UNIVERSITÀ
DI PAVIA

PhD IN BIOMEDICAL SCIENCES

DEPARTMENT OF BRAIN AND BEHAVIORAL SCIENCES

UNIT OF NEUROPHYSIOLOGY

Building the cerebellum (and the Brain Scaffold Builder)

PhD Tutor: Claudia Casellato

PhD dissertation of
Robin De Schepper

a.y. 2020/2021

Building the cerebellum (and the Brain Scaffold Builder)

Robin De Schepper

November 29, 2023

Contents

Acknowledgements

It is nice to have a little personal space to write out my thanks to everyone that contributed to my years as a PhD student. From all the scientists I've met that inspired hallucination-like mind adventures, also known as "new thoughts". To all the friends made along the way. The most important people have been my loving partners Elide, Maria, and Katha. I love you all. You give me the warmth, love, space, and security that you will always be there for me, allowing me to discover and grow into my unconventional self, loved whole. Without them this PhD project would have ended on a runaway panic-fueled existential crisis rather quickly.

Professionally, it has been the hard work of Claudia, keeping me somewhat sane in a system I couldn't find myself in, and strategically guiding me away from my self-destructive confrontational habits[1]. Thanks! Although our relationship is work-based, I consider you both a good colleague and a good friend.

Finally, special thanks goes out to all the people that gave me a place to feel home: Lissa and Mattia, for being the best friends. Pier, for giving me family in Pavia. And all of my queer and kinky friends in Milan, for giving me a safe space to love myself.

[1] In my opinion, academia as an institute is flawed, and much work is published not for the pursuit of quality science, but for the proliferation of an academic's career and their survival in a capitalist society. Most published research findings are likely false [1]. Furthermore it is my opinion that computational neuroscience suffers from a reproducibility crisis where code is not reviewed or reviewable, making peer review of these works nearly pointless, and with the perceived worthlessness of replication studies, the publication of false statements is much likelier than anyone disproving them. I am displeased but will keep further reflections of these opinions limited to blog posts and bar rants, as I am not in a position to provide scientific proof for the statements, and they do not belong in this doctoral thesis.

Chapter 1

Introduction

1.1 Background and Motivation

In this thesis we present the development of the Brain Scaffold Builder (BSB), a black-box component framework for multiscale bottom-up brain modeling, and apply it to build a reusable model of the cerebellum. The development of this scientific software was guided by software design and engineering principles, and the needs of other researchers in our field. On top of that, we'd like to present you the first scientific findings and projects that have already been carried out while the framework was still in development, by the growing community of early-adopters of the framework. We want to critically discuss what can be improved, and what others may learn from our efforts.

Launched in 2013, the Human Brain Project (HBP) aspires to a comprehensive understanding of the human brain. Aiming to construct multilevel brain models from all of the available life science's multiomics datasets, the HBP envisions integrating data and knowledge about brain structure and function to create models validated through supercomputer simulations. They set out to drive supercomputing developments for the life sciences and provide the community with new tools for informatics, modeling, and simulation of the brain [2].

Embedded in this broader initiative, Egidio D'Angelo's research group contributes significantly by gathering data and constructing models of the cerebellum, progressively advancing towards the HBP's overarching goal. The group has a long standing history of creating, improving, and validating single-cell models of the cells in the cerebellum [3–7], and has in the last 2 decades scaled these efforts up to some of the first large-scale network level simulations of the

cerebellum [8–12]. This dissertation aligns with this ongoing work, focusing on the latest developments in network level cerebellar modeling investigated by the research group.

EBRAINS, created as a part of the HBP project, gathers an extensive range of tools and imaging data that scientists around the world can then use to run simulations and digital experiments. It melds the data from individual neurons with information on brain anatomy, connectivity, and function to help translate the latest scientific discoveries into innovation for the benefits of patients and society. Its aim is to dramatically increase the efficiency and productivity of European research, by making findable, reusable data and state-of-the art digital research tools available to the scientific community within Europe, and offering them openly for use and expansion across the planet. In fact, EBRAINS is a unique infrastructure worldwide, in that it provides access to the most comprehensive set of brain data and software tools yet made available [13–20]. The BSB was selected during its development as an EBRAINS software component and is available to the neuroscience community in EBRAINS' Network simulation category [21].

1.2 The Cerebellum

1.2.1 Neuroanatomy

The cerebellum is the infratentorial part of the brain, separated from the supratentorial cerebrum by the tentorium cerebelli (Fig. 1.1), an invagination of the dura mater, the outermost layer of connective tissue around the brain (with more internally the arachnoid and pia), and is situated behind the brain stem and fourth ventricle, on top of the posterior cranial fossa, the most dorsal region of the cranial bone which forms the floor of the cranial cavity [22].

The cerebellum consists of 2 hemispheres joined on the midline by the vermis.There are 3 lobes separated by 2 transversal fissures, each lobe consists of a central part in the vermis and the adjacent parts in the hemispheres, further divided into 10 lobules, consisting of many transversal folds called folia. The many folia arise from the self-similar folding pattern of the cerebellar cortex to maximize cortical surface area while minimizing tract length and cerebellar volume, leading to a tree-like white matter structure with each branch surrounded by a fold of cortical gray matter [22]. The transversal folds give it its distinct outward appearance, constrasted by the tortuous convolutions of the neocortex, and arise from the organization of the cerebellar cortex in long parallel bundles of fibre with microcircuit connections organized orthogonally to these parallel fiber bundles, giving it a constrained transversal symmetry. This rigorous folding gives the cerebellar cortex 80% the surface area of the cerebrum, while only being 10% of its volume [24, 25].

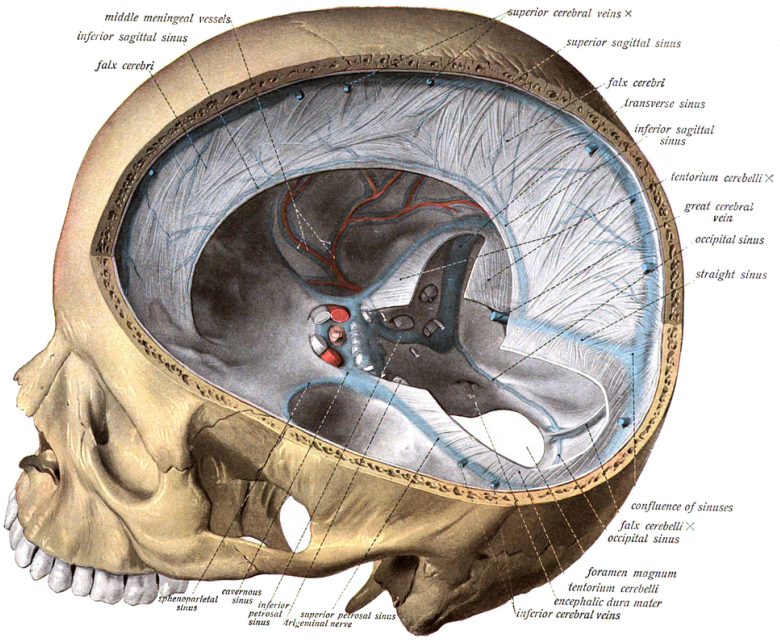


Figure 1.1: Sobottas Anatomy Handbook [23] plate 589, anatomy of the connective tissue in the skull, vasculature, and sinuses.

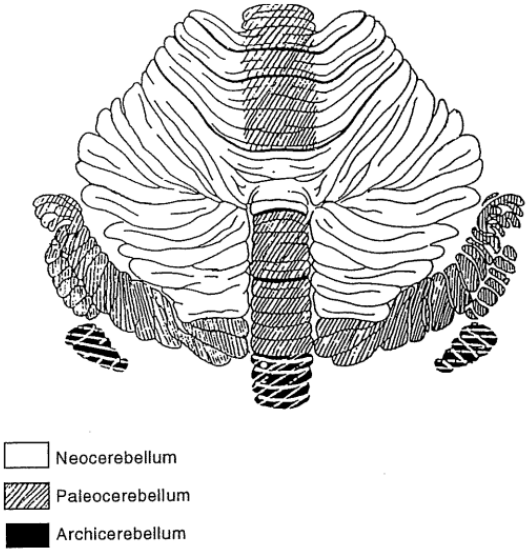


Figure 1.2: Schematic drawing of the monkey cerebellum [26] with the major evolutionary subdivisions.

The cerebellar cortex can be divided into 3 evolutionary regions, ordered ancient to modern: the archicerebellum, paleocerebellum, and neocerebellum (see Fig. 1.2). The ancient regions serve the classical role of motor control and coordination, while the neocerebellum has seen massive expansion in primates and is associated to control of higher associative functions [26, 27].

Nestled at the roots of the white matter tree of the neocerebellum are the 3 pairs of deep cerebellar nuclei (DCN). The most medial nuclei are the fastigial nuclei, more laterally the globose nuclei, and most laterally the dentate nuclei. The dentate nuclei are the largest, so named for their resemblance to a row of teeth. The DCN contain the only efferent fibers of the cerebellum, inhibited by the Purkinje cell axon, the only efferent fiber of the cerebellar cortex [22].

The cerebellum is connected via 3 pairs (superior, middle, and inferior) of cerebellar peduncles in the pons to the cerebrum, medulla oblongata, and spinal cord. The main cerebral pathway is the cerebello-thamalo-cortical pathway connecting the cerebellum to the cerebrum via the thalamus [22].

1.2.2 Function

The cerebellum plays a crucial role in controlling movement, maintaining balance, and facilitating locomotion. Consequently, walking ataxia is seen as a

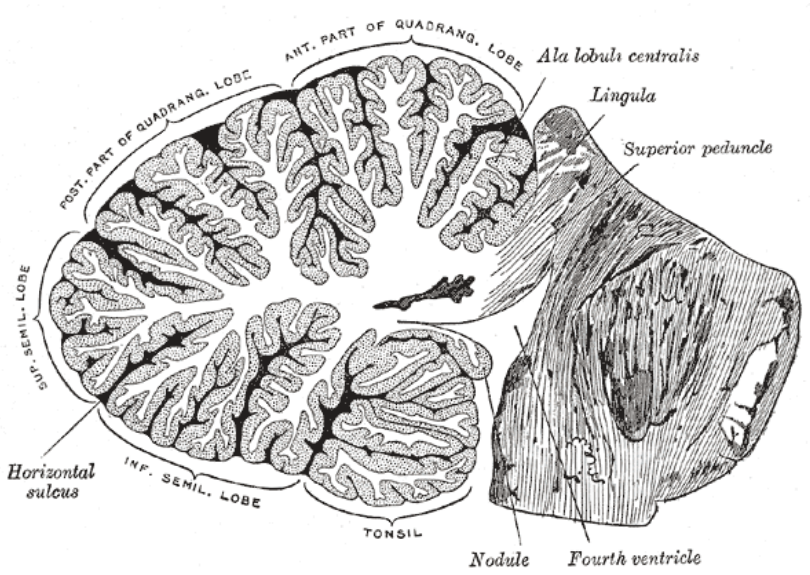


Figure 1.3: Cross-section of the cerebellum, showing the white matter and gray matter distribution, and the pons. Gray's Anatomy plate 704 [28]

distinctive maker of cerebellar damage. During tasks like reaching for a moving target, humans need to predict both motion of the target and movement of the limb to achieve optimal synchronization. The cerebellum's fundamental function involves making predictions and validating them against real sensory inputs, acting as a sophisticated conductor within the complex symphony of movement. Its significance extends beyond mere execution: indeed the cerebellum plays a pivotal role in refining and fine-tuning motor functions to achieve precision, coordination, and balance. Through this continuous calibration process, the cerebellum contributes to the seamless execution of everyday actions, from the intricacies of finger dexterity to the fluidity of walking.

At the core of its functions lies the cerebellum's capacity to compare intended movements with actual execution. This real-time feedback loop enables the cerebellum to detect discrepancies and swiftly adjust motor commands, ensuring that movements align with the intended goals. Recent computational evidence suggests that the cerebellum predicts present and future state estimates of the body and of its environment by integrating an estimate of previous states and efference copies of motor signals. This suggests that the cerebellum employs an internal forward model [29]. Given that sensory feedback signals from sensory organs experience unavoidable delays of tens to hundreds of milliseconds before reaching the central nervous system,the forward model's computation aids in predictive control, especially when dealing with significant sensory feedback delays [30]. This predictive ability enables the brain to adapt movements in

real-time, seamlessly accommodating changes in the environment and ensuring stability even when confronted with unforeseen obstacles.

The forward model represents a computational framework for voluntary motor control that emphasizes the critical role of comparing our intended actions with their actual outcomes. This model assesses the input-output function of bodily segments involved in movements, relying on two core functions: prediction and error processing. In the context of optimal feedback control (OFC) models, both predictive control (internal feedback) and error control (sensory feedback) hold central importance [31]. According to this theory, an efference copy, which is a representation of motor intention, is generated during the preparation of the voluntary movements to predict the sensory outcome of the impending action. This efference copy is dispatched to brain regions responsible for assessing the congruence between the copy and the actual sensory feedback elicted by the movement. Importantly, the gain in sensory feedback is not fixed but rather adaptable. The process of adjusting feedback gain in response to the task likely occurs within transcortical feedback loops connecting cortical sensorimotor areas, particularly the primary motor cortex and spinal motor circuits.

Furthermore, the cerebellum plays an integral role in motor learning. It encodes the rules of movement, enabling the acquisition of new skills through repetition and practice. This aspect is particularly evident in tasks that demand rapid adjustments, such as learning how to ride a bike or mastering a musical instrument. The cerebelum fine-tunes motor patterns over time, allowing for smoother and more efficient execution as proficiency develops.

While historically linked primarily to movement, the cerebellum's influence extends beyond pure motor domains. Its intricate connections with various brain regions hint at its involvement in cognitive processes, including language and emotional regulation [32]. This interplay underscored the cerebellum's role as a multifaceted contributor to overall brain function. The peculiar architecture of the cerebellum could be at the basis of its involvement in such a diverse range of functions. The connectivity between the cerebellum and the cerebral cortex is organized in parallel loops: different regions of the cerebellum and the cerebral cortex receive inputs from a large set of cerebral regions, not only primary sensory areas but also from associative areas, through the pontine nuclei (PN). In return the deep cerebellar nuclei (DCN) send back projections to the same cerebral regions through the thalamus, thus forming a cerebro-ponto-cerebello-dentato-thalamocortical pathway.

1.2.3 Microcircuit organization

The cerebellar cortex is histologically divided into 3 layers, distinct enough in appearance to be discernible under a microscope, first described by Camillo Golgi in his book *Sulla fina anatomia degli organi centrali del sistema nervoso*

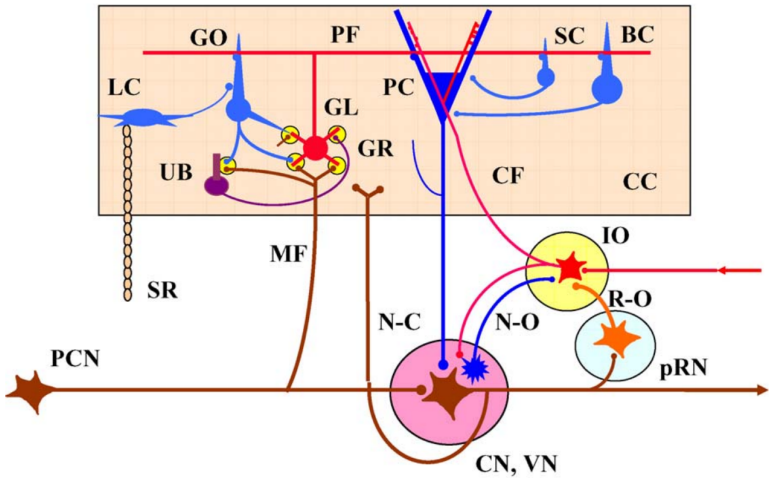


Figure 1.4: Wiring diagram for a cerebellar corticonuclear microcomplex. BC, basket cell; CC, cerebellar cortical microzone; CF, climbing fiber; CN, cerebellar nucleus; GR, granule cell; GL, glomeruli; GO, Golgi cell; IO, inferior olive; LC, Lugaro cell; MF, mossy fiber; N-C, nucleocortical mossy fiber projection; N-O, nucleo-olivary inhibitory projection; PC, Purkinje cell; PCN, precerebellar neuron; PF, parallel fiber; pRN, parvicellular red nucleus; R-O, rubro-olivary excitatory projection; SC, stellate cell; SR, serotonergic fiber; UB, unipolar brush cell; VN, vestibular nucleus. Taken from *Cerebellar circuitry as a neuronal machine* [33].

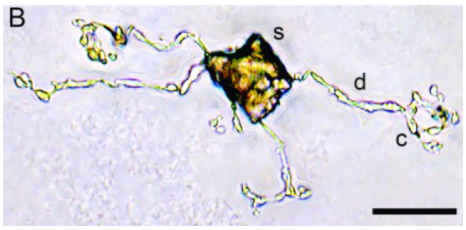


Figure 1.5: The dendrites of a single granule cell visualized by the Golgi-Cox method. Figure 2 panel B of Lackey et al. 2018 [34].

[35] in 1885. The outermost layer is the molecular layer, followed by the Purkinje layer and finally the innermost granular layer.

The granular layer is notable microscopically for its densely packed arrangement of granule cell bodies, the small but numerous cell type accounts for more than half of the human brain's neuron in number, and gives the layer a finely

granulated appearance under most staining methods, such as Nissl body staining [36–38]. The granular layer contains the granule cells, Golgi cells, unipolar brush cells, and Lugaro cells.

The Purkinje layer is a thin transitional layer between the granular and molecular layer that contains the Purkinje cell bodies. The Purkinje cell bodies are arranged in straggered lines in order to pack their dendritic trees as close together as possible along the parallel fiber axis, while maintaining maximal shared parallel fiber input between adjacent cells, and enough space for the fairly large cell bodies [39].

Figure 1.6: Drawing of a cat Purkinje cell by Santiago Ramón y Cajal [40]. **a.** soma, **b.** collateral, **c/d.** dendrites

The molecular layer is the outermost layer and contains the parallel fiber bundles, Purkinje cell dendritic trees, Golgi cell dendritic trees, stellate cells, and basket cells. Within this layer the parallel fibers all run in the same direction, and the dendritic trees of all the cell types extend orthogonal to it, with little extension along the parallel fiber axis.

The granular layer, as part of the cortex, is folded in accordion-like parallel folia to increase its surface area, and when unravelled forms a continuous sheet. The granule cells themselves have a T-shaped axon, with the rising ascending axon, and 2 protruding parallel fibers. The parallel fiber is a particularly long neurite,

in humans up to 2 millimeter long, and both the ascending axon and parallel fiber have a low tortuosity, forming straight thin fiber bundles. All fibers of the different granule cells run parallel to one another in the same direction; this is where the parallel fiber gets its name from [40, 41].

Each granule cell typically has 4 to 5 dendrites, up to 40 micrometer long, that end in claw-like structures (Fig. 1.6) and intermingle with glomeruli. Each dendrite exclusively forms contacts with glomeruli of different mossy fibers. The glomerulus is composed of a mossy fiber terminal bouton, several granule cell dendrites, and Golgi cell axons [42], and is surrounded by a glial sheet of the velate astrocytes [43] to form a bulbous microdomain characterized by shared synaptic dynamics such as glutamate and GABA spillover [44, 45]. These glomeruli form the main component of the synaptic input to the cerebellar cortex. Inside of the glomerulus the mossy fiber forms excitatory synapses on the granule cell dendrites, and the Golgi cells form inhibitory synapses on the granule cell dendrites as well. Outside of the glomerulus the mossy fibers also form excitatory synapses on the Golgi cell basal dendrites.

When we convolute the granule cell anatomy and orientation with the shape of the layer, the granule cells form a tightly packed layer of cell bodies, and along the normal of the sheet the ascending axons rise up into the Purkinje and molecular layer above, where the parallel fibers all run in the same direction and intersect orthogonally with the neurites of the other cell types, remarkably, these cells all show extensive branching, but strictly orthogonal to the parallel fibers. Take the Purkinje cell for example, it forms a massive dendritic tree extending 250 micrometers in both orthogonal dimensions, but only $3 - 6$ micrometers along the parallel fiber dimension [41]. This way, the cytoarchitectonic rules of the cerebellar cortex wire each cell up to as many different, but spatially adjacent, parallel fibers as possible. Taking into account that the granule cells are the most numerous, that the parallel fiber is remarkably long, and that the dendritic tree of the Purkinje cell is among the largest: the number of parallel-fiber-Purkinje-cell synapses is the highest in the human brain, forming its largest synaptic reservoir for computing. The reservoir is not entirely uniform: along each $50 - 150$ micrometer of parallel fiber bundle, distinct microzones exist. The zones alternative between a positive and negative type and have different gene expression patterns, and have a much higher degree of shared inputs and outputs within the microzone than between microzones (i.e., they form a network module), and an extensive one-to-one somatotopy exists between the sensorimotor space and the microzones [46–48].

1.2.4 History

The scientific history of the cerebellum begins with the first mentions of the brain as an organ in ancient history by Hippocrates and Galen, and later anatomically described by early anatomists such as Andreas Vesalius' *De Hu-*

mani Corporis Fabrica [49], Thomas Willis' *Cerebri Anatome* [50], Gottfried Wilhelm Leibniz's *Protogaea* [51], and Johann Friedrich Meckel's *Handbuch der Menschlichen Anatomie* [52], during the Renaissance and modernity. These early descriptions were limited by the techniques available to study them at the time through dissection, and later microscopy. The first neuroanatomical descriptions of the microcircuitry were provided by Camillo Golgi after developing his seminal Golgi staining method and created detailed drawings in *Sulla fina anatomia degli organi centrali del sistema nervoso* [35]. The staining method was quickly adopted by Santiago Ramon y Cahal for drawings in *Histologie du Système Nerveux de l'Homme et des Vertébr és* [40], who were both awarded the 1906 Nobel Price of Medicine for their work on the structure of the nervous system. Golgi for the development of the staining method, and Cajal for his detailed observations of individual neurons which led to his proposition of the neuronal theory that neurons are non-contiguous entities that communicate through contact points, which contradicted the prevailing reticular theory that the brain was a single connected web.

The scientific view on the function of the cerebellum has changed throughout time as well. During ancient times, due to the smaller size of the cerebellum, it was considered of lesser importance to the larger cerebrum, while during medieval times a new view emerged that the cerebellum might be related to play a role in movement. As anatomical studies of the brain intensified during the Renaissance, so too did the interest in figuring out the specific functions of brain regions, but no new strides in understanding were made [50, 53–57].

During the 19th century the first works started appearing on different localization theories that tried to identify which brain regions were responsible for which functions through clinical observations, autopsies, post-mortem studies, comparative anatomy, and experimental lesions. While many of them carried out cerebral experiments, they did contribute both directly and indirectly to establishing a localization theory for the cerebellum as a coordinator of motor control [58–61].

In the late 19th century and the beginning of the 20th century novel experimental studies focussed on the function of the cerebellum and new technologies became available to perform detailed brain stimulation and lesion research. From the combined understanding of the anatomy, microcircuit organization of the cerebellar cortex, the function, and electrophysiology, several groundbreaking computational theories on the information processing and function of the cerebellum were developed during the second half of the 20th century [62–64].

The three seminal models of cerebellar functions, proposed by James Albus [63], Masao Ito [64], and David Marr [62], all share the essential concept that parallel-fiber-Purkinje-cell synapses undergo plastic changes, guided by climbing-fiber activities during sensorimotor learning. However, they differ in several impor-

tant respects, including the role of the cerebellum in motor control, the computational objective of learning, the type of synaptic plasticity, the role of the climbing fiber, the coding scheme used by the granule cells, and the use of internal models.

The Marr model of the cerebellum is a hierarchical model that views the cerebellum as a pattern recognition system. The model proposes that the cerebellum learns to associate sensory inputs with motor outputs by comparing the actual output with the desired output. The climbing fiber is thought to signal the error between the actual and desired outputs, and this error signal is used to modify the parallel-fiber-Purkinje-cell [62] synapses.

The Ito model of the cerebellum is a feedback control model that views the cerebellum as a feedback controller. The model proposes that the cerebellum learns to adjust the gain of a motor control loop by comparing the actual output with the desired output. Just like the Marr model the error signal used to modify the parallel-fiber-Purkinje-cell synapses is calculated between the actual and desired outputs [65].

The Albus model of the cerebellum is a predictive control model that views the cerebellum as a predictive controller. The model proposes that the cerebellum learns to predict the future state of the motor system based on its current state and the sensory inputs. The climbing fiber is thought to signal the error between the predicted and actual states, and this error signal is used to modify the parallel-fiber-Purkinje-cell synapses [63].

The three models of Marr, Ito, and Albus share the essential concept that parallel-fiber-Purkinje-cell synapses undergo plastic changes, guided by climbing-fiber activities during sensorimotor learning. However, they differ in several important respects.

The Marr model views the cerebellum as a pattern recognition system, while the Ito and Albus models view the cerebellum as a feedback or predictive controller, with the same aim reflected in the computational objective of learning. The Marr model proposes that long-term potentiation (LTP) is the primary form of synaptic plasticity, while the Ito and Albus models propose that long-term depression (LTD) is the primary form of synaptic plasticity. The Marr model views the climbing fiber as a signal that indicates an error, while the Ito and Albus models view the climbing fiber as a signal that indicates a prediction error. The Marr model proposes that the granule cells use a sparse coding scheme, while the Ito and Albus models propose that the granule cells use a dense coding scheme. The Marr model does not explicitly use internal models, while the Ito and Albus models explicitly use internal models [62, 63, 65].

The three models have been successful in explaining some cerebellar functions, such as oculomotor control and classical conditioning. However, they have been criticized for being too simplistic and being unable to explain other cerebellar functions, such as whole body movements or cognitive functions [66].

1.2.5 Contemporary research

One of the most active areas of research remains the role of the cerebellum in motor control. Different models with different focuses exist, and consensus agrees on their value in different research contexts, but a truly unified theory of the function of the cerebellum for all motor control is still lacking and remains an area of investigation for theorists [67–69].

Another active area of research is the mechanisms of synaptic plasticity in the cerebellum. The cerebellum serves as an excellent model system for studying synaptic plasticity due to its well-defined neural circuitry and regular structure compared to other brain regions. Yet new findings such as new afferent, efferent and recurrent synaptic sites lead to appreciation of new computational and plastic capacities of the cerebellum [70, 71].

While the cerebellum is commonly perceived as a homogeneous brain region, it presents a challenging puzzle: how can its seemingly uniform microcircuit organization contribute to such a wide array of diverse functions? Contemporary computational neuroscience research into the cerebellum focusses on more accurately modeling reconstructions of the cerebellum, for example, to model disease states [8–10], or on eludicating how the signal processing properties of the cerebellum can lead to, for example, learning, prediction, control, or formation of internal models of the world [72, 73].

1.3 Bottom-up modeling

1.3.1 What is bottom-up modelling?

Bottom-up modeling is a computational and theoretical approach used in neuroscience to simulate and understand the behavior of neural systems, by starting from the smallest components and building upward. This approach seeks to elucidate the complex and emergent properties of the brain through the systematic integration of information from lower levels of organization to higher levels. At the foundation of bottom-up modeling are the individual neural components, which include neurons, synapses, ion channels, and receptors. Each of these components has a specific function and behavior, and they interact with one another to generate neural activity. Bottom-up models represent the biophysical and electrical properties of neurons, including action potentials, membrane potentials, the dynamics of ion flow through channels, and incorporate detailed descriptions of synapses. The synaptic models simulate the release and reception

of neurotransmitters, synaptic plasticity, and the impact of synaptic strength on neural communication. Bottom-up models can replicate the connectivity patterns of brain microcircuitry, and in this way, at the largest scale, may replicate the entire brain [74].

In principle, bottom-up models aim for biological realism by incorporating as much known biological detail as possible. This involves the inclusion of accurate neuronal morphologies, channel kinetics, and synaptic properties. Mathematical equations describe how neurons generate and propagate electrical signals, how synapses transmit information, and how networks of neurons interact. Bottom-up modeling often employs a multiscale approach, where information is integrated across different levels of organization, from the molecular and cellular levels to the network and systems levels. This allows for the exploration of emergent properties. Models are rigorously validated and calibrated using experimental data. Parameters within the models are adjusted to match observed physiological responses, aiming for the simulations to be biologically accurate [74].

Bottom-up modeling helps researchers gain a deeper understanding of how individual neurons and synapses contribute to overall brain function. It provides insights into neural computation, signal processing, and information encoding. These models are instrumental in studying synaptic plasticity, which is crucial for learning and memory. Bottom-up models can simulate neurological and psychiatric disorders, aiding in the study of their underlying mechanisms and the development of potential treatments. These models can be used to simulate cognitive processes and behaviors, allowing researchers to explore the neural basis of various cognitive functions [74].

1.4 State of the Art

1.4.1 Definition of a Multiscale Brain Modeling Framework

A "framework" is an often used term, with no standardized meaning. Often attributed key distinctions from software libraries are inversion of control, extensibility, and some level of opinionation (imposing certain choices on the user to establish productive idioms) [75]. Where a library fits the needs of user code, a framework expects user code to fit its architecture, and together with the user code forms a custom application. The framework runs the operation by invoking the user code at the right times, fitting the framework architecture.

Applied to the multiscale modeling problem, a framework should provide abstractions for some or all of the problems associated with multiscale modeling, and should have well defined interfaces where the user can provide encapsu-

lated model specific code, which the framework uses at well defined moments to execute an extensible workflow.

This definition would exclude software that simply provides functions or classes that perform certain operations that the user calls when they see fit (*libraries*), software that executes a single specific task (*tools*), or any software that strictly runs a preprogrammed *workflow* (perhaps even configurable, but not extensible).

The goal of multiscale modeling frameworks should not just be the simulation of multiscale models, but to cover the entire multiscale modeling workflow, which begins with the ingestion of multimodal datasets and their transformation into the many individual parameter sets of each element modeled at the multiple scales of the model.

This will be referred to as *generation*, which goes beyond *specification*: In multiscale models many parameters aren't merely constants or distributions from literature, but may need to be *generated* from multimodal datasets that are processed in multiple steps, with algorithms determining the resultant parameters for each individual element specific to that scale (e.g., each synapse in the model may have its parameters generated from its location on the cell, or each cell may have its morphology generated from brain atlas information such as the orientation of the histological layer and the cell's location within the layer).

1.4.2 Existing Frameworks in Multiscale Modeling

Tools that are not classified as a *framework* are still listed here when they can at least offer similar value to the neuroscience community for at least one or more parts of the multiscale modeling problem.

Brain simulators have been excluded from the list, as that part of the worfklow has been properly addressed by several competing well-established simulation frameworks that cover the needs of the community.

The state of the art is reviewed to see whether the multiscale workflow is covered outside of simulation as well as for simulation, or whether a need for generative frameworks covering the entire workflow remains. To this end, the source code and documentation of the existing tools were analyzed for the following criteria:

- **Framework**: Does the software meet the framework criteria?
  - Inversion of control: Does the software invert control over user code?
  - Architecture: Does the software have and impose its own architecture?
  - Extensible: Is the software open for extension to a point that it can be closed to modification [76]?

    - Components: Are all of the above captured by a component-based system[1]?
- **Worfklow**: What parts of the multiscale problem are covered?
    - Data: Can multiscale data be handled by the software (declared, retrieved, processed, used, ...)?
    - Specification: Can general multiscale brain networks be specified?
    - Generation: Can parameters be generated from data for individual elements of a scale?
    - Simulation: Can the specified network be instantiated on brain simulators?
    - Analysis: Can the simulation results be analysed or stored in a standardized format?[2]
    - Visualization: Can the multiscale elements be visualized or stored in a standardized format?
- **Scale**: Which scales are covered by the software?
    - Microscale: Can microcircuit elements such as cells and synapses be described?
    - Microscale (non-neuronal): Can non-neuronal elements such as glial, vascular, or non-central elements such as cellular microdomains or extracellular microstructures be described?
    - Mesoscale: Can neuron population dynamics be described?
    - Macroscale: Can whole brain dynamics be described?

1.4.3 Limitations of Current Frameworks

Looking at Table 1.1, the available tools for multiscale modeling center strongly around the specification of neurons and connections, simulation, analysis, and visualization. With the exception of Snudda, no support was encountered for the generation of multiscale parameters, beyond stereotyped spatial distributions and distance dependency. No support was encountered for the incorporation of data and data processing into a multiscale workflow, except for dedicated tools like Sumatra [88].

Instead the user is expected to write its own workflows and pipelines, and the available tools provide only abstractions for the software's take on nodes and edges. The available tools take this network specification and instantiate multicompartmental or point neuron networks in the supported brain simulation engines. Sometimes, configuration files are allowed to separate some of the more numerous or bulky parameter specification from the declarations.

Inversion of control is generally weak and did not go beyond what inevitably

[1] Components here meaning software centered around blocks that define the invariant behaviour that can easily be extended to implement model-specific variant behaviour

[2] Although analysis and visualization of simulation results is an integral part of the multiscale modeling workflow, the criterium can also be met if the software supports standardized formats that are interoperable with analysis and visualization tools.

			BMTK	mozaik	NetPyNE	neuroConstruct	Neurodamus	NeuroML	NineML	PyCabnn	PyNN	Snudda	SONATA	Sumatra	The Virtual Brain
Framework	IoC		✓	✓	✗	✗				✓	✓	✗			✓
	Extensible		✗*	✗	✗	✗	✗✢	✗	✗	✗	✓	✗	✗		✓
	Components		✗	✓	✗	✗	✗✢	✗	✓	✗	✓	✗	✗		✓
	Documented		✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓
Workflow	Data		✗✣	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	Specification		✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗	✓
	Generation		✗◆	✗	✗◆	✗	✗	✗	✗	✗	✗◆	✓	✗	✗	✗
	Simulation		✓	✓	✓	✓✧	✓			✗	✓	✓		✗	✓
	Analysis		✓	✓	✓	✓	✗			✗	✓	✓		✗	✓
	Visualization		✓	✓	✓	✓	✗			✗	✗	✓		✗	✓
Scale	Micro	N	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✗
		NN	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓		✗
	Meso		✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗		✓
	Macro		✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		✓

Table 1.1: Assessment of multiscale tools. At the top, the considered software in alphabetical order: BMTK [77], mozaik [78], NetPyNE [79], neuroConstruct [80], Neurodamus [81], NeuroML [82], NineML [83], PyCabnn [84], PyNN [85], Snudda [86], SONATA [87], Sumatra [88], The Virtual Brain [89]. Label colors indicate software category. Green: framework. Yellow: tool. Orange: worfklow. Blue: library. Pink: specification format. Green checkmarks indicate the criterium was met, red cross indicate it did not, empty gray boxes indicate the criterium did not apply to the software. The microscale is split up in neuronal (N) and non-neuronal (NN). Notes: (*) Extension might be possible through modification or addition of a new simulation backend which was considered to be too involved, undocumented, and not granular enough to qualify as an extension system. (✢) The BMTK includes special support for the NRRD data format. (✧) Generates scripts the user then has to run in the target simulator. (✙) The component system targets simulator control, not the multiscale problem. (◆) Supports spatial distributions and distance dependent properties.

follows from an object oriented and/or declarative paradigm; no special design efforts went into providing structure or architecture in user code. This does not exclude diligent users from writing clean and organized code, but the design of the software never promoted it.

Existing frameworks are also lacking integrations with other tools, although frameworks as a backbone of a project are in the perfect position to do so. Having multiple integrations per problem vastly increases the amount of modeling needs that can be addressed with a single framework. Even when offered integrations are functionally equivalent, they may address more user preferences. Only NetPyNE [79] offered several integrations, and the BMTK [77] offered some additional support for the NRRD data format, which the developing institute, the Allen Institute for Brain Science, often uses.

There are some alternatives to using a framework at all: Some standards exist such as NeuroML [82] and SONATA [87], with enough convergence in the multiscale modeling world that several tools can create model descriptions in these standards, or run models described by those standards on a simulator. A modeler can mix and match several tools together to arrive at a working workflow, but the possibility of reuse with homemade workflows is low since many hidden pieces or implications may exist: hard to automate steps might have been ran manually, specific code may have to be run in specific order, the environment may not be accounted for (software, versions, operating system, ...). In summary there's no guarantee that it is a total description of the workflow that can be reproduced.

The software landscape focuses on neurons and synapses, but no abstractions were encountered to deal with either subcellular or extracellular information (e.g., marking where dendritic spines are located, marking the extent of myelination, modeling tripartite synapses, glial environments, vascular environments, ...) unless it could be integrated into a cell or synapse model, or captured as a parameter thereof. No network specification format was encountered that can accomodate such information, except perhaps for some creative use of SONATA's [87] node & edge properties, or NineML's [83] component system.

Except for dedicated tools like Sumatra, no software was encountered that performed data provenance, parameter tracking, or otherwise bundled the experimental environment with the created output, although it is an essential part of creating reproducible results. Custom logic to retrieve, process and provide input data has to be written by the user. Generally tools supported importing, exporting, or saving recorded data to a standardized format, most often through NeuroML or SONATA [87] for network specification, or Neo [90] for recorded data storage.

When comparing the software landscape of multiscale modeling, to for example that of web development, or astrophysics, or closely related multiscale *simulation*; there isn't a set of competing well-established frameworks that cover the entire workflow, offer abstractions for each piece of the puzzle and offer undeniable advantages to the point that their use has become ubiquitous.

1.5 Research Problem Statement

For the cerebellum

> Can we develop a base model for the cerebellar cortex, and validate basic emergent properties so that it can be used for hypothesis-driven research into the cerebellum?
>
> Specifically, how does physiological input propagate through the cerebellar cortex? How do mossy fiber afferents activate granule cells, and how do different populations in the cortex contribute to the propagation and spatiotemporal processing of activated granule cells with regards to the vertical organization of the cerebellar cortex [91]?

For the multiscale modeling framework

> Can we capture the multiscale modeling workflow in a framework that lets users express their needs starting from multimodal datasets that need to be processed, and the generation of derivated parameter sets, with explicit declaration and parametrization of the used modeling strategies, and promotion of encapsulation, user code organization, quality, and reusability?
>
> Let that framework generate and store multiscale data, and transfer it to brain simulator engines, supporting various paradigms without imposing the restrictions of any paradigm. Let model declaration and configuration be the central value and leverage accessible extension mechanisms to address heterogenous modeling needs.
>
> It should be proper to the framework that described models are a strict sum of components each consisting of their own modeling strategy, input data, and parameters, to lead to easily understood and reproducible descriptions. Model descriptions need to include all information required to deterministically (re)run the workflow to be reused, built upon, and replicated, or altered for the investigation of another hypothesis.

Chapter 2

The Brain Scaffold Builder: Design and Architecture

2.1 Requirements and design goals

The state of the art shows that currently the field covers network simulation and specification, but leaves other areas, most notable data handling, processing, and derived parameter generation up to the user. User code is not managed by the software (i.e., inversion of control and encapsulation are only weakly present), reuse of created user code is low, shared idioms are few, and there is little attention for the promotion of qualitative, understandable, testable, or reproducible user code.

To address this, the software should be designed to provide a framework in which users are helped to write encapsulated code, i.e., code where the code boundaries (coupling between functions, classes, modules, ...) overlap with the boundaries of biological rules, constraints, or modeling choices, so that modelers that face similar constraints can reuse any part of interest of the model in a plug-and-play fashion.

