UNCERTAINTY QUANTIFICATION IN NEUROSCIENCE

by

Simen Tennøe

THESIS
for the degree of

PHILOSOPHIAE DOCTOR

Faculty of Mathematics and Natural Sciences
University of Oslo

January 2019
Acknowledgements

The work in this thesis was carried out at the Center for Integrative Neuroplasticity (CINPLA) and the Department of Informatics at the University of Oslo in the period 2014 – 2019, under the supervision of Dr. Geir Halnes, Prof. Morten Hjorth-Jensen and Prof. Gaute T. Einevoll. The work was funded by the University of Oslo and the Research Council of Norway.

I want to express my sincere gratitude to my supervisors. Your supervision along the way has been vital and you have taught me a lot. You have also been very encouraging and have helped guide the projects in the directions that I have found the most interesting. I would also like to acknowledge the vital contribution to the vision of this work by Prof. Hans-Petter Langtangen. He started as my main supervisor and set the path and long-term goal for the work in this thesis. He has also been a huge source of inspiration. To our deep regret, he passed away before he could see the work in this thesis completed.

I want to extend a special thanks to Svenn-Arne Dragly, Milad H. Mobarhan and Andreas Våvang Solbrå with whom I have shared an office for the last four years. We started our Ph.D.s at the same time, and it has been a pleasure to complete them together. You have provided valuable insight and feedback during my Ph.D., along with plenty of interesting scientific and less-scientific discussions. I would also like to thank Ivan Myhre Winje and all current and former members of CINPLA for valuable collaborations and for creating an excellent environment in which to work. Additionally, I want to thank Dr. Jonathan Feinberg for his help in teaching me the basics of polynomial chaos expansions, as well as how to use Chaospy.

I also want to thank Thea and Anne Caroline for suffering through my writing and helping me improve it. Last but not least, I want to thank Thea, my family, and friends for all the support, patience, and encouragement along the way.

Simen Tennøe
Oslo, May 6, 2019
## Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>List of papers</td>
<td>3</td>
</tr>
<tr>
<td><strong>Part I  Background</strong></td>
<td>5</td>
</tr>
<tr>
<td>1  Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2  Objectives</td>
<td>11</td>
</tr>
<tr>
<td>3  Computational neuroscience</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Benefits of computational modeling</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Creating computational models</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Computational models used in this thesis</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1 Conductance-based models</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2 Integrate-and-fire models</td>
<td>20</td>
</tr>
<tr>
<td>3.3.3 Computational models of endocrine pituitary cells</td>
<td>21</td>
</tr>
<tr>
<td>3.4 Estimating the parameters of a model</td>
<td>22</td>
</tr>
<tr>
<td>3.4.1 Deterministic parameter estimation</td>
<td>23</td>
</tr>
<tr>
<td>3.4.2 Bayesian parameter estimation</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Replicability in computational neuroscience</td>
<td>29</td>
</tr>
<tr>
<td>3.5.1 Difference between reproduction and replication</td>
<td>30</td>
</tr>
<tr>
<td>3.5.2 Challenges for reproducibility</td>
<td>30</td>
</tr>
<tr>
<td>3.5.3 Challenges for replicability</td>
<td>31</td>
</tr>
<tr>
<td>4  Uncertainty quantification and sensitivity analysis</td>
<td>33</td>
</tr>
<tr>
<td>4.1 Origins of parameter uncertainty</td>
<td>35</td>
</tr>
<tr>
<td>4.1.1 Measurement uncertainty</td>
<td>35</td>
</tr>
<tr>
<td>4.1.2 Biological variability</td>
<td>35</td>
</tr>
</tbody>
</table>
Abstract

The complexity of the nervous system has made computational science an invaluable tool in order to understand how the nervous system functions. The overarching goal of this thesis has been to develop software tools to improve areas of neuroscience that are currently lacking, which include uncertainty analysis of computational models (Paper I), data storage (Paper IV), and education (Paper V). The major area of focus has been that of uncertainty analysis. Computational models always contain parameters that describe the system to be modeled. These parameters are for various reasons often uncertain. An uncertainty analysis provides rigorous procedures to quantify how the model depends on this parameter uncertainty. To reduce the barrier of performing uncertainty analysis in neuroscience we have created a toolbox for uncertainty analysis (Paper I). We then used this toolbox on a selected set of models (Paper I, II and III).

In Paper I we introduced Uncertainpy, a Python toolbox for performing uncertainty quantification and sensitivity analysis. Uncertainpy is tailored for neuroscience applications by its built-in capability for calculating characteristic features in the model output. We provided a detailed user guide for Uncertainpy and illustrated its use by showing four different case studies.

In Paper II we presented a reimplementation of a model for endocrine pituitary cells in rats. We qualitatively replicated the computational results in the original publication and confirmed the key conclusions, namely that big conductance K⁺ (BK) ion channels are important for the bursting activity of endocrine pituitary cells in rats. Additionally, we performed an uncertainty analysis of the model using Uncertainpy, which further strengthened the findings in the original publication.

In Paper III we created a computational model for endocrine pituitary cells in medaka, a species of Japanese rice fish. The reimplementation and results in Paper II were used as a basis for the computational work in this paper. We discovered that the BK conductance has the opposite effect on the action potential shape in medaka pituitary cells compared to in the rat pituitary cells in Paper II. The BK channels makes the action potentials generated in the medaka
model narrower, but they make the action potentials generated in the rat model broader. An uncertainty analysis of the two models was performed in order to examine differences in the sensitivity of the models to changes in their ion channel conductances.

In Paper IV we developed a specification for organizing data in a hierarchy by using file-system directories to represent the hierarchy. We used the same data abstraction as in the HDF5 file format. We provided a reference implementation in Python and described how to use this implementation.

In Paper V we introduced Neuronify, an educational app for easily creating neural networks by dragging and dropping neurons onto the canvas and then simulating the networks. Neuronify is available for iOS and Android, as well as Mac, Linux, and Windows.
This thesis consists of the following five papers, with the main part of the work being Paper I.

Paper I

**Uncertainpy: A Python Toolbox for Uncertainty Quantification and Sensitivity Analysis in Computational Neuroscience**
Simen Tennøe, Geir Halnes, and Gaute T. Einevoll.

Paper II

[Re] **Fast-Activating Voltage- and Calcium-Dependent Potassium (BK) Conductance Promotes Bursting in Pituitary Cells: A Dynamic Clamp Study**
Simen Tennøe, Kjetil Hodne, Trude M. Haug, Finn-Arne Weltzien, Gaute T. Einevoll, and Geir Halnes
*ReScience (submitted).*

Paper III

**BK channels have opposite effects on sodium versus calcium mediated action potentials in endocrine pituitary cells**
Geir Halnes, Simen Tennøe, Trude M. Haug, Gaute T. Einevoll, Finn Arne Weltzien, and Kjetil Hodne.
*PLOS computational (submitted).*
Paper IV

Experimental Directory Structure (Exdir): An Alternative to HDF5 Without Introducing a New File Format
Svenn-Arne Dragly†, Milad Hobbi Mobarhan†, Mikkel Lepperød†, Simen Tennenøe, Marianne Fyhn, Torkel Hafting, and Anders Malthe-Sørenssen.


Paper V

Neuronify: An Educational Simulator for Neural Circuits


†These authors have contributed equally to this work.
Part I

Background
The nervous system is responsible for making us think, feel, and move. It is heavily studied but poorly understood. The ultimate goal of neuroscience is to understand how the entire nervous system functions. This requires a deep understanding of all levels of detail in the nervous system, from how a single ion channel works on a molecular level to how aging affects the brain as a whole. The spatial scale of neuroscience goes from the molecular dynamics of single ion channels at nanometer scales ($\sim 10^{-9}$ m) up to the length of the longest nerves at meter scales. Similarly, the temporal scale goes from molecular dynamics at picosecond scales ($\sim 10^{-12}$ s) to aging over the lifespan of humans ($\sim 100$ years). The complexity of the nervous systems and the wide range of different spatial and temporal scales we want to combine makes computational science an irreplaceable tool.

Ever more extensive computational models aiming to explain increasingly complex phenomena in the nervous system are developed and models now exist for large networks of biophysically detailed neurons (Izhikevich and Edelman, 2008; Merolla et al., 2014; Markram et al., 2015). The creation of these extensive computational models is fuelled by the ever increasing amount of experimental data generated due to technological innovations. However, in order to draw conclusions from the increasing amount of experimental data we need to use computational models, rigorous data handling, and advanced data analysis (Council, 2005). This enables a feedback loop between experiments and computational science that enhances both and leads to an ever deeper understanding of the nervous system. However, while neuroscience research has become increasingly dependent on computational tools, neuroscience education is lagging behind. The computational aspects of neuroscience are not yet integrated into the current neuroscience education, making future generations of neuroscientists
Introduction

Chapter 1

ill-equipped for performing research.

The challenges of computational modeling, data handling, and computational education fall within the field of neuroinformatics, the field where the work in this thesis resides. Neuroinformatics is concerned with the development and use of tools for data handling (which includes analysis, visualization, and sharing) and computational modeling (Sterratt et al., 2011). The goal of this thesis has been to develop software tools to improve areas of neuroinformatics that are currently lacking, which include uncertainty analysis of computational neuroscience models (Paper I), data storage (Paper IV), and education (Paper V).

A large part of neuroinformatics is to develop and analyze computational models of specific systems within the nervous system. Computational models enable us to accurately state theories using mathematical formulations and predict outcomes and test hypotheses that for various reasons are too costly, complicated or unethical to test experimentally. Furthermore, computational models enable us to gain new insight, predict parameters that we are unable to measure, guide and refine experiments, examine the sensitivity of the system to changes in parameters, build an intuition for the modeled system, and more (Council, 2005; Sterratt et al., 2011; Brodland, 2015).

The major area of focus has been uncertainty analysis of computational models in neuroscience, with the main part of the work being spent on the development of a toolbox for uncertainty analysis (Paper I). Computational models always contain a number of parameters that describe the system to be modeled. It is common that these parameters for various reasons are uncertain, for example, due to measurement uncertainty. Uncertainty quantification and sensitivity analysis provide rigorous procedures to quantify how the model output depends on the parameter uncertainty. However, no commonly accepted practice for taking parameter uncertainties into account exists within the field of neuroscience. Due to the prevalence of inherent variability in the parameters of biological systems, uncertainty analysis is of especial importance in neuroscience. Unfortunately, the application of such methods is not yet standard within the field of neuroscience and there is a pressing need for systematic approaches to quantify what confidence we can have in the model output. One goal of this thesis has been to help remedy this situation by creating a Python toolbox, tailored to perform uncertainty quantification and sensitivity analysis of computational neuroscience models (Paper I). This toolbox was then used on a selected set of neuroscience models (Paper I, II, and III), providing additional insight into the examined models.

