_6OHDA_by_equation_hist_%s_sliced.png'
                                            ↪  % (eq_name)),
261                            dpi=FIG_DPI,
262                            bbox_inches='tight')
263
```

```
264         base_lesion = 'lesion_L6OHDA'
265         kind = ['SHAM', BL[base_lesion], '+EXT_DRN', '+EXT_LC', '+EXT_DRN+EXT_LC']
266         for fitted_individual, pop in enumerate(populations):
267             (boxplot, plot) = plot_population(pop[0], pop, None, t0, T,
                ↪  linthresh=PLOT_LINTHRESH, plot_target=False)
268             boxplot.savefig(
269                     os.path.join(POPULATION_BASE_PATH,
270                                  'cure_population_%s_target_%s.png' % (base_lesion,
                                     ↪  kind[fitted_individual])),
271                     dpi=FIG_DPI,
272                     bbox_inches='tight')
273             plot.savefig(os.path.join(POPULATION_BASE_PATH,
274                                       'cure_population_%s_target_%s_plot.png' % (
275                                           base_lesion, kind[fitted_individual])),
276                          dpi=FIG_DPI,
277                          bbox_inches='tight')
278 
279         for fitted_individual, pop in enumerate(sliced_populations):
280             (boxplot, plot) = plot_population(pop[0], pop, None, t0, T,
                ↪  linthresh=PLOT_LINTHRESH, plot_target=False)
281             boxplot.savefig(
282                     os.path.join(POPULATION_BASE_PATH,
283                                  'cure_population_%s_target_%s_sliced.png' % (
284                                      base_lesion, kind[fitted_individual])),
285                     dpi=FIG_DPI,
286                     bbox_inches='tight')
287             plot.savefig(os.path.join(POPULATION_BASE_PATH,
288                                       'cure_population_%s_target_%s_plot_sliced.png' % (
289                                           base_lesion, kind[fitted_individual])),
290                          dpi=FIG_DPI,
291                          bbox_inches='tight')
292 
293         # CURABLE VS UNCURABLE
294         stats_individuals_cure_combined = list()
295         stats_individuals = list()
296         stats_individuals_L6OHDA = list()
297 
298         for i in individuals:
299             sham = i.lesion_SHAM()
300             l6ohda = Healthy_combined_fit()
301             l6ohda.update(i.lesion_L6OHDA().copy())
302             l6ohda = l6ohda.lesion_SHAM()
303             sham['fitness_history'] = i['fitness_history']
304             l6ohda['fitness_history'] = i['fitness_history']
305             stats_individuals.append(sham)
306             stats_individuals_L6OHDA.append(l6ohda)
307 
308         for i in individuals_cure_combined:
309             cured = i.cure_DRN_LC().lesion_SHAM()
310             cured['fitness_history'] = i['fitness_history']
311             stats_individuals_cure_combined.append(cured)
312 
313         cured_threshold = 5
314 
315         cured = [i for i in stats_individuals_cure_combined if
316                  -np.log10(1 - i['fitness_history'][-1][1]) >= cured_threshold]
317         not_cured = [i for i in stats_individuals_cure_combined if
318                      -np.log10(1 - i['fitness_history'][-1][1]) < cured_threshold]
319 
320         figure = boxplot_population_parameters(stats_individuals,
            ↪  linthresh=BOXPLOT_LINTHRESH, color='black')
321         figure = boxplot_population_parameters(stats_individuals_L6OHDA,
            ↪  linthresh=BOXPLOT_LINTHRESH, color='orange',
```

```
                                        figure=figure, alt_title='SHAM vs 6OHDA
                                        ↪  (%s - %s)' % (
            len(stats_individuals), len(stats_individuals_L6OHDA)))
    figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                'cure_population_parameters_SHAM_vs_6OHDA.png'),
                   dpi=FIG_DPI,
                   bbox_inches='tight')

    figure = boxplot_population_parameters(stats_individuals,
    ↪  linthresh=BOXPLOT_LINTHRESH, color='black')
    figure = boxplot_population_parameters(stats_individuals_cure_combined,
    ↪  linthresh=BOXPLOT_LINTHRESH,
                                        color='orange',
                                       figure=figure, alt_title='SHAM vs TREATED
                                        ↪  (%s - %s)' % (
            len(stats_individuals), len(stats_individuals_cure_combined)))
    figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                'cure_population_parameters_SHAM_vs_TREATED.png'),
                   dpi=FIG_DPI,
                   bbox_inches='tight')

    figure = boxplot_population_parameters(stats_individuals_L6OHDA,
    ↪  linthresh=BOXPLOT_LINTHRESH, color='black')
    figure = boxplot_population_parameters(stats_individuals_cure_combined,
    ↪  linthresh=BOXPLOT_LINTHRESH,
                                        color='orange',
                                      figure=figure, alt_title='6OHDA vs TREATED
                                        ↪  (%s - %s)' % (
            len(stats_individuals_L6OHDA), len(stats_individuals_cure_combined)))
    figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                'cure_population_parameters_6OHDA_vs_TREATED.png'),
                   dpi=FIG_DPI,
                   bbox_inches='tight')

    figure = boxplot_population_parameters(cured, linthresh=BOXPLOT_LINTHRESH,
    ↪  color='green')
    figure = boxplot_population_parameters(not_cured, linthresh=BOXPLOT_LINTHRESH,
    ↪  color='red', figure=figure,
                                       alt_title='Cured vs Not-Cured (%s - %s)' % (
                                            len(cured), len(not_cured)))
    figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                'cure_population_parameters_Cured_vs_NotCured.png'),
                   dpi=FIG_DPI,
                   bbox_inches='tight')

    figure = boxplot_population_parameters(stats_individuals_L6OHDA,
    ↪  linthresh=BOXPLOT_LINTHRESH, color='black')
    figure = boxplot_population_parameters(cured, linthresh=BOXPLOT_LINTHRESH,
    ↪  color='orange',
                                       figure=figure, alt_title='6OHDA vs Cured
                                        ↪  (%s - %s)' % (
            len(stats_individuals_L6OHDA), len(cured)))
    figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                'cure_population_parameters_6OHDA_vs_CURED.png'),
                   dpi=FIG_DPI,
                   bbox_inches='tight')

    figure = boxplot_population_parameters(stats_individuals_L6OHDA,
    ↪  linthresh=BOXPLOT_LINTHRESH, color='black')
    figure = boxplot_population_parameters(not_cured, linthresh=BOXPLOT_LINTHRESH,
    ↪  color='orange',
                                        figure=figure, alt_title='6OHDA vs
                                        ↪  Not-Cured (%s - %s)' % (
            len(stats_individuals_L6OHDA), len(not_cured)))
```