The software should be designed so that all parts of the workflow can be expressed in semantically intuitive abstractions in a declarative paradigm where the important model component and parameter declarations can be separated from implementation details, to provide a clear overview of the model.

The software should be designed with the heterogeneity of the modeling chal-

lenge in mind: each brain region will have its specific cytoarchitectonic constraints that need to be catered to by allowing variant code to be effortlessly expressed and integrated into the workflow.

Putting the design goals together, a declarative paradigm, and a strategy pattern combined with a strong configuration pattern can address the needs to write reusable well-parametrized encapsulated components that carry out heterogeneous variant biological tasks, that can be summed together to describe an entire multiscale modeling workflow.

What follows is a list of requirements that should be met by the software:

- **Workflow**: Parts of the multiscale modeling workflow that should be covered by the software
  - **Parallelization**: The entire workflow must be parallelizable, to scale well with problem size.
  - **Data handling**
    - **Provenance**: The models must declare all data sources in a uniquely identifying manner
    - **Availability**: The software must be able to guarantee that all data required to recreate model instantiations remains available[1]
    - **Abstractions**: Abstractions should cater towards commonly required data formats and data representations[2]
    - **Integrations**: The software should offer integrations with common data sources to obviate custom logic to glue pieces of the workflow together
    - **Processing**: The software must accomodate step-by-step processing pipelines from base data sources into prepared input data for model generation
- **Specification**: The framework should support model specification across the following scales and paradigms
  - Subcellular
  - Neuronal
  - Non-neuronal
  - Populations
  - Connections
    - Individual (microscale)
    - Probabilistic (mesoscale)
    - Population (macroscale)
- **Generation**: The framework must operate under the assumption that parameters may need to be generated from data

[1] Due to large size of datasets, adding an opt-out mechanism may work beneficially towards feasible model sizes.

[2] For example manipulation utility libraries for cell morphologies and rasterized 3D brain images. Considerable advantages should be offered over simply supporting a file format.

    - **Cells**: Capable of hosting generated cell information, to arrive at plausible electrophysiology and morphology.
    - **Subcellular**: Capable of hosting generated subcellular information, such as microdomains or microstructures on cells.
    - **Populations**: Support reducing information on cells into population level information.
    - **Non-neuronal**: The software should not restrict to a neuronal dogma, and allow for example glia cells and vasculature to be described.
    - **Connections**: Capable of hosting generated information on the interactions between multiscale elements, for example chemical, electrical, or tripartite synapses, and neurovascular coupling.
- **Simulation**:
    - **Instantiation**: Should support instantiating the described model in a simulator
    - **Multiparadigm**: Should support simulation at multiple scales and in the common paradigms of each scale
        - Multicompartmental
        - Point-neuron
        - Populations
    - **Cosimulation**: Allow the multiscale model be simulated at multiple scales at the same time, with different parts of the model represented at different scales.
    - **Recording**: Record the obtained results into a standardized format
- **Function**:
    - **Inversion of control**: Manage the workflow, have user code meet the imposed framework expected structure.
    - **Generic**: Assure the user needn't break architecture to express their intentions
    - **Extensible**: Allow extension of framework function. Any functionality offered by the framework should use the same extension mechanisms a user would.
    - **Component-based**: All functionality should be offered in well-encapsulated base components that the user can extend with a well-defined interface.
    - **Architecture**: Promote higher-order non-imperative code organization in user code
    - **Utility**: Offer utility libraries to simplify common tasks
- **Model description**:
    - **Separated**: A model description should exist separate, or separatable from the model implementation.
    - **Composable**: Can models be added together by adding their descriptions together?
    - **Complete**: It must always be sufficient to read the model description, to reproduce instances of the model, or to at least establish and

collect the missing pieces.[3]

2.2 Overview of the Brain Scaffold Builder Framework

This overview serves to introduce all major concepts in the framework. More in-depth explanation and examples will follow in their own sections. "See also" notes are provided for easier navigation.

The framework is a black-box component framework for large and multi-scale bottom-up modeling. Black-box component frameworks invert control by encapsulating each task into a taxonomy of components and lets users implement their functional interfaces. By developing against an interface the user provides strictly the variant part of the task, while the framework can take over control of the invariant backbone of the task, and stably invokes the variant parts by operating on the components' interface.

By far the largest part of the bottom-up modelling code effort is invariant (i.e., remains the same for all tasks in the domain, and would need to be repeated by each individual). Solving the software design for this invariance while providing a layer of abstraction of semantically intuitive component types is the quintessential value provided by the framework. By taking over control of these invariant parts the framework can guarantee well-tested optimal solutions for common problems, and the modeller is left only with a fraction of the work.

2.2.1 Overview of the workflow

Models are created in the context of directory structures called projects. A project contains the model configuration and all the model code and data sources, either as a structured remote reference in the configuration, or as a relative local file. Projects may contain multiple alternative model configurations. Template projects can be generated through the Command Line Interface (CLI) and demonstrate idiomatic framework practices.

Models are described in their totality by declaring the model components and their parameters. Declarations happen in the confines of a structured component tree and can be made in Python, or a configuration format. The component tree begins at the root as a named collection of root nodes, that all house components of a specific type (see Code Snippet S7.1).

Placing a component in a root node declares a new piece of the model and adds a

[3] If any implicit dependencies or references are not described, the model description is not complete: For example, all exact versions of the software, and any dependencies of user code; or references to local files, ... may be missing, and reconstruction would not be possible

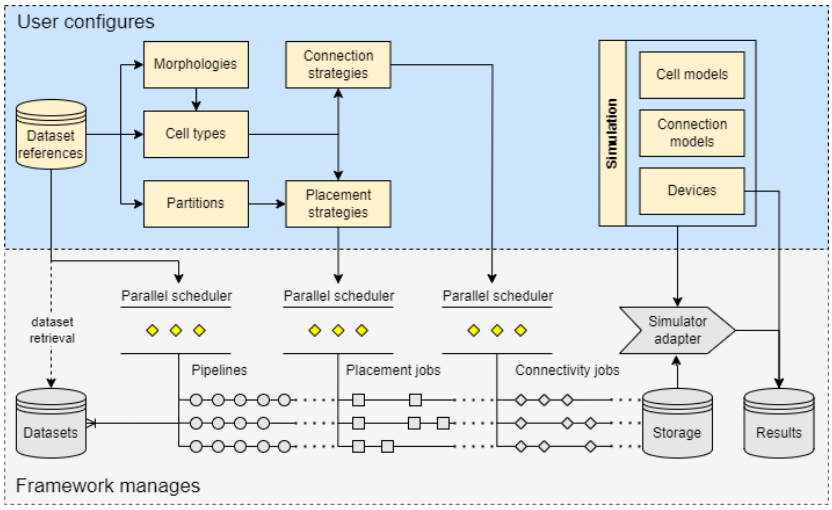


Figure 2.1: Diagram of the main framework worfklow. In the blue area: the components configured by the user, and used by the framework to understand the objective; arrows indicate dependencies between component types. In the gray area: Tasks performed by the framework based on the configuration: datasets are retrieved, jobs are scheduled to process the data, and passed to their dependent placement and connectivity jobs. The model description, inputs and the output are all stored in the storage object. The storage object can then be used to run a managed simulation workflow, by loading the stored data into simulator specific adapter components, which will instantiate the model on the simulator backend. Results are recorded by device components and stored into a result object.

new objective to the workflow. E.g., placing a component in the cell types node defines a new cell type (Code Snippet S7.2 contains some basic declarations).

Each component is polymorphic and can specify its own attributes and subnodes in the component tree. This helps users to explicitly parametrize their components, and to separate the important model parameters and dependencies from the implementation.

N.B.: This will lead to the final structure of the component tree to depend on the components a user declares. In the text, when we speak of "components", this always implies that a user can also provide their own (see 2.3 for in-depth explanation on the component system).

The component tree is from here on out also referred to as the "configuration" of a model, or the "model description" (see Code Snippet S7.5 for Chapter 4's cerebellar cortex model description, and corresponding diagram in Fig. S5.2). The configuration is an instruction manual for multiscale data processing, generative algorithms, and simulator specific instructions that can be followed to run predictable workflows for the reconstruction of the model and its simulation. (Groups of) stochastically generated model instances produced by the model generation workflow will be referred to as "model samples", or "model reconstructions".

Quick note: on top of components, which are bootstrapped from the configuration, the framework can also be extended by plugins. Plugins are automatically discovered so they do not need to be bootstrapped from the configuration and can provide components to the user, or extend framework functions (see section 2.9 for more on plugins).

A first example of such a plugin category are the configuration parsers. A component tree can be programatically declared in Python, or specified in a configuration format which needs to be parsed. Current parser plugins offered by the framework can parse JSON and YAML.

The model description in Code Snippet S7.5 is vertically lengthy and spans multiple pages in this document. In real projects one would navigate them with a proper IDE with collapsable nodes and use the built in dot diagrams[4]. The YAML and JSON parsers are also extended to support matrix expansions, multi-document layouts, and reusable subdocuments (see Code Snippet S7.3). Together with the possibility to write short procedures to generate the declarative nodes, this can maintain compactness and hierarchical overview of the model description even when the total number of lines required to describe the model grows.

[4] `https://graphviz.org/`

It's a parser's task to convert the configuration format into a component tree, or vice-versa. Since the parsing (from format to Python) and reverse parsing (from Python to format) operations are bijective and stable one can programmatically generate, modify compose and then serialize, or deserialize model configurations. This is ideal for programmatic use such as transfer to remote machines or processes through sockets or pipes (e.g., "pickling" in Python), batch job submission, or other tooling on top of the framework.

The framework offers the user a scaffold to tie together all the abstract information, and concrete data, into a client that is capable of processing all requested workflow operations on the model. The main operation called "compilation" generates data from the description (see Fig. 2.1 for a depiction of the 3 stages of the compilation workflow managed by the parallel job pool schedulers).

Compilation runs the model workflow and writes the generated data into a storage file. The stored information includes the cells with positions, rotations, arbitrary generated user properties, and individually grown morphologies, with specific synapses on precise subcellular targets, again with their own arbitrarily generated properties, aware of the network topology, space and cells around themselves.

Compilation happens from a single source of truth: the declarative component based configuration (see Fig. 2.1, blue area, for the main configuration blocks related to compilation). From a description representable by a diagram of blocks with explicit inputs, parameters, and outputs, large and intricate multiscale brain models with detailed neurons and synapses can be generated and simulated. Scientific readers that have no affinity or interest in decyphering large amounts of imperative code can still glance and understand the configuration underlying the model, and choose to inspect any of the well encapsulated single blocks of interest for any algorithm's implementation details.

The framework understands complex hierarchical network topologies for the spatial layout of the network. The volume is described as a set of *partition* components (such as layers, meshes, voxelsets, ...). Partitions can be laid out using a tree of *region* components (such as groups, stacks, spacers, ...). Several operations can be performed on the volume definition, such as translation /rotation to compose them with other models, or scaling/cropping to create descriptions of smaller test networks[5].

The topology is the basis to chop the volume into equal pieces for parallel processing. Postprocessing steps are supported to deal with any border artifacts

[5]N.B.: This is at the level of the model description, not the data. Without separating the topology from the other components, one would have to specify and adjust all the spatial confines all throughout the configuration.

that follow from this approach or global constraints that can not be computed from a local parallelized context.

A compilation workflow consists of multiple stages, each stage transparently parallelized by the framework, made possible by careful design of the component interfaces, so that the framework can request scoped information from components amenable to parallel job submission (Fig. 2.1).

As the first stage (Fig. 2.1, dataset & pipeline blocks), the dependency pipelines are checked. Each pipeline specifies a series of input data sources, operators and parameters (see section 2.7.2). A caching system checks whether the data sources need to be retrieved, and a parallel scheduler distributes the operations across the available workers. For each dependency the used data and a description of the processing operations are stored so that model reconstructions contain all data required to rerun, reparametrize, reference, or otherwise reproduce any aspect of the model reconstruction.

After the data dependencies have been checked and processed, the placement stage is ready to be run. Each placement component schedules multiple parallel placement jobs to fill partitions with certain cell types according to the algorithm implemented by the component. During the placement the components generate positions for the cells, and can be configured with subcomponents (called *distributors*, see 2.5.3), for example to generate de novo morphologies, or other properties. The components have access to all the spatial information and can generate properties accordingly. e.g., morphologies can be generated on the fly that are constrained by the histological layer, or grow towards another layer to innervate their targets.

Arbitrary properties can be stored on each point of the morphology to emulate subcellular features such as dendritic spines, or properties that depend on the position of the neurite in the histological context. Individual cells and synapses support generating arbitrary properties too, and can be labelled to create subpopulations.

These arbitrary properties are important to escape entrenchment in a presupposed modeling context. While specific semantics like “cell position” and “cell morphology” ease the basic workflow, if a user were restricted to them, they’d have to break the framework architecture to accomodate their model specific semantics. This is why the framework allows a user to declare components for the generation of model-specific information, associated to a name. E.g., a user’s distributor component (let’s call it the “SpineDistributor”, can generate a boolean value per point on the morphology indicating whether there’s a dendritic spine there, and store it on the cell morphologies under the “spine” name.

Although the framework doesn't know by default what to do with this data, it can be given meaning during simulations by creating a `Parameter` component that processes the data into parameters of existing simulation elements, or represent the data by creating new elements. Returning to the example of dendritic spines, one could take the boolean value and change the membrane capacitance of cell models wherever a spine occurs, or create a new compartment there with its own dynamics. If a property is only going to be used to (sub)select model entities, then a specialized system exists to add labels to points of interest across scales and later select groups based on their labels. E.g., one could label all the points on a generated morphology that should form synapses only with specific cell types, or for debugging purposes label all the cells that were placed under irregular conditions, such as borders, to validate their properties.

After the generation of the cells and their spatial information, the connectivity stage is ran. The connectivity stage follows the same tiling paradigm, but regions of interest composed of any number of tiles can be subselected. This allows for more efficient scaling on very large reconstructions. e.g., an algorithm that connects cells within 100 micrometer does not need any information of cells in chunks that are further away than that, and can implement appropriately restricted regions of interest.

See section 2.7 for a deep dive on multiscale data generation and placement and connectivity data structures, which explains what inputs a user's placement and connectivity strategy components are given, the interplay with the tiling paradigm, what output is expected, and how the framework processes the information.

Note on modeling paradigms. There are no assumptions or requirements made regarding the modeling paradigm used to represent cells or connections[a]. Minimally, a user declares a component that generates cells without any information, akin to declaring a population of a certain size. To the framework, cells are black-boxes that it generates and stores black-box information and interactions for. Most semantics are merely implied, and are reified only when the information needs to be transferred to a simulation backend[b].

[a] A current limitation is that only 2 cells can partake in a connection, although workarounds exist

[b] e.g., to create a "glia cell", a user calls their cell type "astrocyte", generates astrocyte-like positions and connections using their own black-box astrocyte components, and during simulation uses a cell model component that performs astrocyte-like duties in the simulator. The framework follows the exact same workflow it would for any other type of cell, by calling the component interfaces.

Connections can be formed on any point of the morphology, and connection components that wish to perform morphology intersection can rely on optimal bounding box intersection of the cell types to preselect cell pair candidates,

or preselect which pieces of the morphology they regard presynaptically and postsynaptically, and then continue to implement more use-case specific constraints from there. The default creates typical axodendritic connections, but any morphology label the user generated can be specified

The generated network storage files are random samples of the model description and can be used to run simulations. The storage format is encapsulated in a component called a storage engine, and currently supports only our homemade HDF5 engine as no other format seemed to suit our need for multiscale detail[6]. The simulation stage is entirely disjoint from the reconstruction stages, and exists to seemlessly transfer all the generated data to a simulator that matches the modeling paradigm. Arbor and (Core)NEURON are supported for multicompartmental simulations, and NEST for point neuron simulations. The data transfer is also encapsulated by a component called a simulation adapter, and its interface can be implemented for any simulator. For example, initial enthusiasm from early-adopters has resulted in a community maintained Brian2 adapter package.

The generated information is used to create the simulator specific representations. Unlike PyNN, the BSB has no unified simulator interface. It expects of each simulation adapter that they, as part of their interface, implement 3 base types of components to represent any cellular information (called a cell model component), connection information (connection model component), and experimental protocol information (device component). These components represent the transfer of generated data, over to the simulator, and from there these components together with the adapter are responsible for managing the simulator state and executing simulator specific instructions to run the simulation. This allows for more paradigm specific information to be transferred, and for more specific components to be provided per adapter, but has as a drawback that models require multiple simulator-specific simulation configurations if they want to simulate on multiple simulation backends[7]. The simulation workflow ends with the collection of data from the simulator, stored using Neo, which supports roughly 40 neuroscience information exchange formats.

2.3 The Configuration and Component System

```
@config.node
class Example:
    min = config.attr(type=types.int(min=0), required=True)
    mode = config.attr(type=types.in_(["slow", "fast"]), default="fast")
    children = config.dict(type=str)
```

[6]SONATA is a good next candidate because it would drastically increase the interoperability of the framework, but the standard would need to be extended

[7]That is, until someone implements a PyNN adapter to piggy-back on.

Code Snippet 2.1: Example of a node class and its configuration attributes. The `config` module is the `bsb.config` module, and the `types` module is the `bsb.config.types` module.

```
min: 50
mode: slow
children:
  - A
  - B
  - C
```

Code Snippet 2.2: Example of a valid configuration for the node in Code Snippet 2.1 using YAML.

The base elements of the configuration system are lists, dictionaries, and nodes. Lists contain ordered values, dictionaries contain key-value pairs, and nodes contain a set of attributes, as defined by the node class. The `bsb.config` module contains multiple node class decorators and descriptor factories with which node classes can define their configuration form (compare the analogy between Code Snippet 2.1 and Code Snippet 2.2).

List, dictionary, and attributes values are validated using type handling components[8]. Each list, dictionary or attribute specifies their own type handler. Node classes function as type handlers as well, and recursively convert a piece of the configuration tree into a component with nested lists, dictionaries, attributes each again perhaps containing subnodes.

A model configuration is thus determined by starting from the root node class, checking for configuration descriptors on the class, and passing the configuration values to the type handlers of the descriptors. Since descriptors can specify type handlers resolving to nodes, lists of nodes, or dictionaries of nodes, a recursive process ensues that traverses the entire configuration tree, and converts it into a component tree (See flowchart Fig. 2.2). After construction of the component tree, each node is visited to resolve references between nodes that were specified in the configuration (see 2.3.2 for more on configuration references), replacing the configuration reference value with a reference to the component object in the tree that the reference pointed at. E.g., the string reference `"celltype_A"` might be resolved to the cell type component named `celltype_A`.

Using the `config.node` decorator a class is modified to incorporate this recursive

[8]Type handlers are callable components, or even functions in simple cases. The provided type handlers already cover most use cases, including boolean operators such as “type A OR type B”.

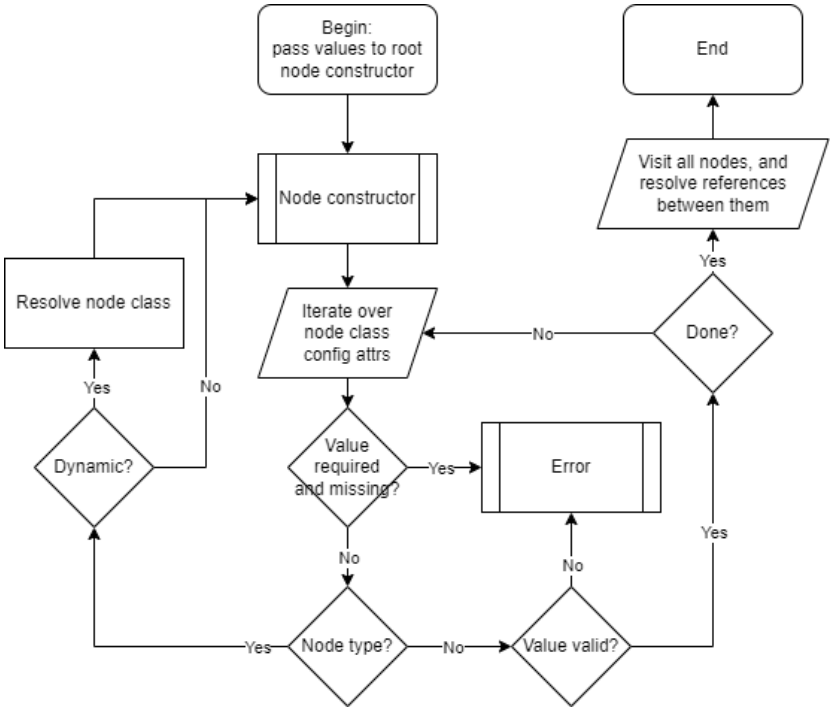


Figure 2.2: Flowchart of component tree construction: flow of the recursive node-constructor based algorithm that converts the user specified tree of configuration values into a tree of components ready to perform their framework function.

algorithm in the class protocol[9]. Users can integrate their own components into the configuration by inheriting from a dynamic parent node class (explained below), and then creating configuration attribute descriptors using the various descriptor factories to prescribe and access configuration values (see 2.3.2 for the different descriptor factories).

The component system is further enriched with a strategy pattern [92] that allows users to bootstrap their own components implementing the parent component interface. Using the `config.dynamic` decorator[10], a class can be marked as a host for a component type with a strategy pattern. The dynamic decorator will add a configuration attribute to the class that will be resolved to the class that should be invoked, rather than the parent node class, wherever another configuration attribute expects a parent-typed node (see 2.3.1 for an example of a parent and child node, and configuration examples)

The default resolution mechanism (which can be overridden by passing a type handler to the dynamic decorator) treats a dotted path as a location to an importable Python module, with the last part of the path being the name of class variable to retrieve from the module. The config system then imports the module and fetches the class. Some component types use a *classmap*, which maps short recognizable names to the fully qualified class name[11]. Code is treated as a dependency of the model description, and should be referenced through the appropriate data source dependencies (see 2.7.2).

In conclusion, we now have the mechanisms in place for users to define, declare, encapsulate, and parametrize their custom algorithms for cell placement, connectivity, or instantiation of simulator elements based on that data, or any other function the framework supports with the component system. The available component types are described in section 2.5. The structure of the configuration tree is largely unknown, and the framework itself only relies on a couple of root dictionaries and lists to perform its workflows, while the exact dependencies or algorithms are specified by the strategies implemented in the components, and these components may specify whatever configuration attributes, nodes, or data dependency nodes (see 2.7.2) they may need to perform their function.

In the framework, the component configuration and invocation usually happens within a scaffold bootstrapping flow, see the scaffold section 2.4 for more information.

[9] By modifying the class, metaclass, `__new__`, `__init__`, and `__init_subclass__` methods

[10] So named because by using the dynamic decorator, classes can be dynamically loaded

[11] Just declaring a class somewhere is not enough for it to be picked up by the class map, but listing it in the `components` list solves this bootstrapping problem. Components can also be provided through plugins, which are discovered, but should still be listed in the `packages` dependency list

2.3.1 Node class decorators

The behaviour of nodes in the configuration is established by modifying class objects.

`bsb.config.node`

The `node` decorator modifies the class to accept either a dictionary of given configuration values, or a set of keyword arguments of either configuration values or runtime values. This allows nodes to be in any Python context, be it during framework workflows, or by advanced users using the framework in their own workflow scripts.

The algorithm set up by the node constructors is demonstrated in Fig. 2.2. The configuration attributes are determined by traversing the parent classes according to Python's Method Resolution Order[12] to respect the entire parent hierarchy of configuration attributes as well.

`bsb.config.root`

The `root` decorator modifies a class to be the starting point of a configuration tree. This node is responsible for bootstrapping the code dependencies listed in the `components` root attribute. Then, it can initiate instantiation of all its other attributes. After its attribute processing, due to the recursive nature, component tree construction is complete, and the root class will finalize the component tree by resolving any configuration references. Configuration nodes without this modifier can construct a component tree as well, but it will contain unresolved references which error out when accessed, until the node is inserted into a configuration tree that does have a root node class at its root.

[12] `https://www.python.org/download/releases/2.3/mro/`

```
bsb.config.dynamic

@config.dynamic(
    attr_name="strategy",
    auto_classmap=True,
    required=False,
    default="random"
)
class MorphologyDistributor(Distributor):
    @abc.abstractmethod
    def distribute(self, positions, morphologies, context):
        pass

class RandomDistributor(
    MorphologyDistributor,
    classmap_entry="random"
):
    def distribute(self, positions, morphologies, context):
        return self.scaffold.random.np.integers(
            len(morphologies),
            size=len(positions)
        )
```

Code Snippet 2.3: Example of a dynamic parent class, `MorphologyDistributor`, implementing a strategy pattern prescribing a component interface that needs to be implemented (namely the abstract `distribute` method), and a child class that implements the interface, and assumes the `"random"` label in the parent classmap. Since it is also the default value, not assigning a distributor, would assign the random distributor.

The `dynamic` decorator implements the polymorphism of the configuration nodes, and allows a parent node to be resolved to any of its child node classes by locating and importing the class as explained in section 2.3.

The decorator adds a configuration attribute to the parent node that can be customized. The default convention is that if the purpose of the polymorphism is to implement the strategy pattern (i.e., a variant algorithm to be carried out) that the name of the attribute is "strategy", and if it is more generic than that it will be named "type".

The dynamic attribute can be customized with the same properties as any other node attribute (see 2.3.2).

Dynamic classes support a class map as well, where mnemonic devices can be associated to their importable class name, e.g. `"cone"` could map to

`bsb.topology.partitions.Cone`. Regular class maps need to be defined on the parent class, and are not extensible, but class maps may also be *automatic*, and can be extended simply by inheriting from the parent class. The automatic map infers a snake cased name from the class name, or a specific name can be set with the idiom `class MyClass(Parent, classmap_entry="custom"):`, which is processed by the parent's `__init_subclass__` to add it to the class map.

`bsb.config.pluggable`

```
@config.pluggable(key="simulator", plugin_name="simulation backend")
class Simulation:
    name: str = config.attr(key=True)
    cell_models: cfgdict[CellModel] = config.slot(
        type=CellModel,
        required=True,
    )
    connection_models: cfgdict[ConnectionModel] = config.slot(
        type=ConnectionModel, required=True
    )
    devices: cfgdict[DeviceModel] = config.slot(
        type=DeviceModel,
        required=True,
    )
```

Code Snippet 2.4: Example of the `bsb.simulation.Simulation` class using the `pluggable` decorator to load available simulation nodes based on the installed simulation backends, and three slots for the cell models, connection models, and devices, which are required by the framework, but need to be provided by the simulation backend.

The `pluggable` decorator (see Code Snippet 2.4) modifies a class so that it will be dynamically loaded, but not by importing a class, but by loading a plugin (see 2.9) of a certain category. Pluggable nodes may contain `slot` attributes (see 2.3.2) which the plugin should provide[13].

2.3.2 Configuration descriptor factories

The configuration system is descriptor based. In Python the descriptor protocol[14] describes the actions taken when a property of a class instance is accessed. Using a set of descriptor factories the framework, and the user can create node classes that convert configuration trees into component trees.

[13] This decorator, like the others, are part of the public API, so users may leverage the integration between the configuration and plugin system in their own components as well.

[14] `https://docs.python.org/3/reference/datamodel.html#implementing-descriptors`

`bsb.config.attr`

The `config.attr` is the basis for declaring an attribute on a node. Attributes have a type handler that validates and transforms the configuration value to its component value, and the inverse operation to convert them back to configuration values . Attributes can be required (or set conditional requirement functions), can have a default value (or value factory), and can contain documentation and hints for automated documentation and validation schemas[15]

Configuration attributes can raise `CastError`s or `RequirementError`s, when the value can not be cast, or a required value is missing, respectively.

The type handler can contain complex types such as `config.types.or_(config.types.dict(type=config.types.list(type=int)), config.types.list(type=int))` to describe the expected type of the attribute. Usually for nested types it is better to specify a node class as the type handler: `config.attr(type=MyNode)`, so that the `MyNode` class can contain additional logic that pertains to the structure.

`bsb.config.list` and `bsb.config.list`

The list and dictionary descriptor factories can be used for ordered or keyed collections of items respectively. They contain additional logic for the runtime mutation of the component tree, so that when nodes are added and removed the rest of the tree can be notified.

`bsb.config.file`

The `file` descriptor creates an attribute which will be treated as a URI. The appropriate URI scheme will be used to retrieve, cache and bundle the file dependency in the storage object. This way, any user components with external dependencies, be it data, code, ..., can effortlessly bundle their dependencies with model reconstructions, making them self-sufficient distributions of the model. The descriptor contains a `cache` attribute which can be set to `False` to opt-out of the caching and bundling mechanisms. This is usually not recommended, but some users may prefer not to bloat their model reconstructions.

[15] The bsb-json package (`https://github.com/dbbs-lab/bsb-json`) can for example generate JSON schema's (`https://json-schema.org/specification`) from any config object to validate it, or integrate with IDE autocompletion.

`bsb.config.ref` **and** `bsb.config.reflist`

```
@config.dynamic(attr_name="strategy", required=True)
class PlacementStrategy(abc.ABC, SortableByDeps):
    cell_types: list["CellType"] = config.reflist(
        refs.cell_type_ref, required=True
    )
    partitions: list["Partition"] = config.reflist(
        refs.partition_ref, required=True
    )
    depends_on: list["PlacementStrategy"] = config.reflist(
        refs.placement_ref
    )
```

Code Snippet 2.5: Example of a the placement strategy node containing reference lists to cell types, partitions, and a self referential list of dependencies on other placement strategies. `refs` is the `bsb.config.refs` module.

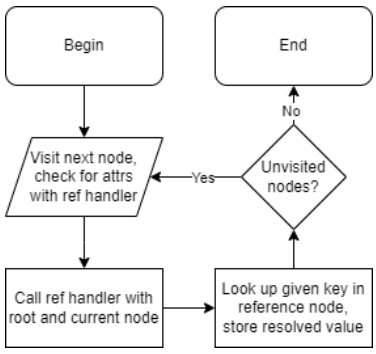


Figure 2.3: Flowchart of the reference resolution node visiting algorithm. The key aspect is the reference handler, which based on the root and/or current node (i.e., through an absolute or relative location reference) determines the parent node in which to look up the given reference key.

```
cell_types:
  cell_type_A: ...
placement:
  placement_A:
    cell_types:
      - cell_type_A
```

Code Snippet 2.6: Example of the `cell_types` reference list, containing a reference to cell type A in YAML. At runtime, the generated configuration object `placement.placement_A.cell_types` will have resolved the string `"cell_type_A"` to the cell type object.

The reference descriptor factories allow one attribute to point to other locations in the tree. Using a callable class or lambda, each descriptor can describe where to look for referenced values in the tree (see Code Snippet 2.5, Code Snippet 2.6, and Code Snippet S7.4 for an example of a reference handler). The `reflist` descriptor works the same way, but resolves a list of configuration values instead of a single value.

`bsb.config.slot`

Configuration slots work together with the pluggable node classes to establish an expected configuration interface for plugins. For example, the storage backend plugins rely on it to provide a storage node, that describes the location of the storage object. This allows filesystem based engines to expect a node with a file path, while remote storage objects may specify all the attributes they require to connect to a remote database. The other example in the framework are the simulation backends, that are expected to fill in slots for the cell model, connection model, and device nodes, so that some uniformity between simulation backends can be established[16]

Others

There's also a `property` attribute that lets users customize the getting and setting behaviour of the configuration value; `provide` attributes that use a getter function to provide the configuration values required by parent nodes; `catch_all` attributes that collect surplus/unknown attributes into a data structure (e.g., collect all unknown given attributes into a "parameters" dictionary); and `unset` attributes that strip the declaration of a parent attribute from a child class.

2.4 The Scaffold

While the model description is the central point for declarations and is embodied by the configuration object and its component tree, the `Scaffold` class is the central client for operations using that model description, and executes any operation requested by the user. It interacts with the configuration to get the components whose functions and strategies determine the outcome of the operation, and with the storage object should any operation be stateful and needs to read from or write to the state contained within the storage object.

The scaffold forms the most important public API in the framework, you can view its API reference documentation[17] to form a general idea of the operations the `Scaffold` client can perform.

[16] The name of the nodes `cell_model` and `cell_type` are probably the only semantic coupling of the BSB as primarily a microcircuit builder centered around cells.

[17] `https://bsb.readthedocs.io/en/latest/bsb/bsb.html#bsb.core.Scaffold`

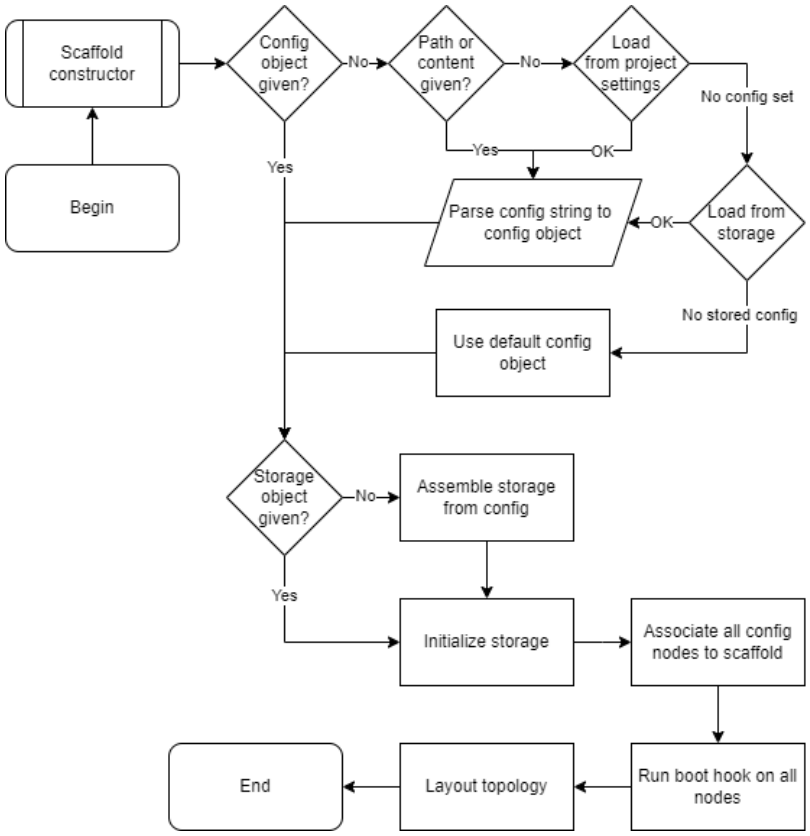


Figure 2.4: Bootstrapping process of a scaffold object. The process begins by determining the config and storage objects to use for the scaffold, and then visiting and preparing all the components in the component tree, and laying out the topology (see 2.6.1).

To construct a scaffold the user calls the constructor either without arguments, to initialize an empty scaffold, and adds component declarations during runtime, and then calls the desired operation methods; or, passes in a path-like Python object, to load a configuration file; or, passes in a content string, to parse it; and, may provide a storage object to use. After the config and storage object have been established, the scaffold inserts a reference to itself to establish a bidirectional relationship between the scaffold and every component, bootstrapping the scaffold. For further initialization of components that requires this bootstrapped scaffold context, a `boot` hook is provided that components may implement. The bootstrapping is finalized by passing through the topology component tree in order to update the layout. From this point on the scaffold and all of its components are ready to carry out their functions.

Since the framework strongly inverts control the user will likely only interact with the scaffold from inside a component context through the component's `self.scaffold` attribute that the scaffold inserted during bootstrapping. However, should the user, or tool developer, want to, they can use the scaffold's API to modify and generate the configuration, interact with the stored data, or run their own workflows.

2.5 Core Component Types

2.5.1 Main component assemblies

There's 2 component assemblies in the framework, these assemblies are components themelves and manage a structure of subcomponents. The configuration root assembles the parsed configuration tree into a tree of components. The storage engine is an example of a plugin component; In the framework plugin components have both a direct interface with the framework, but also have an expected set of plugin specific subcomponents they must provide to the framework. The storage engine must provide components for storage APIs. These APIs offer the user an interface to interact with the generated data in identical fashion no matter which storage format is used. The tree of components declared by the user to describe its model, and the storage engine with subcomponents that bridge to the datasets of the model are then assembled into the main `Scaffold` object, that can process all of the client's requested operations on the model (Fig. 2.5).

2.5.2 Storage objects

The storage object coordinates all manipulations requested by the scaffold on the data, such as read, write, move, delete, copy, merge, and clear. The storage `root` and `engine` attributes form the sufficient ground to describe a storage: the engine determines the storage format that will be used, while the root should uniquely identify and locate the storage. For filesystem based formats these may

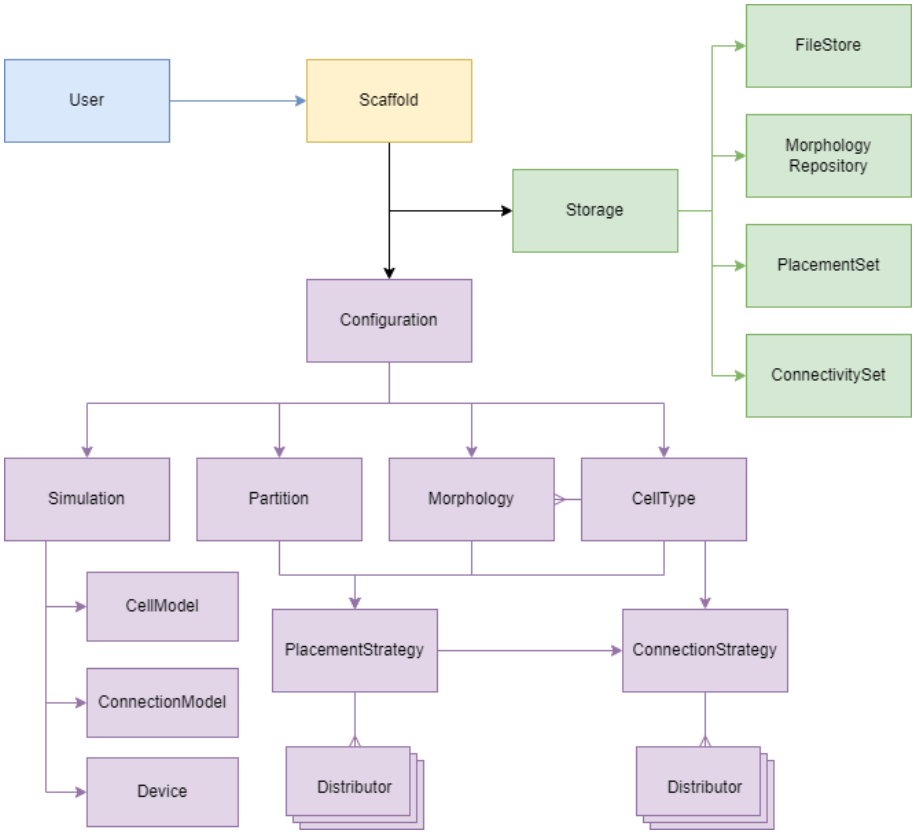


Figure 2.5: Component class diagram of the main component types of the BSB. The user (blue) interacts with the `Scaffold` top-level component assembly (yellow), which manages the main component assemblies that the user has configured: the `Storage` assembly (green) and `Configuration` assembly (purple).

be a path, while for online formats the root may be a URI, database connection string, or any compound data structure.