In order to create computational models, we require experimental data from the systems we want to model. Technological innovations have made it so vast amounts of experimental (and computational) data are generated and new experimental techniques put additional demands on existing data formats. These demands necessitate the use of good and flexible standards to store the data and
to be able to easily share the data to facilitate reproducible research. In order to improve upon limitations of existing data formats in use in neuroscience, a new specification for storing data was developed in Paper IV.

As mentioned, neuroscience has become increasingly dependent on data analysis and computational modeling. However, this is not reflected in much of the current neuroscience education (Goldman and Fee, 2017). One of many solutions to help remedy this situation and improve the computational education in neuroscience is to create neuroscience simulators tailored towards education. Such simulators enable students without computational experience to perform simulations and give a great introduction to computational modeling that these students can benefit from. Educational simulators can be used to both teach neuroscience, in that the students explore how the biological system behaves, and computational modeling, in that they can learn how the implemented models work. Such an educational simulator was developed in Paper V.

The outline of this thesis is as follows. We start this thesis by stating the objectives of each paper in chapter 2. Then we give a brief introduction to computational neuroscience in chapter 3. In this chapter we go into details on the types of computational neuroscience models encountered in Paper I, II, III, and V. This section gives a background in order to easier interpret the results in Paper I, II and III. We then give a brief introduction to parameter estimation, which is related to uncertainty analysis. We end the chapter with a discussion of the challenges related to replicability in computational neuroscience, which concerns the work in Paper II where we replicate the results of a previously studied computational model.

In chapter 4 we give an introduction to uncertainty quantification and sensitivity analysis and a brief overview of some methods for uncertainty quantification and sensitivity analysis. This introduction gives a background for the uncertainty analysis performed in Paper I, II and III, and for the uncertainty quantification and sensitivity analysis methods implemented in the toolbox in Paper I. In chapter 5 we talk about the challenges of data standards and data sharing which is relevant for the data specification developed in Paper IV. Then, in chapter 6, we talk about the lack of computational education in neuroscience, which we aimed to help address with the educational simulator developed in Paper V. We give a brief summary of the results of each paper in chapter 7 before we end with a discussion and some future prospects in chapter 8.
2 Objectives

The overarching goal of this thesis has been to develop tools to improve areas of neuroscience that are currently lacking, with the main focus being to develop a toolbox for performing uncertainty quantification and sensitivity analysis of computational neuroscience models (Paper I). In detail, the goals of each individual paper were:

**Paper I**

Develop a toolbox for performing uncertainty quantification and sensitivity analysis tailored towards neuroscience and use the toolbox on a selected set of use cases relevant for neuroscience. Additionally, give an introduction to uncertainty quantification and sensitivity analysis for neuroscientists.

**Paper II**

Tabak et al., 2011 created a computational model for endocrine pituitary cells in rats. The goal of this paper was to reimplement their model, replicate their computational results, and perform an uncertainty quantification and sensitivity analysis of the model to further enhance the conclusions in the original publication.

**Paper III**

Develop a computational model for endocrine pituitary cells in Japanese rice fish from the model in Paper II. Then examine the role of the big conductance K⁺ ion channel on bursting in these cells by, among other methods, performing an
uncertainty quantification and sensitivity analysis. Furthermore, compare the bursting in endocrine pituitary cells in fish and rat (Paper II).

**Paper IV**

HDF5 (Hierarchical Data Format 5) is a popular data format for storing data in a hierarchy within a single binary file. The objective of this paper was to develop a new specification for data storage that has the same type of data hierarchy as HDF5, but which uses the file system to store the data instead of storing everything in a single binary file.

**Paper V**

Develop an educational neural network simulator that is intuitive to use and enables students to easily explore how neural networks behave in “real-time”.
“All models are wrong but some are useful.”

George E. P. Box

3

Computational neuroscience

Computational neuroscience can be considered a subset of neuroinformatics and is focused on the development of computational models and the use of mathematical tools and theories to understand how the brain functions (Nature, 2018). Computational neuroscience ranges from analyzing experimental data from extracellular recordings (Rey et al., 2015) to the theoretical analysis of constraints of neural network architecture (Wen and Chklovskii, 2005). One of the larger aspects of computational neuroscience is the development and analysis of computational models of specific systems in the brain. An example of a common type of models in neuroscience is models for the propagation of action potentials. One of the most famous models of this kind was created already in 1952 (Hodgkin and Huxley, 1952).

3.1 Benefits of computational modeling

There are many reasons for doing computational modeling. We state a few of them here but this is by no means an exhaustive record.

By creating computational models we accurately state how we believe the modeled systems behave, which removes any ambiguity that arises when formulating a verbal theory. The assumptions of models are made explicit, and we are able to test how these assumptions affect the models. Explicit assumptions make it easier to find flaws in our understanding, and others are easier able to fully grasp how we believe the systems behave. Computational models also make it easier for others to verify our work. By simulating models we easily see if the models provide the dynamics observed in experiments. If we observe different dynamics we know that at least some parts of our models are wrong, and our
understanding of the system is flawed and must be revised.

Certain hypotheses can be too difficult, costly, or unethical to test experimentally. There can be many reasons for this, one example being when there are multiple interconnected systems and it is impossible to isolate the elements specific to the hypotheses. Computational models can be used to test such hypotheses, as we have complete control over the modeled systems. Additionally, we can use models to test the effect specific parts of the modeled systems have on the systems as a whole, and thereby identify key factors of the systems. Well-tested models can also be used to predict outcomes, one example being weather forecasting.

The speed and ease of running computational models compared to performing the equivalent experiments make computational models an invaluable tool to help guide and refine experiments. By exploring the models we are able to build an intuition for how the systems behave under various conditions in a way that would not be feasible by only performing experiments. Simulations of models can be used to find hypotheses, which can be tested experimentally. This can help reduce the number of experiments required. Computational models can help guide which experiments to perform and experimental techniques to use, as the experiments and techniques best suited to the simulated dynamics can be chosen.

In addition to the reasons stated above, we can use computational models to find basic concepts of wide applicability, uncover new phenomena to examine, link different levels of detail, and more (Hillis, 1993; Council, 2005; Sterratt et al., 2011; Brodland, 2015).

From the above, it might seem that modeling is the be-all and end-all. However, models do not replace experiments, but rather complement them. To build models of specific systems we require experimental data on the systems. Additionally, models are only able to prove that a specific mechanism results in a particular dynamic. They are unable to prove that the mechanism is the reason we observe that particular dynamic in nature. For that we require experiments. It is important to note that all models are simplifications and abstractions of the system to be modeled. In the words of George E. P. Box, “All models are wrong but some are useful” (Box, George E. P., 1976). Models can never include all details found in nature but instead focus on including the most important parts of a specific system. What is considered important depends on the phenomena we want to examine.

3.2 Creating computational models

In Paper III we developed a new computational model. Creating a computational model is a complicated task, and the process depends heavily on the phenomena of interest. The first step is to formulate a conceptual model of the system of interest, that is, what the basic components and interactions of the system are.
Here we make several assumptions of the system we want to model, which can be correct or incorrect and need to be tested.

We must also decide on how detailed we wish the model to be, that is, how many details do we require to reproduce the dynamics of interest. More details can always be added to a model to make it more realistic, but the cost is a more complex model, which has several drawbacks. A more complex model is more computationally costly, so we might be limited in either the size, timescale, or both, of the system we can simulate. It is also generally more difficult to analyze and draw conclusions from more complex models. A more complex model also requires more effort to construct and is more prone to errors.

Adding extra detail might not be crucial for the phenomena we want to examine, and only makes the model unnecessarily complicated. As an example, if we want to examine how the connectivity in a neural network affects the spike timing, we do not need to know the shape of the action potentials, only the times when the spikes arrive. By adding more detail to the model, for example details providing a correct action potential shape, the model becomes more computationally costly, and we might have to reduce the number of neurons in the network. As such, adding that level of detail to the model is detrimental to our goal.

The second step when creating a model is to translate the conceptual model to a mathematical form (stated as a set of equations), which must be implemented on the computer. A suitable method for solving the equations must be chosen, examples being: finite element, finite difference, Monte Carlo, lattice Boltzmann, evolutionary computation and multiscale approximation, to mention a few. Choosing a suboptimal method, or computer language, can be costly in terms of model efficiency and quality of the results.

Once the model is implemented it must be tested to ensure that the mathematical equations are correctly implemented and that it gives correct results. We can test the model by running it for cases where the results are known and compare the computational and experimental results. If the model and experimental results correspond we can be reasonably sure that our model accurately reflects the system of interest.

One important aspect of creating a computational model is to determine the values of the many parameters that describe the system. Some of these parameters can be found by experimental measurements, while others are free parameters that must be determined by the modeler. One cause of free parameters is a lack of experimental techniques that enable us to measure the free parameters. Another cause of free parameters is parameters that are phenomenological abstractions that do not represent directly measurable physical entities e.g., because they represent the combined effect of several distinct physical processes. The free parameters are tuned by the modeler to values that make the model output match a set of experimental constraints, a process called parameter estimation. Pa-
Parameter estimation is an active field of research which is related to uncertainty analysis. We will go into further detail on parameter estimation in section 3.4.

3.3 Computational models used in this thesis

The essential aspect of neurons is that they are connected to other neurons and communicate by sending action potentials (also called spikes) between each other. Computational models in neuroscience typically aim to model different aspects of this behavior. A large number of computational models are in use in neuroscience today, from simplified models of spiking neurons (e.g., Izhikevich, 2003) to models of large networks of biophysically detailed neurons (e.g., Markram et al., 2015). There are large databases with thousands of models ready for researchers to use (Peterson et al., 1996; Le Novère, 2006). The computational models in neuroscience can be roughly divided into three main types, conductance-based models (used in Paper I, II and III), integrate-and-fire models (used in Paper I and V), and firing-rate models (Dayan and Abbott, 2001). In the next two sections, we go into detail on how conductance-based models and integrate-and-fire models work, the two types of neuron models encountered in the work in this thesis.

3.3.1 Conductance-based models

The goal of conductance-based models is to model the biophysical mechanisms that give rise to action potentials in neurons and thereby get accurate predictions of neuron dynamics, such as the action potential shape. The various components related to the generation of action potentials are modeled as electrical circuit elements, and we are typically interested in how the membrane potential evolves with time. Conductance-based models were used in Paper I, II and III.

The notable feature of the conductance-based models is that we model the conductances of various ion channels in the cell membrane. Action potentials are generated by current across the cell membrane. This current is caused by ion channels letting ions through. Many of these channels are highly selective to which ions they can let through and are typically labeled by the ion they are most permeable to. A neuron generally has a dozen or more different types of ion channels. Ion channels are either active or passive. Active channels change their conductance in response to the membrane potential, whereas passive channels have a constant conductance.