```
371         figure.savefig(os.path.join(POPULATION_BASE_PATH,
372                                     'cure_population_parameters_6OHDA_vs_NotCured.png'),
373                        dpi=FIG_DPI,
374                        bbox_inches='tight')
375
376         a_EXT_DRN = list()
377         a_EXT_LC = list()
378         CDRN___a_EXT_DRN = list()
379         CLC___a_EXT_LC = list()
380         FITS = list()
381         for i in [i for i in individuals_cure_combined if
382                   -np.log10(1 - i['fitness_history'][-1][1]) >= cured_threshold]:
383             a_EXT_DRN.append(i.P['a_EXT_DRN'])
384             a_EXT_LC.append(i.P['a_EXT_LC'])
385             CDRN___a_EXT_DRN.append(i.P['CDRN___a_EXT_DRN'])
386             CLC___a_EXT_LC.append(i.P['CLC___a_EXT_LC'])
387             FITS.append(-np.log10(1 - i['fitness_history'][-1][1]))
388
389         a_EXT_DRN = np.array(a_EXT_DRN)
390         a_EXT_LC = np.array(a_EXT_LC)
391         CDRN___a_EXT_DRN = np.array(CDRN___a_EXT_DRN)
392         CLC___a_EXT_LC = np.array(CLC___a_EXT_LC)
393
394         D_DRN = (CDRN___a_EXT_DRN - a_EXT_DRN) / a_EXT_DRN
395         D_LC = (CLC___a_EXT_LC - a_EXT_LC) / a_EXT_LC
396
397         FITS = np.array(FITS)
398         FITS = 255 * (FITS - min(FITS)) / (max(FITS) - min(FITS))
399         FITS = [plt.cm.plasma(int(f)) for f in FITS]
400         figure = plt.figure()
401
402         plt.scatter(D_LC, D_DRN, c=FITS)
403         plt.title('Treatment Combined Relative $\Delta$a_EXT_LC vs $\Delta$a_EXT_DRN
            ↪  (%s)' % len(D_LC))
404         plt.xlabel('$\Delta$a_EXT_LC')
405         plt.ylabel('$\Delta$a_EXT_DRN')
406         plt.xscale('symlog', linthresh=1e-1, subs=SUBS)
407         plt.yscale('symlog', linthresh=1e-8, subs=SUBS)
408         plt.grid(which='both')
409         figure.savefig(os.path.join(POPULATION_BASE_PATH,
410                                     'cure_combined_relativedelta_parameters.png'),
411                        dpi=FIG_DPI,
412                        bbox_inches='tight')
413
414         figures_dict = histplot_population_parameters(populations, title_postfix='
            ↪  whole population', linthresh=False)
415         for param, figure in figures_dict.items():
416             figure.savefig(os.path.join(POPULATION_BASE_PATH,
417                                         'cure_populations_by_parameter_%s.png' % (param)),
418                            dpi=FIG_DPI,
419                            bbox_inches='tight')
420
421         figures_dict = histplot_population_parameters(sliced_populations,
            ↪  title_postfix=sliced_populations_desc,
422                                                       linthresh=False)
423         for param, figure in figures_dict.items():
424             figure.savefig(os.path.join(POPULATION_BASE_PATH,
425                                         'cure_populations_by_parameter_%s_sliced.png' %
                                            ↪  (param)),
426                            dpi=FIG_DPI,
427                            bbox_inches='tight')
428
429         for i in cured:
430             i['name'] = i['name'].split(' ')[0] + ' CURED'
```

```
        for i in not_cured:
            i['name'] = i['name'].split(' ')[0] + ' NOT_CURED'

        populations = [individuals, individuals_6OHDA, cured, not_cured]
        figures_dict = histplot_population_parameters(populations, title_postfix='
        ↪  whole population', linthresh=False)
        for param, figure in figures_dict.items():
            figure.savefig(os.path.join(POPULATION_BASE_PATH,
                                        'cure_vs_not_cured_populations_by_parameter_%s.png'
                                            ↪  % (param)),
                           dpi=FIG_DPI,
                           bbox_inches='tight')


if __name__ == '__main__':
    # main(fit=False, plot=True)
    main(fit=True, plot=False)
```

B.8 Statistics tables

```
from CONF import SIMULATION_TIME, POPULATION_BASE_PATH
from models import *
from plotting import *
from s02_population import slice_populations
from s04_treatment import BL, POPULATION_BASE_PATH___CURE_COMBINED,
↪  POPULATION_BASE_PATH___CURE_DRN, \
    POPULATION_BASE_PATH___CURE_LC
import os

PLOT_DERIVATIVES = False
PLOT_LINTHRESH = False  # 10 ** -4
BOXPLOT_LINTHRESH = 1e-4


def tukey__str__(tukey_stat):
    # Note: `__str__` prints the confidence intervals from the most
    # recent call to `confidence_interval`. If it has not been called,
    # it will be called with the default CL of .95.
    if tukey_stat._ci is None:
        tukey_stat.confidence_interval(confidence_level=.95)
    s = ("Tukey's HSD Pairwise Group Comparisons"
         f" ({tukey_stat._ci_cl * 100:.1f}% Confidence Interval)\n")
    s += "Comparison  Statistic   p-value     Lower  CI   Upper CI\n"
    for i in range(tukey_stat.pvalue.shape[0]):
        for j in range(i, tukey_stat.pvalue.shape[0]):
            if i != j:
                s += (f" ({i} - {j}) {tukey_stat.statistic[i, j]:>12.3e}"
                      f"{tukey_stat.pvalue[i, j]:>12.3e}"
                      f"{tukey_stat._ci.low[i, j]:>12.3e}"
                      f"{tukey_stat._ci.high[i, j]:>12.3e}\n")
    return s


def main(fit=True, plot=False):
    t0 = 0
    T = SIMULATION_TIME
    if fit:
        pass
    else:
        files = sorted(filter(lambda x: x.startswith('S_'),
        ↪  os.listdir(POPULATION_BASE_PATH + '')))
```