Once the storage has been initialized the scaffold may request instances of the following component type, to manage certain data sets.

Engine

The engine coordinates access to the resources to guarantee atomic concurrent access is possible, and performs any read/write operations whose responsibility does not belong to other more specific APIs listed below. The default engine is the BSB HDF5 engine. We consider it temporary until a standard format can be adopted. The format is unspecified and adjusted to the needs of the framework.

A storage engine may provide support for any of the following interfaces. If the framework attempts to use a feature for which the engine provides no support, either a warning or error is raised, if the missing feature is essential to the operation.

FileStore

The least domain specific interface is the generic file store. The file store tracks files and their metadata, for provenance so that all inputs of the model can be bundled along with the generated data. In the file store you'll find the active configuration of the model, ad-hoc code dependencies, input data (online data may be retrieved and stored as well), input morphology files, and the results of any processing pipelines on those inputs.

The 'FileScheme' component implements URI schemes, and some are provided for file paths (file://), http(s) (http://, https://), NRRD raster data for 3D brain images (nrrd://), connections to online repositories such as NeuroMorpho (nm://) or the Allen Brain Atlas (allen://). These schemes are used to customize the identification, retrieval, storage, encoding, and caching behaviours of the resources in the file store. Any node of the configuration tree can declare a `file` attribute (see 2.3.2) and the provided value will be treated as a URI and incorporated as a dependency of the model. This means that even user components can fluently declare their own file dependencies and leverage the full framework support. Further file support exists in the form of `FileDependencyNode` components, which is a taxonomy of more specialized nodes (e.g., the aptly named `CodeDependencyNode`, `MorphologyDependencyNode`, `NrrdDependencyNode`, ...). More than a single attribute on a node, these nodes have specialized attributes and/or subnodes that can customize behaviour, and specify how to load the runtime data structure from the data, as well as data processing pipeline (see Code Snippet 7.6 for examples).

The core API are the `all`, `store`, and `load` functions, to list all available files, store one, or load one, respectively. There's also additional functions to support file metadata (`get_meta`) and specific cases of metadata such as encoding or the time it was last modified (`get_encoding`, and `get_mtime`).

MorphologyRepository

The morphology repository contains all the morphologies that have been loaded from data sources or generated de novo during a workflow for the model. The morphologies are stored in the framework's own representation as essentially an (NxM) matrix and a DAG, where N is the number of points, and M is the number of columns. The number of columns is established by the 3 spatial columns XYZ, the radius column, the label column, and the additional property columns defined by the user (see 2.8).

The repository stores metadata alongside the morphologies, and using the `MorphologySelector` components suitable morphologies can be searched for in the repository. One can for example select all the morphologies that fit certain size constraints, by looking through the metadata without having to load the morphologies. The selector component type helps to decouple the morphology from its technical identifier (filename, data source, ...), to focus on the biological reason that makes it a useful morphology to consider.

The core API are the `save`, `load`, and `all` methods to store and retrieve morphologies, and list available morphologies, respectively. There's also the `preload` function which should return a `StoredMorphology` wrapper that can peek into the metadata and when required also load the full morphology data.

PlacementSet

The placement set is one of the main data APIs and contains all placement information on a type of cells. There is always one placement set per cell type. To form subpopulations or groups cells can be labelled with one or more arbitrary labels. The set stores the cell positions, rotations, and morphologies. These 3 sets have particular meaning associated to them in the framework, but additional arbitrary datasets can be stored as well, usually by defining a `Distributor` (see2.5.3, distributor, and Code Snippet 3.5). The rotation and morphology sets are optional, for point neurons, and even the positions themselves are optional; a set of cells can be defined simply by its count, for research contexts that do away with spatial detail, or for other edge cases. These counted cell populations can still have extensive user implied representations during connectivity, or simulation[18].

Placement sets can support the tiling paradigm and may choose to support

[18] Most simulators discard Euclidean space information during simulation either way

storing the data in seperated units that correspond to a tile, called a chunk. This ensures that even when the scale grows beyond the memory capacity of a single computer, that the framework can operate in a distributed cluster by assigning chunks to parallel nodes, and processing the data chunk per chunk. Storage engines can opt-out of chunking by reading/writing all the data to a single canonical chunk, even when the placement context would stipulate elsewise.

Placement sets can be scoped to a set of chunks, and/or filtered by specific cell labels, and/or filter morphologies to certain subregions.

The core API of the `PlacementSet` to be implemented for child components are the data read/write functions (`append_data`, `load_positions`, `load_morphologies`, and `load_additional`), and the context filtering functions (`set_chunk_filter`, `set_label_filter`, `set_morphology_filter`), and auxiliary housekeeping functions e.g. to clear those filters, create, copy, clear, ldots the set.

See 2.7.3 for more info on the data model and role of the placement set.

ConnectivitySet

The connectivity set is largely analogous to the placement set but stores information on relationships between placed entities, usually chemical or electrical synapses, but they may be glial or neurovascular interactions as well.

For the following explanation, remember that the reconstructed volume is split into equal pieces, called *chunks*.

The data is stored as 2 sparse matrices with 6 fixed columns for the pre- and post sorted by blocks of connections between 2 specific chunks, for faster querying of connections between certain regions of interest, and can be duplicated into 2 sparse matrices representing the presynaptic to postsynaptic connectome and the postsynaptic to presynaptic connectome.

The core API that should be supported are the connection functions (`connect` based on a from-PlacementSet and to-PlacementSet, and the `chunk_connect` based on a source and target chunk), and some data loading functions: `load_block_connections` for the connections between a source and target chunk, `load_local_connections` for all the connections either incoming or outgoing from a chunk, and `load_connections` to load all the connections as a `ConnectivitySetIterator`. Connectivity set iterators support iterating over the data from a certain perspective (incoming or outgoing), from/to a certain source/target chunk, or simply loading all of the connections. Storage engines should override the base `load_connections` method if they wish to return a

different `ConnectivitySetIterator`, should they not support some of these features. There are also several housekeeping functions to create, require, move, clear, ... the set.

See 2.7.4 for more info on the data model and role of the connectivity set.

2.5.3 Configuration nodes

`network`, `component` and `package` nodes

The network node contains the size hints, used to roughly contain and scale the topology, the network name, and may contain free metadata about the reconstructed network.

```
{
  "components": ["my_model.placement", "my_model.connectivity"],
  "packages": ["numpy==1.22.0", "bsb-atlas==1.0.0"]
}
```

Code Snippet 2.7: Example of component and package declarations in JSON. The component modules will be imported, and their source code will be bundled with the model reconstructions. The presence of the listed packages will be asserted when using the model in any workflow.

The component and package nodes contain component module and package dependencies respectively (Code Snippet 2.7). The package list should contain all version specifiers of any packages the model depends on. The framework asserts that the environment the model runs in contains the necesary packages or raises a warning.

For most simple component use cases the Python packaging system is an added burden and offers no advantages, and users can instead reference standalone component modules in the component node, which will be bundled with the model.

Storage node

```
{
  "storage": {
    "engine": "hdf5",
    "root": "network.hdf5"
  }
}
```

Code Snippet 2.8: Example of a storage node in JSON. The plugin system will load the `"hdf5"` storage plugin from the `"bsb.storage.engines"` plugin category. The `root` is used by the plugin to uniquely locate/connect with the storage object.

The storage node contains the information required to bootstrap the storage engine (Code Snippet 2.8), it specifies the format and root descriptor. The root descriptor is a format dependent syntax to uniquely identify each storage object. For local engines this would likely be a file path, while for remote engines it could be a connection string, or even multiple pieces of information like a dialect, connection options and so on.

Morphologies block

```
{
  "morphologies": ["morphologies/my_morpho.swc", "nm://Trh-M-200003"]
}
```

Code Snippet 2.9: Example of a simple listing of morphologies with several URI schemes in JSON.

```
morphologies:
  - preprocessed.swc
  - sources: ["unprocessed.swc", "nm://Trh-M-200003"]
    pipeline:
      - center
      - func: "pipelines.process_morpho"
        # User comment: turn on strict processing flag
        parameters: [true]
```

Code Snippet 2.10: Extended example showing how processing pipelines can be set up for data dependencies, including stepwise function calls, with parameters, and user comments for clarifications in YAML. For morphology dependency nodes, the entire `bsb.morphologies` utility library is exposed as exemplified by the `center` shortcut for `bsb.morphologies.Morphology.center`.

In this block users can specify morphologies and their processing pipelines (Code Snippet 2.9). They are later associated to cell types using morphology selector components, that can select morphologies based on either technical or biological qualifiers (such as filename, or morphology size, respectively). When morphologies are generated, these input morphologies can still serve as templates, or be omitted.

The morphologies are loaded during the dependency workflow phase, and con-

verted to an internal format that can store all the required multiscale data, and can utilize the framework's morphology utility library (see 2.8). The user can set up different processing pipelines, for example to sanitize or unify morphologies from different sources to the same reference space, or soma conventions (Code Snippet 2.10).

Topology block

```
partitions:
  VAL:
    - type: allen
    - struct_name: VAL
  layer1:
    - thickness: 100
  layer2:
    - thickness: 200
regions:
  stack:
    type: stack
    children: [VAL, layer1, layer2]
```

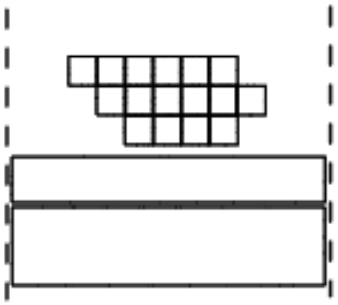


Figure 2.6: Example of some partitions and a region grouping them together, and a diagram of the resulting topology. The layout is obtained because the stack places the elements on top of each other, the VAL partition is retrieved from the Allen Brain Atlas, and the layers occupy their specified thickness along one of the axes, bounded in the other dimensions by the outer perimeter of their parent region, which in this case is determined by the bounding box of the VAL partition.

The `partition` and `region` nodes contain descriptions of geometric shapes that outline partitions of the volume to fill with cells, and how to assemble them into larger regions of the brain, respectively. Out-of-the-box the framework supports geometric primitives like rhomboids, cones, and layers, and irregular shapes with voxelsets, or meshes. See Fig. 2.6 for a configuration example, and section 2.6 for an explanation of the layout process and component interface.

Cell types, placement, and connectivity blocks

These are straightforward declarations of the cell types, and how to place and connect them. Cell type blocks contain cascading information: the information here are mostly hints to help other pieces query information; e.g, you can specify a radius, but it won't necessarily be the radius of every particle, it will instead be passed as a contextual cue for placement algorithms to rely on.

The hard parameters are defined in the placement and connectivity blocks. Some parameters can be omitted if instead they can be found in the cell type. A cell type is effectively a label on some information, permitting various paradigms

and interpretations of the generated network information. The placement and connectivity blocks form the core of the description of the model.

Importantly, the placement and connectivity blocks may contain distributor nodes. Distributors are special jobs that run after each placement or connectivity job to assign additional properties to the block of data that that job has just generated. Any number of properties can be distributed.

The placement block has 2 special types of distributor: the morphology and rotation distributors. They accomodate many of the use-cases and workflows surrounding morphologies. For example, in the simplest case one can disable them, or randomly distribute existing morphologies, or generate new morphologies. The rotation distributor makes it so that rotating a morphology does not require you to create a copy of it, the same morphology can be assigned to different cells with different rotations, for example to reuse and orient template morphologies along the orientation of a layer's surface.

Postprocessing block

Placement and connectivity jobs are scoped and processed independently in parallel. As a consequence no correct global information is available during either placement or connectivity. Each of these phases does however have its own postprocessing phase which is ran when all the information has been generated, and so the global information is known.

These postprocessing blocks invoke their own strategies for example to prune connectivity to a desired global distribution. By default they are not parallelized, but placement, connectivity, and postprocessing component jobs are all scheduled using the same mechanism which can be overridden from the component code.

Simulation block

The simulation block contains simulator specific cell model, connection model, and device configuration. All three types implement a strategy pattern for the instantiation of their respective representations into the simulator environment. These components are decoupled from the framework's cell and connection representations, and are bridged together with the Parameter component, which lets the framework generated cell and connection data be mapped to parametrize the simulator cell and connection models, respectively. This decoupling ensures flexibility to support the simulator's paradigm and user needs, while the framework can generate data without needing any formalisms for what that data is going to be used as.

The Parameter component type is the final piece in a chain of components that

allow the framework not to prescribe anything to the user. The further away the needs of the user from the framework's vision, paradigms and opinions, the more components one will need to implement, but it is highly likely that it will still fit inside the framework's architecture and leverage all of its support and advantages.

The hashing system tells the framework when updates exist for a cached intermediary. E.g., some data sources rely on URI's to locate the resource, and several URI schemes (such as filesystem paths, http, and https URIs) have their own mechanisms of telling whether a resource was updated since the last run. If any of the inputs have changed, or any of the parameters of the pipeline changed, the pipeline will be rerun, otherwise cached results are used by the rest of the workflow.

2.6 The Topology System

Understanding the spatial topology of the network helps the framework make informed decisions on the parallelization of the volume,

2.6.1 Layout

The topology is initialized by a layout process that begins from the root region (a no-op group is added at the root if there are multiple regions or partitions without a parent), and the network size hint is passed to the root region. Based on the size hint and its own behavior, each node proposes a set of operations on its children (`translate`, `scale`, `rotate`), who in turn propose operations on their children.

Each node can also reject the operation, the parent can then continue proposing alternative operations until a suitable layout is found, or ultimately an error is raised if all options are exhausted.

For example, a voxelset will refuse to rotate if the rotation would not result in axis aligned voxels (so only multiples of 90° rotations around an axis), but will accept any scaling or translating proposal.

A spacing element might refuse a translation that would violate the minimum distance between elements. Each topology component should aim to implement as many operations as possible, to increase the likelihood of a succesful layout[19]

[19] So far, the capacity of the layout system seems to be much greater than the complexity of actual model descriptions, who usually just import shapes from brain atlases, or create simplified descriptions such as a cortical stack of layers

2.6.2 Partition interface

To support the parallelization of jobs, the volume is divided into chunks, and each partition component should implement the interface to determine its own axis-aligned bounding box, and be able to intersect itself with an axis-aligned rhomboid.

The bounding boxes are combined to establish the outer perimeter of the model. The default job queueing method for placement strategies will use this boundary to query all the partitions they should place cells in, and queue a job for every chunk for which a non-empty intersection is returned[20].

Splitting the volume into tiles is mostly a load balancing mechanism, so computation of exact intersections or approximations are not required, and cells may be placed at coordinates outside of the chunk, and still be stored in that chunk in the storage object. The only important part is that everything is approximately equally divided, otherwise scaling will detoriate as some workers might be busy much longer than others and stall jobs that depend on it.

Each partition component should also implement as many functions as possible to support placement indications that both the framework and users may use to guess or estimate priors for parameters of algorithms. These are optional, but most notably they are used to estimate cell counts. If the indications are available, then the framework can estimate cell counts that are relative to the count of another cell type, without having to make the placement of one depend on the final result of the other (because the estimation can be used instead).

These indications are generic, so a user can ask for an indication by any name, and the 2 indications the framework relies on are the `"volume"` and `"surface"` area, intersected with a given rhomboid. Placement indicators may return a single value or a voxelset with one value per voxel, for more fine grained results within a larger volume. This is for example used by the brain atlas partitions to return `"density"` indications per voxel in the data source rasterized brain image, which combined with the `"volume"` indication can then be used to calculate the number of cells that should be placed per voxel in the brain atlas structure.

2.6.3 Brain atlas integration

Since much multiscale data on the brain is provided to the community in the form of brain atlas images, the framework provides an integration with the Allen Brain Atlas, and abstractions to deal with rasterized brain image data.

[20] This `queue` method can be overridden on all of the parallelizable component types for parallelization more suitable to the nature of the algorithm

First, let's understand the data abstraction: A dedicated class called the `bsb.voxels.VoxelSet` lets users define 3 types of sets of voxels: a set of voxels identified by their index coordinates on a regular 3D grid (i.e., "pixel" coordinates in an image, but in 3D), called a *regular* voxelset, a set of voxels identified by spatial coordinates and a shared size, called an *irregular voxelset*, or individual sizes. Either set can be cubic or rhomboid, where atlas data is usually cubic. Each voxelset can also have any number of data columns associated to it, which can either be a value per voxel, or an object per voxel (using numpy's `object` dtype).

The topology system supports 2 types of irregularly shaped shapes out-of-the-box: meshes (using `pymesh`[21]), or voxelsets (usually loaded from NRRD files, with `pynrrd`[22]). Brain atlases usually provide rasterized images of the brain with certain resolutions per voxel, usually ranging from 100 micron to 10 micron.

To process the Allen Brain Atlas data into a framework-useful form the `AllenStructure` component is provided which turns the brain images into a voxelset scopes to an Allen Structure that the user specifies by a `struct_name` or `struct_id`. The identifier has to match the name, acronym, or ID of the structure in the Allen Atlas Ontology, which is a tree of structures. Every structure may contain child structures, and an ontology image can be downloaded specific to each version and resolution of the atlas, where each voxel is labelled with the ID of the terminal child structure it belongs to. To obtain the mask of a non-terminal node all voxels labelled with the id of any of its terminal child nodes must be retrieved. Then, the user may associate any number of `source` NRRD files to which the mask will be applied, and whose data will be loaded as a data column of the voxelset (see top snippet of Code Snippet S7.6).

These elements allow a user to effortlessly use shapes defined by the atlas ontology, and populate the shape with atlas data, that can then be used as the basis for placement and connectivity algorithms.

2.7 The Data Generation System

2.7.1 Parallel scheduling

All the generation phases of the framework described below use the same parallel scheduling mechanism. A `JobPool` component allows jobs to be `submit`ted, and `execute`d. Jobs have an id, a function to execute (i.e., the job to do), and may depend on other jobs, which means they will be scheduled after it, and may request its return value.

[21] `https://pymesh.readthedocs.io/en/latest/`
[22] `https://pynrrd.readthedocs.io/en/stable/`

The job pool requires one manager, and can manage many workers. The framework provided job pool uses a pool of MPI processes, and a thread on one of the worker processes fulfills the manager role (otherwise one worker would be lost while the manager mostly idles).

The framework initiates job scheduling by opening the pool, and asking all relevant components to `queue` their jobs, components may specify that they would like to depend on the jobs of another component, and when queueing is complete the framework executes 2 tasks: one, it resolves the component dependencies to job dependencies (with detection of circular dependency loops), and two, it creates a description of the planned run, so that job execution may error out or be interrupted, and later errored, incomplete, or skipped jobs may be retried/resumed[23].

Another important aspect of parallelization is parallel resource access. The framework provides a locking component to storage engines for multiple read and single write (MRSW[24]) access across MPI processes, so that all user code can transparently call read and write operations without worrying about parallel access conflicts.

2.7.2 Data dependencies & pipelines

Any node can mark itself with a `file` descriptor, which will treat the provided value as a data dependency, retrieve it, cache it, and bundle it with any generated outputs. Dedicated nodes exist within the framework for most use-cases, but users are free to define their own component's data dependencies.

Most dependency nodes support a `pipeline` attribute, a list of `Operation`s, which in turn are composed of a reference to an importable function, and a parameter list. Putting these elements together, users can specify a stepwise data processing pipeline, which will import the function of each step, and call it with the input object, the arguments, and pass the output to the next step (Fig. 2.7, code example in Code Snippet S7.6).

Each dependency component implements a `load_objects` function which must parse the data into a list of values (or `load_object` as a shorthand for a single value list). The default `queue` method will schedule 1 parallel job per input value and apply the processing steps to the input value. Users can override the `queue` method to implement better suited parallelization schemes.

The `CodeDependencyNode` can be used for code dependencies, and will automat-

[23] It occurred that entire lengthy workflows were lost because of a single edge-case error, so this is a nifty feature.

[24] `https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock`

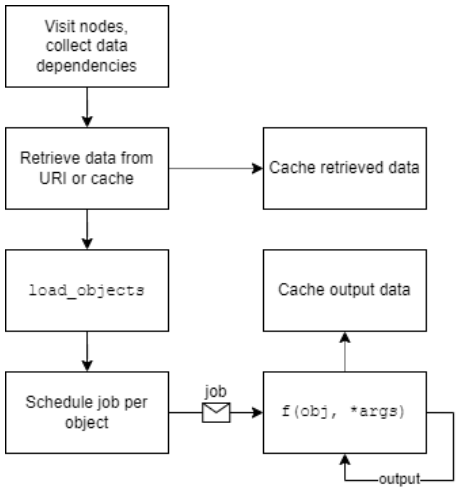


Figure 2.7: Flowchart of data pipelines

ically include the target module as a Python file in the storage's `FileStore`, and load the module from there, as the module it was initially loaded as. e.g., you can load a local file `"my_brain/my_conn.py"` as the `my_brain.my_conn` module, and then relocate the model reconstruction anywhere and the module will still be loaded from the file store as the `my_brain.my_conn` module[25].

2.7.3 Placement

The placement system generates information for multiscale units in the model. It begins by collecting the placement strategy components, the most common purpose of a placement strategy component is to, in some way, fill or associate a collection of partitions with cell types.

The setup for this phase is relatively simple: Each component places jobs in the job pool with a `queue` method, which by default looks for chunks occupied by any of the given partitions, and schedules a job per occupied chunk.

Then, each job calls the `place` method of the strategy component and passes it the chunk, and a context object that can answer questions like: how many cells should be placed? Which morphologies should be used? To simplify queries for users that depend on many configurable factors. The placement strategy is expected to call the `self.place_cells` function[26] with the generated positions (or counts) of the different cell types. The job continues by calling the

[25] This mechanism is limited, as any dependencies or relative imports won't be included, so treat them as "standalone modules" and list any dependencies they have in the `packages` list.

[26] As little as possible, with as much data per call as possible, because it requests a write lock and writes to storage, which incurs overhead.

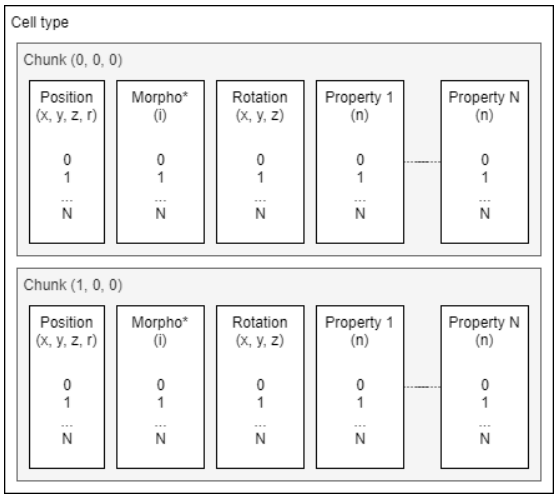


Figure 2.8: The placement data model. Each inner white square denotes a dataset object, with in parenthesis the data per record in the set. All datasets are parallel to each other, e.g., position 1 is associated to rotation 1. (*) A `MorphologySet`: A set of morphology names, and a multiset containing the index of the associated morphology name per cell. The morphologies themselves are stored separately by the `MorphologyRepository`. The rotations are similarly wrapped by a `RotationSet` which lets users use the convenient `scipy.Rotation` per item in the dataset.

`distribute` method of the configured distributors, and passes them the same context object, and the generated positions. Each distributor is supposed to return one value per generated position.

The user returns a dataset with xyz + radius from the placement strategy component, and /or a `MorphologySet` (which maps cells to a set of morphologies by associating each cell to the index of the morphology in the set), and/or a `RotationSet` with the 3D Euler rotation in degrees, and/or additional vectors from the distributors (see Fig. 2.8) which are handed to the storage engine and contained within a compound dataset called the `PlacementSet`.

`PlacementSet`s can read/write the stored data, and can apply several scopes: the data can be restricted to specific chunks, to specifically labelled cells, and the morphologies can be restricted to specifically labelled points.

2.7.4 Connectivity

Connectivity strategy components consist of 2 main nodes called `HemitypeNode`s: one for the presynaptic specification and one for the postsynaptic one (N.B.: components can be extended). Each hemitype specifies a list of cell types, and optionally a list of target cell labels, and target morphology labels.

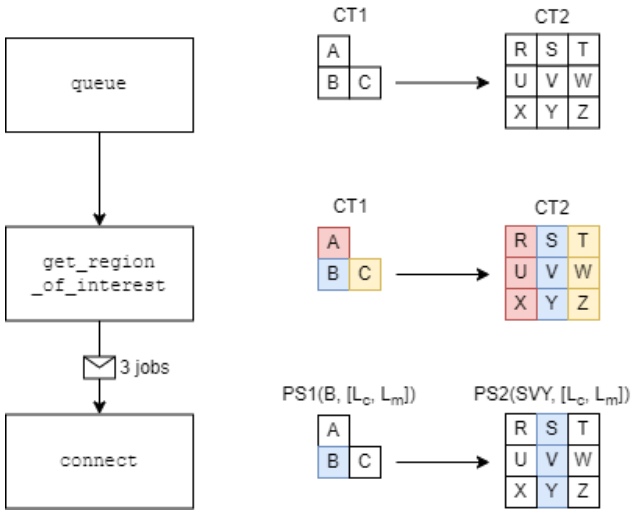


Figure 2.9: Schematic representation of the scoped job submission based on regions of interest. CT: cell type. PS: `PlacementSet`, between parenthesis in order: chunk scope, cell label scope (Lc), morphology label scope (Lm)

The phase begins once more by calling the `queue` method of the components.

The default implementation relies on calling the `get_region_of_interest` method[27] for each presynaptic chunk occupied by any of the presynaptic cell types. The region of interest (RoI) method takes a set of presynaptic chunks as argument, and must return the set of postsynaptic chunks whose placement information is relevant to it (Fig. 2.9). For each pair of RoIs a job is submitted. Each submitted job then calls the `connect` method of the strategy component with 2 arguments: `pre` and `post`[28] representing mini-worlds of the placement information scoped to the selected RoIs and labels of the job.

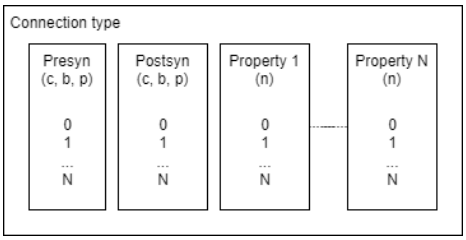


Figure 2.10: The connectivity data model. Parallel arrays represent the presynaptic site and postsynaptic site, and additional properties. Each site is described by 3 coordinates all scoped to the `PlacementSet` that produced it: the cell id in the set, the branch id in the filtered morphology, and the point id on the branch. This information is only valid within the context of the current job, and is decoded by the storage engine, which also recodes it whenever the framework or user loads the data in another scope.

The user can then use the scoped PlacementSets to `load_positions`, `load_morphologies`, or `load_additional(name)` information on the cells, and generate connections between them by calling `self.connect_cells` with 2 3xN datasets with the 3 columns being the cell id, the id of the branch in the morphology, and the id of the point on the branch for both the presynaptic and postsynaptic site. The same arguments and the generated connections are then passed to the distributors to generate additional information.

2.7.5 Data storage

The framework chooses to defer the reading, writing, encoding, and decoding of data to a collection of components, provided by the storage engine, so that through a rather simple interface of parallel arrays the storage engine can store and load it in various ways. No standards were found that could support the wide range of multiscale information we generate, and an in-house format was developed.

[27] Whose default implementation is to select everything as the region of interest

[28] So, only by overriding the `queue` method, to have it schedule a different type of job, could a user break away from the bipartite synapse, which is not great user experience. A mixin class could be provided for the desired behaviour, but it does not exist yet.

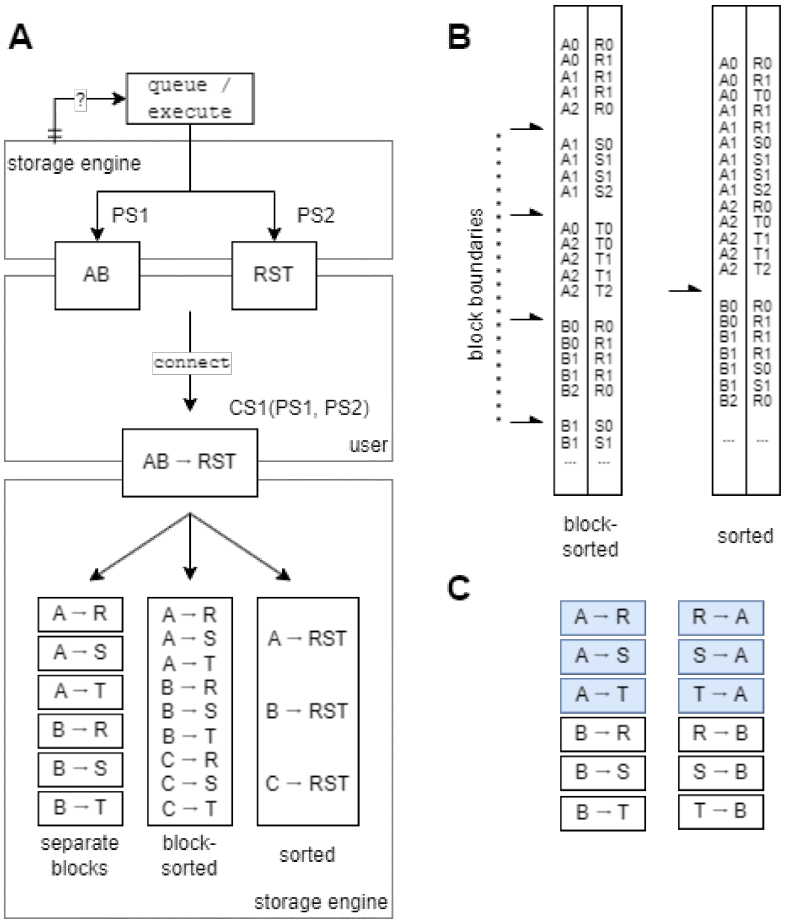


Figure 2.11: **A)** Dependency of the connectivity flow on the storage engine: The connectivity strategy `queue`s parallel jobs by querying the `PlacementSet` (PS) , the job creates the scoped PS1 and PS2 and calls the `connect` function. The user code retrieves data from PS1 and PS2 to produce the output result, oblivious to any scope. The storage engine assembles the `ConnectivitySet` (CS) based on the user output and PS1 & PS2 information (CS(PS1, PS2)). In the bottom part some example data layouts are decoded from the user output. **B)** Sort-order examples of possible demultiplexing of user output to either globally sorted (sorted) or locally sorted (block-sorted) data. **C)** Example of duplicated storage: No sort order can sort both paired columns simultaneously, but by duplicating data, this time sorted by the second column, both the incoming and outgoing connections can be retrieved from a sorted dataset.

To decouple the framework – and more importantly, user code – from storage assumptions, we opted for a simple user interface of parallel arrays (Fig. 2.8 & Fig. 2.10): the user code always operates on dataset objects that either directly are, or facilitate access to, a contiguous vector or matrix, as if the data within the job context was the entirety of data that exists. The user then creates their own dataset, referencing the input data by their index in the dataset, and the storage engine decodes the result.

It's the task of the storage engine both to encode the datasets that are passed to the user, and to decode the returned values from the user code. Note in Fig. 2.11A how the entire flow depends on the PlacementSet, and is stored in the ConnectivitySet, both provided by the storage engine. This double deference to the storage engine for both the inputs and outputs of user code should allow the user to operate in a consistent simplified space[29], while the storage engine can arrange more complicated optimized data layouts to store the information according to the format. Different formats will surely have different performance and scaling in different situations.

The only coupling between the framework and the storage that could cause friction are the chunks, since they are a storage feature closely intertwined to the parallelization scheme. Users that use storage engines that do not support them, would have to either lose some degree of parallelization (not all: each declared component will still schedule at least 1 parallel job, or might not use rely on chunks at all for their parallelization), or need to rethink and override the parallelization logic in many of the components they use.

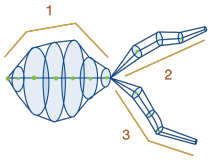
For use within our research group we chose for a duplicated block-sorted layout in the BSB-HDF5 engine (Fig. 2.11A& B), so that during simulator network setup, we needn't iterate the entire connectome. Since the connectivity data is paired, no matter how the cell pairs are sorted either the source or target cells remain unsorted and need to be iterated through entirely to find all the incoming or outgoing connections of a cell owned by the current machine (Fig. 2.11B, connections belonging to cell R0 are spread throughout the dataset). By duplicating they can be sorted both ways, and both the incoming and outgoing connections of a cell can be retrieved from a subset of the connectome (Fig. 2.11C). By using the duplicated block-sorted layout and storing the block boundaries in a hashmap keyed by the chunk, all relevant data could be easily sliced out of the HDF5 datasets. No performance analysis was performed as the network setup time was negligible compared to network simulation time either way[30], but significant reduction in the peak memory consumption and time spent in framework code was observed.

[29]The placement phase does not have this complication: being the first phase, there is no dependency on stored storage-format dependent information yet.

[30]The doctoral scholarship has a fixed duration, and getting the framework into a usable state was a bigger priority.

2.8 Morphologies

Cell morphologies are ubiquitous in multicompartmental modeling, yet no standardized format exists[31]. There were no existing formats that supported spatial coordinates, multiple labels, and arbitrary user properties per point. Therefor we created a data model similar to our others, with parallel arrays that can be represented by a single data matrix, and a directed acyclic graph (DAG).



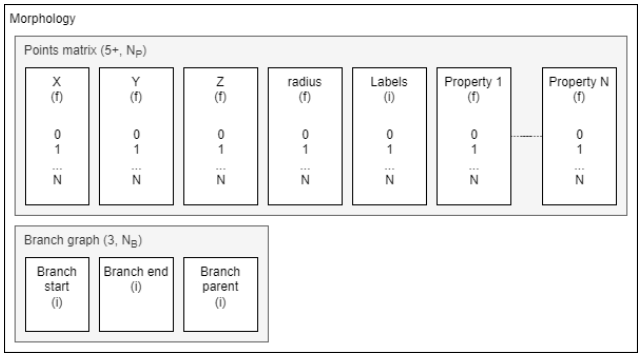


Figure 2.12: Left: schematic representation of a morphology, formed by a series of conical frustrums, described by a series of points with radii. Green lines: line segments inferred between subsequent points of a branch; green dots: points; red numbers: branch identifiers; orange: indicators showing the aproximate extent of the branches in the morphology. Right: Data model of a morphology showing the parallel arrays of data points, and the branch graph, which associates data points together into branches with child-parent relationships.

A morphology in the framework is a matrix of data points with the following columns: 3 spatial coordinates X, Y, and Z, a radius, labels (encoded by a single integer that refers to a map of lists of labels), and any amount of properties (Fig. 2.12). Additionally, a branch graph keeps track of the beginning and end points of the series of line segments to be formed between subsequent points, and to which parent branch it should be (electrically) connected (no line segment is inferred between the head of the child and tail of the parent, gaps are allowed). Branches connected to a branch with id -1 have no parent branch and are *roots* of the morphology. There is no exception for the soma, which can be approximated with a bulbous branch. The branches are laid out in the matrix in depth-first order, so the parent branch always precedes the child branch during iterative algorithms.

This data model allows the framework to load and operate on a morphology as a single contiguous array of data, using `numpy`'s multidimensional arrays. The `bsb.morphologies.Morphology` class wraps the data matrix, and creates

[31] SWC is barely specified, and multiple conflicting conventions exist. NeuroLucida is proprietary

a graph of `bsb.morphologies.Branch` objects. By relying on `numpy`'s array interface, we can create memory views into the main matrix, allowing direct modification of the array while the user can operate on a `numpy` array that appears to contain only the branch's data. Additional methods on the `Branch` class facilitate morphology specific operations and metrics. Any operations that would mutate the structure of the branch graph or would introduce or remove points require the morphology to be laid out into a new data matrix; this is usually avoided internally by the `Branch` class, and only when certain algorithms, like saving the morphology, require a contiguous memory layout is a new matrix allocated: several operations were more than a thousand times faster this way.

2.8.1 Utility library

The morphology utility library helps users perform various operations on morphologies, and directed acyclic graphs in general. Several geometrical transformations are supported[32], such a translation, rotation, scaling, centering, ... Various selection and masking operations are supported to retrieve operate on portions of the morphology. Depth- and breadth-first child branch and point iterator support iterative user algorithms.

Another aspect of the utility library is to provide metrics, such as total branch length, branch tortuosity, or to find special points like all terminal points/branches, all roots, the center of mass, and many more[33].

Interoperability is provided with Arbor's [93] morphologies, MorphIO[34], and DAG packages like networkx[35].

2.9 The Plugin System

The preferred way for third parties to provide components and framework extensions to users is through plugins. First of all, providing components through plugins has several advantages over standalone component modules:

1. Plugins are installed as Python packages so they are (usually semantically[36]) versioned, and can install (binary) dependencies.

2. They can perform relative imports for better code organization into multiple files.

[32] `https://bsb.readthedocs.io/en/latest/morphologies/intro.html#subtree-transformations`

[33] See most utility functions here: `https://bsb.readthedocs.io/en/latest/bsb/bsb.morphologies.html`

[34] `https://morphio.readthedocs.io/en/latest/`

[35] `https://networkx.org/`

[36] `https://semver.org/`

3. They are automatically discovered

4. They can be lazy-loaded (known to the framework, but only imported when used)

5. They can lazily extend classmaps

This list is also a summary of the drawbacks of modules the user doesn't package and merely lists in the `components` of the model. Secondly, many framework functions can be extended in ways that do not belong in the model description.

```
[project.entry-points."bsb.config.parsers"]
json = "bsb_json.parser"

[project.entry-points."bsb.config.templates"]
json_templates = "bsb_json.templates"
```

Code Snippet 2.11: Example of entry point metadata in a plugin's `pyproject.toml` configuration file, part of the Python packaging system. The entry points expose advertised objects to the framework.

The plugin system uses Python *entry points* (Code Snippet 2.11),which are package metadata specifiers that can be scraped using Python's `importlib.metadata` API . Each entry point belongs to a *group*, and the framework categorizes plugins by using different groups, e.g., the available storage engines are discovered with `importlib.metadata.entry_points.get("bsb.storage.engines")`. Entry points return a list of *advertised objects*. Each plugin category has different expectations on what the advertised object should be and each package can specify as many entry points as it wants.

2.9.1 Plugin categories

Component plugins

```
[project.entry-points."bsb.components"]
shapes = "my_plugin.shapes"
```

```
# Eagerly loaded registered classmap component
class Icosahedron(Partition, classmap_entry="icosahedron"):
    pass

# Classmap extension
shapes = {
    "bsb.topology.partition.Partition": {
        "ellipsoid": "my_plugin.ellipsoid.Ellipsoid"
    }
}
```

Code Snippet 2.12: Example of registration mechanisms for component plugins. Top: plugin package metadata. Middle: eagerly loaded registered classmap component. Bottom: Object advertised in package metadata, extending the `Partition`'s classmap with `"ellipsoid"` aliassing `"my_plugin.ellipsoid.Ellipsoid"`

Installation of a plugin component suffices for them to be referenced by their module name in the configuration, but the `"bsb.components"` category is provided so that objects can be advertised to extend classmaps. When the package of Code Snippet 2.12 is installed, any user could use `"ellipsoid"` to lazy-load the `my_plugin.ellipsoid.Ellipsoid` partition.