The current $I_X$ per unit area through an ion channel of type $X$ is given by

$$I_X = g_X(V - E_X),$$  \hspace{1cm} (3.1)

where $g_X$ is the conductance of the specific ion channel per unit area, $V$ the membrane potential and $E_X$ the reversal potential of the ion. Much of the complexity
in the behavior of neurons arises because of the active ion channels, where the conductance $g_X$ can be dependent on the voltage, various ion concentrations, or a host of other mechanisms, and thereby varies with time.

The total current per unit area across the cell membrane ($I_{\text{membrane}}$) due to the ion channels is found by summing up the current contributions from the different types of ion channels

$$I_{\text{membrane}} = \sum_x g_x (V - E_x). \quad (3.2)$$

**Single-compartment models**

Conductance-based models can be divided into single-compartment models and multi-compartment models. Single-compartment models were used in Paper I, II, and III. In single-compartment models, the neuron is considered to consist of only a single compartment, modeled as a single electrical circuit. There is only a single membrane potential $V$ we are interested in, and we only have to solve one main equation. The basic equation for how the membrane potential of a neuron evolves with time is (see e.g., Dayan and Abbott, 2001 for a derivation):

$$c_m \frac{dV}{dt} = -I_{\text{membrane}} + I_{\text{ext}}. \quad (3.3)$$

Here, $c_m$ is the membrane capacitance per unit area. $I_{\text{ext}}$ is any external current into the neuron, for example, from an electrode in an experimental setting or from synapses, given as current per unit area.

**The Hodgkin-Huxley model**

One of the most famous models in neuroscience is the Hodgkin-Huxley model (Hodgkin and Huxley, 1952), which was the first quantitative neuron model with active membrane mechanisms. The Hodgkin-Huxley model is an example of a conductance-based, single-compartment model. We performed an uncertainty quantification and sensitivity analysis of the Hodgkin-Huxley model in Paper I. The Hodgkin-Huxley model contains three membrane currents, a sodium current ($I_{\text{Na}}$) consisting of Na$^+$ ions, a potassium current ($I_{\text{K}}$) consisting of K$^+$ ions, and a leak current ($I_L$) which takes care of the current from other types of ion channels not explicitly modeled.

The potassium conductance is dependent on the membrane potential. Hodgkin and Huxley modeled the potassium ion channel as a set of four gates that can be either open or closed. The gating variable $n$ describes the probability for one gate to be open, and is described by a differential equation with parameters fitted to experimental results (see Sterratt et al., 2011 or Dayan and Abbott, 2001 for the full description). The potassium conductance is modeled as

$$g_K = \tilde{g}_Kn^4, \quad (3.4)$$
where $\tilde{g}_K$ is the maximum conductance per unit area if all potassium channels are open.

The sodium conductance is modeled as having two different gating variables, $m$ and $h$. $m$ behaves similarly to $n$, while $h$ represents the level of inactivation of the ion channel. Each of these gating variables is described by a differential equation with parameters fitted to experimental data (again, see Sterratt et al., 2011 or Dayan and Abbott, 2001 for the full description). The equation for the sodium conductance is:

$$g_{Na} = \tilde{g}_{Na}m^3h,$$  \hspace{1cm} (3.5)

where $\tilde{g}_{Na}$ is the maximum conductance per unit area if all sodium channels are open.

The leak current is the least interesting. It is passive and constant, and thereby only described by its constant conductance per unit area $\tilde{g}_L$:

$$g_L = \tilde{g}_L.$$  \hspace{1cm} (3.6)

Combining equations (3.2) to (3.6) gives us the following main equation for the Hodgkin-Huxley model:

$$c_m \frac{dV}{dt} = -\tilde{g}_L (V - E_L) - \tilde{g}_{Na}m^3h(V - E_{Na}) - \tilde{g}_K n^4(V - E_K) + I_{ext}.$$  \hspace{1cm} (3.7)

### Multi-compartment models

Neurons spread across large areas in order to communicate. As such, modeling them as a single compartment is an approximation that removes the spatial extent of the neuron. Multi-compartment models are an extension to single-compartment models which takes the spatial extent of neurons into account. The model of an interneuron in the dorsal lateral geniculate nucleus (dLGN) examined in Paper I is an example of a multi-compartment model. In multi-compartment models we model the neuron as a series of connected compartments. Figure 3.1 illustrates how the real morphology of the neuron is approximated with a sequence of fewer and fewer compartments. Each of these compartments is modeled as a separate electrical circuit.

Each compartment $i$ has its own membrane potential $V_i$ and receives input from the neighboring compartments. We thereby have to solve an equation for the membrane potential of each compartment. The current in each compartment is modeled similarly to the single-compartment model, with the addition of a current from each neighboring compartment. The current between compartment $i$ and $i+1$ is given by (see e.g., Dayan and Abbott, 2001 for a derivation):

$$I_{i,i+1} = \frac{d}{4R \pi l^2} (V_{i+1} - V_i),$$  \hspace{1cm} (3.8)
where $R_a$ is the resistivity of the intracellular medium, $l$ is the length of a compartment and $d$ the diameter of the compartment, assuming the diameter of a compartment is constant.

For a non-branching cable each compartment has two neighbors (except at the end), and we can use equation (3.3), adding the current from the two neighboring compartments using equation (3.8):

$$
\frac{c_m}{d} \frac{dV_i}{dt} = - \sum_X g_{i,X} (V - E_X) + \frac{d}{4R_a l^2} (V_{i+1} - V_i) + \frac{d}{4R_a l^2} (V_{i-1} - V_i) + I_{ext}. \tag{3.9}
$$

Compartments where the neuron branches have three or more neighboring compartments, while compartments at the end of a branch only have one neighboring compartment. Multi-compartment models end up providing a large set of coupled differential equations that must be solved.

There are several simulators for modeling multi-compartmental neurons, two examples being GENESIS (Bower and Beeman, 1998) and NEURON (Hines and Carnevale, 1997). We have used NEURON in Paper I, II, and III. NEURON (Hines and Carnevale, 1997) is a commonly used simulator for multi-compartmental neural models. NEURON uses the Hoc scripting language (Kernighan and Pike, 1984) but can be used with a graphical user interface or through the Python interface (Hines, Davison, et al., 2009). It can simulate individual neu-
3.3.2 Integrate-and-fire models

Integrate-and-fire models are a simplification of the conductance-based models. Instead of modeling the biophysical properties of neurons, integrate-and-fire models aim to capture the essential dynamics of neurons, namely that neurons generate action potentials once the membrane potential reaches a specific threshold. The difference between the two types of models is illustrated in Figure 3.2. The conductance-based model (in red) captures the shape of the action potentials, while the integrate-and-fire model (in blue) only captures the timing of each action potential. The integrate-and-fire model was used in Paper I and V.

Integrate-and-fire neurons are modeled as point neurons. The simplest version is modeled as a RC-circuit:

$$c_m \frac{dV}{dt} = - \frac{V - E_m}{r_m} + I_{ext},$$  \hspace{1cm} (3.10)
where $E_m$ is the resting membrane potential and $r_m$ is the specific membrane resistance (membrane resistance per unit area).

This equation is commonly written using the membrane time constant $\tau_m = c_m r_m$:

$$\tau_m \frac{dV}{dt} = E_m - V + r_m I_{ext}.$$  \hspace{1cm} (3.11)

Once the membrane potential reaches a specified threshold $V_{\text{thres}}$ the neurons are said to fire an action potential, and we reset the membrane potential to a value $V_{\text{reset}}$.

Integrate-and-fire models are far less complex than the conductance-based models and are therefore much faster to evaluate. Due to their increased simulation speed, integrate-and-fire models are popular in large neural networks. One of the larger networks simulated had $1.51 \times 10^9$ neurons with $16.1 \times 10^{12}$ synapses (Jordan et al., 2018), which is a network the size of a gray parrot’s brain (Olkowicz et al., 2016).

The integrate-and-fire model can be extended in many different ways. For example, an adaptive integrate-and-fire model can be created by adding an hyperpolarizing current to equation (3.11) (Brette and Gerstner, 2005). Examples of other non-linear variants are the quadratic integrate-and-fire model (Latham et al., 2000; Hansel and Mato, 2001), and the exponential integrate-and-fire model (Fourcaud-Trocmé et al., 2003).

There are several different simulators for creating networks of neurons, such as Brian (Goodman and Brette, 2009), NEST (Peyser et al., 2017), and Emergent (Aisa et al., 2008). In this thesis, we have performed an uncertainty quantification and sensitivity analysis of a NEST network model (Brunel, 2000) in Paper I. NEST is a simulator for large networks of spiking neurons, which focus on the dynamics and structure of the networks. NEST is implemented in C++ and can either be used as a standalone application or through its Python interface (Eppler et al., 2008). NEST has several types of integrate-and-fire neurons and synapses implemented.

3.3.3 Computational models of endocrine pituitary cells

Action potentials are not only generated by neurons, but also by other types of cells such as heart cells (Feher, 2012), skeletal muscle cells (Hopkins, 2006), and cells in the endocrine system (Kidokoro, 1975). The mechanics for how these cells generate action potentials are mostly the same as in neurons. As such, computational models of these cells share many of the same traits as neuron models.

Two of the computational models examined in this thesis are models of endocrine pituitary cells in rats (Paper II) and Japanese rice fish (Paper III). Endocrine pituitary cells are cells in the pituitary gland, a part of the endocrine
system. The endocrine system is a chemical messenger system that operates through the use of hormones. The pituitary gland produces hormones responsible for controlling processes such as metabolism, nervous system functions, and growth and development (Ooi et al., 2004; Stojilkovic et al., 2010). There exist several types of hormone-producing cell types in the pituitary gland. One of these types of cells is the gonadotropes (modeled in Paper II and III), which release hormones that act on the reproductive system (Pierce and Parsons, 1981; Ooi et al., 2004).

Endocrine pituitary cells behave much like neurons. They have various voltage-gated ion channels and can fire spontaneous action potentials (Kidokoro, 1975). However, unlike neurons which use action potentials for communication, endocrine cells primarily use action potentials to regulate their intracellular Ca\(^{2+}\) level, which in turn controls hormone release rate (Stojilkovic et al., 2010). The action potentials in neurons are generally mediated by Na\(^{+}\)-channels, while Ca\(^{2+}\) dependent action potentials are common in endocrine pituitary cells (Van Goor, Zivadinovic, et al., 2001). Studies in different species have found that the action potentials are mediated by either Ca\(^{2+}\)-channels or a mix of Ca\(^{2+}\) and Na\(^{+}\)-channels (Van Goor, Goldberg, et al., 1996; Van Goor, Zivadinovic, et al., 2001). Most computational models of endocrine pituitary cells have not included Na\(^{+}\)-channels as they have been based on data from rats, where Na\(^{+}\)-channels generally are inactivated (Stojilkovic et al., 2010).