```
40          individuals = [Healthy_combined_fit.load(POPULATION_BASE_PATH + '%s' % f) for f
            ↪  in files]
41          reject_threshold = 1 - 2e-8
42          individuals = [ind for ind in individuals if ind['fitness_history'][-1][1] >
            ↪  reject_threshold]
43          individuals_6OHDA = [ind._impose_target(ind.lesion_L6OHDA()) for ind in
            ↪  individuals]
44
45          files = sorted(filter(lambda x: x.startswith('S_'),
            ↪  os.listdir(POPULATION_BASE_PATH___CURE_DRN + '')))
46          individuals_cure_DRN = [Cure_DRN.load(POPULATION_BASE_PATH___CURE_DRN + '/%s' %
            ↪  f) for f in files]
47          for i in individuals_cure_DRN:
48              i.apply()
49
50          files = sorted(filter(lambda x: x.startswith('S_'),
            ↪  os.listdir(POPULATION_BASE_PATH___CURE_LC + '')))
51          individuals_cure_LC = [Cure_LC.load(POPULATION_BASE_PATH___CURE_LC + '/%s' % f)
            ↪  for f in files]
52          for i in individuals_cure_LC:
53              i.apply()
54
55          files = sorted(filter(lambda x: x.startswith('S_'),
            ↪  os.listdir(POPULATION_BASE_PATH___CURE_COMBINED + '')))
56          individuals_cure_combined =
            ↪  [Cure_combined.load(POPULATION_BASE_PATH___CURE_COMBINED + '/%s' % f) for
            ↪  f in
57                                      files]
58          for i in individuals_cure_combined:
59              i.apply()
60
61      if plot:
62          print_title("STEP 05: statistics", 'STEP 05')
63          reject_threshold = 1 - 2e-8
64          rejects = [ind for ind in individuals if ind['fitness_history'][-1][1] <=
            ↪  reject_threshold]
65          individuals = [ind for ind in individuals if ind['fitness_history'][-1][1] >
            ↪  reject_threshold]
66
67          populations = list()
68
69          lesions_list = ['lesion_SHAM', 'lesion_L6OHDA', 'lesion_LpCPA', 'lesion_LDSP4',
70                          'lesion_L6OHDA_LpCPA',
71                          'lesion_L6OHDA_LDSP4']
72          for lesion in lesions_list:
73              populations.append([i.__getattribute__(lesion)() for i in individuals])
74
75          populations = slice_populations(populations)
76          populations_pCPA = [populations[0], populations[1], populations[2],
            ↪  populations[4]]
77          ll_pCPA = ['lesion_SHAM', 'lesion_L6OHDA', 'lesion_LpCPA',
78                     'lesion_L6OHDA_LpCPA',
79                     ]
80          ll_DSP4 = ['lesion_SHAM', 'lesion_L6OHDA', 'lesion_LDSP4',
81                     'lesion_L6OHDA_LpCPA',
82                     ]
83          populations_DSP4 = [populations[0], populations[1], populations[3],
            ↪  populations[5]]
84
85          for pop, ll in [(populations, lesions_list), (populations_pCPA, ll_pCPA),
            ↪  (populations_DSP4, ll_DSP4)]:
86              for eq in pop[0][0]['equations']:
87                  eq_data = np.array(
88                          [np.array(
```

```
                                [m.simulate(m.target_as_y0(), t0,
                                ↪  T)['y'].transpose()[-1][m['equations'].index(eq)]
                                ↪  for
                                 m
                                 in
                                 p]) for p in pop])

                DFB = len(eq_data) - 1
                DFW = len(eq_data.flatten()) - len(eq_data)
                f, p = stats.f_oneway(*eq_data)
                text = ["Groups: %s" % ' '.join(str(i) + ':' + BL[g] for i, g in
                ↪  enumerate(ll))]
              text.append("ANOVA: F=%.3e, p=%.3e, dofB=%s, dofW=%s" % (f, p, DFB, DFW))
                text.append(tukey__str__(stats.tukey_hsd(*eq_data)))
                text = '\n'.join(text)
                with open(os.path.join(POPULATION_BASE_PATH,
                ↪  'ANOVA_lesions_%s___%s.txt' % (eq, '-'.join(ll))),
                          'w') as f:
                    f.write(text)
                print(eq + ':\n')
                print(text)

        populations = slice_populations([individuals, individuals_6OHDA,
                                         [i.cure_DRN() for i in individuals_cure_DRN],
                                         [i.cure_LC() for i in individuals_cure_LC],
                                         [i.cure_DRN_LC() for i in
                                         ↪  individuals_cure_combined],
                                         ])

        groups = ['SHAM', BL['lesion_L6OHDA'], '+cure_DRN',
                  '+cure_LC', '+cure_DRN+cure_LC']

        sim_data = [Parallel(n_jobs=-1)(delayed(m.simulate)(m.target_as_y0(), t0, T)
        ↪  for m in p) for p in
                    populations]

        eqs_index = populations[0][0]['equations'].index

        for eq in populations[0][0]['equations']:
            eq_data = np.array(
                    [np.array(
                            [m['y'].transpose()[-1][eqs_index(eq)] for m
                             in
                             p]) for p in sim_data])

            DFB = len(eq_data) - 1
            DFW = len(eq_data.flatten()) - len(eq_data)
            f, p = stats.f_oneway(*eq_data)
            text = ["Groups: %s" % ' '.join(str(i) + ':' + g for i, g in
            ↪  enumerate(groups))]
            text.append("ANOVA: F=%s, p=%s, dofB=%s, dofW=%s" % (f, p, DFB, DFW))
            text.append(tukey__str__(stats.tukey_hsd(*eq_data)))
            text = '\n'.join(text)
            with open(os.path.join(POPULATION_BASE_PATH, 'ANOVA_cure_%s.txt' % eq),
            ↪  'w') as f:
                f.write(text)
            print(eq + ':\n')
            print(text)