Storage engines

Storage engine plugins (`"bsb.storage.engines"`) should advertise a module that contains implementations of the classes in `bsb.storage.interfaces`; any variable that inherits from any of the classes there will be considered a supported feature. The `Engine`, `StorageNode`, `PlacementSet`, and `ConnectivitySet` are required features.

Simulation backends

Simulation backend plugins (`"bsb.simulation.adapters"`) should advertise an object with 2 attributes: a `Simulation` node, and a `SimulatorAdapter`. The simulation node should fulfill three slots: the `CellModel`, `ConnectionModel`, and `Device` who convert placement data, connectivity data, and the experimental setup into simulator-specific elements, respectively. The framework provides base abstractions and a generic simulator adapter flow that can be inherited from and followed for easy adaptation of new simulator backends, but ultimately the plugin developer is free to manage the transfer entirely by themselves, as they are simply passed the configured `Simulation` node that the user wishes to simulate, and the reconstructed scaffold network.

Configuration parsers

Parsers (`"bsb.config.parsers"`) advertise a class with 2 methods. `parse` and `generate` that need to transform the configuration content or path-like object to the configuration tree (N.B.: not the component tree, just a structure of Python lists, dicts and values that the root node constructor then converts into the component tree), and vice-versa.

They can also contain a class attribute with a list of extensions they preferably parse.

Configuration templates

Templates (`"bsb.config.templates"`) should advertise a list of paths to directories that contain configuration templates. The filenames can then be used with the `bsb make-config` command to create new configuration files.

URI schemes

URI schemes (`"bsb.storage.schemes"`) should advertise classes that inherit from the `bsb.storage.UriScheme` class. Through the `find`, `should_update`, and `get_content` functions the retrieval, caching, and reading of the data objects can be controlled; while `provide_stream` and `get_local_path` provide alternatives to `get_content` to read the object as a buffer, or from a local path respectively.

Usually most of the interface can be inherited from the base `UriScheme` class through a `super` call, but this should be confirmed explicitly.

Command line commands

Templates (`"bsb.commands"`) should advertise classes that inherit from the `bsb.commands.BsbCommand` class. They contain a `handle` function that executes the command, and `add_to_parser` to add the command to a Python standard lib `argparse` argument parser.

2.9.2 Listeners

Listeners (`"bsb.listeners"`) should advertise functions that process events the framework emits. These can then be used in user's project settings to configure where and how they wish to see framework events. Events include event hooks, logging, progress reporting, warnings, and captured stdout output.

2.9.3 Framework options

Options (`"bsb.options"`) should advertise objects that inherit from the `bsb.option.Option` class. Options are a descriptor based system, where each

descriptor describes a different source from where the option can be get/set (e.g., command line arguments, scripts, project settings, environment variables, ...). Options are used to change the way the framework (or plugins) behave, but should not be used in any way that affects a user's model.

2.9.4 Auditing rules

Auditing rules (`"bsb.audits"`) should advertise functions that take the project root directory as an argument, and should return a list of lists with the inner list containing a name for the violation, a severity (`"info"`, `"warn"`, `"error"`), the source code file, and finally line number. They will be visited during the `bsb audit` command to report on quality/linting problems.

Chapter 3

Methodology: Modelling Workflow using the Brain Scaffold Builder

3.1 Project setup

Create a new project using the CLI command `bsb new` and fill out the project settings.

Take a look at the project settings in `pyproject.toml`, which for the time being only includes the default configuration file to use when none are passed to the CLI.

```
name: My Model
storage:
    engine: hdf5
    root: network.hdf5
network: {x: 100, y: 100, z: 100}
```

Code Snippet 3.1: Example of the pre-workflow settings of a model configuration

After chosing a configuration template, fill out the model's configuration settings such as the name, size hint, and the storage node (Code Snippet 3.1).

Place any code or files you need for the remainder of the workflow inside of the

project folder.

3.2 Data sourcing and preprocessing

The workflow begins with locating and sourcing the required datasets. Incorporate the datasets with a `file://` or `https://` URI, or set up components for your own URI schemes:

1. Implement or inherit `find`, `should_update` and `get_content`, `get_meta`, `provide_stream`, `get_local_path` from `bsb.storage.UriScheme`: although they're all part of the required interface, most can be skipped and inherited after thoughtful review of the parent code.

2. Register the class under the `"bsb.storage.schemes"` category, keyed by the scheme identifier (e.g., to process "nm://some-uri-path", register it as "nm").

Once the plugin is registered, any URI with the specified protocol will be processed by it.

```
@config.node
class MyDataNode(PipelineMixin, bsb.placement.PlacementStrategy):
  my_data = config.file()

  def load_object(self):
       return self.pipe(self.file)
```

Code Snippet 3.2: Example of a user defined data dependency, with a pipeline mixed in. The pipeline mixin adds a `pipeline` attribute and provides the `pipe` method to apply it.

You can incorporate your data dependencies wherever you need them by using the `bsb.config.file` descriptor factory to create a configuration attribute on your node class (Code Snippet 3.2).

The framework offers the opportunity not only to explicitly locate, but also modify your data with a pipeline. Set up small pure functions that modify the data step by step, and parametrize them well. This will yield a neatly parametrized processing pipeline that can be repeated whenever the input source changes, when parameters must be changed, or whenever someone else needs to rerun your workflow.

You can include the `bsb.mixins.PipelineMixin` class to add a `pipeline` at-

tribute, and `pipe` method that processes the pipeline (Code Snippet 3.2). Reference target pipeline functions (including your own) by their importable name (module path + variable name, e.g. `"my_module.func1"`).

Morphologies are treated as a data dependency as well: you can use either files or more specialized morphology URI schemes (like the NeuroMorpho URI scheme). `MorphologyDependencyNode`s can access about 50 different operations you can perform on branches, subtrees or the entire morphology, from the `bsb.morphologies.Morphology` class directly by their function name.

3.3 Declare network topology

In this stage of the workflow you can declare a hierarchical tree of simple building blocks such as rhomboids, cones, axis-aligned layers, or define arbitrary 3D shapes in the form of either sets of voxels, or meshes (Fig. 2.6). You can also integrate with brain atlases: partitions can for example be defined by Allen Brain Atlas structure IDs or names, and can then later on be filled with cells according to the density of that cell type in each voxel of the atlas (Code Snippet S7.6).

3.4 Determine cell types, placement and connectivity strategies

```
@config.node
class MyPlacement(PlacementStrategy):
    extra_label = config.attr(type=str, default="extra")
    def place(self, chunk, indicators):
        for ct, indicator in zip(self.cell_types, indicators):
            self.independent_place(chunk, ct, indicator)

    def independent_place(chunk, ct, indicator):
        count = indicator.guess("count", chunk)
        pos = self.scaffold.random.integers(
            high=10, size=(count, 3)
        ) / 10 * chunk.dimensions + chunk.ldc
        extra_data = self.scaffold.random.integers(len(pos))
        self.place_cells(
            chunk,
            pos,
            additional={
                self.extra_label: extra_data
            }
        )

@config.node
class MyConns(ConnectionStrategy):
    def connect(self, pre, post):
        for ps_pre in pre:
            for ps_post in post:
                self.independent_connect(ps_pre, ps_post)

    def independent_connect(self, pre, post):
        conns_pre = np.fill(
            (
                self.scaffold.random.integers(
                    size=(1,), high=min(len(pre), len(post))
                )[0],
                3
            ), -1
        )
        conns_post = conns_pre.copy()
        conns_pre[:, 0] = self.scaffold.random.integers(
            high=len(pre), size=(len(conns_pre),)
        )
        conns_post[:, 0] = self.scaffold.random.integers(
            high=len(post), size=(len(conns_pre),)
        )
        self.connect_cells(pre, post, conns_pre, conns_post)
```

Code Snippet 3.3: Example of a custom placement and connectivity component that form random positions, and random connections. The placement component also associates an additional random dataset whose name the user can configure. Both the placement and connectivity component process multiple input components separate from each other, but components combine the given inputs any way they see fit, i.e., to unpack the combinations into `independent_place` and `independent_connect` is a component choice.

```yaml
components: [example.py]
cell_types:
    cell_type_A:
        spatial: {count: 100, radius: 1}
partitions:
    layer_A:
        thickness: 100
placement:
    place_all:
        strategy: example.MyPlacement
        cell_types: [cell_type_A]
        partitions: [layer_A]
connectivity:
    connect_all:
        strategy: example.MyConnections
        presynaptic:
            cell_types: [cell_type_A]
        postsynaptic:
            cell_types: [cell_type_A]
```

Code Snippet 3.4: Example of a minimal reconstructive workflow, connecting 100 randomly placed type A cells to themselves randomly. Assuming Code Snippet 3.3 is available in the root project folder as `"example.py"`.

With a topology in place, all the elements to be considered in the model can be defined. These can span from regular neuronal elements, to glial, vasculature, or afferent fibers, and synapses. The cell types can have some indicative properties inherent to them, such as soma radius and a wide assortment of count estimation mechanisms (a fixed count, fixed density, count/density relative to another type, ...), but most model specific properties are assigned later.

Now define the placement strategies: declare which algorithm should be used to fill which regions or partitions with which cell types. This is followed by the connection strategies: declare which algorithm should be used to form presynaptic and postsynaptic location pairs on presynaptic and postsynaptic cells.

Code Snippet 3.4 shows a minimal setup, add more declarations to arrive at a full model description, and add more strategy components based on your needs (Code Snippet 3.3).

3.4.1 Distribute additional properties

```
placement:
    place_all:
        strategy: example.MyPlacement
        cell_types: [cell_type_A]
        partitions: [layer_A]
        distribute:
            conductance:
                strategy: my_code.SomaConductanceDistributor
                depends_on: morphologies
```

```
class SomaConductanceDistributor:
    factor = config.attr(type=types.float(min=0), required=True)

    def distribute(self, positions, context):
        m_set = context.morphologies
        return [
            m.select(["soma"]).radius.average() * self.factor
            for m in m_set
        ]
```

Code Snippet 3.5: Example of a distributor that calculates a conductance value from the average radius of all the morphology points tagged with a `"soma"` label. It uses a dependency on the implicit `"morphologies"` distributor, and uses the dependency to retrieve the morphologies from the context passed to the distributor.

If you need more than the regular data (positions, morphologies, rotations for placement; or pre/post-synaptic site for connections) then assign distributors to the relevant strategies. Placement and connectivity strategies only form the nodes and edges of the network, by declaring additional distributors per strategy you can calculate unique properties per edge or node, which you can later use to modify or create simulator elements (see Code Snippet 3.8).

3.5 Generate model samples

With all of the declarations in place, you should have an entire description of your model structure. Now any number of samples can be generated with the CLI command `bsb compile`. They will be stored in network storage files of your chosen format.

The framework will parallelize using a Message Passing Interface (MPI) implementation if one is available and set up for the process:

```
mpirun -n 4 bsb compile
```

Code Snippet 3.6: Command to run the compilation workflow with 4 parallel MPI processes.

At this point a lot of interesting data can already have been generated. e.g., using a realistic morphology generator that respects the topology, atlas cell density data for the placement, and a morphology intersection algorithm for the connectivity, a plausible connectome can be extracted. Most of the time the bottom-up workflow doesn't end here though and we continue on with model simulation.

This stage forms the first iterative checkpoint, where after analysis of the generated samples, parameters can be tuned, or modelling approaches can be improved/replaced if the obtained output is not satisfactory.

3.6 Describe cell and connection models

We now focus on how to represent this data in a simulator backend. The cell and connection models form components around the representations of these concepts in the simulator backend.

When declaring simulation components the workflow diverges depending on the chosen simulation paradigm. The core supported paradigms are multicompartmental and point neurons. There's no requirements that cells which had morphologies assigned to them during reconstruction need to be represented as multicompartmental neurons: if not interested in that level of detail you may discard it during simulation, but retain perhaps a more detailed connectome than one could have established without relying on morphologies.

3.6.1 Multicompartmental workflow

The framework enforces an architecture in the form of the cell model, connection model and device components, and provides a default component which uses the *arborize*[1] library as the common description format of models for both the NEURON and Arbor simulator. Code Snippet S7.7 contains an example of an arborized cell model description.

Using `arborize`, each model definition consists of a set of rules to assign labels

[1] `https://arborize.readthedocs.io/en/latest/`

to the pieces of the morphology, and a set of mechanisms and parameters that should be applied to each label. These definitions can then combined with a morphology schema and constructed in either of the simulator backends, with automatic integration with the framework.

The *definition* defines cable types with the cable properties, ionic properties, density mechanisms, and available synapses for pieces of cable marked with certain labels. These descriptions can then later be combined with a *schema*, which is a blueprint for a specific morphology, painted with labels and point-specific cable or mechanism parameter changes, and can be built into model specific instances by a *builder* . E.g., for an Arbor workflow in the BSB, the user's definition would be combined with a schema constructed from the generated data by the BSB for each cell, and built into a model instance by arborize's Arbor builder.

Alternatively, should the user want to plug in arbitrary models that don't fit the arborize format, they can still write their own cell model component and instantiate their own models inside of the component.

The connection models are usually less involved; the default component works with a transceiver paradigm that places a transmitter presynaptically and a receiver postsynaptically. They rely on an interface function of the cell model to find the simulator object that corresponds to the pre and postsynaptic pieces of cable. Both chemical (i.e., event-driven) and electrical (i.e., continuous transfer) synapses can be created.

```yaml
simulations:
    simA:
        simulator: arbor
        temperature: 32
        duration: 1000
        resolution: 0.1
        cell_models:
            cell_type_A:
                model: my_models.cellA
        connection_models:
            cell_type_A_to_cell_type_A:
                synapses: [AMPA, NMDA]
        devices:
            pg:
                device: poisson_generator
                interval: 0.1
                targetting:
                    strategy: by_sphere
                    origin: [50, 50, 50]
                    radius: 5
                    locations:
                        strategy: branch_location
                        label: soma
                        x: 0.5
            spikes:
                device: spike_recorder
                targetting:
                    strategy: all
```

Code Snippet 3.7: Example configuration for an Arbor [93] simulation continued from Code Snippet 3.4. Thanks to the `arborize` package the simulation config and models are compatible with the NEURON simulator as well. The relationship between both the cell model and `PlacementSet` and the connection model and the `ConnectivitySet` are implied by name, but can be specified explicitly in unconventional cases.

In Code Snippet 3.7, the framework passes the cell's morphology, and any constant/parameter overrides to *arborize*'s cell builder, and stores the return value in the population. The connection model component cooperates with the cell model component to create the specified synapses (i.e., the used cell model and connection model should be compatible[2]) on all locations described by the `ConnectivitySet`.

[2]The arborized component supports a simple interface based on a map of *locations* (branch and point id on the BSB morphology), to a descriptor that can retrieve the piece of cable, mechanisms, and synapses in NEURON

This is where previously generated user properties can be mapped onto specific cable properties, mechanism properties, or synaptic properties on each individual cell or connection:

```
cell_models:
    cell_type_A:
        model: my_models.cellA
        parameters:
            conductance:
                type: ArborizedCellDataParameter
                source: conductance
                target: soma.mechanisms.Nav1_6.gmax
```

Code Snippet 3.8: Example application of a `ArborizedCellDataParameter` mapping the additional `"conductance"` dataset to the `"gmax"` parameter of the `"Nav1_6"` mechanism in the `"soma"` region of the *arborized* model description.

3.6.2 Point neuron workflow

```yaml
simulations:
    simA:
        simulator: nest
        duration: 1000
        resolution: 0.1
        cell_models:
            cell_type_A:
                model: iaf_psc_alpha
                constants:
                    ...
        connection_models:
            cell_type_A_to_cell_type_A:
                model: static_synapse
                constants:
                    weight: 1
                    delay: 0.1
        devices:
            pg:
                model: poisson_generator
                rate: 10
                targetting:
                    strategy: by_sphere
                    origin: [50, 50, 50]
                    radius: 5
            spike_recorder:
                device: spike_recorder
                targetting:
                    strategy: all
```

Code Snippet 3.9: Example simulator configuration for the NEST simulation adapter. The ellipsis should be replaced with a valid set of parameters expected by the NEST model.

The point neuron modelling approach is far simpler and the default set of components exposes the NEST API quite directly: neuron, synapse, and device models can be taken directly from the NEST model directory[3] and their parameters set through the `constants` attribute, or using a `Parameter` component.

3.7 Run simulations, validate, iterate

In the final stage of the workflow the framework simulates the reconstructed network according to the configured representations of the data in the sim-

[3] `https://nest-simulator.readthedocs.io/en/stable/models/index.html`

ulator, and manages the entire process. The simulated data the devices create can be stored in any standardized format that Neo supports.

```
bsb simulate network.hdf5 simA
mpirun -n 4 bsb simulate network.hdf5 simA
```

Code Snippet 3.10: CLI commands to run a serial simulation (top) and parallel simulation on 4 nodes (bottom).

The framework opts to end the workflow here as a wide landscape of neuroscience data analysis tools already exist, and can be combined with Data Version Control systems to manage handling and analysis of the generated data.

Chapter 4

Cerebellar Cortex Microcircuit Model

The results presented in this chapter come from De Schepper et al. [94]

4.1 Abstract

The cerebellar network is renowned for its regular architecture that has inspired foundational computational theories. However, the relationship between circuit structure, function and dynamics remains elusive. To tackle the issue, we developed an advanced computational modeling framework that allows us to reconstruct and simulate the structure and function of the mouse cerebellar cortex using morphologically realistic multi-compartmental neuron models. The cerebellar connectome is generated through appropriate connection rules, unifying a collection of scattered experimental data into a coherent construct and providing a new model-based ground-truth about circuit organization. Naturalistic background and sensory-burst stimulation are used for functional validation against recordings in vivo, monitoring the impact of cellular mechanisms on signal propagation, inhibitory control, and long-term synaptic plasticity. Our simulations show how mossy fibers entrain the local neuronal microcircuit, boosting the formation of columns of activity travelling from the granular to the molecular layer providing a new resource for the investigation of local microcircuit computation and of the neural correlates of behavior.

4.2 Introduction

The relationship between structure, function and dynamics in brain circuits is still poorly understood posing a formidable challenge to neuroscience [95]. The core of the issue is how to deal with the distribution and causality of neural processing across multiple spatio-temporal scales. While experimental measurements remain essential, they can now be supported and complemented by realistic computational models. In principle, such models could take into account multi-modal datasets representing morphology, connectivity and activity of different cell populations and make it possible to simulate the propagation of microscopic phenomena into large-scale network dynamics [96–98]. These models can incorporate a broad range of biological data becoming highly constrained and providing the best proxies of the corresponding natural circuits. Eventually, once properly configured and validated, these models can generate their own ground-truth by binding the many parameters, provided by independent measurements and intrinsically prone to experimental error, into a coherent construct, and can be used to test various functional hypotheses [99] using specific simulations platforms, like NEURON [100] and NEST [101]. There are several examples of advanced computational models that have been mostly developed to simulate activities in the cerebral cortex [77,79,102]. Here we have developed a framework, the Brain Scaffold Builder (BSB), to cope with the organization of the cerebellar network.

The cerebellar cortical microcircuit has inspired foundational theories on brain functioning [62] but still challenges realistic computational modeling [103]. Previous network models using ionic conductance-based neurons have been developed only for the granular layer [8, 104]. The only model encompassing the granular and molecular layer altogether made use of single-point neurons with a simplified representation of membrane excitability [105]. Although those models showed a remarkable predictive power against specific target parameters, their main limitation was that connectivity was set independently from neuronal morphology [8,104,105] preventing a direct link between microcircuit structure, function and dynamics. In the meanwhile, detailed computational models of the main cerebellar cortical neurons, which were based on morphological reconstructions embedding multiple membrane ionic channels and synaptic receptors, have been developed, tested and validated [7,106–108]. Thus, with the BSB, we have been able to generate the first computational model of the entire cerebellar cortical microcircuit including both the granular and molecular layer, in which multicompartmental neuron models were wired through a connectome defined by the anisotropy of dendritic and axonal processes through principled rules. The model allowed then to simulate network dynamics and validate it against naturalistic inputs [109–111].

This work generates de facto a new model-based ground truth for the cerebellar cortical microcircuit, predicting the weight that some connections should

have to balance the internal activity. On the scale used here, we observed a set of emerging spatio-temporal dynamics. First, background mossy fiber bombardment induced coherent oscillations throughout the granular layer under gap-junction control. Secondly, collimated mossy fibre bursts mimicking punctuate sensory stimulation generated dense clusters of granule cell activity that propagated vertically invading the overlaying molecular layer, where inhibitory interneurons controlled the emission of burst-pause patterns from Purkinje cells. Finally, synaptic changes mimicked the long-term plasticity of neuronal discharge observed during cerebellar learning. Thus, simulations unveil local microcircuit computations explaining the neural correlates of behaviour, suggesting that the BSB cerebellar model provides a valid resource for future experimental and theoretical investigations.

4.3 Methods

The methods published alongside this journal article were omitted from this chapter because they largely repeat information explained in more details throughout other parts of this dissertation. The originally published methods are available at `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9663576/#Sec19title`.

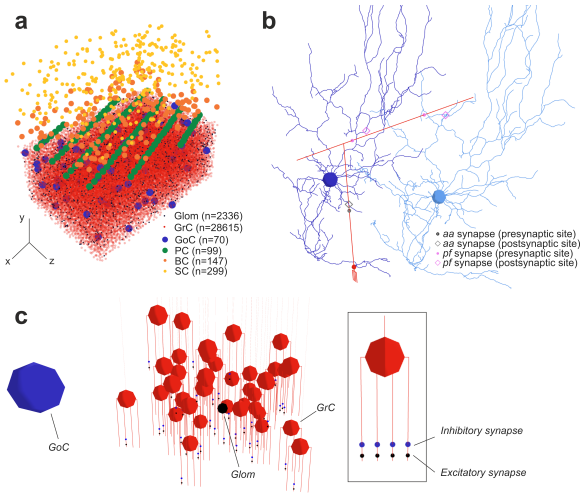
4.4 Results

The BSB was applied to the mouse cerebellar cortical network, which has a geometrically organized architecture that has been suggested to imply its computational properties [62, 103].

The reconstruction and simulation of a network volume of $17.7 \cdot 10^3 mm^3$ is reported, including the following cell and fiber types: mossy fiber (mf), glomerulus (glom), granule cell (GrC) with ascending axon (aa) and parallel fiber (pf), Golgi cell (GoC), Purkinje cell (PC), and molecular layer interneurons (MLI) comprising stellate cells (SC) and basket cells (BC).

4.4.1 Neuron placement

The network elements summed up to 29230 neurons (GrC, GoC, PC, SC, BC) plus 2453 other elements (mf, glom), which were placed in the network volume according to anatomical data [10, 103, 112] (Fig. 4.1a). The density values matched the targets given in the configuration file, the nearest neighbour and the pairwise distance distribution always exceeded cell diameter, and radial distribution function demonstrated the homogeneity of cell distribution without overlapping (Fig. S7.1).



d

	Convergence	Divergence	Synapses	Synapses/pair
mf-Glom	1 ± 0	20 ± 8	2300	1 ± 0
Glom-GrC	4 ± 0	49 ± 26	110000	1 ± 0
Glom-GoC	56 ± 21	1.7 ± 1.4	3900	1 ± 0
GoC-GrC	2.4 ± 0.88	1000 ± 460	97000	1.4 ± 0.66
GoC-Glom	0.84 ± 0.37	28 ± 14	2000	1 ± 0
GoC-GoC	16 ± 6.5	16 ± 7.1	180000	160 ± 5
GrC (*aa*)-GoC	320 ± 230	0.78 ± 1.1	22000	1 ± 0
GrC (*aa*)-PC	82 ± 24	0.28 ± 0.52	20000	2.4 ± 1.1
GrC (*pf*)-GoC	910 ± 360	2.2 ± 1.6	64000	1 ± 0
GrC (*pf*)-PC	1500 ± 300	5.1 ± 2.6	140000	1 ± 0
GrC (*pf*)-BC	740 ± 130	3.8 ± 2.1	110000	1 ± 0
GrC (*pf*)-SC	480 ± 160	5.1 ± 3.2	140000	1 ± 0
BC-PC	20 ± 9.9	14 ± 8	2000	1 ± 0
SC-PC	5.4 ± 2.7	1.8 ± 1.5	530	1 ± 0
BC-BC	14 ± 6.7	14 ± 6.6	200000	100 ± 4
SC-SC	14 ± 6.1	14 ± 5.9	430000	100 ± 4
GoC-GoC (*gap*)	8.4 ± 3.5	8.4 ± 3.5	2100	3.5 ± 1.6

Figure 4.1: Reconstruction of the microcircuit of cerebellar cortex. **a** Positioning of cell bodies in a 3D slab ($300 \times 295 \times 200 \mu m3$) of mouse cerebellar cortex. Cell numbers are indicated (the symbols reflect soma size). In this and the following figures, the xyz reference system is defined by x-y (sagittal plane), x-z (horizontal plane), z-y (coronal plane), as in standard anatomical representation. Thus, y measures cortex thickness (aa direction), while z identifies the major lamellar axis (pf direction). **b** Example of 3D morphologies illustrating GrC-GoC connections through aa and pf. One GrC and two GoCs are shown: the synapse along aa is identified by touch detection, while synapses along pf are identified by fiber intersection. **c** glom -GrC and GoC-GrC connections. A glom contacts a group of 38 GrCs forming an excitatory synapse on the terminal compartment of 1 of their 4 dendrites. The glom, in turn, is contacted by a GoC nearby, which forms an inhibitory synapse on the preterminal dendritic compartment of the same GrCs. The inset shows a GrC with 1 excitatory synapse and 1 inhibitory synapse on each dendrite. **d** The cerebellar cortical connectome generated by BSB reporting convergence (on the postsynaptic element), divergence (from the presynaptic element), total number of synapses, and number of synapses for each connected pair. It should be noted that mf-glom is not a proper synapse but just a branching.

4.4.2 Neuron connectivity

The network connections summed up to 1500000 chemical synapses and 2100 electrical synapses. The cerebellar connectome was modelled combining probabilistic and geometric rules that were chosen depending on available data and the nature of fiber (axon and dendrites) crossing (Fig. 4.1b–d; see Methods for details). This flexible management of connection rules is unique and fixes problems not easy to solve with cerebral cortex simulators, which deal with isotropic cellular organizations and adopt a limited number of intersection rules for all neurons and connections [77, 79, 102]. The well-known connectivity of mf and glom was entirely accounted for by literature data. The BSB generated local anisotropic glom clusters extending $60\mu m$ along the x-axis and $20\mu m$ along the z-axis [113], with 20 gloms per mf [114]. Imposing that each GrC sends its 4 dendrites to gloms belonging to different mfs within about $30\mu m$, the BSB yielded 49 GrCs per glom on average [42, 115]. Each of the 4 GrC dendrites, in addition to a single excitatory synapse on the terminal compartment, also hosted 1 inhibitory synapse on the preterminal compartment, mostly originating from different GoCs (Fig. 4.1c) [9, 116].

The connectivity of GoCs was faced using either literature data (glom-GoC) or adopting various intersection rules (aa-GoC, pf-GoC, GoC-GoC). In fair agreement with literature, each GoC received excitation from 56 different gloms and each glom collected basolateral dendrites from 2 GoCs [117]. There were 320 aa synapses on basolateral dendrites and 910 pf synapses on apical dendrites per GoC, all from different GrCs (Fig. 4.1b) [118]. Moreover, each GoC received inhibition from 16 other GoCs [119] on basolateral dendrites (subsequent functional calibration implied 160 synapses per pair, see below). Finally, there were 8 GoCs that formed gap junctions on other GoCs, with 3.5 gap junctions per pair [120].

The connectivity of PCs and MLIs was recovered using suitable intersection rules (aa-PC, pf-PC, and all MLI synapses). The BSB identified 1500 pf synapses per PC (this figure was limited by the $200\mu m$ network size along z-axis but it would range up by 1 order of magnitude in an unbounded volume [121,122]) and 197 aa synapses per PC from 82 different GrCs [123]. There were 480 pf synapses per SC and 740 pf synapses per BC, while MLI reciprocal inhibition [124] involved 14 SCSC and 14 BCBC connections with 100 synapses per pair. The SC axon, mainly extending on the coronal plane, innervated 2 PCs [125] and each PC received synapses from 5 SCs (Fig. S7.2). The BC axon, mainly extending on the sagittal plane, innervated 14 PCs and each PC received synapses from 20 BCs (akin with the figure of 350 baskets around the PC soma and 710 PCs per BC) [125,126]. These predictions of structural parameters were further assessed and tuned through functional simulations (see below).

4.4.3 Cerebellar network simulations

Network simulations were carried out using detailed neuronal and synaptic models written in NEURON for GrC [5], GoC [4], PC [3, 7], SC and BC [6]. Local microcircuit responses to input patterns were tracked back to individual neurons and used to follow signal propagation with unprecedented resolution. All simulations were carried out in the presence of background noise to improve comparison with recordings in vivo. The emerging spatio-temporal dynamics provided functional model validation beyond constructive validity based on internal connectivity and single neuron responses (Movie S1).

4.4.4 Resting state activity of the cerebellar network

A random input at low frequency (4Hz Poisson) on all mfs [110] was used to simulate the cerebellar network in resting state in vivo. Since anatomical data about the connectivity of cerebellar neurons are incomplete, but their resting discharge frequency is known, we finetuned the number of connections per pair against target values of basal discharge. The turning point was to calibrate GoC-GoC inhibition, which influenced resting state activity of the entire network. Since the synaptic conductance (3200 pS) and the number of interconnected GoCs (about 15) are known [119], we tuned the number of GoC-GoC synapses until basal discharge frequency was achieved. Eventually, the background frequency of all cerebellar neuron types fell in the ranges reported in vivo in anaesthetized rodents (mfs: $4.21 \pm 2.6Hz$; GrCs: $0.81 \pm 1.3Hz$; GoCs: $19 \pm 15Hz$; PCs: $31 \pm 1.6Hz$; BCs: $11 \pm 5.1Hz$; SCs: $9.4 \pm 12Hz$) [GrCs [127], GoCs [4, 128, 129], PCs [130], SCs and BCs [131–133]]. Granular layer oscillations and synchrony Background mf activity is known to generate synchronous low -frequency oscillations in the granular layer [134]. Indeed, in the model, the FFT of GoC and GrC firing revealed a synchronous oscillatory behaviour in the theta band, with the first harmonic peaking at $9.7Hz$. When GoC-GoC gap junctions were disabled, the regularity of the oscillation decreased and the first FFT harmonic moved out of theta band (Fig. 4.2a) [135].

To investigate the sensitivity of Golgi cell synchrony to gap junction density [136], we compared the cross-correlation of Golgi cell discharge with the degree of coupling (electrotonic distance, Fig. S7.2) in GoC pairs, when the network was activated with $4Hz$ Poisson mossy fibre activity. The cross-correlation of Golgi cell discharge decreased smoothly with the increase of electrotonic distance (Fig. 4.3a), tending toward a non -zero level. This non-zero level, that indicates the vanishing of gap-junction effects, corresponded to that observed by disabling the gap junctions and unveiled the synchronizing effect of the feedback loops passing through the granule cell – Golgi cell circuit reported earlier [8, 137] (see below, Fig. 4.3a). This loose synchronization due to shared input from GrCs was still correlated to spatial proximity. In Golgi cell pairs with direct coupling ($n =$ 384 out of 4830 pairs), increasing the gap-junction density by 2.5 times caused two discrete peaks (at $-1ms$ and $+1ms$) in the mean cross-correlogram (Fig.

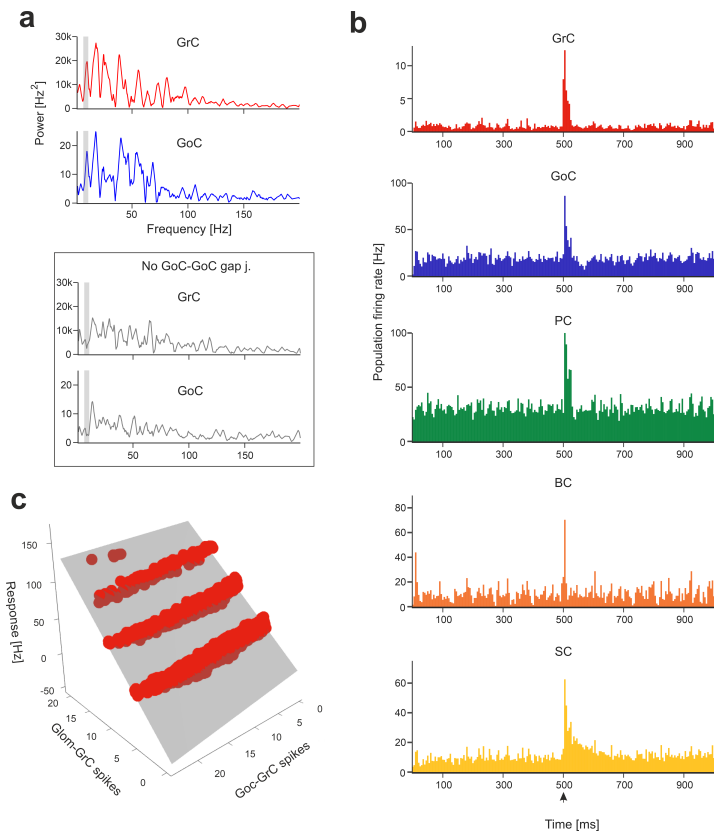


Figure 4.2: Network responses to background noise and mf bursts. **a** Power spectra of GrC and GoC activity are computed with Fast Fourier Transform (FFT) of spike time series (total population spike-counts in 2.5ms time-bins). The periodicity of peaks in power spectra reveals synchronous low-frequency oscillations in the granular layer. The grey curves represent the power spectra when GoC-GoC gap junctions were disabled, showing a marked decrease in periodicity. The grey bands correspond to mouse theta-band ($5 - 10Hz$). **b** The Peri-Stimulus-Time-Histograms (PSTH) of each neuronal population show the effect of the localized mf burst (onset indicated by arrowhead) emerging over background noise. The PSTHs show number of spikes/5ms time-bins normalized by the number of cells, averaged over 10 simulations. **c** Example of multiple linear regression of GrC responses (firing rate) against the number of synaptic spikes from gloms and GoCs, during 40ms after stimulus onset. The grey surface is the fitted plane to the points (each point corresponds to a GrC receiving the mf burst on at least 1 dendrite).

4.3b). A spike could either precede or follow the one emitted by a neighbouring Golgi cell with millisecond precision as observed experimentally [136]. In Golgi cell pairs with indirect coupling (i.e., 2 or more cells away, n=842 pairs), the two peaks in the mean cross-correlogram disappeared, as much as when gap junctions were disabled. The percentage of synchronous spikes across all GoC pairs located within $100\mu m$ reached about 27% with a $5ms$ time lag window, again consistent with experimental findings [136] (Fig. 4.3c). Following this functional validation, the model was used to compute the probability density of spike coincidence in the granular layer, predicting that the effects of Golgi cell coupling can extend over an ellipsoidal volume over $100 \times 200\mu m$.

4.4.5 Impulsive response of the cerebellar network

Short stimulus bursts were delivered to a bundle of 4 mfs connected to 80 gloms to emulate whisker/facial sensory stimulation in vivo [110, 127]. The burst propagated through the network, temporarily raising neuronal firing (Fig. 4.2b, Movie S1). The relationship between the number of spikes at afferent synapses and the response frequency to the mf burst was robustly captured by multiple linear regression (Fig. 4.2 c; Fig. S7.4a; Table S1). GrC responses Fundamental predictions on how GrCs respond to incoming bursts derive from current clamp recordings in situ [138] and simulations [4], which revealed the role of synaptic receptors and ionic channels. In BSB simulations, bursts on a collimated mf bundle activated a dense cluster of GrCs [105, 109, 139]. The relationship between the number of input spikes (both at GoC-GrC and glom -GrC synapses) and GrC response frequency unveiled 4 groups of GrCs with a corresponding number of synaptically activated dendrites (Fig. 4.2c). The number of GrC spikes, first spike latency and dendritic [Ca2+] in correlated with the number of active dendrites (NMI=0.71, 0.86, 0.59, respectively) (Fig. 4.4a, b).

When the inhibitory mechanisms (comprising transient and persistent inhibition) were disabled to simulate a pharmacological GABAA receptor blockade, (i) GrC baseline frequency increased, (ii) a tail discharge appeared after the burst, (iii) responses including more spikes appeared, (iv) the first spike latency decreased, and (v) response variability decreased (Fig. 4.4a, b). The number of GrC spikes, first spike latency and dendritic [Ca2+]in still correlated with the number of active dendrites (NMI=0.79, 0.85, 0.61, respectively) (Fig. 4.4b). Interestingly, inhibition caused a reduction in the number of active GrCs (i.e., those firing¿= 1 spike in the 40ms after the mf burst onset were 3390 ± 431, and 8348 ± 1724 with GABA-A off; n=10 simulations; p¡0.001, unpaired ttest) but enriched the spike pattern, as predicted theoretically [39, 62].

Recordings in vivo disclosed precise integration of quanta and high-fidelity transmission in the granular layer [110,140–143]. In BSB simulations, GrCs receiving maximum excitation generated one action potential for each spike of the input

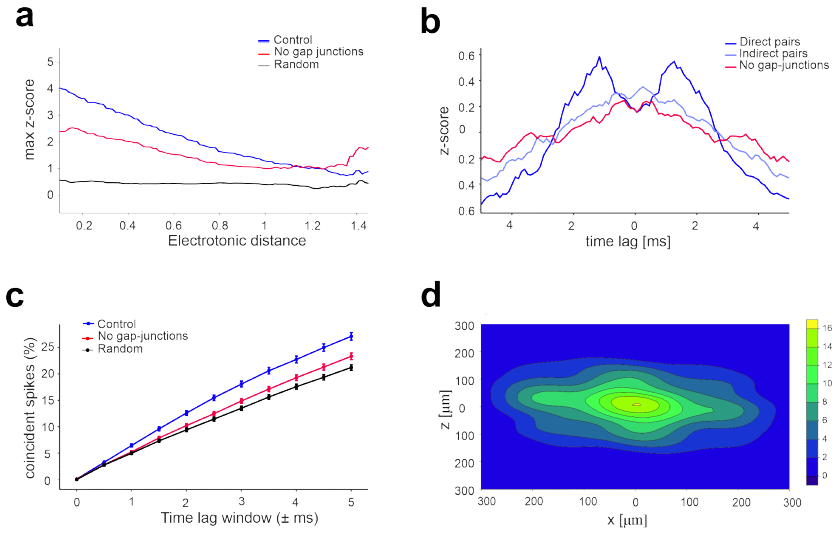


Figure 4.3: GoC millisecond synchronization by gap junctions. **a** Maximum cross -correlation in pairs of Golgi cells as a function of electrotonic distance. The three curves represent control condition (4-Hz Poisson mossy fibre activity), with gap-junctions disabled, and with random spike patterns of GoCs. All values were calculated using a sliding window of ± 0.2 electrotonic distance. At large electronic distances, the z-score in control conditions tends toward the value set by random input patterns. **b** The average cross-correlograms (0.5ms bins) is calculated in control condition for GoC pairs at $< 100\mu m$ distance with either direct coupling (n=384), indirect coupling (n=842), all pairs located $< 100\mu m$ distance from each other when gap junctions were disabled. The z-score shows two distinct peaks indicating GoC-GoC correlation with ms spike precision with on average 7.5 gap-junctions per direct pair. **c** The percentage of spikes that fall within distinct time-lag windows across all pairs located $< 100\mu m$ distance in control condition, with gap-junctions disabled, and with random spike patterns of GoCs . Points are mean $\pm$ SEM (n=1181). **d** Probability density of spike coincidence in the granular layer horizontal plane. This plot indicates that, with a GoC spike in [0,0], there is a certain probability that GoCs around it will fire a spike within a $\pm 5ms$ time-window. The integral of the probability density function over the whole network corresponds to the average spike coincidence for the same time window in (c).