### 3.4 Estimating the parameters of a model

As mentioned, all computational models contain parameters that describe the system being modeled, for example, the parameters describing the ion channel dynamics. Some of the parameters are free parameters that must be fitted so the model matches a set of experimental constraints. Parameter estimation is the process of finding these free parameters. One example is fitting the dynamics of an ion channel so the voltage trace of a neuron model is close to the measured voltage trace. Most neuroscience models contain free parameters.

Historically, the free parameters were tuned manually by trial and error where the modeler tested different parameters until the desired results were obtained. However, the increased complexity of computational models makes this approach less effective. Today a variety of automated parameter estimation algorithms have gradually taken over (Bhalla and Bower, 1993; Vanier and Bower, 1999; Druckmann et al., 2007; Van Geit, Achard, et al., 2007; Van Geit, De Schutter, et al., 2008; Taylor et al., 2009; Hay, Hill, et al., 2011; Svensson et al., 2012; Bahl et al., 2012; Friedrich et al., 2014; Pozzorini et al., 2015; Van Geit, Gevaert, et al., 2016; Mäki-Marttunen et al., 2018).

Two approaches to parameter estimation are the classical deterministic approach and the Bayesian approach. The classical approach finds a set of param-
eter values that best reproduce a particular experimental result. The Bayesian approach, on the other hand, finds a probability distribution for the parameters that most likely explain the experimental results. Bayesian parameter estimation is also able to take the uncertainties of the experimental results into account.

No matter which method is used, the parameter estimation does not guarantee a unique solution corresponding to one “correct” parameter set. It is often the case that a wide range of different parameter combinations gives rise to similar model dynamics, called biological degeneracy (Marder and Taylor, 2011). The biological degeneracy makes it so many different sets of parameters can reproduce one experimental result, and necessitates the use of uncertainty quantification. Due to the effects of biological degeneracy, it is likely that the Bayesian approach is better suited, as we are not limited to fixed sets of parameters. The Bayesian approach also provides the probability distributions for the parameters that we require in an uncertainty analysis.

3.4.1 Deterministic parameter estimation

Deterministic parameter estimation is commonly used in neuroscience. The general idea behind deterministic parameter estimation is simply to run the model with various parameters, compare the simulated results to the experimental results, and select the set of parameters that give the most similar results.

Performing a deterministic parameter estimation consists of two mostly independent steps. The first step is to decide on an error function, which quantifies how well our results match the experimental data. It is this error function that decides what is the best result. The solution space of the error function of a model is illustrated in Figure 3.3. The second step is to select an optimization algorithm that searches this space and finds the minima of the error function.

Error functions

The choice of an error function is important because it influences the final parameter set, as well as the performance of the parameter search. The error function should reflect the properties of the experimental data that we want the model to reproduce. It should preferably also be quick to evaluate, as we have to evaluate the error function for each iteration in the optimization algorithm. The solution space of the error function should also be smooth, to make it easier for the optimization algorithm to converge to the global minimum.

**Point-to-point comparison.** The simplest form of an error function is simply to perform a point-to-point comparison of the model results and experimental data. As an example, a simple point-to-point comparison is to calculate the root
mean square (rms) of the difference between the two:

\[
\text{rms} = \sqrt{\frac{1}{N} \sum_{i=0}^{N} (V_{\text{model},i} - V_{\text{data},i})^2}. \tag{3.12}
\]

Here \(V_{\text{model}}\) is the voltage trace of the model and \(V_{\text{data}}\) the voltage trace of the experimental data. \(N\) is the number of time points in the voltage traces.

One problem with a point-to-point comparison is that it is extremely sensitive to shifts in the spike timing between two voltage traces. A small time-shift between essentially the “same” spike into two voltage traces can cause an error that is larger than if the voltage trace was compared to a constant voltage trace. This effect is illustrated in Figure 3.4. Figure 3.4A shows the voltage trace (membrane potential) of the Hodgkin-Huxley model compared to the same voltage trace shifted 2 ms in time. There is little overlap between the two voltage traces. The absolute difference in Figure 3.4B shows the large difference caused by the time shift. The absolute difference (and the root mean square of the difference) is greater between the two Hodgkin-Huxley model voltage traces than between the Hodgkin-Huxley model voltage trace and the constant voltage trace. As such, the constant voltage trace is the best fit. However, from a biological perspective, the constant voltage trace is a much worse fit than a slightly time-shifted voltage
trace. This is due to the fact that neurons use action potentials to communicate. The constant voltage trace, therefore, contains “no information”, while the time-shifted voltage trace contains almost the same “information” as the original voltage trace.

Another problem with the point-to-point comparison is how to select the experimental result to compare to. When measuring the membrane potential of a neuron, the precise timing of action potentials often varies between recordings, even if the experimental conditions are the same. As such, selecting a single experimental voltage trace to compare to is to some extent an arbitrary choice. Since the experimental data generally display a large variation, it is often meaningless and misleading to base the success of a computational model on a direct point-to-point comparison between a particular experimental recording and model output (Druckmann et al., 2007; Van Geit, De Schutter, et al., 2008). Comparing to the mean of several experimental voltage traces does not work either, due to the aforementioned sensitivity to small time shifts.

**Feature based.** Another set of error functions compares features of the voltage traces, instead of directly comparing the voltage traces. Examples of features useful in neuroscience are action potential width, spike rate, and afterhyperpo-
A memorandum that is not limited to using a single error function. The simplest method of using multiple error functions is to combine them using a weighted sum

$$E(\mathbf{X}) = \sum_i w_i E_i(\mathbf{X}),$$  \hspace{1cm} (3.13)
where $X$ is the model parameters, $w_i$ is the weights and $E_i$ is the error functions. By using multiple error functions we can optimize the model to reproduce several features important for a specific system, and not just optimize the model towards a single feature. Different features can also be assigned varying importance by changing the weights.

**Optimization algorithms**

Many different optimization algorithms exist for exploring the solution space of the error functions in order to find the optimal solution. The optimization algorithms can generally be divided into local and global methods. Local methods are able to find local minima in the solution space, and thereby only reliably find the global minimum when there is one minimum in the solution space. However, it is difficult to know the shape of the solution space before we start the exploration, and many biological optimization problems have multiple local minima (Achard and De Schutter, 2006). Additionally, due to the biological degeneracy, many biological systems have a complex parameter landscape where several parameter sets give rise to similar solutions (Prinz, Bucher, et al., 2004; Achard and De Schutter, 2006; Taylor et al., 2009; Marder and Taylor, 2011). As such, we should consider there to be multiple global “minima” and we should generally use global optimization algorithms.

**Brute force.** The conceptually simplest optimization algorithm is the brute force search. This method scans the entire parameter range and calculates the model output for each set of parameters. The error function is then calculated for each model evaluation and we select the parameter set that gives the smallest error.

The disadvantage of the brute force search is that it is slow and suffers from the curse of dimensionality. The brute force algorithm scales as $O(N^D)$, where $D$ is the number of parameters and $N$ is the number of samples for each parameter. The brute force method has been used in neuroscience (Bhalla and Bower, 1993; Prinz, Billimoria, et al., 2003).

**Evolutionary algorithms.** Evolutionary algorithms are a class of methods that are based on natural evolution and use the principle of “survival of the fittest” to improve the solution (Eiben and Smith, 2015). Points in the parameter space are considered to be individuals with a fitness as determined by the error function.

Evolutionary algorithms start by selecting a population of points in the parameter space. Parents are selected from these individuals according to their fitness. These parents reproduce by mixing their parameters to create children, which are individuals with new parameter combinations. Mutations are then
introduced by randomly modifying the parameters of individual children. The fitness of the children is then evaluated and a selection of the individuals with the best fitness is chosen as the new population.

This process is repeated until a specific stop criterion is met. Many different types of evolutionary algorithms exist, such as genetic algorithms, evolution strategies, and differential algorithms.

**Other algorithms.** Various other methods for finding the minima exists, examples being simulated annealing (Kirkpatrick et al., 1983), gradient descent methods, and particle swarm optimization (Kennedy and Eberhart, 1995), to mention a few.

### 3.4.2 Bayesian parameter estimation

Bayesian statistics is a field of statistics where probabilities are interpreted as the degree of belief in an event, which can be updated as we gather more information. The goal of Bayesian parameter estimation is to find the *posterior* probability distribution of the model parameters that best explain the observed experimental results. This is unlike more traditional deterministic approaches, where we only find a specific parameter set. Bayesian parameter estimation is able to take into account both measurement uncertainty, as well as the biological variability coming from the previously mentioned biological degeneracy.

In Bayesian parameter estimation we take into account all *prior* information that we know of the modeled system. For example, in neuroscience, this prior information may include the maximum and minimum observed values for an ion channel conductance. Once we have defined the prior probability distribution we can calculate the *likelihood* function, which describes how well the data is explained by the model predictions, when given some parameters. Once we have calculated the likelihood we use it and the prior probability to find the posterior probability distribution. For a brief introductory review of Bayesian parameter estimation see Allmaras et al., 2013, and for a more practically oriented approach see Davidson-Pilon, 2016.

One class of methods for performing the Bayesian parameter estimation is Markov Chain Monte Carlo methods. Several variants of the Markov Chain Monte Carlo methods exist, examples being the Metropolis algorithm (Metropolis et al., 1953), Metropolis-Hastings (Hastings, 1970), Gibbs Sampling (Geman and Geman, 1984), Hamiltonian Monte Carlo (Duane et al., 1987), and No-U-Turn Sampler (Hoffman and Gelman, 2011). The general idea of these methods is to sample the parameter space and calculate the likelihood that the measured experimental data is matched by the model when using the sampled parameter set. The exploration of the parameter space randomly moves towards the areas with the highest likelihood and spends the most time in these areas. We collect
the parameter samples and use these to map out the posterior distribution of the parameter space. After enough samples, we have an accurate representation of how this distribution looks. Other methods for performing the Bayesian parameter estimation also exist, for example, Laplace approximations and variational Bayes methods (Davidson-Pilon, 2016).

Bayesian parameter estimation has been used in neuroscience (Stevenson et al., 2009; Cavanagh et al., 2011; Park and Pillow, 2011), but its use is not widespread. On the other hand, Bayesian models have started to gain traction in computational neuropsychology (Parr et al., 2018).

3.5 Replicability in computational neuroscience

Replicability is one of the cornerstones of science and is the ability to perform the exact same experiments and simulations as done by another researcher and obtain the same results and conclusions. The ability to verify the validity of scientific claims and conclusions made by other scientists is critical (Crook et al., 2013; Collberg and Proebsting, 2016; Plesser, 2018). If no one can reproduce my results, why should anyone trust them? Many fields of science are currently experiencing challenges in regards to replication (Collberg and Proebsting, 2016; Baker, 2016; Munafò et al., 2016; Rougier et al., 2017) and computational neuroscience is no exception (Topalidou et al., 2015; Manninen et al., 2018; Pauli et al., 2018).