if __name__ == '__main__':
    main(fit=False, plot=True)
```

Bibliography

[1] E. R. Kandel, J. Koester, S. Mack, and S. Siegelbaum, eds., *Principles of neural science*. New York: McGraw Hill, sixth edition ed., 2021.

[2] Wikipedia contributors, "Dendrite — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Dendrite&oldid=1088360037, 2022.

[3] F. J. and K. M. Harris, "Dendrite structure." https://synapseweb.clm.utexas.edu/sites/default/files/synapseweb/files/1999_dendrites_fiala_harris_dendrite_structure.pdf, 1999.

[4] Wikipedia contributors, "Axon — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Axon&oldid=1106847638, 2022.

[5] D. Purves, G. Augustine, and F. D. et al., "Increased conduction velocity as a result of myelination.," 2001. https://www.ncbi.nlm.nih.gov/books/NBK10921/.

[6] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: from single neurons to networks and models of cognition*. Cambridge, United Kingdom: Cambridge University Press, 2014.

[7] E. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, pp. 1569–1572, Nov. 2003.

[8] R. Kurzweil. https://www.kurzweilai.net/ibm-scientists-emulate-neurons-with-phase-change-technology, 0.

[9] Wikipedia contributors, "Synapse — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Synapse&oldid=1100248063, 2022.

[10] S. Curti, F. Davoine, and A. Dapino, "Function and Plasticity of Electrical Synapses in the Mammalian Brain: Role of Non-Junctional Mechanisms," *Biology*, vol. 11, p. 81, Jan. 2022.

[11] Blue Brain Project, EPFL. https://www.epfl.ch/research/domains/bluebrain/, 0.

[12] H. Markram and E. e. a. Muller, "Reconstruction and Simulation of Neocortical Microcircuitry," *Cell*, vol. 163, pp. 456–492, Oct. 2015.

[13] Kenhub GmbH. https://www.kenhub.com/en/library/learning-strategies/parts-of-the-brain-learn-with-diagrams-and-quizzes, 0.

[14] Wikipedia contributors, "Cerebellum — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Cerebellum&oldid=1111178465, 2022.

[15] Wikipedia contributors, "Amygdala — Wikipedia, the free encyclopedia." https://en.wikipedia.org/w/index.php?title=Amygdala&oldid=1114522923, 2022.

[16] M. F. Bear, B. W. Connors, and M. A. Paradiso, *Neuroscience: exploring the brain*. Philadelphia: Wolters Kluwer, fourth edition ed., 2016.

[17] T. Scarabino, ed., *Atlas of morphology and functional anatomy of the brain*. New York, NY: Springer, 2006.

[18] J. C. Tamraz and Y. G. Comair, *Atlas of regional anatomy of the brain using MRI: with functional correlations*. Berlin Heidelberg: Springer, 2006.

[19] Dana Foundation. https://dana.org/article/neuroanatomy-the-basics/, 0.

[20] Wikipedia contributors, "Parkinson's disease — Wikipedia, the free encyclopedia," 2023. [Online; accessed 21-January-2023].

[21] M. Pourhamzeh, F. G. Moravej, M. Arabi, E. Shahriari, S. Mehrabi, R. Ward, R. Ahadi, and M. T. Joghataei, "The Roles of Serotonin in Neuropsychiatric Disorders," *Cellular and Molecular Neurobiology*, vol. 42, pp. 1671–1692, Aug. 2022.

[22] A. Woźniak-Kwaśniewska, "Optimisation of repetitive transcranial magnetic stimulation using electroencephalographic measurements in patients suffering from mood disorders," 10 2013.

[23] D. Meder, D. M. Herz, J. B. Rowe, S. Lehéricy, and H. R. Siebner, "The role of dopamine in the brain - lessons learned from parkinson's disease," vol. 190, pp. 79–93, 2019.

[24] S. E. Leh, M. Petrides, and A. P. Strafella, "The neural circuitry of executive functions in healthy subjects and parkinson's disease," vol. 35, no. 1, pp. 70–85, 2010.

[25] B. S. Connolly and A. E. Lang, "Pharmacological Treatment of Parkinson Disease: A Review," *JAMA*, vol. 311, p. 1670, Apr. 2014.

[26] D. Gagnon, S. Petryszyn, M. G. Sanchez, C. Bories, J. M. Beaulieu, Y. De Koninck, A. Parent, and M. Parent, "Striatal Neurons Expressing D1 and D2 Receptors are Morphologically Distinct and Differently Affected by Dopamine Denervation in Mice," *Scientific Reports*, vol. 7, p. 41432, Jan. 2017.

[27] A. Ledonne, M. Massaro Cenere, E. Paldino, V. D'Angelo, S. L. D'Addario, N. Casadei, A. Nobili, N. Berretta, F. R. Fusco, R. Ventura, G. Sancesario, E. Guatteo, and N. B. Mercuri, "Morpho-Functional Changes of Nigral Dopamine Neurons in an α-Synuclein Model of Parkinson's Disease," *Movement Disorders*, p. mds.29269, Nov. 2022.

[28] A. Tozzi, M. Sciaccaluga, V. Loffredo, A. Megaro, A. Ledonne, A. Cardinale, M. Federici, L. Bellingacci, S. Paciotti, E. Ferrari, A. La Rocca, A. Martini, N. B. Mercuri, F. Gardoni, B. Picconi, V. Ghiglieri, E. De Leonibus, and P. Calabresi, "Dopamine-dependent early synaptic and motor dysfunctions induced by α-synuclein in the nigrostriatal circuit," *Brain*, vol. 144, pp. 3477–3491, Dec. 2021.

[29] D. Caligiore, R. C. Helmich, M. Hallett, A. A. Moustafa, L. Timmermann, I. Toni, and G. Baldassarre, "Parkinson's disease as a system-level disorder," *npj Parkinson's Disease*, vol. 2, p. 16025, Dec. 2016.

[30] A. Dovzhenok and L. L. Rubchinsky, "On the Origin of Tremor in Parkinson's Disease," *PLoS ONE*, vol. 7, p. e41598, July 2012.

[31] D. Paré, R. Curro'Dossi, and M. Steriade, "Neuronal basis of the parkinsonian resting tremor: A hypothesis and its implications for treatment," *Neuroscience*, vol. 35, pp. 217–226, Jan. 1990.

[32] D. Caligiore, M. A. Arbib, R. C. Miall, and G. Baldassarre, "The super-learning hypothesis: Integrating learning processes across cortex, cerebellum and basal ganglia," *Neuroscience & Biobehavioral Reviews*, vol. 100, pp. 19–34, May 2019.