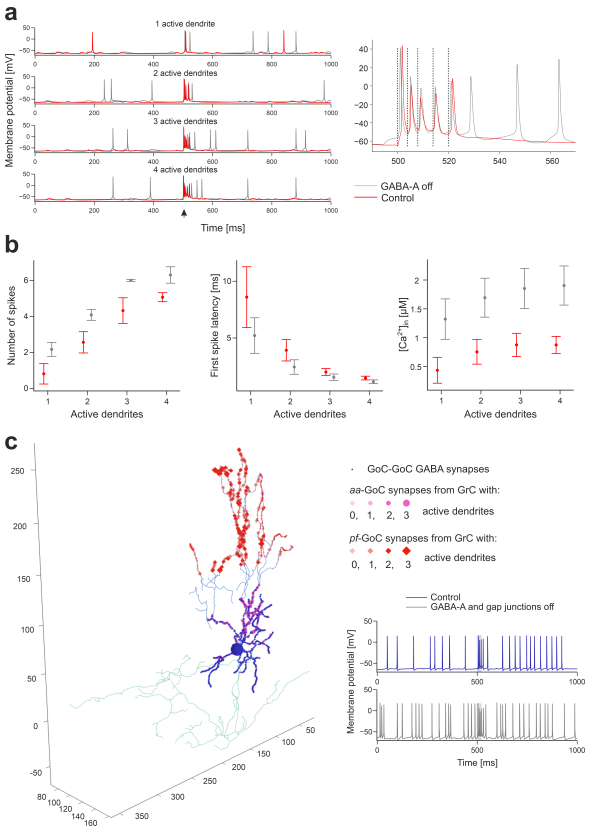


Figure 4.4: Granular layer activation. **a** Membrane potential of 4 representative GrCs with 1 to 4 dendrites activated by the mf burst ($20ms@200Hz$ over background noise, onset indicated by arrowhead), in control condition and after GABA-A receptors blockade ("GABA-A off"). The burst response of the GrC with 4 active dendrites is enlarged on the right to highlight spike-timing (dashed lines indicate the mf burst spikes). **b** Number of spikes (measured in the 40ms from mf burst onset), first spike latency, and dendritic [Ca2+]in (measured in the 500ms from mf burst onset) in subgroups of GrCs with the same number of activated dendrites (Ndend). Means ± sd are reported (n=21068 with Ndend=0, n= 2361 with Ndend =1, n=892 with Ndend=2, n=164 with Ndend=3, n= 6 with Ndend=4). The graphs compare responses in control and during "GABA-A off ". c Synapses of a GoC activated by GrCs. Bigger markers correspond to presynaptic GrCs more activated by the mf burst. The GABAergic synapses from other GoCs are on basolateral dendrites, aa synapses are on basolateral dendrites, pf synapses are on apical dendrites. In this example, the GoC receives 30% of its aa synapses and 6% of its pf synapses from GrCs with at least 2 active dendrites. Traces on the right show the GoC membrane potential in response to the mf burst (same stimulation as in (a), grey band) in control and during GABA-A receptors and gap junctions switch-off.

burst, with short latency (¡ 2ms), and faithfully followed the input up to 250Hz (Fig. 4.4a) (Movie S2).

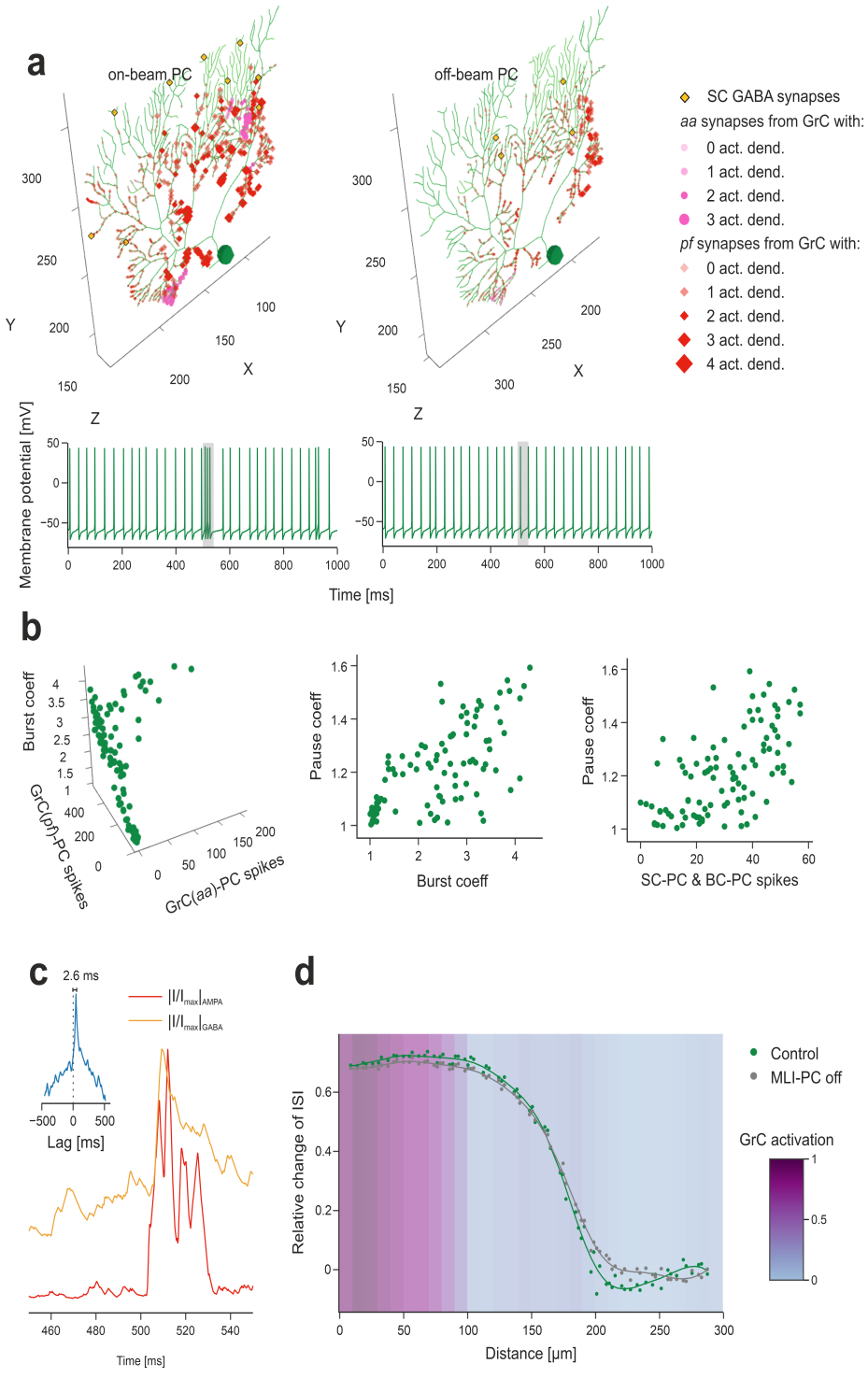
4.4.6 GoC responses

Following punctuate sensory stimulation in vivo, GoCs have been reported to respond with short bursts of 2–3 spikes at up to 200–300Hz [144]. In BSB simulations, GoCs immersed in the GrC active cluster generated a burst of 25 spikes with a maximum instantaneous frequency of $213 \pm 29Hz$ (Fig. 4.4c). When GABA synapses and gap junctions between GoCs were disabled, the response bursts showed up to 6 spikes, with a higher maximum instantaneous frequency ($308 \pm 16Hz$) (n=70 GoCs; p¡0.001, paired t-test) (Fig. 4.4c). The burst was caused by synaptic excitation relayed by gloms and GrCs (through both aas and pfs), which generated AMPA and NMDA currents in GoC dendrites (Movie S3). The "silent pause" appearing after the burst was caused both by an intrinsic phase-reset mechanism [144–146] and by reciprocal inhibition between GoCs, demonstrating marked dendritic processing capabilities [4].

4.4.7 PC and MLI responses

PCs in vivo are known to respond to punctuate stimulation with burst-pause patterns [111,147]. In BSB simulations, PC responses depended on cell position relative to the mf active bundle (Fig. 4.5a). The PCs placed vertically on top of the GrC active cluster received the largest number of aa and pf synaptic inputs producing typical burst-pause patterns [7]. The burst coefficient was correlated with the number of synaptic inputs from pf and aa (multiple regression analysis: R2=0.91) (Fig. 4.5b). The pause coefficient was correlated with the burst coefficient (NMI=0.79) and with the number of spikes from MLIs (NMI=0.66) (Fig. 4.5b), reflecting the origin of the pause from both intrinsic after -hyperpolarizing mechanisms and MLI inhibition [148]. Indeed, MLIs are known to narrow the time window and reduce the intensity of PC responses [131]. In BSB simulations, the PC AMPA current arose soon after the spikes emitted by GrCs, while the PC GABA current was delayed by 2.6ms (Fig. 4.5c). In summary, the di-synaptic IPSCs produced by MLIs quickly counteracted the monosynaptic EPSCs produced by aas and pfs, providing precise time control over PC activation [139,149].

BCs in vivo are known to generate lateral inhibition reducing PC discharge below baseline causing contrast enhancement [39,131]. In BSB simulations, this pattern emerged during stimulation of a mf bundle (100ms @ 50Hz stimulation on 24 neighboring mfs). The PCs placed in a band 150-200μm beside the active cluster along the x-axis were inhibited, bringing their frequency below baseline. When MLI-PC synapses were disabled, the effect disappeared revealing contrast enhancement due to lateral inhibition (Fig. 4.5d).


a
on-beam PC
off-beam PC
SC GABA synapses
aa synapses from GrC with:
0 act. dend.
1 act. dend.
2 act. dend.
3 act. dend.
pf synapses from GrC with:
0 act. dend.
1 act. dend.
2 act. dend.
3 act. dend.
4 act. dend.
Membrane potential [mV]
Time [ms]
b
Burst coeff
GrC(pf)-PC spikes
GrC(aa)-PC spikes
Pause coeff
Burst coeff
Pause coeff
SC-PC & BC-PC spikes
c
2.6 ms
Lag [ms]
Time [ms]
d
Relative change of ISI
Distance [µm]
Control
MLI-PC off
GrC activation

The response of MLIs in vivo is only partially known [39]. In BSB simulations, SCs and BCs intersected by active pfs responded to input bursts and their activity remained higher than baseline for several hundreds of milliseconds, especially in SCs [6] (Fig. S7.4b).

4.4.8 Modification of model parameters to simulate neural correlates of behavior

Two conditions modifying PC firing patterns and their modulation were explored in order to test whether our network model was able to predict neural correlates of behavior: i) knock-out (KO) of MLI inhibition on PCs, which

---

Figure 4.5 *(preceding page)*: Purkinje cell activation. **a** The PC placed on top of the GrC active cluster and the PC placed at its margin show different synaptic inputs. GABAergic synapses from SCs are on medium-thickness dendrites (those from BCs on PC soma are not shown), aa synapses are located on thin dendrites and pf synapses on thick dendrites. Bigger markers correspond to presynaptic GrCs more activated by the mf burst. In this example, the on-beam PC receives 23% of its aa synapses and 6% of its pf synapses from GrCs with at least 2 active dendrites, the off -beam PC 0% of its aa synapses and 0.6% of its pf synapses from GrCs with at least 2 active dendrites. The corresponding membrane potential traces are shown at the bottom (the 20ms mf burst is highlighted by grey band). **b** Analysis of the burst-pause response of PCs to the mf burst (20ms @200Hz over background noise). The burst coefficient (i.e. the shortening of the inter-spike interval due to the mf burst, with respect to baseline) is reported against the number of spikes from aas and from pfs (multivariate regression analysis: $R^2 = 0.91$). The pause coefficient (i.e. the elongation of the inter-spike interval after the mf burst response, with respect to baseline) is reported against either the burst coefficient (NMI=0 .79) or the number of spikes from SCs and BCs (NMI=0.66). **c** Synaptic currents recorded from the PC on top of the GrC active cluster (same as in (a)) , in voltage -clamp. The traces are the sum of all excitatory (AMPA) and inhibitory (GABA) dendritic currents during the mf burst. They are rectified, normalized and cross -correlated (inset) unveiling a GABA current lag of 2 .6ms with respect to AMPA current. **d** By stimulating a mf bundle (100ms @50Hz Poisson stimulation on 24 adjacent mfs), the PC response (modulation with respect to baseline) was quantified by the relative change of Inter-Spike -Interval (ISI), during the stimulus, where 0 corresponds to baseline. The two series of points compare PC response modulation when SCs and BCs were either connected ("control") or disconnected from PCs ("MLI-PC off"). The curves are regression fittings to the points (Kernel Ridge Regression using a radial basis pairwise function, from Python scikit-learn library). The GrC active cluster "(GrC activation") was identified by a threshold on the stimulation-induced activity by using kernel density estimation.

impacts on vestibulo-ocular reflex (VOR) adaptation [150], and ii) long-term plasticity at pf- PC synapses, which drives learning in associative tasks like eye-blink classical conditioning (EBCC) [151]. KO of MLI inhibition on PCs GABAA receptor–mediated synaptic inhibition was selectively disabled in Purkinje cells (KO condition), and a single stimulus pulse was delivered to a bundle of 13 mfs. The burst response of PCs was broader and with a strong temporal dispersion (jitter) of simple spikes in KO than control condition (control: 0.49ms; KO: 1.01ms; p¡0.01 t-test) (Fig. 4.6). These alterations of PC activity patterns reproduced the dysregulation of cerebellar signal coding and adaptation observed in PC-$\delta\gamma$ 2, a mouse line in which GABAA receptor-mediated synaptic inhibition was selectively knocked-out in Purkinje cells [150].

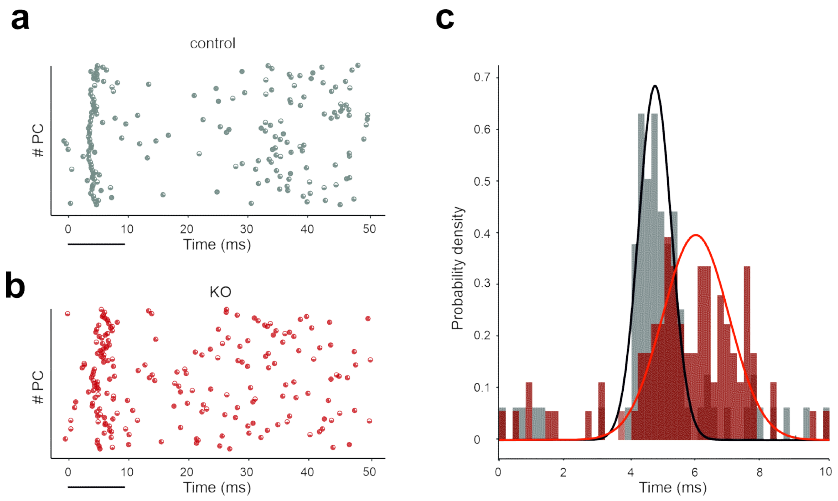


Figure 4.6: Molecular layer interneurons modulate PC discharge. Raster plots of PC spikes following an impulse on a bundle of 13 mfs (time 0) in (a) control condition and in (b) KO condition, in which GABAA receptors are blocked from PCs to uncover the neural correlates of dysfunctional VOR adaptation in the PC-$\delta\gamma$ KO mouse line (c) Probability density functions of spike count (time bins of 0.2 ms) in the 10-ms window following stimulation in the two conditions. Note the more scattered firing response in KO condition.

4.4.9 Long-term plasticity at pf-PC synapses

Reduced values of the AMPA receptor-mediated synaptic conductance (gsyn) were used to simulate long-term depression (LTD) at pf-PC synapses. In EBCC, a level of suppression of about 15% was found to correlate with a stable generation of associative blink responses at the end of the learning process [151]. In BSB simulations using a stimulus at 50Hz on a mf-bundle, different LTD levels caused a corresponding amount of PC simple spike suppression (Fig. 4.7). A 15% PC simple spike suppression emerged with pf-PC LTD of about 35%, predicting the number of synapses that should undergo LTD in order to explain

the experimental observation.

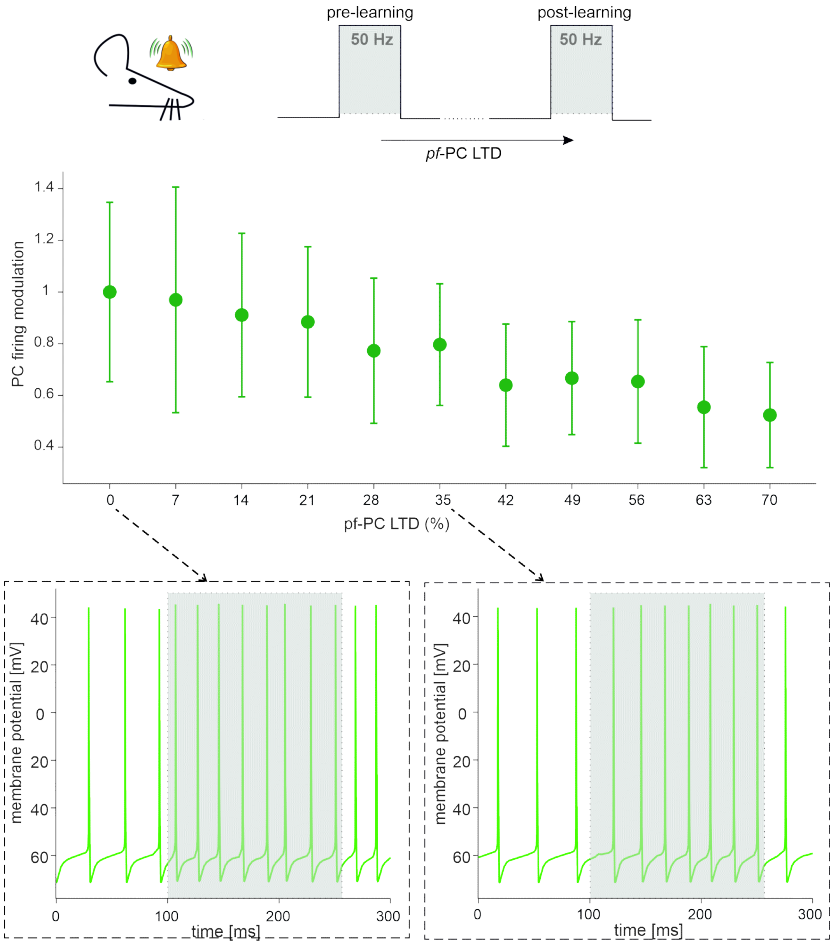


Figure 4.7: pf-PC plasticity modulates PC discharge. The scheme on top shows the simulation protocol that emulates an EBCC paradigm, in which a conditioned stimulus (CS) is delivered to the mfs. Our simulations reproduce the final state ("post -learning") by exploring multiple levels of pf-PC LTD. The relationship between these LTD levels and PC firing modulation (relative to "pre-learning" state with 0% LTD) is shown. The points are mean±SEM across all PCs (n=99). Two representative traces illustrating PC discharge are shown for 0% LTD and 35% LTD in the insets.

4.5 Discussion

This work shows the first detailed model reconstruction and simulations of the cerebellar cortical network and predicts neuronal activities involved in the prop-

agation of mossy fiber input signals from the granular to the PC and molecular layer. By means of the BSB model, we have combined heterogenous data using suitable placement and connectivity rules with accurate multi-compartmental neuron models. In the optimization process, the model extracted information from the interdependence of parameters, bound at high-level through ensuing network dynamics, allowing us to fill gaps in knowledge through constructive rules. In the validation process, the model demonstrated its compatibility with a wealth of experimental literature data collected over the last decades and a parameters sensitivity able to uncover the neural correlates of specific physiopathological conditions.

4.5.1 A model-based ground-truth for the cerebellar cortical network

The statistical and geometrical rules derived from anatomical and physiological works [103, 114] almost completely anticipated network connectivity at the cerebellar input stage. In the BSB model, each glom hosted 50 excitatory and 50 inhibitory synapses on as many GrC dendrites, plus 2 excitatory synapse on basolateral dendrites of as many GoCs, summing up to 102 synapses per glom, in agreement with the anatomical upper limit of 20030. Each one of the 4 GrC dendrites received an excitatory and (in most cases) an inhibitory input from as many different mfs and GoCs, respectively [9, 117]. Each GoC received 320 aa synapses on basolateral dendrites and 910 pf synapses on apical dendrites, according to the figure of 400 and 120032, and there were 3 electrical synapses per GoC-GoC pair [135]. Functional tuning suggested that the number of gap-junctions could actually be 2.5 times higher, i.e., $7 - 8$ per GoC-GoC pair [135]. Only the number of GoC-GoC GABAergic synapses, which amounted to a figure of 160 after functional tuning, lacked any experimental counterpart. In the molecular layer, under geometric and functional constraints, the BSB model placed limits to the debated numbers determining PC and MLI connectivity. The model predicted that 25% of aas contacted the distal dendrites of the overlaying PCs (7˙133 out of 28˙615 GrCs), each aa forming 2.4 synapses on average, supporting the important role predicted for the aa [149, 152], while pfs formed 1 synapse per PC dendritic intersection. In summary, each PC received 12% of the whole GrC inputs from aas, matching the empirical estimate of 724% [153]. The BSB generated 25 SC-PC and BC-PC synapses altogether, which compares well with the experimental estimate of 2068. Moreover, there were 17˙600 pf-MLI-PC synapses (2˙600 pf-SC-PC and 15˙000 pf-BC-PC synapses), compatible with the prediction that the pf - MLIs-PC input is larger than the pf-PC input on the same PC [154]. In general, since all dendritic trees in the molecular layer are orthogonal to pfs, the BSB reconstruction ranked the number of synapses according to dendritic size - PC (1 ˙500) ¿ GoC (900) ¿ BC (700) ¿ SC (500) – a figure that would increase proportionately by scaling the model slab to include full-length pfs [155].

Accurate single-neuron models with realistic morphology proved also critical to carry out simulations allowing us to finetune the connectome. In particular, the number of inhibitory synapses per GoC-GoC pair was increased in order to make them fire at $19Hz$ $[2-30Hz$ range $[127, 144]]$. Similarly, the number of inhibitory synapses per SC-SC and per BC -BC pairs was tuned in order to make them fire at $10Hz$ $[1-35Hz$ range $[131, 132]]$ and to bring PCs into their resting state frequency range of $31Hz$ $[36.4 \pm 11.5Hz$ $[130, 156]]$ in vivo. The number of gap -junctions per GoC-GoC pair was tuned to obtained millisecond synchrony [136].

Thus, a reconstruction of model connectivity purely based on geometrical rules was not sufficient and a careful tuning against functional data was needed. This two-pronged (structural and functional) approach ensured that all parameters were bound at high-level through the basal neuronal firing frequency at rest in vivo [95]. Eventually, the network connectome is in fair agreement with a wealth of disparate anatomical and functional determinations, suggesting that the emerging picture provides a new model -based ground-truth for the cerebellar cortical network.

4.5.2 Cerebellar network model validation and predictive capacity

The functional validation of single neuron models was previously reported in specific studies [4–6, 148], so that these neurons could be directly plugged in and used to simulate spatio-temporal network dynamics in vivo. The functional validation of the cerebellar network model implied first to analyse responses to diffused background noise, which is reported to generate coherent large-scale oscillations [134]. The BSB model showed indeed that GrCs and GoCs were entrained into low -frequency coherent oscillations in resting state and, interestingly, this happened under gap junction control as observed experimentally [135]. Furthermore, the BSB model showed that Golgi cell synchronization through gap junctions occurred with millisecond precision [136]. Thus, gap junctions refined and potentiated the synchronizing effect of massive shared excitatory inputs from GrCs reported earlier [8, 137]. As a whole, these simulations predict that the spatial organization of Golgi cell inhibitory control depends on the distance among GoCs and on their specific morphology and orientation supporting a modular circuit organization: a marked correlation and synchronicity can be observed within an assembly, while it tends to decrease between assemblies, indicating Golgi cells coordinate segregation and integration of activities in the granular layer of cerebellum [157].

The functional validation was extended by simulating responses to naturalistic mf bursts, which rapidly propagated through the GrC-PC neuronal chain (Fig. 4.8) (Movie S1). GrCs responded in a dense cluster [139] regulated by GoCs and activated soon thereafter the overlaying PCs and MLIs. In the cluster, 45% of

the GrCs fired at least one spike, in agreement with results reported previously [103, 139]. Not unexpectedly, SCs and BCs effectively reduced activation of PCs placed either along or beside the active pfs, respectively, generating feedforward and lateral inhibition [62, 103].

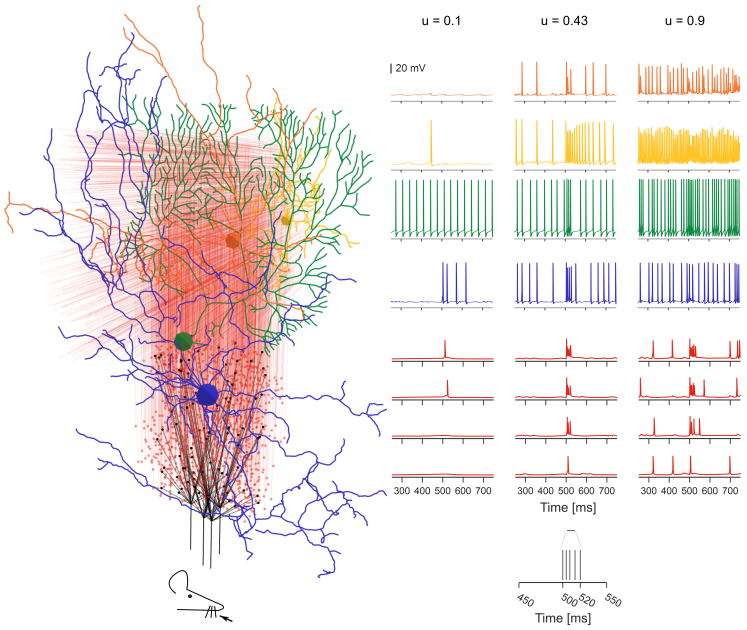


Figure 4.8: Activation of a vertical neuronal column in the cerebellar cortex. A whisker air-puff stimulus (the mf burst) is delivered to 4 adjacent mfs, which branch in 4 glom clusters. GrCs respond rapidly with a burst when at least 2 dendrites are activated. A GrC dense cluster is formed and the signal propagates up through an aa bundle and transversally along a pf beam. GoCs receive the signal both on basolateral and apical dendrites. PCs vertically on top of the active cluster are invested by aa and pf synaptic inputs. On-beam SCs and BCs receive signals through pf synapses; SC axons inhibit mainly on-beam PCs, while BC axons inhibit mainly off-beam PCs. The membrane potential traces (mf burst starts at 500ms) are shown for each neuronal population. Traces in the three columns correspond to three different release probabilities at the mf-GrC synapses: u=0.1, u=0.43 (control condition used in the rest of the paper), u=0.9. The lower and higher u-values are typical of long-term synaptic depression and potentiation in the granular layer.

4.5.3 Model predictions of neural correlates of behavior

Network simulations with the BSB cerebellar model predicted the neural correlates of behaviour in different physio-pathological conditions. First, PCs showed

the typical burst-pause responses that are thought to correlate with cerebellar-dependent motor control [147]. These responses were seriously altered by changing mf-GrC neurotransmitter release probability [5], whose effect propagated from the cerebellar input stage throughout the whole thickness of the cerebellar cortical network, suggesting a possible substrate for pattern regulation in the cerebellum [158]. Secondly, selective removal of GABAA receptor -mediated synaptic inhibition from PCs reproduced the neuronal alterations correlated to dysfunctional VOR adaptation in the PC-$\delta\gamma$ 2 mouse line [150]. Thirdly, plasticity remapping predicted that LTD in 35% of pf-PC synapses could explain the 15% PC simple spike suppression observed during EBCC [151].

4.5.4 Comparison with previous cerebellar models

Since Marr's work [62], the cerebellum has been amongst the most intensely modelled brain microcircuits and has provided a workbench to test biophysical principles of excitability and connectivity. The incorporation of biologically realistic features into models has progressed along the last three decades, as sketched below by considering just some of the many published works.

Spiking models of the granular layer with active membrane mechanisms in neurons

The first one [137] had only the granular layer, neurons were single compartment and with generic excitable mechanisms, synapses did not have short-term plasticity. A second model used cell-specific membrane mechanisms and synapses with short-term plasticity [8]. However, neurons did not have realistic multi-compartment morphology yet. In both cases, connections respected proportions reported in literature without prescribed connectivity rules.

Spiking models including both the cerebellar cortical network and deep cerebellar nuclei

The first model [159] used integrate and fire point neurons and a canonical formulation of neuronal numbers and connectivity. A second set of models used a cerebellar network scaffold strategy with general rules for cell positioning and connectivity based on the probability cloud algorithm [10]. An extended version [160] included deep cerebellar nuclei and the inferior olive. Neurons were single compartment with non -linear discharge properties and synapses did not have short-term plasticity.

The current model of the cerebellar cortical network integrates and extended all the previous realizations by featuring an integrated reconstruction and simulation strategy, using multi-compartment neurons with cell-specific membrane mechanisms, using synapses with intrinsic neurotransmitter release dynamics and short-term plasticity, and adopting multiple connection rules including morphology-based touch-detection and voxel-intersection. These advancements

reflect into the ability of the model to capture a large set of biological properties of the network under various physio-pathological conditions.

4.5.5 Limitations and future challenges

The most relevant problem of this kind of microcircuit models is to incorporate variables that remain underconstrained. Here we have 5 cell types, 16 synaptic types and as many ranges for synaptic density. Almost all of them were carefully validated beforehand, except the BC model with its synapses and the density of reciprocal interneuron inhibitory synapses, which warrants specific investigation. Thus, although the parameterization of the cerebellar network model relies on one of the best-defined anatomical and physiological datasets in the brain [8, 10, 69, 103], it cannot be excluded that other parameter combinations might also be effective. Indeed, some structural data were missing (e.g., not measured experimentally) or error-prone and their estimates were provided by network reconstruction and simulation. The emergent structural parameters were then confronted with available knowledge for constructive validity. The scaffold configuration allows to easily host new data and to update the existing ones when new experimental data become available.

Here we have enforced a connectivity principle largely based on proximity rules between neurites and tuned the connection algorithms to bring the connectivity within the anatomo-physiological range (see Methods). Alternative algorithms for automatic parameter tuning may also be used to predict the cerebellar cortical network connectome [161] and compared to the present results. Finally, while we have used two most representative functional templates (background oscillations and response to sensory-burst stimulation in vivo), others could be envisaged. It should be noted that often functional validation relies on sparse experimental data quantifying single-neuron responses to sensory stimuli. Therefore, multi-layer mesoscale recordings would be useful to further validate model predictions about the mechanisms of microcircuit computation in the cerebellum, e.g., following whisker stimulation or along EBCC training.

Although it is validated on a small network scale (30 k neurons and 1.5M synapses), the model is about 1000 times smaller than the whole mouse cerebellum. This would not be a problem if the model would be a small-scale representation of the cerebellar cortex, but this is not the case given the anisotropy of cerebellar network architecture. The first issue is that signal propagation along the transverse plane would require longer modules. Here we have observed the formation of vertical columns [149, 153] but it would be important now to assess [162, 163] the beam hypothesis along with spatial signal filtering and plasticity [4–7, 39, 62]. Moreover, the cerebellar cortex is subdivided into microzones with different biochemical and functional properties, while the present model can just be taken as a good proxy of the Z+microzone [164–166]. Therefore, the model should be extended and diversified to explore effects on a

larger scale.

Another issue is that, in the model, all neurons of the same type are identical one to another. However, there is morphological and functional variability among neurons of the same type. Moreover, there are known variants of granule cells, Golgi cells and Purkinje cells [3–5,7]. It would therefore be important to explore the impact of neuronal variability, which can bring about relevant computational effect [167]. The same also applies to synapses, which now have the same release probability and gain at homologous connections but are tuned by plasticity in real life [4, 69, 109, 111, 122, 154, 165] and could therefore change network dynamics. The future introduction of plasticity, which now is present only in simplified models [160, 168] and cerebellar subnetworks [8, 10, 62, 103, 105], will allow to refine the effective functional organization of the connectome and test hypotheses on network functioning.

Finally, the operations of the cerebellar cortex are tightly bound to those of the deep cerebellar nuclei and of the inferior olive. However, to date the only available representations on the mesoscale are reconstructed using simplified single point neurons [168,169] and a fine grain realistic representation is missing. Therefore, the model could be extended to the mesoscale to investigate how the cerebellar cortex operates inside the olivo-cerebellar system.

In aggregate, the BSB model shows that the geometrical organization of neurites largely determines cerebellar cortical connectivity and microcircuit dynamics, supporting the original intuition of J.C. Eccles in the late 60's [39,62]. A similar conclusion was recently reported for the cortical microcolumn [97]. With appropriate extension, the model could allow to simulate cerebellar modules including differentiated microzones and microcomplexes [164–166] and more complex patterns of stimuli in the sensorimotor and cognitive domain [69]. Given the "scaffold" design, new neurons and mechanisms can be plugged-in to address ontogenesis, species differences (for example in humans) and pathology. For example, the model may be used to predict the emerging dynamics caused by genetic or epigenetic alterations in neuron (morphology and function) and synaptic properties, as it is supposed to happen in ataxia, dystonia and autism [170, 171]. The model may also be used to predict the impact of drugs acting on ionic channels and synaptic receptors. In conclusion, the model can be regarded as a new resource for investigating the structure-function-dynamics relationships in the cerebellar network.

Chapter 5

Applications of the cerebellar cortex model and BSB framework

5.1 Olivocerebellar Microcomplex Circuit

The cerebellum receives 2 main types of afferent fibers, the mossy fibers, carrying somatosensory or cognitive information, and the climbing fiber originating from the inferior olive (IO), carrying error information. The cerebellum uses these 2 sources of information to form predictive models of understanding of the internal and external worlds, and assists in predictive planning of motor commands and predictive cognitive control.

So far no integrated multicompartmental models exist comprising a model of the cerebellar cortex with all main cell types represented, the deep cerebellar nuclei, and the inferior olive. Such a model could form a closed loop of predictive error based learning.

Here we integrate the existing cerebellar cortex (CC) model [94], and a full scale model of the inferior olive (IO) model [172] (unpublished thesis), by adding the deep cerebellar nuclei (DCN) and the various connections between the newly joined cell populations.

The integration efforts started after the IO was already modeled, so the IO workflow was integrated using a data dependency component that during the

pipeline phase generates the entire IO model, and uses the public API of the BSB framework to store the generated data in the model. Although strictly speaking it clashes with the foreseen architecture, because placement and connectivity steps are ran outside of the placement and connectivity context, embedding a small adapter component into an early phase of the workflow proved an effective strategy to integrate arbitrary existing workflows into a BSB managed workflow, and may be explored and formalized further as a supported feature.

```
@config.node
class InfOliveModelTransfer:
    def load_object(self):
        pos, morphos, conns = generate_io()
        io = self.scaffold.cell_types.io_cell
        ps = io.get_placement_set()
        for i, morpho in enumerate(morphos):
            self.scaffold.morphologies.save(
                f"io_{i}", my_morpho_to_bsb_morpho(morpho)
            )
        ps.append_data(
            pos,
            morphologies=[
                f"io_{i}"
                for i in range(len(morphos))
            ]
        )
        conns = self.scaffold.require_connectivity_set(
            "io_cell_to_io_cell", io, io
        )
        conns.connect(conns[0], conns[1])
```

Code Snippet 5.1: Integration pseudocode, where the cells and connections generated by `generate_io` are transferred to their respective `PlacementSet` and `ConnectivitySet` with some data conversion functions like `my_morpho_to_bsb_morpho`.

5.1.1 IO model reconstruction

The bounded volume of the IO was extracted from z-stacks of microscope slices using Neurolucida software to create contours which were then converted to a wavefront triangle mesh. IO cells form dendrodendritic electrical gap junctions in microstructures called glomeruli, distributed evenly through space, while the network of the IO cells forms clusters of more numerously connected IO cells. To model this, the glomeruli and IO somata where distributed evenly in the bounded volume using Poisson disk sampling, and clusters of configurable size created with K-means clustering. The morphologies of the IO cells were created

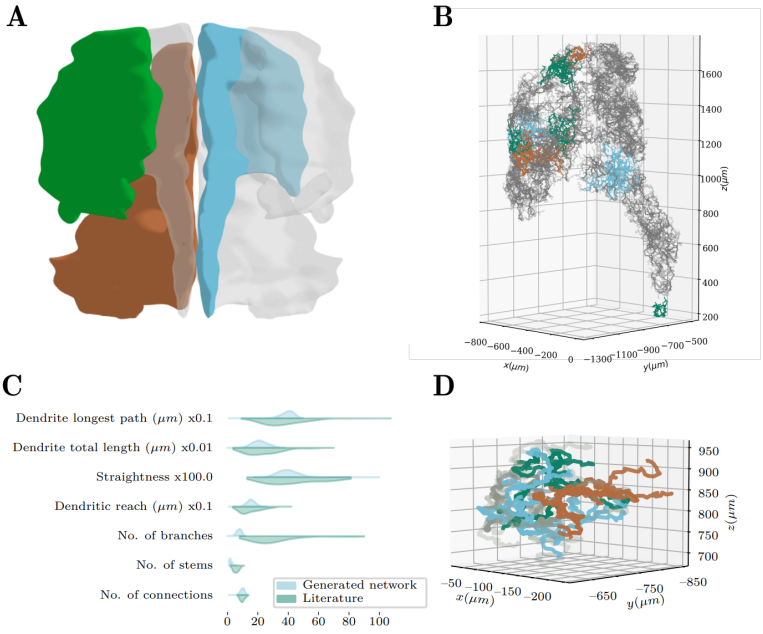


Figure 5.1: **A:** Bounded volume representation of the inferior olive, generated from stacked contours from microscopy slides. Highlighted are the left Primary Olivary nucleus (PO, blue), right Medial accessory olivary nucleus (MAO, brown) and right Dorsal accessory olivary nucleus (DAO, green). Notice the folding of the PO. **B:** Generated network with highlighted clusters with parameters for the left PO. **C:** Comparision between experimental distributions from Vrieler et al.. Straightness is the maximum reach divides by the maximal path length. **D:** Single morphological cluster from (B), with individual cells highlighted. Taken from Landsmeer, LPL 2022 (unpublished).

by constrained random walks from the somata to the glomeruli (Fig. 5.1).