Collberg and Proebsting, 2016 measured the extent of how much of the code and data associated with the papers from the last (at their time of writing) eight Association for Computing Machinery conferences that were available and could be built with a reasonable amount of effort. Of the 402 papers that were accompanied by code, only 32.3% had code that was able to be built within ≤ 30 minutes, and only 48.3% were able to be built with no time constraints. Additionally, they only examined if they were able to build the code in the papers and did not check if the code actually ran. Neither did they examine if the code and the description in the papers were in agreement, or if the methods used, results found, or both were correct.

Various initiatives have been started to improve upon this problem, from journals starting to require code and data to be published alongside articles (for example Plos One1), to the development of tools to improve the ease of replication, such as Zenodo2 which enables version controlled sharing of data and code. One such initiative is ReScience (Rougier et al., 2017), an openly-peer-reviewed journal for replications of previously published computational research. The work in Paper II involved the replication of a previously published neuroscience model (Tabak et al., 2011), and is submitted to ReScience.

---

1journals.plos.org/plosone/s/materials-and-software-sharing
2zenodo.org
3.5.1 Difference between reproduction and replication

It is useful to distinguish between reproduction and replication. However, there is currently no commonly agreed upon definition of the two (Goodman, Fanelli, et al., 2016; Plesser, 2018). Completely opposite definitions exist e.g., see Manninen et al., 2018 and Rougier et al., 2017. Plesser, 2018 gives a brief history and overview of the two terms. In this thesis we use the definition used in ReScience (Rougier et al., 2017), which Plesser refers to as Claerbout terminology (Claerbout and Karrenbach, 1992). In this terminology, reproduction is defined as getting the same results when running the same software on the same input (Rougier et al., 2017). Replication is defined as writing a new implementation of the software from the description in the original publication and obtaining similar enough results. (Rougier et al., 2017). What is similar enough can only be judged by an expert within each field.

3.5.2 Challenges for reproducibility

We might think that computational science should excel at reproducibility. We are in complete control of the computer system and avoid many of the uncontrollable variations and pitfalls that experiments are burdened with. However, computational science has its own problems that limit reproducibility.

For a simulation to be reproducible, the code must be accessible. A common problem hindering reproducibility is simply that the code is not available (Peng, 2011; Rougier et al., 2017). The code is often not posted online, and various causes, such as changes in e-mail addresses and bad archiving systems can make it impossible to get access to the original code. Ideally, all code should be available upon publication to promote reproducibility and make it easier for other researchers to build upon the work in the publication.

A problem related to code sharing is that it is often not known which version of a code produced the results shown in the original publication (Collberg and Proebsting, 2016). This problem can be alleviated by the use of version control software such as git or svn. The code can then be published on services such as Bitbucket³ and GitHub⁴, or databases such as ModelDB (Peterson et al., 1996). Even better is to use specialized scientific repositories, such as Zenodo, which gives specific versions of a code a DOI (Digital Object Identifier) and ensures that the code will not disappear (for example by deleting the GitHub repository). All parameters and input used by the code should also be version controlled. One helpful tool is Sumatra (Davison, 2012), which keeps track of which version of the code, input, and parameters generated the different results.

Computational code generally has a wide variety of dependencies and getting these to work is not necessarily a trivial task (Collberg and Proebsting, 2016).

---

³bitbucket.org
⁴github.com
Differences in the dependencies can also have a large impact on the results (Benureau and Rougier, 2018). Recording details of the execution environment is therefore a necessary step. Another solution is to use containers, such as Docker\textsuperscript{5}, which allows researchers to package code along with all the required dependencies. However, Docker is still not an ideal solution (Boettiger, 2014).

Even when the code is available, it can be hard to understand how it works. There are several best practices that should be followed when writing code, both to make it easier for others to read the code (or yourself in a couple of years’ time), and to reduce the risk of introducing bugs and errors. These best practices include, but are not limited to, choosing descriptive variable names, commenting on the design and purpose (and not the mechanics of the code), modularizing the code, and using a unit testing library (Wilson, Aruliah, et al., 2014; Wilson, Bryan, et al., 2017; Benureau and Rougier, 2018; Pauli et al., 2018).

Many of the best practices mentioned in this section, as well as a few more, have been followed in the work in this thesis, in order to help increase the reproducibility and replicability of the work.

### 3.5.3 Challenges for replicability

To be able to perform a replication of the results in a publication, the publication must contain a complete description of the work performed. However, it is often the case that specific implementation details are missing from the publication (Manninen et al., 2018). Examples of these details include missing parameters, model equations, assumptions, details of the algorithm, and missing numerical method descriptions. Publications also generally aim to convey the main ideas behind the research performed. Adding all details required to implement the code would detract from the main focus of the publication, and would make the publication that much harder to read. Therefore, having access to the code that generated the results in the original publication is generally a necessity when performing a replication. As stated by Buckheit and Donoho, 1995: “An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures”.

Another problem is that the model can be mathematically well defined, but can be numerically sensitive to its implementation details. As shown by Crook et al., 2013, even simple neuron models such as a simple linear first-order ordinary differential equation (Lapicque, 1907) with an analytical solution can have widely varying dynamics dependent on the implementation. Models of neural networks are especially sensitive to small changes in the network and these changes can lead to widely varying spike timings (Crook et al., 2013). This sensitivity means

\textsuperscript{5}docker.com
the model results are dependent on the simulator and numerical methods used (Gleeson et al., 2010; Pauli et al., 2018). It is therefore not sufficient to specify the model equations to ensure replicability. In general, it should be noted that replication of a neural network generally does not aim for a spike perfect replication, but rather a statistical equivalence e.g., similar firing rates, spike train correlations, etc. (Pauli et al., 2018).

In order to decouple models from the numerical implementation details, it is argued that models should be simulator independent (Hinsen, 2014). Tools for creating such simulator independent models in neuroscience exist, for example, PyNN (Davison et al., 2009) and NeuroML (Gleeson et al., 2010). However, to ensure replicability, all numerical implementation details must still be stated (for example which simulator was used).
Uncertainty quantification and sensitivity analysis

All computational models contain parameters that describe the system to be modeled. It is common that models in neuroscience contain a number of parameters that are uncertain. However, despite the parameter uncertainty it is not usual to take these uncertainties into account. In these cases, it is a priori difficult to know to what degree we can trust the model results.

Uncertainty quantification provides rigorous procedures to quantify the uncertainty in the model output that occurs due to uncertainty in the model parameters. Instead of assuming fixed model parameters, and getting a single model output (as depicted in Figure 4.1A), we assign a distribution of possible values to each model parameter. The parameter uncertainty is propagated through the model and gives rise to a distribution of possible model outputs (as depicted in Figure 4.1B).

After an uncertainty quantification has been performed, it is useful to perform a sensitivity analysis. The goal of a sensitivity analysis is to quantify how much of the uncertainty in the model output is caused by the uncertainty in each parameter (Saltelli, 2002b). The higher the sensitivity to a parameter, the more the model output changes when we vary that parameter. In Paper I we developed a toolbox, called Uncertainpy, tailored towards performing uncertainty quantification and sensitivity analysis in neuroscience.
Figure 4.1: Illustration of an uncertainty quantification of a model. 

A: A traditional model where each input parameter has a chosen fixed value, and we get a single output of the model (gray). 

B: An uncertainty quantification of the model takes the distributions of the input parameters into account, and the output of the model becomes a range of possible values (light gray). 

Illustration from Tennøe et al., 2018 (Paper I).
4.1 Origins of parameter uncertainty

The uncertainties in the model parameters have many possible origins and we go into details on some of them in this section.

4.1.1 Measurement uncertainty

All measurements are subject to uncertainty and all measured parameters thereby have an associated uncertainty. As stated by the Joint Committee for Guides in Metrology, 2008 (a part of the International Bureau of Weights and Measures), all measurements of physical quantities should be reported along with the measurement uncertainty, so the quality of the measurements can be determined. If all measured numbers should be reported with their measurement uncertainty, how can we ignore the uncertainty of measurements when we use them as parameters in a model?

4.1.2 Biological variability

A large and important cause of parameter uncertainty in neuroscience is due to biological variability. Unlike parameters in the physical sciences, such as the gravitational constant, many parameters in biological systems are variable (Marder and Taylor, 2011). They can, for example, vary between different neurons of a single species in a neural network (Edelman and Gally, 2001; Hay, Schürmann, et al., 2013), or dynamically within a single neuron, for example, due to homeostatic mechanisms or plasticity (Marder and Goaillard, 2006).

This large amount of biological variability is made possible by biological degeneracy. Biological degeneracy is the ability of different biological structures to perform the same function and have the same output (Edelman and Gally, 2001). In nature there is a wide range of different solutions that give rise to similar (or the same) dynamics (Bhalla and Bower, 1993; Beer et al., 1999; Goldman, Golowasch, et al., 2001; Golowasch et al., 2002; Prinz, Bucher, et al., 2004; Tobin, 2006; Schulz et al., 2007; Taylor et al., 2009; Marder and Taylor, 2011).

This biological variability makes it more challenging to measure the parameter values. As an example, it is not uncommon that the ion channel conductances are measured from sets of 10-20 neurons and then either the best (depends on the experiment) or the average is chosen as the value for the specific conductance (Marder and Taylor, 2011). None of these solutions are ideal. Neither solution takes into account potential correlations between the conductances. They might, for example, be inversely correlated and the first conductance is large when the second is small. The option of averaging the results is also problematic because a typical neuron might not be well represented by the mean (Golowasch et al.,
The best way of representing such parameters is to use probability distributions which capture the observed spread of the parameters.

It is recommended studying a population of neurons instead of creating a model that represents the generic dynamics of a single neuron (Golowasch et al., 2002; Prinz, Bucher, et al., 2004; Achard and De Schutter, 2006; Günay et al., 2008; Prinz, 2010; Marder and Taylor, 2011). One method for studying a population of neurons is to perform an uncertainty quantification. By performing an uncertainty quantification we are able to find the mean and range of possible model results for a population of model neurons with parameters varying within the given parameter distributions.

### 4.1.3 Missing experimental techniques

Parameters that we lack experimental techniques to measure is another cause of uncertain parameters. Instead, we have to fit the parameter value so the model reproduces an experimental result we are able to measure (as described in section 3.4). This process introduces uncertainty as there are several “correct” parameter sets to find due to the biological degeneracy.

### 4.2 Uncertainty analysis in neuroscience

The importance of uncertainty analysis of computational models is well known in a wide variety of fields, for example, ocean wave modeling (Yıldırım and Kariniadakis, 2015), hydrology (Beck, 1987), fluid dynamics (Najm, 2009), economy (Leamer, 1985), meteorology (Rossa et al., 2011), engineering (Oberkampf et al., 2002), chemistry (Turányi and Turányi, 1990), and systems biology (Marino et al., 2008). Uncertainty analysis is of equal, if not likely greater, importance in neuroscience due to the inherent variability in biological systems.