[33] J. A. Obeso, M. C. Rodriguez-Oroz, C. G. Goetz, C. Marin, J. H. Kordower, M. Rodriguez, E. C. Hirsch, M. Farrer, A. H. V. Schapira, and G. Halliday, "Missing pieces in the Parkinson's disease puzzle," *Nature Medicine*, vol. 16, pp. 653–661, June 2010.

[34] J. Jankovic and A. S. Kapadia, "Functional Decline in Parkinson Disease," *Archives of Neurology*, vol. 58, p. 1611, Oct. 2001.

[35] D. Aarsland, K. Bronnick, J. P. Larsen, O. B. Tysnes, G. Alves, and For the Norwegian ParkWest Study Group, "Cognitive impairment in incident, untreated Parkinson disease: The Norwegian ParkWest Study," *Neurology*, vol. 72, pp. 1121–1126, Mar. 2009.

[36] C. H. Williams-Gray, T. Foltynie, C. E. G. Brayne, T. W. Robbins, and R. A. Barker, "Evolution of cognitive dysfunction in an incident Parkinson's disease cohort," *Brain*, vol. 130, pp. 1787–1798, July 2007.

[37] F. Faivre, A. Joshi, E. Bezard, and M. Barrot, "The hidden side of Parkinson's disease: Studying pain, anxiety and depression in animal models," *Neuroscience & Biobehavioral Reviews*, vol. 96, pp. 335–352, Jan. 2019.

[38] L. Miquel-Rio, D. Alarcón-Arís, M. Torres-López, V. Cóppola-Segovia, R. Pavia-Collado, V. Paz, E. Ruiz-Bronchal, L. Campa, C. Casal, A. Montefeltro, M. Vila, F. Artigas, R. Revilla, and A. Bortolozzi, "Human α-synuclein overexpression in mouse serotonin neurons triggers a depressive-like phenotype. Rescue by oligonucleotide therapy," *Translational Psychiatry*, vol. 12, p. 79, Feb. 2022.

[39] E. Cohen, A. A. Bay, L. Ni, and M. E. Hackney, "Apathy-Related Symptoms Appear Early in Parkinson's Disease," *Healthcare*, vol. 10, p. 91, Jan. 2022.

[40] M. Favier, C. Carcenac, M. Savasta, and S. Carnicella, "Dopamine D3 Receptors: A Potential Target to Treat Motivational Deficits in Parkinson's Disease," in *Therapeutic Applications of Dopamine D3 Receptor Function* (I. Boileau and G. Collo, eds.), vol. 60, pp. 109–132, Cham:

Springer International Publishing, 2022. Series Title: Current Topics in Behavioral Neurosciences.

[41] J. Pagonabarraga, J. Kulisevsky, A. P. Strafella, and P. Krack, "Apathy in Parkinson's disease: clinical features, neural substrates, diagnosis, and treatment," *The Lancet Neurology*, vol. 14, pp. 518–531, May 2015.

[42] R. C. Helmich, M. Hallett, G. Deuschl, I. Toni, and B. R. Bloem, "Cerebral causes and consequences of parkinsonian resting tremor: a tale of two circuits?," *Brain*, vol. 135, pp. 3206–3226, Nov. 2012.

[43] T. Wu and M. Hallett, "The cerebellum in Parkinson's disease," *Brain*, vol. 136, pp. 696–709, Mar. 2013.

[44] D. Caligiore, G. Pezzulo, G. Baldassarre, A. C. Bostan, P. L. Strick, K. Doya, R. C. Helmich, M. Dirkx, J. Houk, H. Jörntell, A. Lago-Rodriguez, J. M. Galea, R. C. Miall, T. Popa, A. Kishore, P. F. M. J. Verschure, R. Zucca, and I. Herreros, "Consensus Paper: Towards a Systems-Level View of Cerebellar Function: the Interplay Between Cerebellum, Basal Ganglia, and Cortex," *The Cerebellum*, vol. 16, pp. 203–229, Feb. 2017.

[45] D. Caligiore, F. Giocondo, and M. Silvetti, "The Neurodegenerative Elderly Syndrome (NES) hypothesis: Alzheimer and Parkinson are two faces of the same disease," *IBRO Neuroscience Reports*, vol. 13, pp. 330–343, Dec. 2022.

[46] R. C. Helmich, "The cerebral basis of Parkinsonian tremor: A network perspective," *Movement Disorders*, vol. 33, pp. 219–231, Feb. 2018.

[47] H. Zach, M. F. Dirkx, D. Roth, J. W. Pasman, B. R. Bloem, and R. C. Helmich, "Dopamine-responsive and dopamine-resistant resting tremor in Parkinson disease," *Neurology*, vol. 95, pp. e1461–e1470, Sept. 2020.

[48] K. A. Jellinger, "Pathology of Parkinson's disease: Changes other than the nigrostriatal pathway," *Molecular and Chemical Neuropathology*, vol. 14, pp. 153–197, June 1991.

[49] S. Perez-Lloret and F. J. Barrantes, "Deficits in cholinergic neurotransmission and their clinical correlates in Parkinson's disease," *npj Parkinson's Disease*, vol. 2, p. 16001, Feb. 2016.

[50] H. Wilson, G. Dervenoulas, G. Pagano, C. Koros, T. Yousaf, M. Picillo, S. Polychronis, A. Simitsi, B. Giordano, Z. Chappell, B. Corcoran, M. Stamelou, R. N. Gunn, M. T. Pellecchia, E. A. Rabiner, P. Barone,

L. Stefanis, and M. Politis, "Serotonergic pathology and disease burden in the premotor and motor phase of A53T α-synuclein parkinsonism: a cross-sectional study," *The Lancet Neurology*, vol. 18, pp. 748–759, Aug. 2019.

[51] F. H. Hezemans, N. Wolpe, C. O'Callaghan, R. Ye, C. Rua, P. S. Jones, A. G. Murley, N. Holland, R. Regenthal, K. A. Tsvetanov, R. A. Barker, C. H. Williams-Gray, T. W. Robbins, L. Passamonti, and J. B. Rowe, "Noradrenergic deficits contribute to apathy in Parkinson's disease through the precision of expected outcomes," *PLOS Computational Biology*, vol. 18, p. e1010079, May 2022.

[52] C. Delaville, S. Navailles, and A. Benazzouz, "Effects of noradrenaline and serotonin depletions on the neuronal activity of globus pallidus and substantia nigra pars reticulata in experimental parkinsonism," *Neuroscience*, vol. 202, pp. 424–433, Jan. 2012.

[53] C. Delaville, J. Chetrit, K. Abdallah, S. Morin, L. Cardoit, P. De Deurwaerdère, and A. Benazzouz, "Emerging dysfunctions consequent to combined monoaminergic depletions in parkinsonism," *Neurobiology of Disease*, vol. 45, pp. 763–773, Feb. 2012.

[54] H. Brunnström, N. Friberg, E. Lindberg, and E. Englund, "Differential degeneration of the locus coeruleus in dementia subtypes," *Clinical Neuropathology*, vol. 30, pp. 104–110, May 2011.