5.1.2 Deep Cerebellar Nuclei (DCN)

The cerebellar cortex has only 1 type of efferent fibre, from the Purkinje cell to the DCN cell. Two populations of DCN cells were added, an interneuron which projects back to the Purkinje cell, molecular layer interneurons, and to the IO cells, and outward projecting DCN cells as the output of the model.

The DCN cells were modeled as Leaky Integrate-and-Fire (LIF) models with parameters taken from Geminiani et. al. 2019 [160].

5.1.3 Integrative connection types

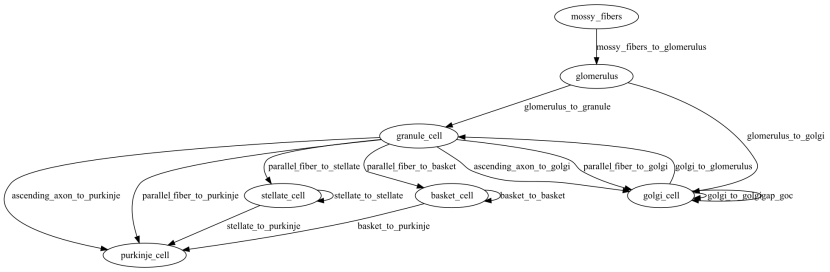


Figure 5.2: Schematic representation of the CC cell populations and connection types between them before addition of the IO components.

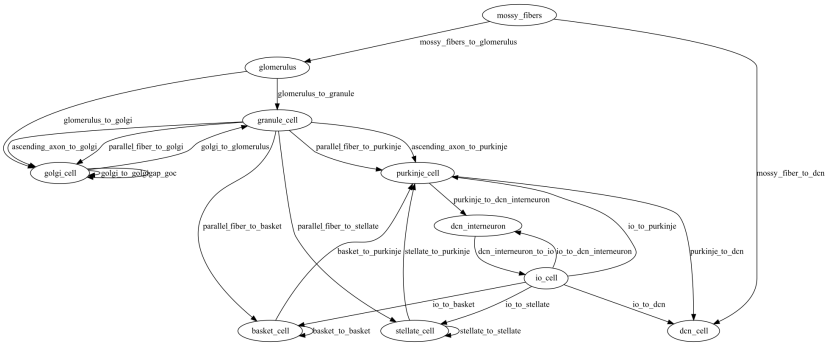


Figure 5.3: Schematic representation of the CC and IO cell populations and connection types between them after integration.

5.1.4 Role of the framework

The framework was used to create a composite unified description. The framework allowed converging on a shared software stack: even though the IO developers were using entirely custom data formats, by relying on the BSB's job pools their workflow could be integrated into a framework component and their data

transferred into it with relative ease. From the model description simulations will be ran on the unified model relying on the framework's Arbor adapter.

5.1.5 Future work

The collaboration is ongoing, and within a month we would like to run our first simulations investigating how the CC-DCN-IO closed loop functions, especially how the subthreshold oscillations of the clusters of IO cells exert control and transmit information to the cerebellar cortex, and back.

Another avenue of exploration is the refinement of the DCN cell model and the connections between the added components with more biophysical restraints, so that more information can be transferred between the two models in a biologically realistic manner.

5.2 Pathological Cerebellar Cortex Microcircuits

5.2.1 Autism spectrum disorders

The cerebellar cortex model was used to introduce perturbations associated to autism spectrum disorders (ASD). The cerebellum projects to multiple regions in the cerebrum that underlie movement, language, and social processing and may in this way contribute to social deficits and stereotyped behaviors [173]. The postsynaptic density of synapses contains scaffolding proteins that link receptors to the cytoskeleton. Mutations in the Shank family of scaffolding proteins is linked to autism and other neurological disorders. Most of the mutations occur in the IB2 gene which encodes a scaffolding protein that interacts with the NMDA receptor, which is abundantly present in the mossy fiber to granule cell synapses, and is the main driver of granule cell plasticity. Research into the effects of the IB2 knockout mouse model for autism on the cerebellar cortex revealed that this caused, among other things, hyperexcitability and hyperplasticity at the mossy fiber to granule cell synapse layer due to larger NMDA currents, and a reduction of the Purkinje cell dendritic arbor [174, 175].

To model these changes we alterated the single cell model of the granule cell, optimizing it to find a suitable change to the NMDA receptor maximum conductance, and the ambient glomerulus glutamate concentration (which relates proportionally to excitability due to glutamate spillover), and found a best match at a maximum conductance of $141nS$ and ambient concentration of $27.8\mu M$.

Future work

In this study, we conducted simulations and created a model to replicate the autistic alterations observed in the experimental findings of Soda et al 2019 [175] our next step will involve incorporating the IB2-GrC within the cerebellar

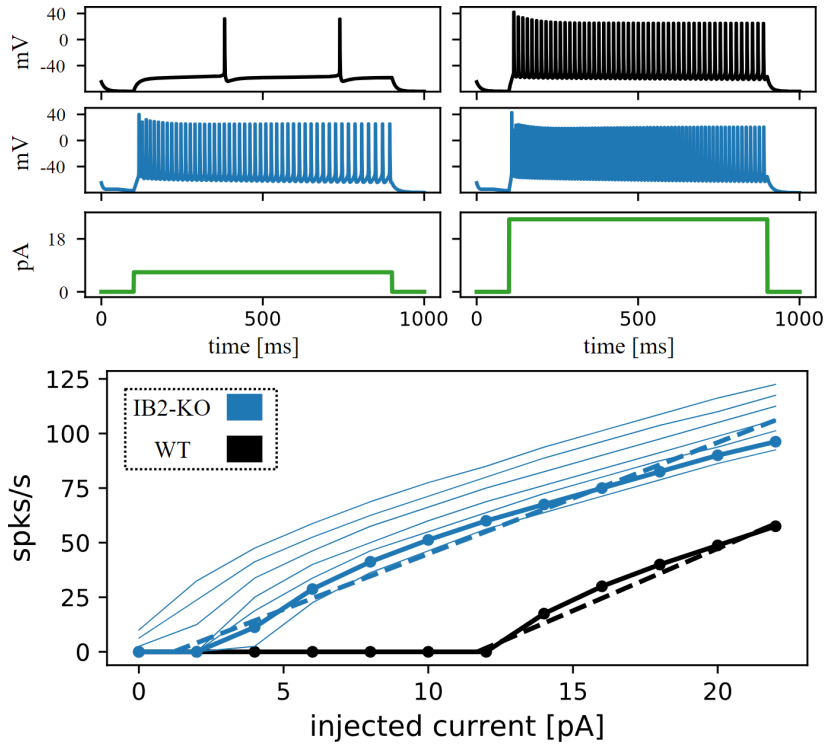


Figure 5.4: Empirical (dashed) and simulated (solid) I-F curves of the wild type and IB2KO granule cell. The maximum conductance of the NMDA receptor and the ambient glutamate concentration were optimized to match the empirical observation.

microcircuit to simulate and analyze its impact at the network level.

5.2.2 Emotional networks and disorders

This section comes from my colleague Dimitri Rodarie's unpublished collaboration.

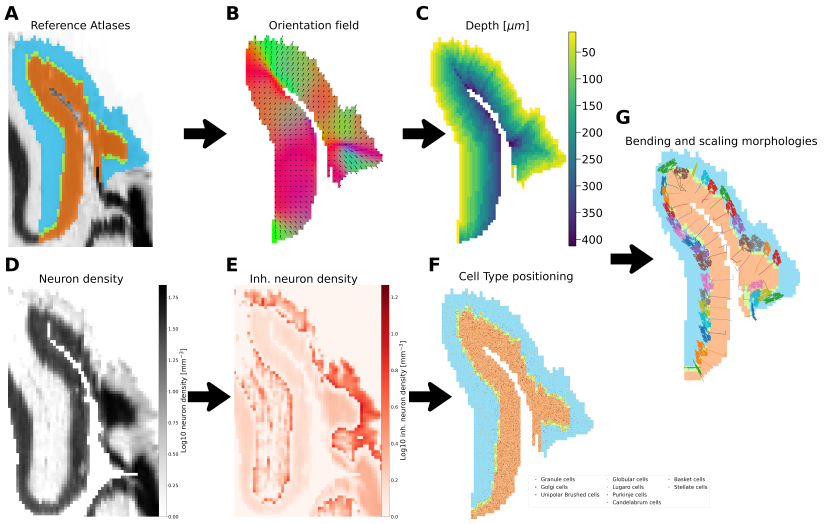


Figure 5.5: Mouse declive cell placement workflow: Each image represents a sagittal slice of the result of the step of our process to reconstruct the mouse declive. Black arrows show the order of the workflow steps. **A.** Our corrected annotation atlas of declive is shown in colors over the Nissl volume in levels of grey. The granular layer appears in orange, the molecular layer in blue and Purkinje layer in green. **B.** To each voxel of the declive, a 3D direction normalized vector is computed corresponding to the main axis of the axons in the region. Colors represent the orientation vectors norm on their respective plane, black arrows their projected vector. **C.** Distance of each voxel of declive to the outside border of the molecular layer, following the orientation field, expressed in micrometers. **D. E.** Respectively neuron and inhibitory neuron density in logarithmic scale. **F.** Soma position of the different neuron types of the declive, displayed over the annotation atlas. Each cell type appears in a different color and size corresponding to its radius. The annotation atlas' colors correspond to A . **G.** Projection of the Purkinje cell morphology displayed in colors over the annotation atlas. Each morphology has been rotated, scaled, and bended following the orientation and depth fields. The annotation atlas' colors correspond to A.

We aim to reconstruct and simulate atlas-mapped cerebellar regions of the mouse. We want to understand the cerebellar contribution to emotional behavior, in the healthy and diseased brain. As we are interested in cerebellar

regions involved in the process of emotions, we focus on Lobule VI. Numerous experimental and clinical evidence in both humans and rodents show that this lobule plays a relevant role in many functions [176–178] including behavioral and emotional tasks. Lobule VI is made up of a vermis part (declive) and a hemispheric part (simple lobule).

We present here a pipeline to reconstruct the declive of the mouse, based on the Blue Brain Cell Atlas model [179] and the Brain Scaffold Builder (BSB). With this pipeline, we were able to estimate for the first time the specific densities of each cell type, including granule, golgi, unipolar brush, lugaro, globular, Purkinje, candelabrum, basket, and stellate cells. In the BSB we placed, oriented, and connected the neurons. The output of this pipeline is a circuit that can be simulated and validated against functional experimental findings.

Methods

We build here a 3D representation of the mouse declive region, embedded into an anatomically realistic whole mouse brain structure (Fig. 5.5). We based our model on the Blue Brain Cell Atlas pipeline, which we extended with the Purkinje layer at the boundary between granular and molecular layers. We also added Unipolar brush cells and lugaro cells based on regional densities from Sekerková et al. [180] and Dieudonné and Dumoulin [181], respectively. Moreover, we proposed a new strategy to place Purkinje, candelabrum and globular cells based on linear density from Osorno et al. [182] (Fig. 5.5A). The remaining cell types and their numbers were estimated using regional distributions from the Blue Brain Cell Atlas [179].

To reconstruct local connectivity, we computed the orientations of each morphology (Fig. 5.5BC) using Rodarie et al.'s method [179]. We leveraged this information to orient neurons including granule cells and their ascending axons. Also, it will be used to bend the parallel fibers of our model following the external surface of the region. We applied voxel intersection and point clouds connection strategies and synaptic in- and out-degree ratios reported in De Schepper et al. [94]

We assigned point-neuron electrical parameters to each cell type, and synaptic parameters to each connection type, according to Geminiani et al. [160] The resulting scaffold model has been simulated using the BSB interfacing with the NEST simulator [94, 101]. The model will be structurally and functionally refined as more data becomes available.

Results

The final scaffold model has 3,113,153 neurons, with 2,877,812 Granule cells, 45257 Golgi cells, 10956 Unipolar brush cells, 234 Lugaro cells, 3240 Globular

cells, 3512 Purkinje cells, 3875 Candelabrum cells, 80952 Basket cells and 87314 Stellate cells (see Fig. 5.5D).

The connectome defines a precise picture of the local connectivity.

Preliminary simulations in NEST [101] are ongoing, to validate the dynamics of the declive region in resting state and under proper stimulations. Our goal is to leverage this circuit to study circuits in cerebellum responsible for emotional experiences. Moreover, our strategy can be extended to reconstruct other cerebellar regions. A full -scale cerebellum network will allow us to analyze subregions specificities and their interactions. Finally, we will investigate mechanisms to simulate emotional states [183], fear learning [177, 184] and pathological states such as anxiety [185], depression [186] and autism [187].

5.2.3 Role of the framework

Both projects are a continuation of the model validated in 4.

Role in the autism disorder project

because of the framework's separated configuration, a new collaborator was able to introduce perturbations into a model they had no previous experience with, by setting new parameter values in the afflicted cell model's configuration; in previous iterations of the model that did not use the framework, this parameter was only found in the NMODL description of the mechanism, and could not be set separately without duplicating the NMODL mechanism description, the model description (both of which would require either refactoring or copy-pasting the entire cell model directory structure provided by another collaborator), and refactoring the population creation logic (which for simplicity's sake would likely result again in copy-pasting and adjusting the code).

This exemplifies how without a supporting framework it is likely that model code is organically organized along the path of least resistance, becoming less maintainable with each iteration, to achieve the scientific milestones, which is not a slight to neuroscientists, but the nature of scientific code.

Role in the emotional network project

Our colleague Dimitri Rodarie had developed a brain atlas pipeline in earlier works [179]. The pipeline starts from several NRRD images, and results in per-voxel density estimates of cell types, and classification and orientation of the cortical layers. Using the framework's pipeline expressions the previously established pipeline was integrated into an atlas-based reconstruction of the cerebellar declive region.

Framework provided placement strategies then rely on this per-voxel density estimate to place the cells. A custom distributor was then developed to generate the morphologies to grow according to the orientations established by the pipeline.

Framework provided connection strategies then intersect the generated morhologies to arrive at the connectome.

The connectome will then be simulated by using the framework provided simulation adapter for the NEST simulator.

5.3 From a Mouse to Human Cerebellar Cortex Model

The results presented in this section are from the currently unpublished work by my colleague Alessio Marta, and were showcased in a poster in CNS2023.

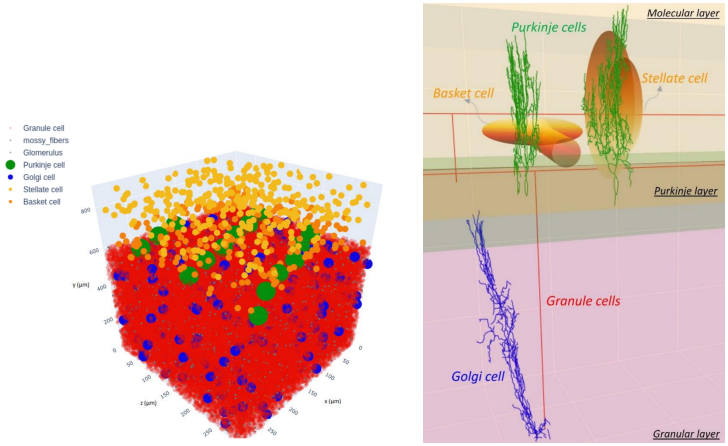


Figure 5.6: **Left:** Reconstruction of a patch of the human cerebellar cortex, showing the somata. **Right:** Detailed view of the morphologies used per cell type: constructed morphology for the granule cells, human morphology for the Golgi and Purkinje cells, and geometrical approximations for the basket and stellate cells.

The mouse cerebellum cortex model was used as a basis, and the key parameter differences between them are being investigated. A tentative reconstruction was created according to layer thicknesses taken from Zhen et al. 2022 [188] resulting in $300 \cdot 900 \cdot 300 = 0.32mm^3$ of reconstructed cortex with 148000 cells of the following neuron types: mossy fibers (270), glomeruli (12000), granule cells (135000) with ascending axon and parallel fiber, Golgi cells (486), Purkinje cells (15), basket stells (195), and stellate cells (381) (Fig. 5.6). Missing

microstructural parameters have been estimated by proportion with the mouse cerebellum. Whenever possible the connectome was based on human neuron morphologies. Otherwise, geometric shapes were used if neurite extensions were known, or it was scaled up from the mouse model if no information was known. The reusability of the mouse model and BSB were exploited to transform the mouse cerebellum into its human variant.

The human model was simulated at point-neuron resolution using the NEST simulator [101]. These initial simulations allow us to validate and further explore new hypothesis about the reconstruction, to couple back, and improve the model. It will allow us to investigate the relationship between the dynamics, function, and structure of the human cerebellum. Special interest goes out to the signal propagation and dynamics of different mossy fiber organization patterns.

5.3.1 Role of the framework

This work is a continuation of the work presented in 4. Because of the separated configuration, the human variant could be reconstructed by replacing the configured dimensions of the layers based on literature references, and to swap out the morphologies used to human morphologies provided by collaborators.

The same workflow could then be reused in its entirety to reconstruct a human variant. As more data becomes available also the cell model dynamics can be adjusted to human variants, by creating derived model descriptions, using *arborize*'s feature to create one model description based on another template, offering inheritance for model descriptions.

5.4 Hippocampus

We collaborated with Italian neuroscience research groups from Palermo and Modena, to add a new geometry based connection strategy to the BSB, and leveraged it to reconstruct the hippocampal CA1 region [189]. The following is an abridged version of our paper Gandolfi et al. [189].

5.4.1 Abstract

Here we propose a method to implement a neuronal network at single cell resolution by using the geometrical probability volumes associated with pre- and postsynaptic neurites. This allows us to build a network with plausible connectivity properties without the explicit use of computationally intensive touch detection algorithms using full 3D neuron reconstructions. The method has been benchmarked for the mouse hippocampus CA1 area, and the results show that this approach is able to generate full-scale brain networks at single cell

resolution that are in good agreement with experimental findings. This geometric reconstruction of axonal and dendritic occupancy, by effectively reflecting morphological and anatomical constraints, could be integrated into structured simulators generating entire circuits of different brain areas facilitating the simulation of different brain regions with realistic models.

5.4.2 Methods

Neuronal placement

Cells placement was performed by downloading neuronal coordinates from the Blue Brain Cell Atlas database [190], which provides 3D coordinates of excitatory and inhibitory classes in the Allen reference atlas already annotated for the four layers of mouse hippocampus: Stratum Oriens (SO), Stratum Pyramidalis (SP), Stratum Radiatum (SR) and Stratum Lacunosum Moleculare (SLM) (Fig. 5.7). Neuronal populations respect the ratio of 10% between inhibitory (inh) and excitatory (exc) classes. The neurons (exc/inh) were then divided into 13 classes (2 exc, 11 inh) according to their relative distribution [191, 192] within layers (see Fig. 5.7). The scaffolding of the neurons in the simulation volume was performed with the BSB framework [94] (Fig. 5.7).

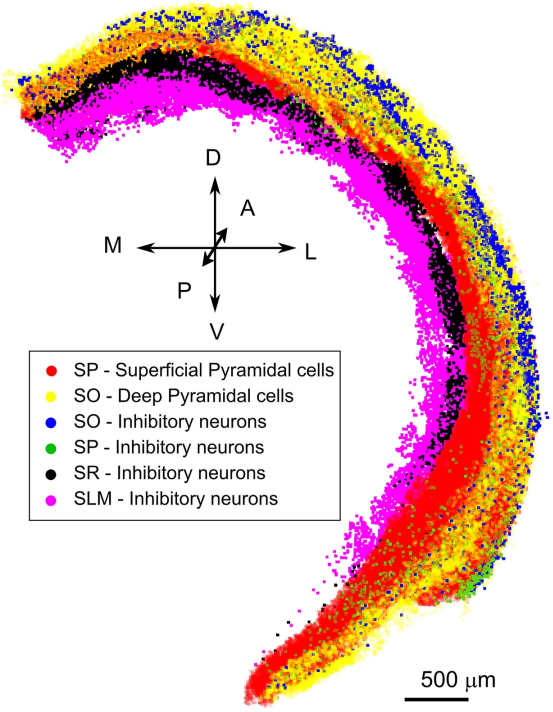


Figure 5.7

Neuronal morphology

In our approach, the rules for the generation of connections between any two neurons were implemented assuming that every neuronal class is characterized by an average shape. Morphological analysis has been performed by collecting the experimentally reconstructed morphologies of CA1 neuron subtypes from the literature and from public databases such as neuromorpho [193], Allen Brain Institute [194], and Janelia Research Campus [195]. In this analysis, we have assumed that each cell class could be represented as a combination of geometrical shapes (ellipsoids and cones). The parameterization of these shapes (axonal and dendritic extensions) was generated by creating normal distributions for each of the parameters with peaks corresponding to the average values derived from the analysis and half-widths of 10% of the peaks.

In addition to the size of geometrical shapes, neurons were endowed with morpho-anatomical features derived from the calculation of the minimum Euclidean distances of neuronal soma from internal (CA1 layers) and external landmarks (CA3 and Subiculum). An automatic iterative algorithm scanned neuronal positions and calculated the minimum distances between neuronal coordinates and CA3, Subiculum meshes and CA1 layers. This allowed the consideration of the experimentally observed preferential orientation of PC axons along the direction of the minimum distance between CA3 and Subiculum, thus implementing the strong directionality in Pyramidal-to-Pyramidal activity propagation along transversal hippocampal slices that has been observed experimentally [196]. These morpho-anatomical features were used to generate the parametric description that allowed the orientation of axonal and dendritic probability clouds along realistic anatomical axes.

Network connectivity

Neuronal connectivity was instantiated by iteratively intersecting a single presynaptic axonal hull and the postsynaptic dendritic points of all the neurons belonging to a particular class. To reduce the computational effort required to explore all the possible connections, we circumscribed axonal and dendritic probability clouds within their minimal bounding-boxes.

The final evaluation of the intersection between axonal clouds and dendritic points was performed only on neurons with overlapping bounding boxes. Connection pairs were computed through an iterative algorithm that calculated if at least one postsynaptic dendritic point was included in a presynaptic axonal cloud. All the neuronal pairs that shared at least one point were included in the pool of potential connection pairs. The final number of connection pairs was obtained through a pruning procedure that followed the estimation of the numerosity of connections between two neuronal classes. This number was estimated by multiplying the number of neurons composing the two classes and the synaptic connection probability obtained from hippocampome.org [197,198].

In particular, the connection probability was generated by taking into account synaptic densities and synaptic contacts. For instance, the estimated number of connections between superficial Pyradmidal Cells and Ivy cells can be obtained by multiplying the total number of neurons in the two classes (216,435—PCs and 5074—IVY) per connection probability of the two cell types (0.000933 as obtained from hippocampome.org). The result is 1,024,612 connections. This number has been obtained by pruning the connection pairs generated with the procedure of probability cloud intersection (9,321, 511).

The time required to calculate the intersection between probability clouds, including the procedure to generate and orient morphologies, was dependent on the following parameters: (1) the number of points in each cloud (2) the number of potential intersections between each neuron and cells belonging to a specific class and (3) the numerosity of each neuronal class.

5.4.3 Results

The developed "Positional-Morpho-Anatomical" modeling (PMA, Fig. 5.8) approach can be adapted and applied to different brain regions with proper morpho-anatomical constraints. We applied the PMA algorithm to the case of the mouse CA1 network, for which data on cell placement and morphological features are available. Several models of hippocampus CA1 have been developed with variable levels of detail ranging from extremely simplified networks, to realistic full-scale networks. However, a network at single cell resolution using connectivity rules based on morpho-anatomical constraints, rather than simple fixed connection probabilities, was not yet available.

5.4.4 Role of the framework

In this project, the hippocampus was reconstructed from an embargoed pipeline, so the collaborators instead used the framework's CSV importing component to import the placement data, and much of the connectome. And then relied on the NEST adapter to run their simulations[1]

The PMA algorithm was initially also developed by them in MATLAB and included in the CSV import, but has now been converted into a component; the initial data remains imported from CSV, but the component is now added to the model configuration to generate the new connections according to the PMA algorithm integrated in the workflow.

[1] They used a previous major version of the framework, which suffered from high memory overhead in NEST, but a custom branch was provided to overcome this.

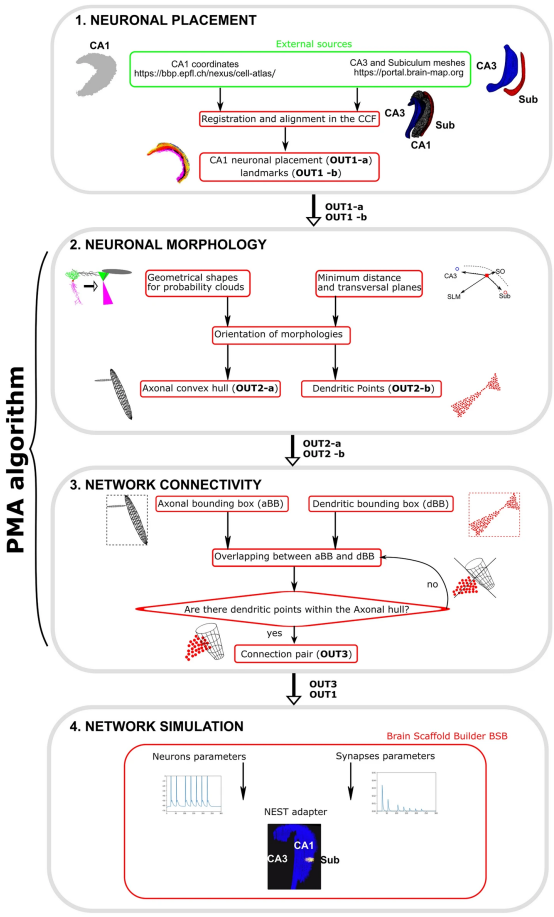


Figure 5.8

5.5 Thalamic nuclei

A collaborator and contributor, Francesco Sheiban, led the development of a pipeline to reconstruct the thalamic nuclei. Here is a report on the ongoing unpublished efforts using the BSB as main modeling tool, in combination with a realtime visualization tool developed by Francesco Sheiban to facilitate debugging (Fig. 5.9).

5.5.1 Methods

Placement

For accurate placement, neural densities from the Blue Brain Project's Mouse Cell Atlas were queried for VAL and RT. Utilizing Allen Mouse Brain Atlas

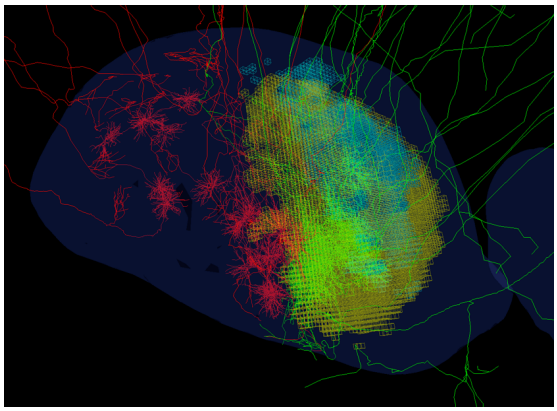


Figure 5.9: The figure shows the outcome of the data integration process followed during the connectivity reconstruction in the thalamic scaffold implementation. The RT volume (represented by the blue mesh) has been segmented using data from the Allen Mouse Brain injections dataset (yellow and light blue voxels, belonging to anterograde injection experiments in the RT and VAL, respectively). It can be seen that the volume identified by such experiments is consistent with the trajectories of the 3D morphological cells reconstructions, as the green axons (representing motor-related cells originating in VAL) pass within the segmented area, while red axons (somatosensory cells originating from a nucleus just next to VAL) are not contacting the RT in the designated area.

segmentation data, we distinguished VAL and RT areas, considering variations in the mouse thalamus akin to humans and primates. The Calb1 gene's density served to differentiate VA and VL voxels.

Manual annotation of experiment IDs led to the download of a 32-bit structural annotation volume at $200\mu m$ resolution. After upscaling $200\mu m$ density data to $25\mu m$, a straightforward threshold mechanism classified subvolumes into primary and secondary subvolumes.

This information facilitated cohesive data integration, with each labeled voxel filled according to BBP densities atlas predictions, preserving VA/VL distinctions for morphological placement.

Connectivity

During the connectivity phase cells within the designated volume are interlinked using a voxel-based geometrical approach, identifying cells, who when their morphology is appromixated by a set of voxels that intersect with one another, as potential synaptic candidates.

The process involves multiple steps: as the VA-VL nuclei only target the RT neurons in a precise sub-nucleus region, the voxel mask of such region has been identified using a combination of detailed morphological 3D cell reconstructions, Allen Mouse Brain's projection experiments and geometrical extrusion of the VA-VL volumes, following a general axonal projection direction. Combining VAL mesh extrusion with projection experiments aligns well with traced data from a literature study focusing on RT to thalamus connections.

With the RT subregion identified, the connection strategy employs a geometrical approach, projecting thalamo-cortical cells onto the RT with a distance-dependent Gaussian profile. Adjustments accommodate soma offsets, ensuring accurate projections onto the desired subvolume. This nuanced approach refines the BSB scaffold connectome construction.

5.5.2 Role of the framework

This is another example of user's relying on the data processing pipelines to formalize and encapsulate the processing of datasets into framework-ready input: with 2 different brain atlases as data source, a reclassification pipeline establishes a new masked brain image that's used as the per-voxel density for framework provided placement strategies. Then, a custom morphology distributor generates morphologies to project and grow towards targets, and connected with framework provided morphology intersection components.

5.6 Arbor simulator benchmarks

I include only the abstract of the preprint. The work still needs to be redone including comparisons to CoreNEURON, and with a better biological match of the simulation outputs. At the time of the comparison the models could not be completely detangled or understood from their NEURON implementation, as NEURON allows for entirely procedural descriptions, with a host of problems arising when attempted to be described in Arbor in more biologically revelant domain specific languages (DSLs), which only support a sane subset of biologically plausible instructions of the entire set of instructions that a Turing-complete programming environment would allow. This means that questionable workarounds and tweaks that existed in these long-lived much passed around models (all of them contained at least one HOC mechanism that had its origin pre-2000) had to be figured out and elegantly solved. This was not possible in the given timeframe, and after finding a suitable biological description of the models, parameter reoptimization will be required.

5.6.1 Abstract

A variety of software simulators exist for neuronal networks, and a subset of these tools allow the scientist to model neurons in high morphological detail.

The scalability of such simulation tools over a wide range in neuronal networks sizes and cell complexities is predominantly limited by effective allocation of components of such simulations over computational nodes, and the overhead in communication between them. In order to have more scalable simulation software, it is therefore important to develop a robust benchmarking strategy that allows insight into specific computational bottlenecks for models of realistic size and complexity. In this study, we demonstrate the use of the Brain Scaffold Builder (BSB; De Schepper et al., 2021) as a framework for performing such benchmarks. We perform a comparison between the well-known neuromorphological simulator NEURON (Carnevale and Hines, 2006), and Arbor (Abi Akar et al., 2019), a new simulation library developed within the framework of the Human Brain Project. The BSB can construct identical neuromorphological and network setups of highly spatially and biophysically detailed networks for each simulator. This ensures good coverage of feature support in each simulator, and realistic workloads. After validating the outputs of the BSB generated models, we execute the simulations on a variety of hardware configurations consisting of two types of nodes (GPU and CPU). We investigate performance of two different network models, one suited for a single machine, and one for distributed simulation. We investigate performance across different mechanisms, mechanism classes, mechanism combinations, and cell types. Our benchmarks show that, depending on the distribution scheme deployed by Arbor, a speed-up with respect to NEURON of between 60 and 400 can be achieved. Additionally Arbor can be up to two orders of magnitude more energy efficient.

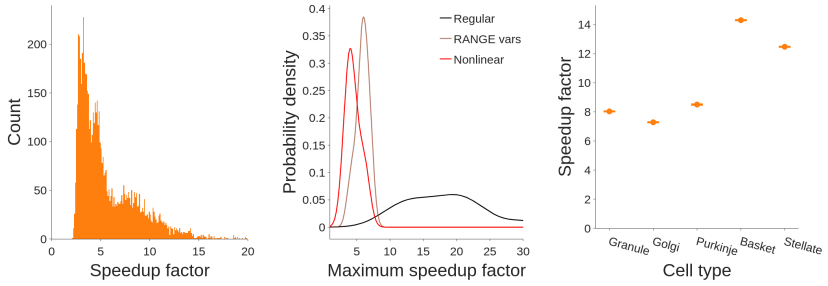


Figure 5.10: A catalogue of 33 mechanisms was benchmarked. **Left:** Histogram of all combinations of 1 to 4 out of 33 mechanisms inserted in a single compartment. **Center:** Speedup distribution of manually annotated categories of mechanisms: Unordinary mechanisms, those containing highly complex nonlinear dynamics, and those containing a high number of RANGE vars (an expensive feature of NEURON's mechanism language NMODL). **Right:** Speedup of cerebellar single cell models.

5.6.2 Role of the framework

The model in 4 was used to provide extensive feature coverage in both benchmarked simulators. The framework was extended with an Arbor simulation

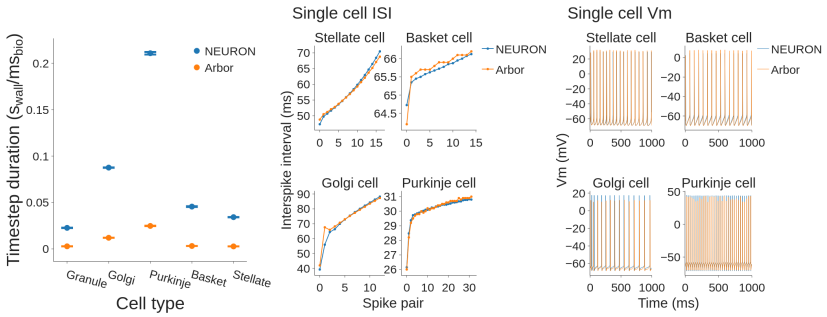


Figure 5.11: **Left:** Comparison of timestep duration of single cell models. **Center:** Validation of single cell inter spike intervals (ISI). **Right:** Validation of membrane voltage.

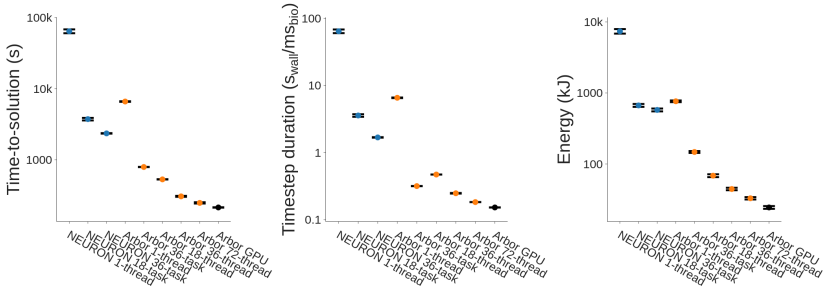


Figure 5.12: Comparison of time-to-solution (left), timestep duration (center), and energy consumed (right) of different benchmark hardware configurations on a single compute node.

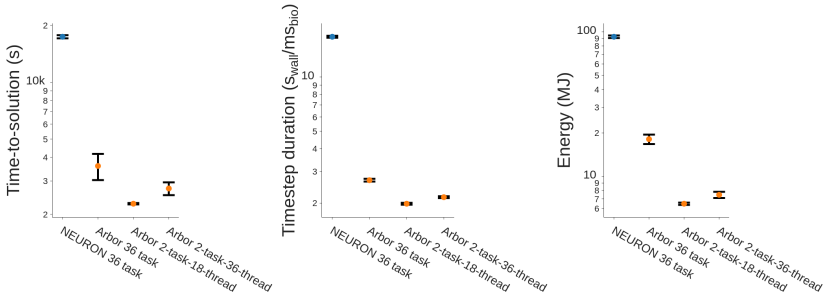


Figure 5.13: Comparison of time-to-solution (left), timestep duration (center), and energy consumed (right) of different benchmark hardware configurations on multiple compute nodes.

adapter, and then leveraged as the only tool able to simulate identical models in both NEURON and Arbor at that time[2]. By extending the *arborize* package to also produce identical cell models in Arbor [3] our laboratory's models could be used for simulation and benchmarking in both Arbor and NEURON.

[2]Arbor now supports a small part of NeuroML, and jNML/pyNML can run NeuroML in NEURON

[3]Arborize started as a package to declaratively describe cell models in NEURON, with a congruently evolved Arbor-like syntax, for which it was later renamed.

Chapter 6

Discussion and Future Work

6.1 Critical Analysis of the Brain Scaffold Builder: Advantages, limitations, and future works

The framework is now nearly a mature software, with a growing community of users and contributors, and is integrated in the EBRAINS Research Infrastructure. The impact of reusable software can not be understated: if the community converges on this software, and develops a healthy ecosystem of tools, novel parts of the multiscale modeling effort would be captured and trivialized, and large amounts of time and resources can be saved. For the first time, data handling and parameter generation could be included in a reproducible declarative multiscale model description, which proved a very useful feature for microscale bottom-up modeling projects.

From the applications and first user stories we can see that the data processing pipelines, brain atlas integration, and offering abstract representations for common data structures, were widely used features for the generation of morphologies, that the placement and connectivity strategies offered by the framework were sufficient to cover the common use cases, and that users were able to implement components to implement their own model-specific tasks.

The component and configuration system proved powerful, and many offspring projects managed to reuse previous models in their entirety, changing only the central configuration file to obtain useful model perturbations for hypothesis

investigation.

Reusing and altering data pipelines, adding/changing cell types, modifying placement and connectivity between them, and putting it all together in a new context for another use case would have required extensive source analysis and modification with imperative code. Thanks to the framework most of the code complexity was entirely transparent to most users who could oversee and understand the model at the diagram level, and zoom in only on well-encapsulated code of interest, and make targetted and well informed code modifications.

Although beneficial, the inversion of control, component code architecture, and project organization imposed by the framework was unfamiliar to most users and represented the largest challenge: the key to adoption of the framework will be a smooth learning curve facilitated by user oriented documentation exposing and introducing them to framework idioms, and demonstrating the long-term advantages of code quality.

In conclusion, work remains to be done to refine the software and documentation to ensure its usability, and the likelihood of adoption. Yet already, the missing coverage of multiscale modeling software tools seen in the state of the art analysis have been adressed succesfully by the developed framework: the software offers a strong framework environment with inversion of control, component encapsulation, code architecture and organization of user code, observing high reuse and understandability in the preliminary models that have been developed. The software facilitates the description of models as a sum of black-box components with a clear separation between the model description and implementation, through the configuration and strategy pattern. The parts of the multiscale workflow not covered by other tools, namely the provenance and incorporation of multimodal multiscale datasets and the generation of parameter values from them (more than their mere specification) is made possible by the framework and put into use succesfully.

6.2 Scientific findings

Through the application of the BSB, several base models were set up for multiple brain regions. These base models then were either used for research into fundamental computational properties, or perturbations were introduced to model disease states.

The cerebellar cortex model forms a leap forward in many aspects: it is the first multicompartmental model that comprises all 3 cortical layers, and all main cell types from the granular input layer with mossy fiber branching, to the sole output of the cortex, namely the Purkinje cell axon, with the main inhibitory interneurons present, forming feedback loops and effecting control over the main

granule-Purkinje cell axis, including several rarely modeled synaptic connections such as the dendrodendritic Golgi cell GABA synapses, electrotonic synapses, and the molecular layer interneuron (MLI) MLI-to-MLI synapses. Other than the completeness, the simulations of the models reveal supporting evidence of the vertical organization of the cerebellar cortex, and demonstrates how several mechanisms contibute to the spatiotemporal focusing of dense bundles of granule cell activation: the mossy fibers contact and activate dense clusters of granule cells, the Golgi cells synchronize themselves with millisecond precision, and inhibit a wide area of granule cells in an oblong shape around the activated cluster, the basket cell inhibits off-bundle Purkinje cells laterally for sharper bundle boundaries, and both MLI through the delays inherent to their disynaptic pathway inhibit the Purkinje cells temporally right after activation for more temporally focused responses to granule cell activation.