However, there is no generally accepted practice of uncertainty analysis in neuroscience. Some steps have been taken in this direction, but the methods used are often rather simple one-at-a-time methods, where one parameter at the time is varied and we examine how the model output is affected (see e.g., De Schutter and Bower, 1994; Blot and Barbour, 2014; Kuchibhotla et al., 2017). These methods are unable to take potential dependencies between parameters into account. Other methods used are local methods, which are limited to examining small perturbations around a single point in parameter space (see e.g., Gutenkunst et al., 2007; Blomquist et al., 2009; O’Donnell et al., 2017). Global methods, which are to able take the complete parameter space into account, have been used, but only sparingly (see e.g., Torres Valderrama et al., 2015; Halnes et al., 2009).
4.3 Uncertainty quantification

An uncertainty quantification quantifies how the model output is affected by the parameter uncertainties. The general approach is that each parameter is assigned a distribution of possible values (illustrated in Figure 4.1B), instead of using fixed values for the model parameters (illustrated in Figure 4.1A). These parameter uncertainties are then propagated through the model and we get a distribution of possible model outputs. The goal of the uncertainty quantification is to describe this output distribution through statistical metrics such as the mean and the variance.

Performing an uncertainty quantification has several benefits. First and foremost, by performing an uncertainty quantification we are able to properly account for the effects of the uncertain parameters. We are able to include the biological variability in the parameters of our model and examine the “true” range of the model output. An uncertainty quantification also helps to improve our understanding of the model, as we know how the model behaves in regards to uncertain parameters and the range of possible model outputs we can expect to observe. This tells us how robust the modeled system is within the explored distribution. An uncertainty quantification also makes it easier to compare the output of a model to experimental results or to other model outputs. When we know the distribution of the model output we can better quantify how similar the two results we compare are.

There are several different methods for uncertainty quantification. These methods can be divided into local and global methods, and intrusive and non-intrusive methods. Local methods are constrained to examining small perturbations around a set of fixed parameters. Global methods are able to consider the whole parameter space and, unlike the local methods, can account for complicated parameter dependencies (Eck et al., 2016).

Intrusive methods require that changes are made to the underlying model equations, and a new set of model equations must be solved, which often is a complex and challenging task. The intrusive methods therefore often require detailed knowledge of both the intrusive method used and how to solve the model equations. Non-intrusive methods, on the other hand, do not require any changes to the model equations or implementation and treat the model as a “black box”. In neuroscience, it is common to create models with the use of complex simulators, for example, NEURON (Hines and Carnevale, 1997) and NEST (Peyser et al., 2017). Changing the underlying equations of models implemented using these simulators is a difficult task not achievable by most neuroscientists. Global, non-intrusive methods are therefore generally the best methods for uncertainty quantification and sensitivity analysis in neuroscience.

Here we go into a brief overview of different methods available for uncertainty quantification. We use the same method classification as Xiu, 2010, however other
classifications also exist (see e.g., Hoon Lee and Chen, 2009)

4.3.1 Sampling-based methods

There are several different sampling-based methods. They all sample the parameter distributions, or a variant of them, and use those parameter samples to evaluate the model and calculate statistics from model evaluations.

Monte Carlo

The most used and well known of the sampling-based methods for uncertainty quantification is the Monte Carlo method. In the Monte Carlo method, we generate independent sets of parameters by randomly sampling the parameter distribution and evaluate the model for each of the parameter sets. From the collection of solutions, we calculate statistical metrics such as the mean and variance.

The disadvantage of the Monte Carlo method is that a large number of samples are required to get reliable statistics. The computational cost can quickly become prohibitive for computationally expensive models. The convergence rate of the error of the Monte Carlo method is $O(1/\sqrt{N})$, where $N$ is the number of samples (Lemieux, 2009). This slow convergence still shows one of the strengths of the Monte Carlo method, it is independent of the number of uncertain parameters. For high dimensional problems, the Monte Carlo method might be the only feasible method. Other strengths of the Monte Carlo method are that it makes no assumptions about the model and is global, non-intrusive, and easy to implement and use. For a comprehensive review of Monte Carlo methods see Lemieux, 2009.

Quasi-Monte Carlo

Various variance reduction methods have been developed to reduce the number of model evaluations needed by the Monte Carlo method. The idea is to distribute the samples more evenly in the parameter space to increase the sample coverage. These methods are collectively called quasi-Monte Carlo methods and use low-discrepancy sequences such as the Sobol sequence (Sobol, 1967) or Hammersley sequence (Hammersley, 1960), instead of randomly sampling the parameter distributions. Quasi-Monte Carlo methods generally converge as $O\left(\frac{\log D N}{N}\right)$, where $D$ is the number of uncertain parameters (Lemieux, 2009). Quasi-Monte Carlo methods scale much better than Monte Carlo methods as long as the model is sufficiently smooth and the number of uncertain parameters is sufficiently small (Lemieux, 2009). It is therefore generally better to use a quasi-Monte Carlo method than the standard Monte Carlo method.
Importance sampling

Another sampling-based method related to the Monte Carlo method is importance sampling. Importance sampling tries to sample the most important regions of the parameter space, which is useful if we model rare events. In importance sampling, we change the underlying parameter distribution so the events of interest happen more often. This new distribution is then used to generate the parameter samples used when evaluating the model. The model evaluations are weighted to correct for the biased parameter distribution. The most challenging step in importance sampling is to choose the biased parameter distribution. One method is exponential twisting/tilting (Siegmund, 1976). However, there is no general way of constructing the best distribution and we are not even guaranteed to find a distribution that is better than using the Monte Carlo method (Lemieux, 2009).

4.3.2 Surrogate-based methods

Surrogate-based methods try to approximate the model by computationally inexpensive functions, such as polynomials. These metamodels can then be used in place of the original model in a (quasi-)Monte Carlo method, speeding up the calculations.

Polynomial chaos expansions

One example of a surrogate-based method is that of generalized polynomial chaos expansions (Xiu and Karniadakis, 2002), which is a generalization of polynomial chaos expansions (Ghanem and Spanos, 1991). In this method, the model is approximated by a polynomial orthogonal of the parameter distributions. In the original polynomial chaos expansions, only Hermite polynomials could be used as the orthogonal polynomials. Generalized polynomial chaos expansions extended the theory to work for other polynomials and thereby work for more parameter distributions. In this thesis, we only work with generalized polynomial chaos expansions and we will for simplicity refer to it as polynomial chaos expansions. Two methods for calculating the polynomials are the discretized Stieltjes method (Stieltjes, 1884) or the so-called three-term recurrence relation (Xiu, 2010). Most of the computational work of polynomial chaos expansions is to determine the expansion coefficients. Different methods for this exist, such as point-collocation, stochastic Galerkin, and pseudo-spectral projection (see e.g., Xiu, 2010).

In the cases where the model is based on a set of differential equations the stochastic Galerkin method can be used. However, this method is an intrusive method and the model equations are reformulated to a coupled set of differential equations. On the other hand, both point-collocation and pseudo-spectral projection are non-intrusive methods.
An advantage of polynomial chaos expansions is that many of the statistical metrics we seek can be calculated directly from the polynomial expansion, and we do not necessarily need to perform a (quasi-)Monte Carlo method step after the metamodel has been calculated.

As long as the number of uncertain parameters is low (typically around 20), polynomial chaos expansions require far fewer model evaluations than the (quasi-)Monte Carlo methods (Xiu and Hesthaven, 2005; Crestaux et al., 2009; Eck et al., 2016). The main disadvantage of polynomial chaos expansions is that the required number of model evaluations scales worse with the number of uncertain parameters than the (quasi-)Monte Carlo methods. Polynomial chaos expansions also have reduced performance when the model has a non-smooth behavior with respect to the input parameters (Eck et al., 2016).

**Gaussian process emulation**

Gaussian process emulation is another surrogate-based method (Sacks et al., 1989). Here, the model is represented as a Gaussian stochastic process, meaning the metamodel itself is stochastic. Owen et al., 2017 performed a comparison of polynomial chaos expansions and Gaussian process emulation. They found that Gaussian process emulation has a slight edge in flexibility and robustness, while polynomial chaos expansions are slightly faster. However, neither method greatly outperformed the other, and the performance of both methods is problem dependent. This comparison only used the most common methods for each method class, in order to make an “off-the-shelf” comparison. More advanced methods than the methods used in Owen et al., 2017 exist for both classes of methods.

**4.3.3 Perturbation methods**

In perturbation methods truncated Taylor expansions are used to approximate the uncertain parameters around their mean value. The problem with the perturbation methods is that they are intrusive and only work when the magnitude of the input and output uncertainties are small, typically smaller than 10% (Xiu, 2010).

**4.3.4 Moment equations**

In this class of methods, the goal is to directly calculate the statistical moments (such as the mean and variance) of the model. The equations for the statistical moments are derived from the stochastic model equations, which then must be solved. As such, these methods are intrusive and non-trivial to use.
4.3.5 Operator based methods

Operator based methods are based on manipulating the stochastic operators in the model equations. One example of such a method is the weighted integral method (Deodatis, 1991; Deodatis and Shinozuka, 1991). The limitation of these methods is that they only work when the input and output uncertainties are small.

4.4 Sensitivity analysis

A sensitivity analysis tells us how much of the model output uncertainty is caused by the uncertainty in each parameter. We are “assigning blame” for the output uncertainty to each parameter.

By performing a sensitivity analysis we gain increased insight into how the parameters, and thereby the biological mechanisms, affect the model output (Marino et al., 2008). A sensitivity analysis can also help to identify interesting or critical regions in the model parameter space. We also find how robust the model results are to changes in the input parameters. Additionally, a sensitivity analysis helps guide the experimental focus (Zi, 2011). We can focus on making accurate measurements of parameters with high sensitivity, whereas cruder methods are acceptable for low sensitivity parameters. A sensitivity analysis is also helpful when performing parameter estimations and model reductions (Degener et al., 2004; Snowden et al., 2017; Zi, 2011). Setting low sensitivity parameters to fixed values hardly affects the variance of the model. This makes it so there are fewer parameters that must be fitted. In some cases, even entire mechanisms can be removed from the model, and we can create simpler models with much of the same dynamics.

A variety of different measures of sensitivity and methods for calculating them exists and we briefly mention a few of them here. For more in-depth reviews see Saltelli, Ratto, et al., 2007 and Zi, 2011.

4.4.1 Scatterplots

One of the simplest methods for examining the sensitivity of a model is to use the Monte Carlo method to sample the parameter distributions and generate a set of model outputs. We can then create scatterplots of the model outputs against each parameter. By examining these scatterplots we can see which parameters have the greatest influence on the model. However, this method is only qualitative.
4.4.2 Linear regression

One common measure of sensitivity, related to the scatterplot method, is to fit the model outputs using a linear regression. A Monte Carlo method is used to sample the parameter distributions and generate a set of model outputs. A linear regression is then performed for each parameter using the parameter samples and model outputs. The regression coefficients, or standard regression coefficients, can be used as a direct measure of sensitivity.