[55] D. C. German, K. F. Manaye, P. K. Sonsalla, and B. A. Brooks, "Midbrain Dopaminergic Cell Loss in Parkinson's Disease and MPTP-Induced Parkinsonism: Sparing of Calbindin-D $_{25k}$?Containing Cells," *Annals of the New York Academy of Sciences*, vol. 648, pp. 42–62, May 1992.

[56] J. Jankovic, "Parkinson's disease tremors and serotonin," *Brain*, vol. 141, pp. 624–626, Mar. 2018.

[57] J. Pasquini, R. Ceravolo, Z. Qamhawi, J.-Y. Lee, G. Deuschl, D. J. Brooks, U. Bonuccelli, and N. Pavese, "Progression of tremor in early stages of Parkinson's disease: a clinical and neuroimaging study," *Brain*, vol. 141, pp. 811–821, Mar. 2018.

[58] A. Muñoz, A. Lopez-Lopez, C. M. Labandeira, and J. L. Labandeira-Garcia, "Interactions Between the Serotonergic and Other Neurotransmitter Systems in the Basal Ganglia: Role in Parkinson's Disease and Adverse Effects of L-DOPA," *Frontiers in Neuroanatomy*, vol. 14, p. 26, June 2020.

[59] M. Politis and F. Niccolini, "Serotonin in Parkinson's disease," *Behavioural Brain Research*, vol. 277, pp. 136–145, Jan. 2015.

[60] S. Prange, H. Klinger, C. Laurencin, T. Danaila, and S. Thobois, "Depression in Patients with Parkinson's Disease: Current Understanding of its Neurobiology and Implications for Treatment," *Drugs & Aging*, vol. 39, pp. 417–439, June 2022.

[61] C. Fornari, C. Pin, J. W. Yates, J. T. Mettetal, and T. A. Collins, "Importance of Stability Analysis When Using Nonlinear Semimechanistic Models to Describe Drug-Induced Hematotoxicity," *CPT: Pharmacometrics & Systems Pharmacology*, vol. 9, pp. 498–508, Sept. 2020.

[62] Z. Shi, W. Yao, Z. Li, L. Zeng, Y. Zhao, R. Zhang, Y. Tang, and J. Wen, "Artificial intelligence techniques for stability analysis and control in smart grids: Methodologies, applications, challenges and future directions," *Applied Energy*, vol. 278, p. 115733, Nov. 2020.

[63] J. Liu, G. P. Shelkar, L. P. Sarode, D. Y. Gawande, F. Zhao, R. P. Clausen, R. R. Ugale, and S. M. Dravid, "Facilitation of GluN2C-containing NMDA receptors in the external globus pallidus increases firing of fast spiking neurons and improves motor function in a hemiparkinsonian mouse model," *Neurobiology of Disease*, vol. 150, p. 105254, Mar. 2021.

[64] H. Kita and T. Kita, "Role of Striatum in the Pause and Burst Generation in the Globus Pallidus of 6-OHDA-Treated Rats," *Frontiers in Systems Neuroscience*, vol. 5, 2011.

[65] S. Damodaran, R. C. Evans, and K. T. Blackwell, "Synchronized firing of fast-spiking interneurons is critical to maintain balanced firing between direct and indirect pathway neurons of the striatum," *Journal of Neurophysiology*, vol. 111, pp. 836–848, Feb. 2014.

[66] C. Cepeda, V. M. André, I. Yamazaki, N. Wu, M. Kleiman-Weiner, and M. S. Levine, "Differential electrophysiological properties of dopamine D1 and D2 receptor-containing striatal medium-sized spiny neurons," *European Journal of Neuroscience*, vol. 27, pp. 671–682, Feb. 2008.

[67] R. S. Feldman, J. S. Meyer, and L. F. Quenzer, *Principles of neuropsychopharmacology*. Sunderland, Mass: Sinauer Associates, 1997.

[68] S. T. Szabo and P. Blier, "Functional and pharmacological characterization of the modulatory role of serotonin on the firing activity of locus coeruleus norepinephrine neurons," *Brain Research*, vol. 922, pp. 9–20, Dec. 2001.

[69] R.-J. Liu, A. N. van den Pol, and G. K. Aghajanian, "Hypocretins (Orexins) Regulate Serotonin Neurons in the Dorsal Raphe Nucleus by Excitatory Direct and Inhibitory Indirect Actions," *The Journal of Neuroscience*, vol. 22, pp. 9453–9464, Nov. 2002.

[70] Y. Kang and S. Kitai, "A whole cell patch-clamp study on the pacemaker potential in dopaminergic neurons of rat substantia nigra compacta," *Neuroscience Research*, vol. 18, pp. 209–221, Dec. 1993.

[71] X. Zhang, N. Cui, Z. Wu, J. Su, J. S. Tadepalli, S. Sekizar, and C. Jiang, "Intrinsic membrane properties of locus coeruleus neurons in *Mecp2* -null mice," *American Journal of Physiology-Cell Physiology*, vol. 298, pp. C635–C646, Mar. 2010.

[72] C. A. Deister, C. S. Chan, D. J. Surmeier, and C. J. Wilson, "Calcium-Activated SK Channels Influence Voltage-Gated Ion Channels to Determine the Precision of Firing in Globus Pallidus Neurons," *Journal of Neuroscience*, vol. 29, pp. 8452–8461, July 2009.

[73] L. Brugnano, *Modelli numerici per la simulazione*. 2018.

[74] J. C. Butcher, *Numerical methods for ordinary differential equations*. Chichester, England ; Hoboken, NJ: Wiley, 2nd ed ed., 2008. OCLC: ocn191024153.

[75] L. Brugnano and C. Magherini, "The BiM code for the numerical solution of ODEs," *Journal of Computational and Applied Mathematics*, vol. 164-165, pp. 145–158, Mar. 2004.

[76] L. F. Shampine and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing*, vol. 18, pp. 1–22, Jan. 1997.

[77] The SciPy community, "scipy.optimize.solve_ivp." `https://docs.scipy.org/doc/scipy-1.8.1/reference/generated/scipy.integrate.solve_ivp.html`, 0.

[78] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential evolution: a practical approach to global optimization*. Natural computing series, Berlin ; New York: Springer, 2005.

[79] The SciPy community, "scipy.optimize.differential_evolution." `https://docs.scipy.org/doc/scipy-1.8.1/reference/generated/scipy.optimize.differential_evolution.html`, 0.