The cerebellar cortex model is now being reused for multiple investigations by multiple collaborators: through parameter and morphology modifications a human variant is being reconstructed, which through differential investigation could highlight human-only properties and their function; through parameter and membrane mechanism modifications the IB2K mouse model for autism is being studied; through atlas-alignment and morphology generation the non-developable surface of the cerebellar cortex is being reconstructed, and the highly detailed reconstruction will be used to study how the same microcircuit organization can lead to different functions in different regions, specifically, how the declive's lobule VI contributes to emotional control; through composition with a model for the inferior olive the error input to the cerebellum can be included and cerebellar learning can be investigated in novel ways.

Many of these projects would have been considerably harder without a reusable and modifiable cerebellar cortex model.

Chapter 7

Supplementary Material

```yaml
name: Skeleton configuration
storage:
  engine: hdf5
  root: network.hdf5
network:
  x: 100
  y: 100
  z: 100
regions: {}
partitions: {}
cell_types: {}
placement: {}
after_placement: {}
connectivity: {}
after_connectivity: {}
simulations: {}
```

Code Snippet 7.1: Example of a minimal configuration, containing a hints for the storage and network size of the model reconstructions, and the root nodes that host the declarations of a user's components.

```yaml
name: Starting example
storage:
  engine: hdf5
  root: network.hdf5
network:
  x: 400
  y: 400
  z: 600
partitions:
  base_layer:
    type: layer
    thickness: 100
cell_types:
  base_type:
    spatial:
      radius: 2.5
      density: 0.00039
    plotting:
      display_name: Template cell
      color: "#E62314"
      opacity: 0.5
placement:
  example_placement:
    strategy: bsb.placement.ParticlePlacement
    cell_types:
    - base_type
    partitions:
    - base_layer
connectivity: {}
simulations: {}
```

Code Snippet 7.2: Example of a starting configuration, containing a single layer, cell type, and placement strategy according to which the cells will be placed into the layer.

```
{
  "cell_types": {
    "$import": [
      {
        "ref": "cortex/cell_types.json#/"
      },
      {
        "ref": "dcn/cell_types.json#/",
        "values": ["DCN", "DCN_interneuron"]
      },
    ],
    "DCN_modified": {
      "$ref": "#./DCN",
      "spatial": {
        "density": 3e-4
      }
    }
  }
}
```

Code Snippet 7.3: Example of multi-document organization using the JSON parser's extensions. The `"$import"` key here specifies 2 references that will be import the root nodes of 2 cell type documents into the cell types node of this document. The first import imports all the remote nodes, while the second import specifies 2 specific nodes to import. The `"DCN_modified"` node then references the DCN cell type node (which was imported), copying its declaration and overwriting the density.

```
class CellTypeReference(Reference):
    def __call__(self, root, here):
        return root.cell_types

    def is_ref(self, value):
        from ..cell_types import CellType

        return isinstance(value, CellType)
```

Code Snippet 7.4: Example of a reference handler. Using the `root` node, and current `here` node, the reference handler can point to the location in the component tree from where to fetch the reference. The `is_ref` method differentiates a config value that needs to be resolved from an already resolved reference.

7.1 Cerebellar cortex model

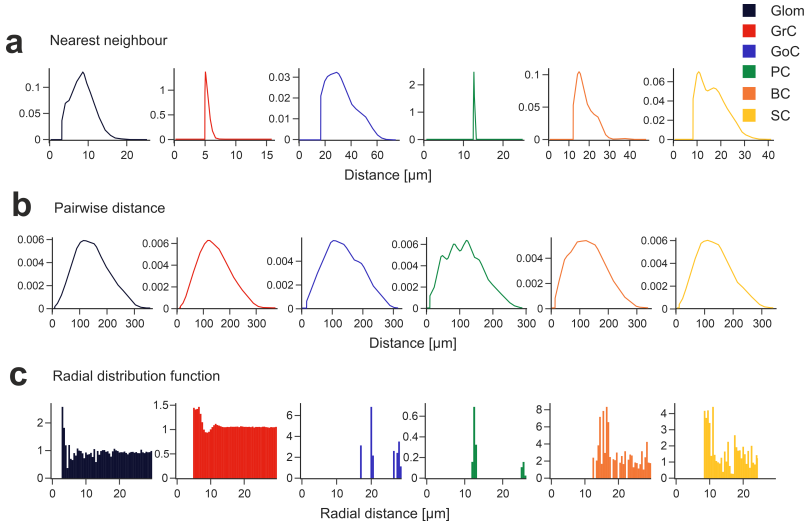


Figure 7.1: Placement metrics. Cell placement is assessed using various metrics for each population, 7 including (a) Nearest Neighbor distance, (b) Pairwise Distance, (c) Radial Distribution Function. These 8 metrics show realistic cell positioning.

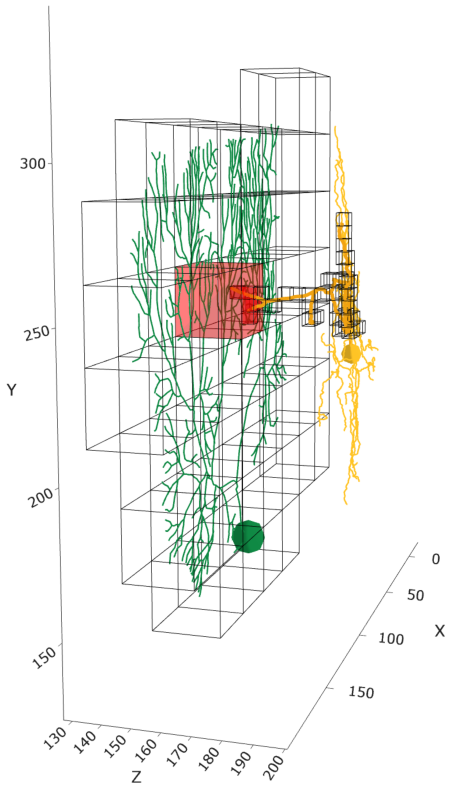


Figure 7.2: Connecting SC-PC by voxel intersection. A mesh of adjacent voxels is used to enwrap 13 the axon of a stellate cell (50 cubes with 4.6 μm side) and the dendritic tree of a PC (50 cubes with 26 14 μm side). The intersecting voxels are in red. The synapses are located on compartments within the 15 intersecting voxels.

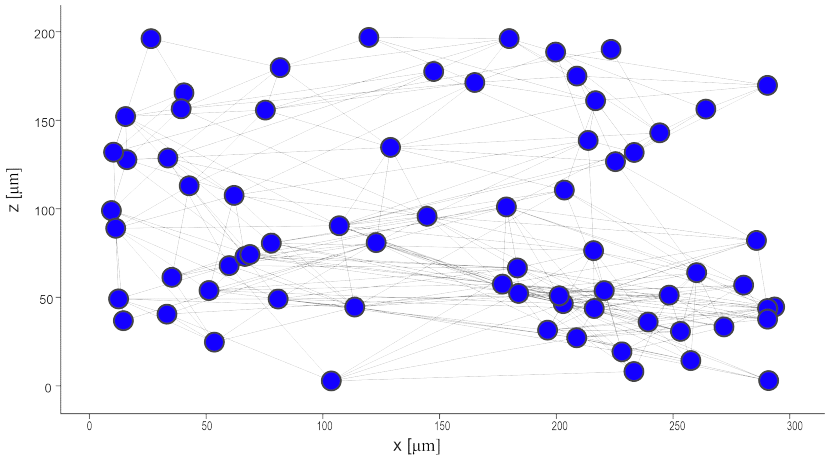


Figure 7.3: Coupling graph for GoCs. Each of the 70 blue dots represents a GoC in the horizontal 20 plane. The grey edges represent connections through gap junctions.

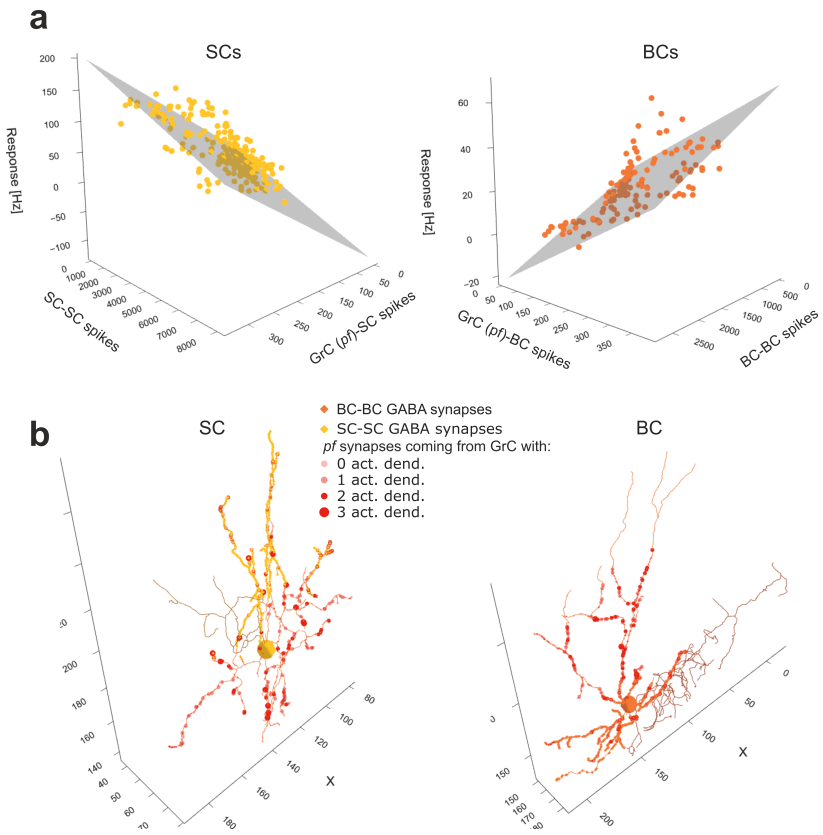


Figure 7.4: MLI responses to mf burst. (a) Multiple linear regression of SCs and BCs in responses to 25 the mf burst against the number of synaptic spikes from pfs and from other SCs or BCs. (b) One SC and 26 one BC crossed by an active pf beam are represented in 3D. The GABAergic synapses from other SCs 27 or BCs are also indicated. Bigger markers correspond to presynaptic GrCs more activated by the mf 28 burst. In this example, the SC receives 8% and the BC 7.5% of their pf synapses from GrCs with at 29 least 2 active dendrites.

7.1.1 Mouse configuration file

```yaml
name: DBBS Mouse cerebellar cortex
storage:
  engine: hdf5
  root: cerebellum.hdf5
network:
  chunk_size: [50, 50, 50]
  x: 300
  y: 200
  z: 295
packages:
  - cerebellum==0.1.4
morphologies:
  - morphologies/GolgiCell.swc
  - morphologies/GranuleCell.swc
  - morphologies/GolgiCell.asc
  - morphologies/BasketCell.swc
  - morphologies/PurkinjeCell.swc
  - morphologies/StellateCell.swc
partitions:
  dcn_layer:
    stack_index: 0
    thickness: 600
  granular_layer:
    stack_index: 1
    thickness: 322
  purkinje_layer:
    stack_index: 2
    thickness: 15
  b_molecular_layer:
    stack_index: 3
    thickness: 50
  t_molecular_layer:
    stack_index: 4
    thickness: 100
```

```yaml
regions:
  cerebellar_cortex:
    type: stack
    children:
      - granular_layer
      - purkinje_layer
      - b_molecular_layer
      - t_molecular_layer
      - dcn_layer
cell_types:
  granule_cell:
    spatial:
      density: 0.0039
      geometry:
        pf_height: 126
        pf_height_sd: 15
      morphologies:
        - GranuleCell
      radius: 2.5
    plotting:
      display_name: Granule cell
      opacity: 0.3
  mossy_fibers:
    spatial:
      count_ratio: 0.05
      radius: 1
      relative_to: glomerulus
  glomerulus:
    spatial:
      density: 0.0003
      radius: 1.5
    plotting:
      display_name: Glomerulus
  purkinje_cell:
    spatial:
      morphologies:
        - PurkinjeCell
      planar_density: 0.0017
      radius: 7.5
    plotting:
      display_name: Purkinje cell
```

```
  golgi_cell:
    spatial:
      density: 0.000009
      geometry:
        axon_radius: 160
      morphologies:
        - GolgiCell
      radius: 8
    plotting:
      display_name: Golgi cell
  stellate_cell:
    spatial:
      density: 0.00005
      morphologies:
        - StellateCell
      radius: 4
    plotting:
      display_name: Stellate cell
  basket_cell:
    spatial:
      density: 0.00005
      morphologies:
        - BasketCell
      radius: 6
    plotting:
      display_name: Basket cell
  dcn_cell:
    spatial:
      count_ratio: 0.090909
      radius: 10
      relative_to: purkinje_cell
    plotting:
      display_name: DCN cell
placement:
  granular_layer_innervation:
    strategy: bsb.placement.RandomPlacement
    partitions:
      - granular_layer
    cell_types:
      - mossy_fibers
```

```
  granular_layer_placement:
    strategy: bsb.placement.RandomPlacement
    partitions:
      - granular_layer
    cell_types:
      - granule_cell
      - golgi_cell
      - glomerulus
  purkinje_layer_placement:
    strategy: bsb.placement.ParallelArrayPlacement
    partitions:
      - purkinje_layer
    cell_types:
      - purkinje_cell
    spacing_x: 130
    angle: 70
  basket_layer_placement:
    strategy: bsb.placement.RandomPlacement
    partitions:
      - b_molecular_layer
    cell_types:
      - basket_cell
  stellate_layer_placement:
    strategy: bsb.placement.RandomPlacement
    partitions:
      - t_molecular_layer
    cell_types:
      - stellate_cell
  dcn_layer_placement:
    strategy: bsb.placement.ParticlePlacement
    partitions:
      - dcn_layer
    cell_types:
      - dcn_cell
connectivity:
  mossy_fibers_to_glomerulus:
    strategy: cerebellum.connectome.mossy_glomerulus.ConnectomeMossyGlomerulus
    presynaptic:
      cell_types:
        - mossy_fibers
    postsynaptic:
      cell_types:
        - glomerulus
    x_length: 60
    z_length: 20
```

```yaml
glomerulus_to_granule:
  strategy: cerebellum.connectome.glomerulus_granule.ConnectomeGlomerulusGranule
  presynaptic:
    cell_types:
      - glomerulus
  postsynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - dendrites
  prepresynaptic:
    cell_types:
      - mossy_fibers
  x_length: 100
  z_length: 100
  max_radius: 40
  convergence: 4
glomerulus_to_golgi:
  strategy: cerebellum.connectome.glomerulus_golgi.ConnectomeGlomerulusGolgi
  presynaptic:
    cell_types:
      - glomerulus
  postsynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - basal_dendrites
  radius: 50
golgi_to_glomerulus:
  strategy: >-
    cerebellum.connectome.golgi_glomerulus_granule.ConnectomeGolgiGlomerulusGranule
  presynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - axon
  postsynaptic:
    cell_types:
      - granule_cell
  intermediate:
    cell_types:
      - glomerulus
  radius: 50
  convergence: 40
```

```yaml
golgi_to_golgi:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - axon
  postsynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - basal_dendrites
  affinity: 0.5
  contacts:
    distribution: norm
    loc: 160
    scale: 5
gap_goc:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - basal_dendrites
  postsynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - basal_dendrites
  affinity: 0.2
  contacts:
    distribution: norm
    loc: 3
    scale: 1
ascending_axon_to_golgi:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - basal_ascending_axondendrites
  postsynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - basal_dendrites
```

```yaml
parallel_fiber_to_golgi:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - parallel_fiber
  postsynaptic:
    cell_types:
      - golgi_cell
    morphology_labels:
      - apical_dendrites
  affinity: 0.1
ascending_axon_to_purkinje:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - ascending_axon
  postsynaptic:
    cell_types:
      - purkinje_cell
    morphology_labels:
      - aa_targets
  affinity: 0.1
parallel_fiber_to_purkinje:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - parallel_fiber
  postsynaptic:
    cell_types:
      - purkinje_cell
    morphology_labels:
      - pf_targets
  affinity: 0.1
```

```
parallel_fiber_to_basket:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - parallel_fiber
  postsynaptic:
    cell_types:
      - basket_cell
    morphology_labels:
      - dendrites
  affinity: 0.1
parallel_fiber_to_stellate:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - granule_cell
    morphology_labels:
      - parallel_fiber
  postsynaptic:
    cell_types:
      - stellate_cell
    morphology_labels:
      - dendrites
  affinity: 0.1
stellate_to_purkinje:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - stellate_cell
    morphology_labels:
      - axon
  postsynaptic:
    cell_types:
      - purkinje_cell
    morphology_labels:
      - sc_targets
  affinity: 0.1
basket_to_purkinje:
  strategy: bsb.connectivity.VoxelIntersection
  presynaptic:
    cell_types:
      - basket_cell
    morphology_labels:
      - axon
  postsynaptic:
    cell_types:
      - purkinje_cell
    morphology_labels:
      - soma
```

```
  stellate_to_stellate:
    strategy: bsb.connectivity.VoxelIntersection
    presynaptic:
      cell_types:
        - stellate_cell
      morphology_labels:
        - axon
    postsynaptic:
      cell_types:
        - stellate_cell
      morphology_labels:
        - dendrites
    affinity: 0.5
    contacts:
      distribution: norm
      loc: 100
      scale: 4
  basket_to_basket:
    strategy: bsb.connectivity.VoxelIntersection
    presynaptic:
      cell_types:
        - basket_cell
      morphology_labels:
        - axon
    postsynaptic:
      cell_types:
        - basket_cell
      morphology_labels:
        - dendrites
    affinity: 0.5
    contacts:
      distribution: norm
      loc: 100
      scale: 4
  purkinje_to_dcn:
    strategy: bsb.connectivity.AllToAll
    presynaptic: { cell_types: [ purkinje_cell ], morphology_labels: [ axon ] }
    postsynaptic: { cell_types: [ dcn_cell ] }
  mossy_fiber_to_dcn:
    strategy: bsb.connectivity.AllToAll
    presynaptic: { cell_types: [ mossy_fibers ] }
    postsynaptic: { cell_types: [ dcn_cell ] }
after_placement:
  microzones:
    strategy: cerebellum.postprocessing.LabelMicrozones
    targets:
      - purkinje_cell
  aa_lengths:
    strategy: cerebellum.postprocessing.AscendingAxonLengths
  dcn_rotations:
    strategy: cerebellum.postprocessing.DCNRotations
```

Code Snippet 7.5: Configuration in YAML of the cerebellar cortex model. The configured components come from a mixture of framework provided components (`strategy` attributes starting with `bsb.`), and versioned user code (`strategy` attributes starting with `cerebellum.`)

```yaml
partitions:
  VAL:
    type: allen
    struct_name: VAL
    sources:
      - depth:
        file: https://some-atlas.org/annotation.nrrd
        pipeline:
          - func: atlas.orientation
            parameters:
              # Estimation roughness
              - 2.4
              # Corner protocol
              - "extrapolate"
          - atlas.depth_from_orientation
      - inhib:
          $ref: pipelines/nissl-to-densities.yaml#/
          file: https://some-atlas.org/inh-nissl.nrrd
      - exc:
          $ref: pipelines/nissl-to-densities.yaml#/
          file: https://some-atlas.org/exc-nissl.nrrd
# pipelines/nissl-to-densities.yaml
pipeline:
  - change_reference_space
  - func: atlas.blur
    # sigma
    parameters: [3]
  - func: atlas.erode
    # kernel
    parameters: [
        [0, 1, 0],
        [1, 1, 1],
        [0, 1, 0]
    ]
```

Code Snippet 7.6: Example of an Allen-atlas-connected voxelset calculating layer depth, inhibitory and excitatory cell density data values for each voxel in the selected atlas structure, by fetching data from the fictitious URL `https://some-atlas.org/` (top snippet). The `"$ref"`s reuse the pipeline steps defined in `pipelines/nissl-to-densities.yaml`, the bottom snippet.

```
from arborize import define_model

BasketCellModel = define_model(
    {
        "synapse_types": {
            "AMPA": {
                "tau_facil": 54,
                "tau_rec": 35.1,
                "tau_1": 6,
                "gmax": 1200,
                "U": 0.4,
            },
            ("NMDA", "stellate"): {
                "tau_facil": 5,
                "tau_rec": 8,
                "tau_1": 1,
                "gmax": 5000,
                "U": 0.15,
            },
            "GABA": {
                "tau_facil": 0,
                "tau_rec": 38.7,
                "tau_1": 1,
                "gmax": 3200,
                "U": 0.42,
                "Erev": -65,
            },
        },
```

```
"cable_types": {
    "soma": {
        "cable": {"Ra": 122, "cm": 1},
        "ions": {
            "na": {"rev_pot": 60},
            "k": {"rev_pot": -80},
            "ca": {"rev_pot": 137.5},
            "h": {"rev_pot": -34},
        },
        "mechanisms": {
            "Leak": {"e": -60, "gmax": 3e-05},
            "Nav1_1": {"gbar": 0.10946415489712},
            "Cav3_2": {"gcabar": 0.0006295539},
            "Cav3_3": {"pcabar": 0.0007543986},
            "Kir2_3": {"gkbar": 0.0012770833},
            "Kv3_4": {"gkbar": 0.0277114781},
            "Kca1_1": {"gbar": 0.0044251081},
            "Cav2_1": {"pcabar": 0.000846789},
            "HCN1": {"gbar": 0.0006902196},
            "cdp5": {"TotalPump": 1e-09},
        },
    },
    "dendrites": {
        "cable": {"Ra": 122, "cm": 1},
        "ions": {"k": {"rev_pot": -80}},
        "mechanisms": {
            "Leak": {"e": -60, "gmax": 3e-05},
            "Cav2_1": {"pcabar": 0.0004965596},
            "Kca1_1": {"gbar": 0.0020575902},
            "Kv1_1": {"gbar": 0.0285137286},
            "cdp5": {"TotalPump": 1e-09},
        },
    },
```

```
            "axon": {
                "cable": {"Ra": 122, "cm": 1},
                "ions": {
                    "na": {"rev_pot": 60},
                    "k": {"rev_pot": -80},
                    "h": {"rev_pot": -34},
                },
                "mechanisms": {
                    "Leak": {"e": -60, "gmax": 3e-05},
                    "Kv1_1": {"gbar": 0.0069654709},
                    "Nav1_6": {"gbar": 0.0072983226},
                    "Kv3_4": {"gkbar": 0.0151487764},
                    "HCN1": {"gbar": 0.0034633208},
                    "cdp5": {},
                },
            },
            "axon_initial_segment": {
                "cable": {"Ra": 122, "cm": 1},
                "ions": {
                    "na": {"rev_pot": 60},
                    "k": {"rev_pot": -80},
                    "h": {"rev_pot": -34},
                },
                "mechanisms": {
                    "Leak": {"e": -60, "gmax": 3e-05},
                    "HCN1": {"gbar": 0.0048096086},
                    "Nav1_6": {"gbar": 0.5724695612},
                    "Kv1_1": {"gbar": 0.0827297077},
                    "Kv3_4": {"gkbar": 0.0300388404},
                    "cdp5": {},
                },
            },
        },
    },
    use_defaults=True,
)
```

Code Snippet 7.7: Example of the basket cell description using *arborize*.

Bibliography

[1] John P. A. Ioannidis. Why Most Published Research Findings Are False. *PLOS Medicine*, 2(8):e124, August 2005.

[2] Henry Markram, Karlheinz Meier, Thomas Lippert, Sten Grillner, Richard Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, Seth Grant, Jean-Pierre Changeux, and Alois Saria. Introducing the Human Brain Project. *Procedia Computer Science*, 7:39–42, January 2011.

[3] Stefano Masoli, Sergio Solinas, and Egidio D'Angelo. Action potential processing in a detailed Purkinje cell model reveals a critical role for axonal compartmentalization. *Frontiers in Cellular Neuroscience*, 9:47, 2015.

[4] Stefano Masoli, Alessandra Ottaviani, Stefano Casali, and Egidio D'Angelo. Cerebellar Golgi cell models predict dendritic processing and mechanisms of synaptic plasticity. *PLOS Computational Biology*, 16(12):e1007937, December 2020.

[5] Stefano Masoli, Marialuisa Tognolina, Umberto Laforenza, Francesco Moccia, and Egidio D'Angelo. Parameter tuning differentiates granule cell subtypes enriching transmission properties at the cerebellum input stage. *Communications Biology*, 3(1):1–12, May 2020.

[6] Martina Francesca Rizza, Francesca Locatelli, Stefano Masoli, Diana Sánchez-Ponce, Alberto Muñoz, Francesca Prestori, and Egidio D'Angelo. Stellate cell computational modeling predicts signal filtering in the molecular layer circuit of cerebellum. *Scientific Reports*, 11(1):3873, February 2021.

[7] Stefano Masoli and Egidio D'Angelo. Synaptic Activation of a Detailed

Purkinje Cell Model Predicts Voltage-Dependent Control of Burst-Pause Responses in Active Dendrites. *Frontiers in Cellular Neuroscience*, 11, 2017.

[8] Sergio Solinas, Thierry Nieus, and Egidio D'Angelo. A Realistic Large-Scale Model of the Cerebellum Granular Layer Predicts Circuit Spatio-Temporal Filtering Properties. *Frontiers in Cellular Neuroscience*, 4:12, May 2010.

[9] Lisa Mapelli, Sergio Solinas, and Egidio D'Angelo. Integration and regulation of glomerular inhibition in the cerebellar granular layer circuit. *Frontiers in Cellular Neuroscience*, 8:55, 2014.

[10] Stefano Casali, Elisa Marenzi, Chaitanya Medini, Claudia Casellato, and Egidio D'Angelo. Reconstruction and Simulation of a Scaffold Model of the Cerebellar Network. *Frontiers in Neuroinformatics*, 13:37, 2019.

[11] Egidio D'Angelo, Sergio Solinas, Jonathan Mapelli, Daniela Gandolfi, Lisa Mapelli, and Francesca Prestori. The cerebellar Golgi cell and spatiotemporal organization of granular layer activity. *Frontiers in Neural Circuits*, 7:93, May 2013.

[12] Petruţ A. Bogdan, Beatrice Marcinnò, Claudia Casellato, Stefano Casali, Andrew G. D. Rowley, Michael Hopkins, Francesco Leporati, Egidio D'Angelo, and Oliver Rhodes. Towards a Bio-Inspired Real-Time Neuromorphic Cerebellum. *Frontiers in Cellular Neuroscience*, 15:622870, 2021.

[13] Michael Schirner, Lia Domide, Dionysios Perdikis, Paul Triebkorn, Leon Stefanovski, Roopa Pai, Paula Prodan, Bogdan Valean, Jessica Palmer, Chloê Langford, André Blickensdörfer, Michiel van der Vlag, Sandra Diaz-Pier, Alexander Peyser, Wouter Klijn, Dirk Pleiter, Anne Nahm, Oliver Schmid, Marmaduke Woodman, Lyuba Zehl, Jan Fousek, Spase Petkoski, Lionel Kusch, Meysam Hashemi, Daniele Marinazzo, Jean-François Mangin, Agnes Flöel, Simisola Akintoye, Bernd Carsten Stahl, Michael Cepic, Emily Johnson, Gustavo Deco, Anthony R. McIntosh, Claus C. Hilgetag, Marc Morgan, Bernd Schuller, Alex Upton, Colin McMurtrie, Timo Dickscheid, Jan G. Bjaalie, Katrin Amunts, Jochen Mersmann, Viktor Jirsa, and Petra Ritter. Brain simulation as a cloud service: The Virtual Brain on EBRAINS. *NeuroImage*, 251:118973, May 2022.

[14] Sadaf Alam, Javier Bartolome, Sanzio Bassini, Michele Carpene, Mirko Cestari, Frederic Combeau, Sergi Girona, Stefano Gorini, Giuseppe Fiameni, Björn Hagemeier, Andreas Herten, Nikoleta Kiapidou, Wouter

Klijn, Dorian Krause, Jacques-Charles Lafoucriere, Cerlane Leong, Thomas Leibovici, Thomas Lippert, Colin McMurtrie, Pavel Mezentsev, Anne Nahm, Boris Orth, Dirk Pleiter, Thomas Schulthess, Benedikt von St. Vieth, Debora Testi, and Gilles Wiber. Fenix: Distributed e-Infrastructure Services for EBRAINS. In Katrin Amunts, Lucio Grandinetti, Thomas Lippert, and Nicolai Petkov, editors, *Brain-Inspired Computing*, Lecture Notes in Computer Science, pages 81–89, Cham, 2021. Springer International Publishing.

[15] Luca L. Bologna, Roberto Smiriglia, Dario Curreri, and Michele Migliore. The EBRAINS NeuroFeatureExtract: An Online Resource for the Extraction of Neural Activity Features From Electrophysiological Data. *Frontiers in Neuroinformatics*, 15:713899, August 2021.

[16] Shailesh Appukuttan, Luca L. Bologna, Felix Schürmann, Michele Migliore, and Andrew P. Davison. EBRAINS Live Papers - Interactive Resource Sheets for Computational Studies in Neuroscience. *Neuroinformatics*, 21(1):101–113, January 2023.

[17] Wouter Klijn, Muhammad Fahad, Kim Sontheimer, Cristian Jimenez-Romero, Rolando Ingles Chavez, Sandra Diaz, Jochen Martin Eppler, Lena Oden, and Abigail Morrison. *Multiscale Brain Co-simulation in the Human Brain Project: EBRAINS Tools for in-Transit Simulation and Analysis*. 2021.

[18] Sandra Diaz, Claudia Bachmann, and Wouter Klijn. *Using EBRAINS for Your Use Cases*. 2021.

[19] Luca Leonardo Bologna, Roberto Smiriglia, Carmen Alina Lupascu, Shailesh Appukuttan, Andrew P. Davison, Genrich Ivaska, Jean-Denis Courcol, and Michele Migliore. The EBRAINS Hodgkin-Huxley Neuron Builder: An online resource for building data-driven neuron models. *Frontiers in Neuroinformatics*, 16, 2022.

[20] Miryam Naddaf. Europe spent €600 million to recreate the human brain in a computer. How did it go? *Nature*, 620(7975):718–720, August 2023.

[21] Brain Scaffold Builder - Tools. https://www.ebrains.eu/tools/bsb.

[22] Tina Roostaei, Arash Nazeri, Mohammad Ali Sahraian, and Alireza Minagar. The human cerebellum: A review of physiologic neuroanatomy. *Neurologic Clinics*, 32(4):859–869, November 2014.

[23] Johannes Sobotta. *Atlas and Text-Book of Human Anatomy v. 2, 1906.* W.B. Saunders Company, 1909.

[24] Anamaria Sudarov and Alexandra L. Joyner. Cerebellum morphogenesis: The foliation pattern is orchestrated by multi-cellular anchoring centers. *Neural Development*, 2(1):26, December 2007.

[25] Martin I. Sereno, Jörn Diedrichsen, Mohamed Tachrount, Guilherme Testa-Silva, Helen d'Arceuil, and Chris De Zeeuw. The human cerebellum has almost 80% of the surface area of the neocortex. *Proceedings of the National Academy of Sciences*, 117(32):19538–19543, August 2020.

[26] RB Ivry and JA Fiez. Cerebellar contributions to cognition and imagery. *The new cognitive neurosciences*, 2:999–1011, 2000. The cerebellar cortex is traditionally divided into three subregions - the archicerebellum, the paleocerebellum, and the neocerebellum.

Especially the neocerebellum seems to be the substrate for influencing cognition.

Tremendous expansion in primates.

[27] Henrietta C. Leiner, Alan L. Leiner, and Robert S. Dow. Does the cerebellum contribute to mental skills? *Behavioral Neuroscience*, 100(4):443–454, 1986. Signals from the older part of the dentate nucleus certainly help the frontal motor cortex to effect the skilled manipulation of muscles, and signals from the newest part of the dentate nucleus may help the frontal association cortex to effect the skilled manipulation of information or ideas. How such mental skills could have evolved in higher primates in the course of phylogenetic and ontogenetic development is shown. The validity of this new concept of cerebellar function can be tested on humans by means of tomographic brain scans.

[28] Gray, Henry, and Warren H Lewis. *Anatomy of the Human Body.* Philadelphia, Lea & Febiger, 1918. https://www.biodiversitylibrary.org/bibliography/20311.

[29] Paul R. Davidson and Daniel M. Wolpert. Widespread access to predictive models in the motor system: A short review. *Journal of Neural Engineering*, 2(3):S313–319, September 2005.

[30] M. Desmurget and S. Grafton. Forward modeling allows feedback control for fast reaching movements. *Trends in Cognitive Sciences*, 4(11):423–431,

November 2000.

[31] Emanuel Todorov and Michael I. Jordan. Optimal feedback control as a theory of motor coordination. *Nature Neuroscience*, 5(11):1226–1235, November 2002.

[32] Peter L. Strick, Richard P. Dum, and Julie A. Fiez. Cerebellum and Nonmotor Function. *Annual Review of Neuroscience*, 32(1):413–434, 2009.

[33] Masao Ito. Cerebellar circuitry as a neuronal machine. *Progress in Neurobiology*, 78(3):272–303, February 2006.

[34] Elizabeth P. Lackey, Detlef H. Heck, and Roy V. Sillitoe. Recent advances in understanding the mechanisms of cerebellar granule cell development and function and their contribution to behavior. *F1000Research*, 7:F1000 Faculty Rev–1142, July 2018.

[35] Camillo Golgi. *Sulla fina anatomia degli organi centrali del sistema nervoso*. S. Calderini, 1885.

[36] Duane E. Haines. *Neuroanatomy: An Atlas of Structures, Sections, and Systems*. Lippincott Williams & Wilkins, 9 edition, 2017.

[37] Freida L Carson and Christie Hladik. *Histotechnology: A Self-Instructional Text*. American Society for Clinical Pathology, 2008.

[38] Eric R. Kandel, James H. Schwartz, and Thomas M. Jessell. *Principles of Neural Science*. McGraw-Hill, 5 edition, 2013.

[39] John C. Eccles, Masao Ito, and János Szentágothai. *The Cerebellum as a Neuronal Machine*. Springer Berlin Heidelberg, 1967.

[40] Santiago Ramón y Cajal. *Histologie Du Système Nerveux de l'Homme et Des Vertébrés*. Maloine, Paris, 1909.

[41] S. L. Palay and V. Chan-Palay. *Cerebellar Cortex: Cytology and Organization*. Springer, 1974.

[42] R. L. Jakab and J. Hámori. Quantitative morphology and synaptology of cerebellar glomeruli in the rat. *Anatomy and Embryology*, 179(1):81–88, October 1988.

[43] Victoria Chan-Palay and Sanford L. Palay. The form of velate astrocytes in the cerebellar cortex of monkey and rat: High voltage electron microscopy of rapid Golgi preparations. *Zeitschrift für Anatomie und Entwicklungsgeschichte*, 138(1):1–19, January 1972.

[44] Cassie S. Mitchell and Robert H. Lee. Synaptic glutamate spillover increases NMDA receptor reliability at the cerebellar glomerulus. *Journal of theoretical biology*, 289:217–224, November 2011.

[45] Simon J. Mitchell and R. Angus Silver. GABA Spillover from Single Inhibitory Axons Suppresses Low-Frequency Excitatory Transmission at the Cerebellar Glomerulus. *Journal of Neuroscience*, 20(23):8651–8658, December 2000.

[46] Roy V. Sillitoe, Seung-Hyuk Chung, Jean-Marc Fritschy, Monica Hoy, and Richard Hawkes. Golgi Cell Dendrites Are Restricted by Purkinje Cell Stripe Boundaries in the Adult Mouse Cerebellar Cortex. *Journal of Neuroscience*, 28(11):2820–2826, March 2008.

[47] G. Andersson and O. Oscarsson. Climbing fiber microzones in cerebellar vermis and their projection to different groups of cells in the lateral vestibular nucleus. *Experimental Brain Research*, 32(4):565–579, August 1978.

[48] W. W. Chambers and J. M. Sprague. Functional localization in the cerebellum. I. Organization in longitudinal cortico-nuclear zones and their contribution to the control of posture, both extrapyramidal and pyramidal. *The Journal of Comparative Neurology*, 103(1):105–129, August 1955.

[49] Andreas Vesalius. *De Humani Corporis Fabrica*. Johannes Oporinus, Basel, 1543.

[50] Thomas Willis. *Cerebri Anatome*. Martyn and Allestry, Oxford, 1664.

[51] Gottfried Wilhelm Leibniz. *Protogaea*. Nicolai Foersteri, Berlin, 1749.

[52] Johann Friedrich Meckel. *Handbuch Der Menschlichen Anatomie*. Vandenhoeck und Ruprecht, Göttingen, 1816.

[53] Larry R. Squire, editor. *The History of Neuroscience in Autobiography, Volume 2*. Society for Neuroscience, Washington, DC, 2001.

[54] Michael S. Gazzaniga. *The Mind's Past*. University of California Press, Berkeley, CA, 2000.

[55] S. Finger, F. Boller, and K.L. Tyler, editors. *The Oxford Handbook of the History of Neuroscience*. Oxford University Press, Oxford, UK, 2008.

[56] A.S. David. *The Search for the Mind: A New History of the Brain*. Harry N. Abrams, New York, NY, 2000.

[57] V.S. Ramachandran. *The Tell-Tale Brain: A Neuroscientist's Quest for What Makes Us Human*. W. W. Norton & Company, New York, NY, 2011.

[58] Paul Broca. Remarques sur le siège de la faculté du langage articulé, suivies d'une observation d'aphémie (perte de la parole). *Bulletin de la Société Anatomique*, 6:330–357, 1861.

[59] Carl Wernicke. *Der Aphasische Symptomencomplex: Eine Psychologische Studie Auf Anatomischer Basis*. Cohn & Weigert, 1874.

[60] Gustav Fritsch and Eduard Hitzig. Über die elektrische Erregbarkeit des Grosshirns. *Archiv für Anatomie, Physiologie und wissenschaftliche Medicin*, 37(1-2):300–332, 1870.

[61] François Magendie. Sur les fonctions du cerveau. *Gazette Médicale de Paris*, 1:141–145, 1822.

[62] David Marr. A theory of cerebellar cortex. *The Journal of Physiology*, 202(2):437–470, 1969.

[63] James S. Albus. A theory of cerebellar function. *Mathematical Biosciences*, 10(1):25–61, February 1971.

[64] Masao Ito. Neurophysiological aspects of the cerebellar motor control system. *International journal of neurology*, 7:162–176, 1970.

[65] M. Ito. *The Cerebellum and Neural Control*. Raven Press, 1984.

[66] Mitsuo Kawato, Shogo Ohmae, Huu Hoang, and Terry Sanger. 50 Years Since the Marr, Ito, and Albus Models of the Cerebellum. *Neuroscience*, 462:151–174, May 2021.