The linear regression assumes that the model is linear, and we are unable to account for any non-linear effects. The coefficient of determination, commonly denoted $R^2$, measures how much of the variance of the model the regression coefficients account for. If the coefficient of determination is close to one, the linear regression accounts for most of the variance of our model. On the other hand, if the coefficient of determination is low, our model is highly non-linear and most of the variance is unaccounted for by the linear regression. Linear regression is also unable to take interactions between parameters into account. Two parameters interact if their effect on the output is non-additive (Saltelli, Ratto, et al., 2007).

4.4.3 Sobol sensitivity indices

Another common sensitivity measure are the Sobol sensitivity indices (Sobol, 1990). This is a variance-based method that decomposes the variance of the model into fractions attributed to each parameter or combination of parameters. As such, the Sobol indices quantify how much of the model variance is caused by the uncertainty in each parameter or combination of parameters, which is illustrated in Figure 4.2.

There exist several orders of the Sobol indices. The first-order Sobol indices measure the effect each single parameter has on the model variance. Higher order Sobol indices measure the effect between a parameter and each combination of the other parameters on the model variance. The total Sobol indices include the sensitivity due to the direct effect from each parameter (first-order) and from all combinations of parameters (higher order) (Homma and Saltelli, 1996). As such, the total-order indices minus the first-order indices gives the sensitivity due to interactions between parameters. If there are no interactions between the parameters, the sum of the first-order and the sum of the total-order Sobol indices are both equal to one (Glen and Isaacs, 2012). When there are interactions present, the sum of the first-order Sobol indices is smaller than one and the sum of the total-order Sobol indices is greater than one (Glen and Isaacs, 2012).

A nice property of the Sobol indices is that both the first- and total-order Sobol indices can be directly calculated from a polynomial chaos expansion (Sudret, 2008; Crestaux et al., 2009). The Sobol indices can also be calculated with the quasi-Monte Carlo approach by using Saltelli’s method (Saltelli, 2002a;
Saltelli, Annoni, et al., 2010). This method requires $N_{MC}(D + 2)$ samples, where $N_{MC}$ is the number of samples required to achieve a specific accuracy with the Monte Carlo method. Saltelli’s method is more computationally expensive than the Monte Carlo method and we lose the independence of the number of uncertain parameters.

### 4.4.4 Morris method of elementary effects

The Morris method of elementary effects is a variant of the one-at-a-time methods (Morris, 1991; Campolongo et al., 2007). In the Morris method, we change each parameter in regular increments a number of times and calculate the elementary effect of each increment. The elementary effect is the difference between the output at the new point in parameter space and the output at the previous point, divided by the change between the two points. The average of the elementary effects of a parameter quantifies how large an influence that parameter has on the output and the standard deviation estimates interactions and non-linear effects.

The main advantage of the Morris method is that it has a low computational cost. It only requires $N_{Morris}D$ model evaluations, where $N_{Morris}$ is the number of times we change each parameter. The Morris method is therefore often used as a screening method, where the goal is to quickly find the most important parameters.
parameters. Such screening methods can be used to efficiently reduce the number of uncertain parameters. The disadvantage of the Morris method is that it is only qualitative.

4.4.5 Differential method

A simple method for performing a sensitivity analysis is to calculate the first-order partial derivatives of the model output with respect to the uncertain model parameters. This can, for example, be done using the finite difference approximation. The differential method only examines a small perturbation around a fixed point and is, therefore, a local method. Additionally, it is a one-at-a-time method and we are unable to examine interactions between parameters.

4.4.6 Other methods

Many other measures and methods for sensitivity analysis exist, examples being the Fourier amplitude sensitivity test (Cukier, Fortuin, et al., 1973; Schaibly and Shuler, 1973; Cukier, Schaibly, et al., 1975), weighted average of local sensitivities (Bentele et al., 2004), random sampling high-dimensional model representation (Li, Wang, et al., 2002; Li, Hu, et al., 2006), and multi-parametric sensitivity analysis (Hornberger and Spear, 1981), just to show the variety.

4.5 Choice of software for uncertainty analysis

Python was the programming language of choice for the work in this thesis. Python is an open-source, high-level programming language with useful scientific libraries such as NumPy (Walt et al., 2011), Pandas (McKinney, 2010) and HDF5 (The HDF Group, 1997-2018; Collette, 2013). Python is in wide use in the scientific community in general (Oliphant, 2007) and in neuroscience in particular (Einevoll, 2009; Muller et al., 2015). A wide variety of popular simulators such as NEST, NEURON, and Brian have built-in Python support. There also exists a wide range of Python libraries for neuroscience analysis, such as Elephant (NeuralEnsemble, 2017), PySpike (Mulansky and Kreuz, 2016), and OpenElectrophy (Garcia and Fourcaud-Trocmé, 2009). A multitude of other tools also exist such as the Blue Brain Python Optimization Library (BluePyOpt) for parameter optimization (Van Geit, Gevaert, et al., 2016), and PsychoPy (Peirce, 2009), a toolbox for generating visual and auditory stimuli.

There is a wide variety of software to perform uncertainty quantification and sensitivity analysis, however, few of them implement polynomial chaos expansions. A few of the more popular software with polynomial chaos expansions are UQLab (Marelli and Sudret, 2013), Dakota (Adams et al., 2014), OpenTURNS (Baudin et al., 2015), MUQ (Conrad and Marzouk, 2012), UQ Toolkit
(Debusschere et al., 2005), PUQ (Hunt et al., 2015), and Chaospy (Feinberg and Langtangen, 2015).

Of these we chose to use Chaospy for most of the uncertainty quantification and sensitivity analysis in the toolbox developed in Paper I. Chaospy is a Python package that implements a broad range of different methods for uncertainty quantification and sensitivity analysis. It is modular and flexible, which makes it easy to use from a developer’s perspective. Chaospy was developed by Feinberg and Langtangen while at Simula Research Laboratory, which has collaborations with our research group. Unfortunately, Chaospy does not implement methods for sensitivity analysis using Monte Carlo methods. We, therefore, use a modified version of methods in the Python package SALib (Herman and Usher, 2017) to perform sensitivity analysis using Saltelli’s method.
Data sharing and data standards

Experiments are one of the primary components of the scientific method and enable us to test hypotheses and theories. Technological developments are enabling more detailed experimental measurements to be performed and an ever greater amount of raw data is produced. Depending on the technique used, electron microscopy with a resolution of $4 \times 4 \times 30$ nm (required to capture the smaller details of a neural network), giving a 100 $\mu$m cube (which can capture roughly 50 neuron somas), requires about 2 terabytes of data (Morgan and Lichtman, 2017). To give a concrete example, Morgan, Berger, et al., 2016 used scanning electron microscopy to generate a dataset of around 100 terabytes of the mouse visual thalamus. This large amount of raw data requires the use of rigorous data handling to keep the raw data organized, accessible and sharable.

Datasets in neuroscience are often bigger than what one research group is able to fully analyze and explore (Choudhury et al., 2014). Sharing data greatly benefits several disciplines of science, such as molecular biology and astronomy. However, data sharing is not yet common in neuroscience (Ascoli et al., 2017). There is a growing recognition of the need to share data (Kennedy, 2006; Teeters, Harris, et al., 2008; Ascoli, 2015). Data sharing enables more information to be extracted from each experiment, helps promote reproducibility and replicability, reduces the cost of science as fewer experiments have to be unnecessarily repeated, and enables novel discoveries through new collaborations (Gardner, Toga, et al., 2003; Poline et al., 2012; Ascoli, 2015).

Although there are several benefits to data sharing, several challenges exist that prevent a full-scale adoption. Some of these are privacy concerns in regards to human trials, costs of hosting the data, heterogeneous data, lack of resources to clean up and organize the data, and concerns about how credits for data are given (Ascoli, 2006; Ferguson et al., 2014; Teeters, Harris, et al., 2008).
Another challenge is that of missing data standards. Although missing a standardized data format does not directly prevent the sharing of data, it impedes the process and makes sharing more complicated than necessary. A wide variety of data formats are in use in neuroscience, and it has been typical for each research group to use their own data formats, and for the data to be distributed in these formats (Gardner, Knuth, et al., 2001; Herz et al., 2008; Teeters, Harris, et al., 2008). Recent efforts have been made to create standardized data formats for neuroscience, two examples being Neo (Garcia, Guarino, et al., 2014) and NWB (Neurodata Without Borders, Teeters, Godfrey, et al., 2015).

There are several challenges when developing a standardized format for neuroscience data. One of the biggest challenges is the diversity of the experimental methods required in order to build an understanding of the brain. Neuroscience includes a wide range of different fields, from physics to psychiatry. Within each of these fields, there is a wide variety of different experimental techniques used, each with different requirements of the data standard. These experimental methods range from electron microscopy to whole brain EEG recordings in spatial scales, while the temporal scales range from microseconds when examining ion channels to studies of development and aging over the lifespan of humans (~100 years). This variety makes creating a standard a difficult task. The data format has to be able to store the variety of data that the wide range of experimental techniques produce.

Another challenge is that metadata often is required to be able to make sense of the experimental data, for example, the physical units, sampling interval or the protocols for the stimulus applied in an electrophysiology experiment. There is also additional metadata that can be included, such as which region of the brain the recording is from.

In Paper IV we developed a specification, called Exdir, for organizing data in a hierarchy by using file-system directories to represent the hierarchy. This work does not try to directly solve the problem of missing data standards in neuroscience. However, Paper IV provides a specification for data storage that we hope is flexible enough to handle the above-mentioned requirements and can be used once a data standard is agreed upon. Additionally, several research groups are already storing data in a directory hierarchy, but no common specification exists. The work in Paper IV thereby helps to enable data sharing and reproducible research by providing a common specification for how to store data in a directory hierarchy.
The use, and need, for computational modeling and advanced data analysis in neuroscience is ever increasing. However, the computational aspect of neuroscience is not reflected in much of the current neuroscience education (Goldman and Fee, 2017). Students do not learn all the tools required in order to perform research, nor all the tools in demand by industry. The solution is to make data analysis and computational modeling an integral part of the education, ideally starting at an early stage. Neuroscientists do not only require knowledge of biology but also of mathematics, as well as computational skills, which should be reflected in the neuroscience education (Goldman and Fee, 2017).

Rejuvenating the neuroscience curriculum is a complicated and time-consuming task. Luckily there are several intermediate steps that can be taken to remedy the situation. One such step is to create simulators that are tailored towards education. Such simulators make computational modeling accessible to students who lack programming and computational experience. We created one such simulator, called Neuronify, for simulating networks of neurons in Paper V. Although many full-scale simulators such as NEURON and NEST exist, few of these enable students without any programming experience to explore various simulations. Educational simulators enable students to explore computational models much earlier in their education.