[80] G. Jeyakumar and C. Shanmugavelayutham, "Convergence Analysis of Differential Evolution Variants on Unconstrained Global Optimization Functions," *International Journal of Artificial Intelligence & Applications*, vol. 2, pp. 116–127, Apr. 2011.

[81] H. Sahai and M. I. Ageel, *The Analysis of Variance: Fixed, Random and Mixed Models*. Boston, MA: Birkhäuser Boston : Imprint : Birkhäuser, 2000. OCLC: 853266620.

[82] D. J. Weiss, *Analysis of variance and functional measurement: a practical guide*. New York: Oxford University Press, 2006. OCLC: 65190673.

[83] The SciPy community, "scipy.stats.f_oneway." `https://docs.scipy.org/doc/scipy-1.8.1/reference/generated/scipy.stats.f_oneway.html`, 0.

[84] The SciPy community, "scipy.stats.tukey_hsd." `https://docs.scipy.org/doc/scipy-1.8.1/reference/generated/scipy.stats.tukey_hsd.html`, 0.

[85] S. E. Maxwell, H. D. Delaney, and K. Kelley, *Designing experiments and analyzing data: a model comparison perspective*. New York, NY: Routledge, third edition ed., 2017.

[86] E. Howell, "Anova test simply explained." `https://towardsdatascience.com/anova-test-simply-explained-c94e4620ec6f`, 0.

[87] A. Bevan, *Statistical Data Analysis for the Physical Sciences*. Cambridge, UK: Cambridge University Press, 2013. OCLC: ocn841794250.

[88] G. Cowan, *Statistical data analysis*. Oxford science publications, Oxford : New York: Clarendon Press ; Oxford University Press, 1998.

[89] D. S. Sivia and J. Skilling, *Data analysis: a Bayesian tutorial; [for scientists and engineers]*. Oxford science publications, Oxford: Oxford Univ. Press, 2. ed., repr ed., 2012.

[90] R. Balestrino and A. Schapira, "Parkinson disease," *European Journal of Neurology*, vol. 27, pp. 27–42, Jan. 2020.

[91] B. P. Guiard, M. El Mansari, Z. Merali, and P. Blier, "Functional interactions between dopamine, serotonin and norepinephrine neurons: an in-vivo electrophysiological study in rats with monoaminergic lesions," *International Journal of Neuropsychopharmacology*, vol. 11, pp. 625–639, Aug. 2008.

[92] C. Miguelez, L. Grandoso, and L. Ugedo, "Locus coeruleus and dorsal raphe neuron activity and response to acute antidepressant administration in a rat model of Parkinson's disease," *The International Journal of Neuropsychopharmacology*, vol. 14, pp. 187–200, Mar. 2011.

[93] N. Berretta, P. S. Freestone, E. Guatteo, D. d. Castro, R. Geracitano, G. Bernardi, N. B. Mercuri, and J. Lipski, "Acute Effects of 6-Hydroxydopamine on Dopaminergic Neurons of the Rat Substantia Nigra Pars Compacta In Vitro," *NeuroToxicology*, vol. 26, pp. 869–881, Oct. 2005.

[94] A. H. Kaya, R. Vlamings, S. Tan, L. W. Lim, P. J. Magill, H. W. Steinbusch, V. Visser-Vandewalle, T. Sharp, and Y. Temel, "Increased electrical and metabolic activity in the dorsal raphe nucleus of Parkinsonian rats," *Brain Research*, vol. 1221, pp. 93–97, July 2008.

[95] X. Zhang, P. E. Andren, and P. Svenningsson, "Changes on 5-HT2 receptor mRNAs in striatum and subthalamic nucleus in Parkinson's disease model," *Physiology & Behavior*, vol. 92, pp. 29–33, Sept. 2007.

[96] R. L. Albin, A. B. Young, and J. B. Penney, "The functional anatomy of basal ganglia disorders," *Trends in Neurosciences*, vol. 12, pp. 366–375, Jan. 1989.

[97] N. Mallet, "Cortical Inputs and GABA Interneurons Imbalance Projection Neurons in the Striatum of Parkinsonian Rats," *Journal of Neuroscience*, vol. 26, pp. 3875–3884, Apr. 2006.

[98] N. Maurice, M. Liberge, F. Jaouen, S. Ztaou, M. Hanini, J. Camon, K. Deisseroth, M. Amalric, L. Kerkerian-Le Goff, and C. Beurrier, "Striatal Cholinergic Interneurons Control Motor Behavior and Basal Ganglia Function in Experimental Parkinsonism," *Cell Reports*, vol. 13, pp. 657–666, Oct. 2015.

[99] J. G. Parker, J. D. Marshall, B. Ahanonu, Y.-W. Wu, T. H. Kim, B. F. Grewe, Y. Zhang, J. Z. Li, J. B. Ding, M. D. Ehlers, and M. J. Schnitzer, "Diametric neural ensemble dynamics in parkinsonian and dyskinetic states," *Nature*, vol. 557, pp. 177–182, May 2018.

[100] D. et al, "Effect of Physical Exercise and Acute Escitalopram on the Excitability of Brain Monoamine Neurons: In Vivo Electrophysiological Study in Rats," *International Journal of Neuropsychopharmacology*, vol. 20, pp. 585–592, July 2017.

[101] B. P. Guiard, M. El Mansari, and P. Blier, "Cross-Talk between Dopaminergic and Noradrenergic Systems in the Rat Ventral Tegmental Area, Locus Ceruleus, and Dorsal Hippocampus," *Molecular Pharmacology*, vol. 74, pp. 1463–1475, Nov. 2008.

[102] A. Ferron, "Modified coeruleo-cortical noradrenergic neurotransmission after serotonin depletion by PCPA: Electrophysiological studies in the rat," *Synapse*, vol. 2, no. 5, pp. 532–536, 1988.

[103] N. Haddjeri, C. De Montigny, and P. Blier, "Modulation of the firing activity of noradrenergic neurones in the rat locus coeruleus by the 5-hydroxtryptamine system: 5-HT and locus coeruleus firing," *British Journal of Pharmacology*, vol. 120, pp. 865–875, Feb. 1997.

[104] A. E. Lang and J. A. Obeso, "Challenges in Parkinson's disease: restoration of the nigrostriatal dopamine system is not enough," *The Lancet Neurology*, vol. 3, pp. 309–316, May 2004.

[105] S. af Bjerkén, R. Stenmark Persson, A. Barkander, N. Karalija, N. Pelegrina-Hidalgo, G. A. Gerhardt, A. Virel, and I. Strömberg, "Noradrenaline is crucial for the substantia nigra dopaminergic cell maintenance," *Neurochemistry International*, vol. 131, p. 104551, Dec. 2019.