[67] Mario Manto, James M. Bower, Adriana Bastos Conforto, José M. Delgado-García, Suzete Nascimento Farias da Guarda, Marcus Gerwig, Christophe Habas, Nobuhiro Hagura, Richard B. Ivry, Peter Mariën, Marco Molinari, Eiichi Naito, Dennis A. Nowak, Nordeyn Oulad Ben Taib, Denis Pelisson, Claudia D. Tesche, Caroline Tilikete, and Dagmar Timmann. Consensus Paper: Roles of the Cerebellum in Motor Control—The Diversity of Ideas on Cerebellar Involvement in Movement. *The Cerebellum*, 11(2):457–487, June 2012. clinical deficits exhibited by cerebellar patients and which are characterized by disturbances in accuracy and coordination: disorders of eye movements, disorders of speech, disorders of limb movements, impairments of posture/gait as well as cognitive deficits

The cerebellar structures controlling eye movements include the so-called oculomotor vermis (lobules VI and VII) and fastigius nucleus, crus I–II of ansiform lobule, flocculus and paraflocculus, uvula, and nodulus. Speech is controlled by the superior paravermal region, the intermediate cerebellar cortex, and the dentate nucleus. Limb movements are under the supervision of the dentate nucleus, the interpositus nucleus, the intermediate cerebellar cortex, and the lateral cerebellar cortex. Stance/gait is controlled by the medial and intermediate cerebellum. Cognitive operations are mainly controlled by the posterior lobe (posterolateral cerebellum) and cerebellar nuclei (mainly parts of dentate nuclei)

7.1.2 The Role of the Cerebellum in Oculomotor Control

The vestibulocerebellum (flocculus, paraflocculus, nodulus, uvula, tonsil, and cerebellar pyramid) is important for steady gaze holding, smooth pursuit, and the vestibulo-ocular reflex; the oculomotor cerebellum (dorsal vermis—lobules VI and VII—and the underlying fastigial nucleus, as well as ansiform lobe—crus I and crus II) is mainly involved in the control of saccades but also contributes to smooth pursuit and vergence

Eye Stability Control

The control of eye stability corresponds to gaze holding processing, slow phase (VOR, smooth pursuit) instability control, and inhibition of unwanted saccades. The best insight into the role of the cerebellum in eye stability control is illustrated by the appearance of gaze-evoked nystagmus, periodic alternating nystagmus, and square wave saccadic intrusion (SWSI) following cerebellar dysfunction.

These concepts suggest that the cerebellum contributes to timing and sensory acquisition and is involved in the prediction of the sensory consequences of action.

[68] Jeremy D. Schmahmann. The cerebellum and cognition. *Neuroscience Letters*, 688:62–75, January 2019.

[69] Egidio D'Angelo and Stefano Casali. Seeking a unified framework for cerebellar function and dysfunction: From circuit operations to cognition. *Frontiers in Neural Circuits*, 6:116, January 2013.

[70] Henrik Jörntell. Cerebellar Synaptic Plasticity and the Credit Assignment Problem. *Cerebellum (London, England)*, 15(2):104–111, April 2016.

[71] Chris I. De Zeeuw, Stephen G. Lisberger, and Jennifer L. Raymond. Diversity and dynamism in the cerebellum. *Nature Neuroscience*, 24(2):160–167, February 2021.

[72] Masao Ito. Control of mental activities by internal models in the cerebellum. *Nature Reviews Neuroscience*, 9(4):304–313, April 2008.

[73] Zhenyu Gao, Boeke J. van Beugen, and Chris I. De Zeeuw. Distributed synergistic plasticity and cerebellar learning. *Nature Reviews. Neuroscience*, 13(9):619–635, September 2012.

[74] Egidio D'Angelo and Viktor Jirsa. The quest for multiscale brain modeling. *Trends in Neurosciences*, 45(10):777–790, October 2022.

[75] Mohamed Fayad and Douglas Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40, October 1997.

[76] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988. Credited for originating the term open-closed principle.

[77] Kael Dai, Sergey L. Gratiy, Yazan N. Billeh, Richard Xu, Binghuang Cai, Nicholas Cain, Atle E. Rimehaug, Alexander J. Stasik, Gaute T. Einevoll, Stefan Mihalas, Christof Koch, and Anton Arkhipov. Brain Modeling ToolKit: An open source software suite for multiscale modeling of brain circuits. *PLoS Computational Biology*, 16(11):e1008386, November 2020.

[78] Antolik Jan and Davison Andrew. Mozaik: A framework for model construction, simulation, data analysis and visualization for large-scale spiking neural circuit models. *Frontiers in Neuroinformatics*, 7, 2013.

[79] Salvador Dura-Bernal, Benjamin A Suter, Padraig Gleeson, Matteo Cantarelli, Adrian Quintana, Facundo Rodriguez, David J Kedziora, George L Chadderdon, Cliff C Kerr, Samuel A Neymotin, Robert A Mc-

Dougal, Michael Hines, Gordon MG Shepherd, and William W Lytton. NetPyNE, a tool for data-driven multiscale modeling of brain circuits. *eLife*, 8:e44494, April 2019.

[80] Padraig Gleeson, Volker Steuber, and R. Angus Silver. neuroConstruct: A Tool for Modeling Networks of Neurons in 3D Space. *Neuron*, 54(2):219–235, April 2007.

[81] James King, Michael Hines, Sean Hill, Philip Goodman, Henry Markram, and Felix Schürmann. A component-based extension framework for large-scale parallel simulations in NEURON. *Frontiers in Neuroinformatics*, 3, 2009.

[82] Robert C. Cannon, Padraig Gleeson, Sharon Crook, Gautham Ganapathy, Boris Marin, Eugenio Piasini, and R. Angus Silver. LEMS: A language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2. *Frontiers in Neuroinformatics*, 8, 2014.

[83] Ivan Raikov, Robert Cannon, Robert Clewley, Hugo Cornelis, Andrew Davison, Erik De Schutter, Mikael Djurfeldt, Padraig Gleeson, Anatoli Gorchetchnikov, Hans Plesser, Sean Hill, Mike Hines, Birgit Kriener, Yann Le Franc, Chung-Chuan Lo, Abigail Morrison, Eilif Muller, Subhasis Ray, Lars Schwabe, and Botond Szatmary. *NineML: The Network Interchange for Neuroscience Modeling Language*, volume 12. July 2011.

[84] Ines Wichert, Sanghun Jee, Erik De Schutter, and Sungho Hong. Pycabnn: Efficient and extensible software to construct an anatomical basis for a physiologically realistic neural network model. *Frontiers in Neuroinformatics*, 14:31, 2020.

[85] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: A common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2, 2009.

[86] J. J. Johannes Hjorth, Jeanette Hellgren Kotaleski, and Alexander Kozlov. Predicting Synaptic Connectivity for Large-Scale Microcircuit Simulations Using Snudda. *Neuroinformatics*, 19(4):685–701, October 2021.

[87] Kael Dai, Juan Hernando, Yazan N. Billeh, Sergey L. Gratiy, Judit Planas, Andrew P. Davison, Salvador Dura-Bernal, Padraig Gleeson, Adrien Devresse, Benjamin K. Dichter, Michael Gevaert, James G. King, Werner A. H. Van Geit, Arseny V. Povolotsky, Eilif Muller, Jean-Denis Cour-

col, and Anton Arkhipov. The SONATA data format for efficient description of large-scale network models. *PLOS Computational Biology*, 16(2):e1007696, February 2020.

[88] Andrew Davison. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science & Engineering*, 14(4):48–56, July 2012.

[89] Paula Sanz Leon, Stuart Knock, M. Woodman, Lia Domide, Jochen Mersmann, Anthony McIntosh, and Viktor Jirsa. The Virtual Brain: A simulator of primate brain network dynamics. *Frontiers in Neuroinformatics*, 7, 2013.

[90] Samuel Garcia, Domenico Guarino, Florent Jaillet, Todd Jennings, Robert Pröpper, Philipp L. Rautenberg, Chris C. Rodgers, Andrey Sobolev, Thomas Wachtler, Pierre Yger, and Andrew P. Davison. Neo: An object model for handling electrophysiology data in multiple formats. *Frontiers in Neuroinformatics*, 8:10, 2014.

[91] J. M. Bower and D. C. Woolston. Congruence of spatial organization of tactile projections to granule cell and Purkinje cell layers of cerebellar hemispheres of the albino rat: Vertical organization of cerebellar cortex. *Journal of Neurophysiology*, 49(3):745–766, March 1983.

[92] Erich Gamma, editor. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Mass, 1995.

[93] N. Abi Akar, B. Cumming, V. Karakasis, A. Küsters, W. Klijn, A. Peyser, and S. Yates. Arbor — A morphologically-detailed neural network simulation library for contemporary high-performance computing architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 274–282, February 2019.

[94] Robin De Schepper, Alice Geminiani, Stefano Masoli, Martina Francesca Rizza, Alberto Antonietti, Claudia Casellato, and Egidio D'Angelo. Model simulations unveil the structure-function-dynamics relationship of the cerebellar cortical microcircuit. *Communications Biology*, 5(1):1240, November 2022.

[95] Michael A. Arbib and Péter Érdi. Précis of iNeural organization: Structure, function, and dynamics/i. *The Behavioral and brain sciences*, 23(4):513–533, August 2000.

[96] Egidio D'Angelo and Claudia Gandini Wheeler-Kingshott. Modelling the brain: Elementary components to explain ensemble functions. *La Rivista del Nuovo Cimento*, 40(7):297–333, July 2017.

[97] Henry Markram, Eilif Muller, Srikanth Ramaswamy, Michael W. Reimann, Marwan Abdellah, Carlos Aguado Sanchez, Anastasia Ailamaki, Lidia Alonso-Nanclares, Nicolas Antille, Selim Arsever, Guy Antoine Atenekeng Kahou, Thomas K. Berger, Ahmet Bilgili, Nenad Buncic, Athanassia Chalimourda, Giuseppe Chindemi, Jean-Denis Courcol, Fabien Delalondre, Vincent Delattre, Shaul Druckmann, Raphael Dumusc, James Dynes, Stefan Eilemann, Eyal Gal, Michael Emiel Gevaert, Jean-Pierre Ghobril, Albert Gidon, Joe W. Graham, Anirudh Gupta, Valentin Haenel, Etay Hay, Thomas Heinis, Juan B. Hernando, Michael Hines, Lida Kanari, Daniel Keller, John Kenyon, Georges Khazen, Yihwa Kim, James G. King, Zoltan Kisvarday, Pramod Kumbhar, Sébastien Lasserre, Jean-Vincent Le Bé, Bruno R.C. Magalhães, Angel Merchán-Pérez, Julie Meystre, Benjamin Roy Morrice, Jeffrey Muller, Alberto Muñoz-Céspedes, Shruti Muralidhar, Keerthan Muthurasa, Daniel Nachbaur, Taylor H. Newton, Max Nolte, Aleksandr Ovcharenko, Juan Palacios, Luis Pastor, Rodrigo Perin, Rajnish Ranjan, Imad Riachi, José-Rodrigo Rodríguez, Juan Luis Riquelme, Christian Rössert, Konstantinos Sfyrakis, Ying Shi, Julian C. Shillcock, Gilad Silberberg, Ricardo Silva, Farhan Tauheed, Martin Telefont, Maria Toledo-Rodriguez, Thomas Tränkler, Werner Van Geit, Jafet Villafranca Díaz, Richard Walker, Yun Wang, Stefano M. Zaninetta, Javier DeFelipe, Sean L. Hill, Idan Segev, and Felix Schürmann. Reconstruction and simulation of neocortical microcircuitry. *Cell*, 163(2):456–492, October 2015.

[98] Katrin Amunts, Alois C. Knoll, Thomas Lippert, Cyriel M. A. Pennartz, Philippe Ryvlin, Alain Destexhe, Viktor K. Jirsa, Egidio D'Angelo, and Jan G. Bjaalie. The Human Brain Project—Synergy between neuroscience, computing, informatics, and brain-inspired technologies. *PLoS Biology*, 17(7):e3000344, July 2019.

[99] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M. Bower, Markus Diesmann, Abigail Morrison, Philip H. Goodman, Frederick C. Harris, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P. Davison, Sami El Boustani, and Alain Destexhe. Simulation of networks of spiking neurons: A review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398, July 2007.

[100] M. L. Hines and N. T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9(6):1179–1209, August 1997.

[101] Marc-Oliver Gewaltig and Markus Diesmann. NEST (NEural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[102] Sergey L. Gratiy, Yazan N. Billeh, Kael Dai, Catalin Mitelut, David Feng, Nathan W. Gouwens, Nicholas Cain, Christof Koch, Costas A. Anastassiou, and Anton Arkhipov. BioNet: A Python interface to NEURON for modeling large-scale networks. *PLoS ONE*, 13(8):e0201630, August 2018.

[103] Egidio D'Angelo, Alberto Antonietti, Stefano Casali, Claudia Casellato, Jesus A. Garrido, Niceto Rafael Luque, Lisa Mapelli, Stefano Masoli, Alessandra Pedrocchi, Francesca Prestori, Martina Francesca Rizza, and Eduardo Ros. Modeling the cerebellar microcircuit: New strategies for a long-standing issue. *Frontiers in Cellular Neuroscience*, 10, July 2016.

[104] Stefano Casali, Elisa Marenzi, Chaitanya Medini, Claudia Casellato, and Egidio DAngelo. Reconstruction and simulation of a scaffold model of the cerebellar network. 13, May 2019.

[105] Stefano Casali, Marialuisa Tognolina, Daniela Gandolfi, Jonathan Mapelli, and Egidio D'Angelo. Cellular-resolution mapping uncovers spatial adaptive filtering at the rat cerebellum input stage. *Communications biology*, 3(1), October 2020.

[106] Stefano Masoli, Marialuisa Tognolina, Umberto Laforenza, Francesco Moccia, and Egidio D'Angelo. Parameter tuning differentiates granule cell subtypes enriching transmission properties at the cerebellum input stage. *Communications biology*, 3(1), May 2020.

[107] Martina Francesca Rizza, Francesca Locatelli, Stefano Masoli, Diana Sánchez-Ponce, Alberto Muñoz, Francesca Prestori, and Egidio D'Angelo. Stellate cell computational modeling predicts signal filtering in the molecular layer circuit of cerebellum. *Scientific reports*, 11(1), February 2021.

[108] Stefano Masoli, Sergio Solinas, and Egidio DAngelo. Action potential processing in a detailed Purkinje cell model reveals a critical role for axonal compartmentalization. *Frontiers in Cellular Neuroscience*, 9, February 2015.

[109] L. Roggeri, B. Rivieccio, P. Rossi, and E. DAngelo. Tactile stimulation evokes long-term synaptic plasticity in the granular layer of cerebellum. *Journal of Neuroscience*, 28(25):6354–6359, June 2008.

[110] Ede A. Rancz, Taro Ishikawa, Ian Duguid, Paul Chadderton, Séverine

Mahon, and Michael Häusser. High-fidelity transmission of sensory information by single cerebellar mossy fibre boutons. *Nature*, 450(7173):1245–1248, December 2007.

[111] K. B. Ramakrishnan, Kai Voges, Licia De Propris, Chris I. De Zeeuw, and Egidio D'Angelo. Tactile stimulation evokes long-lasting potentiation of purkinje cell discharge in vivo. *Frontiers in Cellular Neuroscience*, 10, February 2016.

[112] Fahad Sultan and James M. Bower. Quantitative Golgi study of the rat cerebellar molecular layer interneurons using principal component analysis. *Journal of Comparative Neurology*, 393(3):353–373, April 1998.

[113] Fahad Sultan. Distribution of mossy fibre rosettes in the cerebellum of cat and mice: Evidence for a parasagittal organization at the single fibre level. *European Journal of Neuroscience*, 13(11):2123–2130, June 2001.

[114] Guy Billings, Eugenio Piasini, Andrea Lőrincz, Zoltan Nusser, and R. Angus Silver. Network structure within the cerebellar input layer enables lossless sparse encoding. *Neuron*, 83(4):960–974, August 2014.

[115] Catriona M. Houston, Efthymia Diamanti, Maria Diamantaki, Elena Kutsarova, Anna Cook, Fahad Sultan, and Stephen G. Brickley. Exploring the significance of morphological diversity for cerebellar granule cell excitability. *Scientific reports*, 7(1), April 2017.

[116] József Hámori, Robert L. Jakab, and József Takács. Morphogenetic plasticity of neuronal elements in cerebellar glomeruli during deafferentation-induced synaptic reorganization. *Journal of Neural Transplantation and Plasticity*, 6(1):11–20, 1997.

[117] Sawako Tabuchi, Jesse I. Gilmer, Karen Purba, and Abigail L. Person. Pathway-specific drive of cerebellar golgi cells reveals integrative rules of cortical inhibition. *Journal of Neuroscience*, 39(7):1169–1181, December 2018.

[118] E. Cesana, K. Pietrajtis, C. Bidoret, P. Isope, E. DAngelo, S. Dieudonne, and L. Forti. Granule cell ascending axon excitatory synapses onto golgi cells implement a potent feedback circuit in the cerebellar granular layer. *Journal of Neuroscience*, 33(30):12430–12446, July 2013.

[119] Court Hull and Wade G. Regehr. Identification of an inhibitory circuit that regulates cerebellar golgi cell activity. *Neuron*, 73(1):149–158, January

2012.

[120] Miklos Szoboszlay, Andrea Lőrincz, Frederic Lanore, Koen Vervaeke, R. Angus Silver, and Zoltan Nusser. Functional properties of dendritic gap junctions in cerebellar golgi cells. *Neuron*, 90(5):1043–1056, June 2016.

[121] Stefano Masoli and Egidio D'Angelo. Synaptic activation of a detailed purkinje cell model predicts voltage-dependent control of burst-pause responses in active dendrites. *Frontiers in Cellular Neuroscience*, 11, September 2017.

[122] Eriola Hoxha, Filippo Tempia, Pellegrino Lippiello, and Maria Concetta Miniaci. Modulation, plasticity and pathophysiology of the parallel fiber-purkinje cell synapse. *Frontiers in Synaptic Neuroscience*, 8, November 2016.

[123] Joy T. Walter and Kamran Khodakhah. The linear computational algorithm of cerebellar purkinje cells. *Journal of Neuroscience*, 26(50):12861–12872, December 2006.

[124] Satoru Kondo and Alain Marty. Synaptic currents at individual connections among stellate cells in rat cerebellar slices. *The Journal of Physiology*, 509(1):221–232, May 1998.

[125] Fabrice Ango, Caizhi Wu, Johannes J. Van der Want, Priscilla Wu, Melitta Schachner, and Z. Josh Huang. Bergmann Glia and the Recognition Molecule CHL1 Organize GABAergic Axons and Direct Innervation of Purkinje Cell Dendrites. *PLoS Biology*, 6(4):e103, April 2008.

[126] Antonin Blot and Boris Barbour. Ultra-rapid axon-axon ephaptic inhibition of cerebellar Purkinje cells by the pinceau. *Nature neuroscience*, 17(2):289–295, January 2014.

[127] Christian D. Wilms and Michael Häusser. Reading out a spatiotemporal population code by imaging neighbouring parallel fibre axons in vivo. *Nature communications*, 6(1), March 2015.

[128] Sergio Solinas. Fast-reset of pacemaking and theta-frequency resonance patterns in cerebellar golgi cells: Simulations of their impact in vivo. *Frontiers in Cellular Neuroscience*, 1, 2007.

[129] Lia Forti Elisabetta Cesana, Jonathan Mapelli, and Egidio DAngelo. Ionic

mechanisms of autorhythmic firing in rat cerebellar Golgi cells. *The Journal of Physiology*, 574(3):711–729, July 2006.

[130] Marife Arancillo, Joshua J. White, Tao Lin, Trace L. Stay, and Roy V. Sillitoe. In vivo analysis of Purkinje cell firing properties during postnatal mouse development. *Journal of Neurophysiology*, 113(2):578–591, January 2015.

[131] Jinsook Kim and George J. Augustine. Molecular layer interneurons: Key elements of cerebellar network computation and behavior. *Neuroscience*, 462:22–35, May 2021.

[132] Dan-Anders Jirenhed, Fredrik Bengtsson, and Henrik Jörntell. Parallel fiber and climbing fiber responses in rat cerebellar cortical neurons in vivo. *Frontiers in Systems Neuroscience*, 7, 2013.

[133] Neal H. Barmack and Vadim Yakhnitsa. Functions of interneurons in mouse cerebellum. *Journal of Neuroscience*, 28(5):1140–1152, January 2008.

[134] Mitra J. Hartmann and James M. Bower. Oscillatory activity in the cerebellar hemispheres of unrestrained rats. *Journal of Neurophysiology*, 80(3):1598–1604, September 1998.

[135] Guillaume P. Dugué, Nicolas Brunel, Vincent Hakim, Eric Schwartz, Mireille Chat, Maxime Lévesque, Richard Courtemanche, Clément Léna, and Stéphane Dieudonné. Electrical coupling mediates tunable low-frequency oscillations and resonance in the cerebellar golgi cell network. *Neuron*, 61(1):126–139, January 2009.

[136] Ingrid van Welie, Arnd Roth, Sara S.N. Ho, Shoji Komai, and Michael Häusser. Conditional spike transmission mediated by electrical coupling ensures millisecond precision-correlated activity among interneurons in vivo. *Neuron*, 90(4):810–823, May 2016.

[137] R Maex and E De Schutter. An optimal connection radius for long-range synchronization. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99.(Conf. Publ. No. 470)*, volume 2, pages 557–562. IET, 1999.

[138] E DAngelo, G De Filippi, P Rossi, and V Taglietti. Synaptic excitation of individual rat cerebellar granule cells in situ: Evidence for the role of NMDA receptors. *The Journal of Physiology*, 484(2):397–413, April 1995.

[139] Shyam Diwakar, Paola Lombardo, Sergio Solinas, Giovanni Naldi, and Egidio D'Angelo. Local Field Potential Modeling Predicts Dense Activation in Cerebellar Granule Cells Clusters under LTP and LTD Control. *PLoS ONE*, 6(7):e21928, July 2011.

[140] Paul Chadderton, Troy W. Margrie, and Michael Häusser. Integration of quanta in cerebellar granule cells during sensory processing. *Nature*, 428(6985):856–860, April 2004.

[141] Alexander Arenz, R. Angus Silver, Andreas T. Schaefer, and Troy W. Margrie. The Contribution of Single Synapses to Sensory Representation in Vivo. *Science (New York, N.Y.)*, 321(5891):977–980, August 2008.

[142] Kate Powell, Alexandre Mathy, Ian Duguid, and Michael Häusser. Synaptic representation of locomotion in single cerebellar granule cells. *eLife*, 4, June 2015.

[143] Taro Ishikawa, Misa Shimuta, and Michael Häusser. Multimodal sensory integration in single cerebellar granule cells in vivo. *eLife*, 4, December 2015.

[144] B. P. Vos, A. Volny-Luraghi, and E. De Schutter. Cerebellar Golgi cells in the rat: Receptive fields and timing of responses to facial stimulation. *The European Journal of Neuroscience*, 11(8):2621–2634, August 1999.

[145] Sergio M. G. Solinas, Reinoud Maex, and Erik Schutter. Dendritic amplification of inhibitory postsynaptic potentials in a model Purkinje cell. *European Journal of Neuroscience*, 23(5):1207–1218, March 2006.

[146] Tahl Holtzman, Thimali Rajapaksa, Abteen Mostofi, and Steve A. Edgley. Different responses of rat cerebellar Purkinje cells and Golgi cells evoked by widespread convergent sensory inputs. *The Journal of Physiology*, 574(2):491–507, July 2006.

[147] David J. Herzfeld, Yoshiko Kojima, Robijanto Soetedjo, and Reza Shadmehr. Encoding of action by the Purkinje cells of the cerebellum. *Nature*, 526(7573):439–442, October 2015.

[148] Stefano Masoli, Martina F. Rizza, Martina Sgritta, Werner Van Geit, Felix Schürmann, and Egidio DAngelo. Single neuron optimization as a basis for accurate biophysical modeling: The case of cerebellar granule cells. *Frontiers in Cellular Neuroscience*, 11, March 2017.

[149] J. M. Bower and D. C. Woolston. Congruence of spatial organization of tactile projections to granule cell and Purkinje cell layers of cerebellar hemispheres of the albino rat: Vertical organization of cerebellar cortex. *Journal of Neurophysiology*, 49(3):745–766, March 1983.

[150] Peer Wulff, Martijn Schonewille, Massimiliano Renzi, Laura Viltono, Marco Sassoè-Pognetto, Aleksandra Badura, Zhenyu Gao, Freek E Hoebeek, Stijn van Dorp, William Wisden, Mark Farrant, and Chris I De Zeeuw. Synaptic inhibition of Purkinje cells mediates consolidation of vestibulo-cerebellar motor learning. *Nature neuroscience*, 12(8):1042–1049, July 2009.

[151] Michiel M. ten Brinke, Henk-Jan Boele, Jochen K. Spanke, Jan-Willem Potters, Katja Kornysheva, Peer Wulff, Anna C.H.G. IJpelaar, Sebastiaan K.E. Koekkoek, and Chris I. De Zeeuw. Evolving models of pavlovian conditioning: Cerebellar cortical dynamics in awake behaving mice. *Cell Reports*, 13(9):1977–1988, December 2015.

[152] Huo Lu, Angelica V. Esquivel, and James M. Bower. 3D electron microscopic reconstruction of segments of rat cerebellar purkinje cell dendrites receiving ascending and parallel fiber granule cell synaptic inputs. *Journal of Comparative Neurology*, 514(6):583–594, June 2009.

[153] Girija Gundappa-Sulur, Erik De Schutter, and James M. Bower. Ascending granule cell axon: An important component of cerebellar cortical circuitry. *Journal of Comparative Neurology*, 408(4):580–596, June 1999.

[154] Henrik Jörntell, Fredrik Bengtsson, Martijn Schonewille, and Chris I. De Zeeuw. Cerebellar molecular layer interneurons – computational properties and roles in learning. *Trends in Neurosciences*, 33(11):524–532, November 2010.

[155] James M. Soha, Sugene Kim, James E. Crandall, and Michael W. Vogel. Rapid growth of parallel fibers in the cerebella of normal andStaggerer mutant mice. *Journal of Comparative Neurology*, 389(4):642–654, December 1997.

[156] Haibo Zhou, Kai Voges, Zhanmin Lin, Chiheng Ju, and Martijn Schonewille. Differential Purkinje cell simple spike activity and pausing behavior related to cerebellar modules. *Journal of Neurophysiology*, 113(7):2524–2536, April 2015.

[157] Marta Bisio, Alessandro Bosca, Valentina Pasquale, Luca Berdondini, and Michela Chiappalone. Emergence of Bursting Activity in Connected Neu-

ronal Sub-Populations. *PLoS ONE*, 9(9):e107400, September 2014.

[158] Abigail L. Person and Indira M. Raman. Synchrony and neural coding in cerebellar circuits. *Front. Neural Circuits*, 6, 2012.

[159] Javier F. Medina and Michael D. Mauk. Computer simulation of cerebellar information processing. *Nature neuroscience*, 3(S11):1205–1211, November 2000.

[160] Alice Geminiani, Alessandra Pedrocchi, Egidio D'Angelo, and Claudia Casellato. Response dynamics in an olivocerebellar spiking neural network with non-linear neuron properties. 13, October 2019.

[161] Michael W. Reimann, James G. King, Eilif B. Muller, Srikanth Ramaswamy, and Henry Markram. An algorithm to predict the connectome of neural microcircuits. 9, October 2015.

[162] Jonathan Mapelli, Daniela Gandolfi, and Egidio DAngelo. Combinatorial responses controlled by synaptic inhibition in the cerebellum granular layer. *Journal of Neurophysiology*, 103(1):250–261, January 2010.

[163] Jonathan Mapelli, Daniela Gandolfi, and Egidio D'Angelo. High-Pass Filtering and Dynamic Gain Regulation Enhance Vertical Bursts Transmission along the Mossy Fiber Pathway of Cerebellum. *Frontiers in Cellular Neuroscience*, 4:14, May 2010.

[164] Richard Apps and Richard Hawkes. Cerebellar cortical organization: A one-map hypothesis. *Nature reviews. Neuroscience*, 10(9):670–681, September 2009.

[165] Richard Apps, Richard Hawkes, Sho Aoki, Fredrik Bengtsson, Amanda M. Brown, Gang Chen, Timothy J. Ebner, Philippe Isope, Henrik Jörntell, Elizabeth P. Lackey, Charlotte Lawrenson, Bridget Lumb, Martijn Schonewille, Roy V. Sillitoe, Ludovic Spaeth, Izumi Sugihara, Antoine Valera, Jan Voogd, Douglas R. Wylie, and Tom J. H. Ruigrok. Cerebellar modules and their role as operational cerebellar processing units. *Cerebellum (London, England)*, 17(5):654–682, June 2018.

[166] Chris I. De Zeeuw. Bidirectional learning in upbound and downbound microzones of the cerebellum. *Nature reviews. Neuroscience*, 22(2):92–110, November 2020.

[167] Krishnan Padmanabhan and Nathaniel N Urban. Intrinsic biophysical

diversity decorrelates neuronal firing while increasing information content. *Nature neuroscience*, 13(10):1276–1282, August 2010.

[168] Alice Geminiani, Claudia Casellato, Francesca Locatelli, Francesca Prestori, Alessandra Pedrocchi, and Egidio DAngelo. Complex dynamics in simplified neuronal models: Reproducing golgi cell electroresponsiveness. 12, December 2018.

[169] Alice Geminiani, Claudia Casellato, Egidio D'Angelo, and Alessandra Pedrocchi. Complex electroresponsive dynamics in olivocerebellar neurons represented with extended-generalized leaky integrate and fire models. 13, June 2019.

[170] Miaozhen Huang, Tom J. de Koning, Marina A.J. Tijssen, and Dineke S. Verbeek. Cross-disease analysis of depression, ataxia and dystonia highlights a role for synaptic plasticity and the cerebellum in the pathophysiology of these comorbid diseases. *Biochimica et Biophysica Acta (BBA) - Molecular Basis of Disease*, 1867(1):165976, January 2021.

[171] Saša Peter, Michiel M. ten Brinke, Jeffrey Stedehouder, Claudia M. Reinelt, Bin Wu, Haibo Zhou, Kuikui Zhou, Henk-Jan Boele, Steven A. Kushner, Min Goo Lee, Michael J. Schmeisser, Tobias M. Boeckers, Martijn Schonewille, Freek E. Hoebeek, and Chris I. De Zeeuw. Dysfunctional cerebellar Purkinje cells contribute to autism-like behaviour in Shank2-deficient mice. *Nature Communications*, 7:12627, September 2016.

[172] Lennart Paul Liong Landsmeer. *Influence of Local Dendritic Organization on Synchronization in the Inferior Olivary Nucleus*. PhD thesis, TU Delft, September 2022.

[173] Anila M. D'Mello and Catherine J. Stoodley. Cerebro-cerebellar circuits in autism spectrum disorder. *Frontiers in Neuroscience*, 9, 2015.

[174] Joanna Giza, Michael J. Urbanski, Francesca Prestori, Bhaswati Bandyopadhyay, Annie Yam, Victor Friedrich, Kevin Kelley, Egidio D'Angelo, and Mitchell Goldfarb. Behavioral and Cerebellar Transmission Deficits in Mice Lacking the Autism-Linked Gene Islet Brain-2. *Journal of Neuroscience*, 30(44):14805–14816, November 2010.

[175] Teresa Soda, Lisa Mapelli, Francesca Locatelli, Laura Botta, Mitchell Goldfarb, Francesca Prestori, and Egidio D'Angelo. Hyperexcitability and Hyperplasticity Disrupt Cerebellar Signal Transfer in the IB2 KO Mouse Model of Autism. *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*, 39(13):2383–2397, March 2019.

[176] Kyoung-Doo Hwang, Sang Jeong Kim, and Yong-Seok Lee. Cerebellar circuits for classical fear conditioning. *Frontiers in Cellular Neuroscience*, 16, March 2022.

[177] Mario Manto, Donna Gruol, Jeremy Schmahmann, Noriyuki Koibuchi, and Roy Sillitoe, editors. *Handbook of the Cerebellum and Cerebellar Disorders*. Springer International Publishing, 2020.

[178] Michael Adamaszek, Mario Manto, and Dennis J. L. G. Schutter, editors. *The Emotional Cerebellum*. Springer International Publishing, 2022.

[179] Dimitri Rodarie, Csaba Verasztó, Yann Roussel, Michael Reimann, Daniel Keller, Srikanth Ramaswamy, Henry Markram, and Marc-Oliver Gewaltig. A method to estimate the cellular composition of the mouse brain from heterogeneous datasets. *PLOS Computational Biology*, 18(12):e1010739, December 2022.

[180] Gabriella Sekerková, Masahiko Watanabe, Marco Martina, and Enrico Mugnaini. Differential distribution of phospholipase C beta isoforms and diaglycerol kinase-beta in rodents cerebella corroborates the division of unipolar brush cells into two major subtypes. *Brain Structure and Function*, 219(2):719–749, March 2013.

[181] Stéphane Dieudonné and Andréa Dumoulin. Serotonin-driven long-range inhibitory connections in the cerebellar cortex. *The Journal of Neuroscience*, 20(5):1837–1848, March 2000.

[182] Tomas Osorno, Stephanie Rudolph, Tri Nguyen, Velina Kozareva, Naeem M. Nadaf, Aliya Norton, Evan Z. Macosko, Wei-Chung Allen Lee, and Wade G. Regehr. Candelabrum cells are ubiquitous cerebellar cortex interneurons with specialized circuit properties. *Nature Neuroscience*, 25(6):702–713, May 2022.

[183] Beth M. Turner, Sergio Paradiso, Cherie L. Marvel, Ronald Pierson, Laura L. Boles Ponto, Richard D. Hichwa, and Robert G. Robinson. The cerebellum and emotional experience. *Neuropsychologia*, 45(6):1331–1341, January 2007.

[184] Iris Lange, Zuzana Kasanova, Liesbet Goossens, Nicole Leibold, Chris I. De Zeeuw, Therese van Amelsvoort, and Koen Schruers. The anatomy of fear learning in the cerebellum: A systematic meta-analysis. *Neuroscience &amp Biobehavioral Reviews*, 59:83–91, December 2015.

[185] Pei Wern Chin and George J. Augustine. The cerebellum and anxiety. *Frontiers in Cellular Neuroscience*, 17, 2023.

[186] Malte S. Depping, Mike M. Schmitgen, Katharina M. Kubera, and Robert C. Wolf. Cerebellar Contributions to Major Depression. *Frontiers in Psychiatry*, 9, 2018.

[187] S. Hossein Fatemi, Kimberly A. Aldinger, Paul Ashwood, Margaret L. Bauman, Charles D. Blaha, Gene J. Blatt, Abha Chauhan, Ved Chauhan, Stephen R. Dager, Price E. Dickson, Annette M. Estes, Dan Goldowitz, Detlef H. Heck, Thomas L. Kemper, Bryan H. King, Loren A. Martin, Kathleen J. Millen, Guy Mittleman, Matthew W. Mosconi, Antonio M. Persico, John A. Sweeney, Sara J. Webb, and John P. Welsh. Consensus paper: Pathological role of the cerebellum in autism. *The Cerebellum*, 11(3):777–807, February 2012.

[188] Junxiao Zheng, Qinzhu Yang, Nikos Makris, Kaibin Huang, Jianwen Liang, Chenfei Ye, Xiaxia Yu, Mu Tian, Ting Ma, Tian Mou, Wenlong Guo, Ron Kikinis, and Yi Gao. Three-Dimensional Digital Reconstruction of the Cerebellar Cortex: Lobule Thickness, Surface Area Measurements, and Layer Architecture. *The Cerebellum*, 22(2):249–260, April 2023.

[189] Daniela Gandolfi, Jonathan Mapelli, Sergio Solinas, Robin De Schepper, Alice Geminiani, Claudia Casellato, Egidio D'Angelo, and Michele Migliore. A realistic morpho-anatomical connection strategy for modelling full-scale point-neuron microcircuits. *Scientific Reports*, 12(1):13864, August 2022.

[190] Csaba Erö, Marc-Oliver Gewaltig, Daniel Keller, and Henry Markram. A Cell Atlas for the Mouse Brain. *Frontiers in Neuroinformatics*, 12, 2018.

[191] Carolina Tecuatl, Diek W. Wheeler, Nate Sutton, and Giorgio A. Ascoli. Comprehensive Estimates of Potential Synaptic Connections in Local Circuits of the Rodent Hippocampal Formation by Axonal-Dendritic Overlap. *Journal of Neuroscience*, 41(8):1665–1683, February 2021.

[192] Kenneth A. Pelkey, Ramesh Chittajallu, Michael T. Craig, Ludovic Tricoire, Jason C. Wester, and Chris J. McBain. Hippocampal GABAergic Inhibitory Interneurons. *Physiological Reviews*, 97(4):1619–1747, October 2017.

[193] Masood A. Akram, Sumit Nanda, Patricia Maraver, Rubén Armañanzas, and Giorgio A. Ascoli. An open repository for single-cell reconstructions of the brain forest. *Scientific Data*, 5(1):180006, February 2018.

[194] Linda Madisen, Theresa A. Zwingman, Susan M. Sunkin, Seung Wook Oh, Hatim A. Zariwala, Hong Gu, Lydia L. Ng, Richard D. Palmiter, Michael J. Hawrylycz, Allan R. Jones, Ed S. Lein, and Hongkui Zeng. A robust and high-throughput Cre reporting and characterization system for the whole mouse brain. *Nature Neuroscience*, 13(1):133–140, January 2010.

[195] Johan Winnubst, Erhan Bas, Tiago A. Ferreira, Zhuhao Wu, Michael N. Economo, Patrick Edson, Ben J. Arthur, Christopher Bruns, Konrad Rokicki, David Schauder, Donald J. Olbris, Sean D. Murphy, David G. Ackerman, Cameron Arshadi, Perry Baldwin, Regina Blake, Ahmad Elsayed, Mashtura Hasan, Daniel Ramirez, Bruno Dos Santos, Monet Weldon, Amina Zafar, Joshua T. Dudman, Charles R. Gerfen, Adam W. Hantman, Wyatt Korff, Scott M. Sternson, Nelson Spruston, Karel Svoboda, and Jayaram Chandrashekar. Reconstruction of 1,000 Projection Neurons Reveals New Cell Types and Organization of Long-Range Connectivity in the Mouse Brain. *Cell*, 179(1):268–281.e13, September 2019.

[196] R. Orman, H. Von Gizycki, W.w. Lytton, and M. Stewart. Local axon collaterals of area CA1 support spread of epileptiform discharges within CA1, but propagation is unidirectional. *Hippocampus*, 18(10):1021–1033, 2008.

[197] Vassilis Cutsuridis, Stuart Cobb, and Bruce P. Graham. Encoding and retrieval in a model of the hippocampal CA1 microcircuit. *Hippocampus*, 20(3):423–446, 2010.

[198] Calvin J. Schneider, Hermann Cuntz, and Ivan Soltesz. Linking Macroscopic with Microscopic Neuroanatomy Using Synthetic Neuronal Populations. *PLOS Computational Biology*, 10(10):e1003921, October 2014.