[106] L. M. Butkovich, M. C. Houser, T. Chalermpalanupap, K. A. Porter-Stransky, A. F. Iannitelli, J. S. Boles, G. M. Lloyd, A. S. Coomes, L. N. Eidson, M. E. De Sousa Rodrigues, D. L. Oliver, S. D. Kelly, J. Chang, N. Bengoa-Vergniory, R. Wade-Martins, B. I. Giasson, V. Joers, D. Weinshenker, and M. G. Tansey, "Transgenic Mice Expressing Human α-Synuclein in Noradrenergic Neurons Develop Locus Ceruleus Pathology and Nonmotor Features of Parkinson's Disease," *The Journal of Neuroscience*, vol. 40, pp. 7559–7576, Sept. 2020.

[107] K. Cui, F. Yang, T. Tufan, M. U. Raza, Y. Zhan, Y. Fan, F. Zeng, R. W. Brown, J. B. Price, T. C. Jones, G. W. Miller, and M.-Y. Zhu, "Restoration of Noradrenergic Function in Parkinson's Disease Model Mice," *ASN Neuro*, vol. 13, p. 175909142110097, Jan. 2021.

[108] K. Del Tredici, U. Rüb, R. A. de Vos, J. R. Bohl, and H. Braak, "Where Does Parkinson Disease Pathology Begin in the Brain?," *Journal of Neuropathology & Experimental Neurology*, vol. 61, pp. 413–426, May 2002.

[109] K. Rommelfanger and D. Weinshenker, "Norepinephrine: The redheaded stepchild of Parkinson's disease," *Biochemical Pharmacology*, vol. 74, pp. 177–190, July 2007.

[110] K. S. Rommelfanger, G. L. Edwards, K. G. Freeman, L. C. Liles, G. W. Miller, and D. Weinshenker, "Norepinephrine loss produces more profound motor deficits than MPTP treatment in mice," *Proceedings of the National Academy of Sciences*, vol. 104, pp. 13804–13809, Aug. 2007.

[111] Y. Vermeiren and P. P. De Deyn, "Targeting the norepinephrinergic system in Parkinson's disease and related disorders: The locus coeruleus story," *Neurochemistry International*, vol. 102, pp. 22–32, Jan. 2017.

[112] Wikipedia contributors, "Probability density function — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Probability_density_function&oldid=1116692234`, 2022.

[113] W. Syam, "Demystifying p-value in analysis of variance (anova)." `https://www.wasyresearch.com/demystifying-p-value-in-analysis-of-variance-anova/`, 0.

[114] J. Humble and D. Farley, *Continuous Delivery Reliable Software Releases Through Build, Test, and Deployment Automation*. Hoboken: Pearson Education, Limited, 2010. OCLC: 1348485094.

[115] R. C. Martin and M. C. Feathers, *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, N.J.: Prentice Hall, 2009. OCLC: 297575371.

[116] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley signature series, Boston: Addison-Wesley, second edition ed., 2019. OCLC: on1064139838.

[117] D. Trigiante, *Dispense per il corso di metodi di approssimazione*. 2007.

[118] W. K. Nicholson, "Linear algebra with applications," 2019.

[119] V. Lakshmikantham and D. Trigiante, *Theory Of Difference Equations, Numerical Methods And Applications*. CRC Press, 0 ed., June 2002.

[120] R. E. Mickens, *Difference equations: theory and applications*. New York: Van Nostrand Reinhold, 2nd ed ed., 1990.

[121] C. D. Meyer, *Matrix analysis and applied linear algebra*. Philadelphia: Society for Industrial and Applied Mathematics, 2000.

[122] R. A. Horn and C. R. Johnson, *Matrix analysis*. New York, NY: Cambridge University Press, second edition, corrected reprint ed., 2017.

[123] D. Poole, *Linear algebra: a modern introduction*. Stamford, CT: Cengage Learning, fourth edition ed., 2015.

[124] G. Strang, *Linear algebra and its applications*. Belmont, CA: Thomson, Brooks/Cole, 4th ed ed., 2006.

[125] N. J. Higham, *Functions of matrices: theory and computation*. Philadelphia: Society for Industrial and Applied Mathematics, 2008. OCLC: ocn185095770.

[126] S. Elaydi, *An introduction to difference equations*. Undergraduate texts in mathematics, New York: Springer, 1996.

[127] A. Spitzbart, "A Generalization of Hermite's Interpolation Formula," *The American Mathematical Monthly*, vol. 67, pp. 42–46, Jan. 1960.

[128] I. Gohberg, P. Lancaster, and L. Rodman, *Matrix polynomials*. No. 58 in Classics in applied mathematics, Philadelphia: Society for Industrial and Applied Mathematics, siam ed., [classics ed.] ed., 2009. OCLC: ocn311310303.

[129] C. de Boor, "Divided Differences," Feb. 2005. arXiv:math/0502036.

[130] A. N. Burkitt, "A Review of the Integrate-and-fire Neuron Model: I. Homogeneous Synaptic Input," *Biological Cybernetics*, vol. 95, pp. 1–19, July 2006.

[131] R. Al-Baradie, "Principles of Neuropsychopharmacology," *Epilepsy Research*, vol. 51, p. 205, Sept. 2002.

[132] S. M. Florin, R. Kuczenski, and D. S. Segal, "Regional extracellular norepinephrine responses to amphetamine and cocaine and effects of clonidine pretreatment," *Brain Research*, vol. 654, pp. 53–62, Aug. 1994.

[133] S. R. Knobloch Roman, Mlýnek Jaroslav, "The classic differential evolution algorithm and its convergence properties," *Applications of Mathematics*, vol. 62, no. 2, pp. 197–208, 2017.

[134] T. P. Naidich and H. M. Duvernoy, eds., *Duvernoy's atlas of the human brain stem and cerebellum: high-field MRI: surface anatomy, internal structure, vascularization and 3D sectional anatomy*. Wien ; New York: Springer, 2009. OCLC: ocn304072758.

[135] R. J. Barnes, "Matrix differentiation (and some other stuff)," 2006.

[136] M. Stacy, "Medical Treatment of Parkinson Disease," *Neurologic Clinics*, vol. 27, pp. 605–631, Aug. 2009.

[137] P. Jovanovic, Y. Wang, J.-P. Vit, E. Novinbakht, N. Morones, E. Hogg, M. Tagliati, and C. E. Riera, "Sustained chemogenetic activation of locus coeruleus norepinephrine neurons promotes dopaminergic neuron survival in synucleinopathy," *PLOS ONE*, vol. 17, p. e0263074, Mar. 2022.