Educational simulators are able to give valuable insight into both the biological and computational aspects of neuroscience. Such simulators enable students to get hands-on experience with computational models and see how the models work and what they can be used for. Additionally, being able to explore how neural networks respond to changes in both parameters and network structure in “real-time” enables students to gain a greater intuition for how different biological networks behave. This is unlike traditional textbooks, which are limited to
static illustrations. Educational simulators can also be used to help bridge the gap between modelers and experimentalists, two groups that benefit from closely cooperating, but often lack a common language. Although creating educational simulators does not lessen the need to rejuvenate the neuroscience education, it provides useful tools that fill an important role.
Summary of papers

Paper I

Uncertainpy: A Python Toolbox for Uncertainty Quantification and Sensitivity Analysis in Computational Neuroscience

In this paper, we presented Uncertainpy, an open-source Python toolbox, tailored to perform uncertainty quantification and sensitivity analysis of neuroscience models. Uncertainpy is tailored for neuroscience applications by its built-in capability for calculating the uncertainty of characteristic features in the model output. The toolbox does not only perform a point-to-point comparison of the “raw” model output but can also calculate the uncertainty and sensitivity of notable model response features such as spike timing and action potential width. Uncertainpy makes it easy to perform an uncertainty analysis and enables efficient uncertainty calculations through the use of polynomial chaos expansions. A more traditional quasi-Monte Carlo method is also implemented. Uncertainpy has several common neuroscience models and features built-in, and support for easily including custom models and new features.

This paper gave a thorough explanation of the most important features of Uncertainpy available to the users. We demonstrated the broad applicability of Uncertainpy by performing an uncertainty quantification and sensitivity analysis of three case studies relevant for neuroscience: the original Hodgkin-Huxley point-neuron model, a NEURON model for a multi-compartment model of a thalamic interneuron, and a NEST model for a sparsely connected recurrent network. Additionally, we used Uncertainpy to perform the uncertainty analysis in Paper
II and III, which strengthened previous findings and gave useful insight into the examined models.

**Paper II**


The first use of Uncertainpy was on a computational model for bursting endocrine pituitary cells created by Tabak et al., 2011. They created a computational model for endocrine pituitary cells in rats using XPP, a software for solving differential equations. Tabak et al., 2011 discovered that big conductance $K^+$ (BK) ion channels promote bursting activity in endocrine pituitary cells in rats.

In Paper II we replicated the computational results in the original publication by Tabak et al., 2011 and presented a reimplementation of their model using the Python interface for the NEURON simulator. We also expanded upon the description of the analysis performed in the original paper, as the model analysis was only briefly explained and several details important for replication were missing.

In addition to replicating the results in the original publication, we performed an uncertainty quantification and sensitivity analysis of the model using Uncertainpy. This uncertainty analysis further strengthened the conclusion in the original publication. Additionally, it provided novel insight into the model and we discovered that rectifying $K^+$ channels are more important for the bursting activity than the BK channels, though BK channels still are important. The model reimplementation and results from this paper were used as a basis for the computational work performed in Paper III.

**Paper III**

**BK channels have opposite effects on sodium versus calcium mediated action potentials in endocrine pituitary cells**

In this paper, we created two versions of a computational model for endocrine pituitary cells in medaka, a species of Japanese rice fish. Unlike pituitary cells in rats, where action potentials are mediated by $Ca^{2+}$ currents, action potentials in pituitary cells in medaka are mediated by $Na^+$ currents. To our knowledge, this medaka model is the first model of pituitary cells that fire $Na^+$ mediated action potentials.
Comparing this model to the reimplementation and results from Paper II we found that the BK channels have the opposite effect on the action potential shape in the two models. The BK channels make the action potentials generated in the medaka model narrower but they make the action potentials generated in the rat model broader. The BK channels amplify slow Ca$^{2+}$ generated action potentials, but counteract fast Na$^+$ generated action potentials. We performed an uncertainty analysis of the models using Uncertainpy and found that the natural tendency for action potential firing, regular spiking, and bursting depend on a complex interplay of several mechanisms. However, in all models the bursts were facilitated by the Ca$^{2+}$ current and counteracted by the delayed rectifying K$^+$ current.

Paper IV

Experimental Directory Structure (Exdir): An Alternative to HDF5 Without Introducing a New File Format

This paper introduced Exdir, a specification for organizing data in a hierarchy using file-system directories. Exdir uses the binary NumPy format to store datasets, while attributes and metadata are stored in human-readable YAML files. The hierarchy is represented using the directory structure of the operating system. Raw data is stored directly in subdirectories. Exdir is not a file format in itself, but rather a specification for organizing files in a directory structure that uses already existing file formats.

Exdir uses the same data abstractions as HDF5, a popular file format commonly used in science, which stores data in a hierarchy within a large binary file. However, Exdir has several advantages over HDF5. The data structure is human-readable, enables easy use of version control software, reduces the risk of data corruption, and enables easy access to the raw data from outside the Exdir structure. We have developed a reference implementation in Python, which generally follows the same API (Application Programming Interface) as h5py, a HDF5 API for Python. This Exdir API means that in most cases h5py can easily be replaced by Exdir by changing only one line of code. Support for Exdir is implemented in Uncertainpy.

The solution of storing data in a directory hierarchy is already in use, but no common specification exists, which limits the development of common tools and the ability for data sharing. Exdir thereby helps enable data sharing, rigorous data storage, and reproducible research.
Paper V

Neuronify: An Educational Simulator for Neural Circuits

To help students get a better understanding of how computational models of neural networks behave and function, we have created Neuronify, which was introduced in this paper. Neuronify is an educational simulator for neural networks. Neuronify enables students to easily create their own neural networks by dragging and dropping network components, such as neurons, sensors, and network activators, on to the canvas and then connecting them together. The goal was to make it easy to create a neuronal network and simulate its dynamics in real time. Neuronify enables students to play around with and observe the effect of changing the parameters of the various components in real time, thereby gaining a greater intuition for how neural networks behave.

Neuronify implements the integrate-and-fire model, one of the simpler and more common computational neuron models, and the adaptive integrate-and-fire model. There are several network activators available, such as current sources, spike generators, and touch and camera input. The sensors available are voltmeters, spike detectors, firing rate plots, and loudspeakers. Neuronify comes with several example networks and is available for Android and iOS, as well as Linux, Mac, and Windows.
Discussion and future prospects

With the papers published as a part of the work in this thesis, we have developed software tools that neuroscience lacked. The main work of this thesis has been to develop the Uncertainpy toolbox (Paper I). We used Uncertainpy to gain further insight into the computational models in Paper II and III. Uncertainpy has so far been well received. At the time of writing Uncertainpy has been downloaded $\sim 5,300$ times and the Uncertainpy repository has been starred 39 times on Github.

In addition to Uncertainpy, we have developed Exdir, a specification for organizing data in a hierarchy using directories (Paper IV), which hopefully helps to improve the ease of data sharing. Exdir has been downloaded $\sim 3,300$ times. Finally, we have helped improve the computational aspect of education in neuroscience by developing Neuronify (Paper V), an educational application for creating and exploring neural networks. Neuronify is in active use in several neuroscience courses and has been downloaded $\sim 63,000$ times. In addition to the work presented in this thesis we have during the course of this Ph.D. performed data analysis of perineural nets in Lensjø et al., 2017 and of muscle cells in Winje et al., 2018.

In this chapter, we discuss some of the work in a broader context, with a focus on the challenges we encountered when performing uncertainty analysis of computational neuroscience models. We also mention the improvements made to the software since the time of publication, before we finish with some future prospects.
8.1 Challenges of uncertainty analysis in neuroscience

Performing an uncertainty quantification and sensitivity analysis of computational neuroscience models was more complicated than we initially had hoped. During the work in this thesis, we have experienced some challenges common to uncertainty analysis in neuroscience. We go into detail below, which hopefully enable others to avoid the pitfalls we encountered.

8.1.1 Non-trivial interpretation of the sensitivity analysis

When developing Uncertainpy we had hoped it would be easy to draw conclusions from the uncertainty quantification and sensitivity analysis. As it turns out, drawing conclusions from the uncertainty quantification is quite straightforward. The sensitivity analysis, however, has shown itself to be more difficult to interpret.

Models in neuroscience often have complex dynamics and it may be challenging to interpret the sensitivity. We encountered an example of this problem in Paper II. The burstiness of the model by Tabak et al., 2011 was defined as the fraction of events (spikes) that had a duration longer than 60 ms. However, the sensitivity of the average duration of events of the model was different from the sensitivity of the burstiness of the model. The most important parameter for the burstiness was the BK conductance (big conductance K\textsuperscript{+}), while the average duration was most sensitive to the small K\textsuperscript{+} (SK) conductance. Intuitively we would expect these to be similar, as both are related to the duration of events. A further parameter exploration revealed that although the BK conductance was important for achieving bursts in the first place (it made events last for at least 60 ms), the SK conductance had a much greater effect on the total duration of the event. As long as there already was a burst, the SK conductance could extend the event duration to several thousand milliseconds. This finding is a new insight into the model dynamics, whose cause was not immediately apparent from the sensitivity analysis results.

Another complication arises from the fact that the sensitivity does not give any information on how the model output changes with changes in a given parameter. We do not know in what “direction” the model is sensitive. For example, we do not get information on whether increasing a parameter increases or decreases the event duration.

As briefly mentioned in section 3.4.1, it can be difficult to create unambiguous feature definitions that work in all scenarios. As such, it is important to make sure that the features are fit for examining the phenomena of interest. One example of this problem was encountered during the work on Paper II. Defining the action potential width as the full width at half maximum can give different results of the width of the action potential between different model evaluations.
As illustrated in Figure 8.1, depending on whether an action potential is “tall” (blue) or “short” (red), the full width at half maximum measures two different properties of an action potential. Figure 8.1A measures the width of the short spike, while Figure 8.1B measures roughly the length of a burst of the action potential (as defined in Paper II). Which of these two would be the “correct” width to measure depends upon the phenomena examined. “Randomly” measuring either the width of the short spike or the length of the burst introduces a huge variance in the feature that does not reflect the variance in the biological system. It would be better to have a more robust feature. This example also illustrates that to be able to accurately interpret the sensitivity of a feature, it is important to know to a high degree how the examined feature is defined.

As you do not see the results from the feature evaluations, unless you specifically examine them, it can be tricky to discover if there are problems during the feature calculation. As it turns out, it is useful to look at the model and feature evaluations, both to get an understanding of the model dynamics and to ensure the model and features behave as intended. For convenience, support for plotting all evaluations is available in Uncertainpy.
Figure 8.2: The total-order Sobol indices for the uncertain parameters of the reimplemented model by Tabak et al., 2011. The parameters are uncertain with the percentage wise value around the original conductance values used in Tabak et al., 2011. The exception is $G_{BK}$, whose value was set to $G_{BK} = 0.67$ nS, in order to span more of the parameter region where variance in $G_{BK}$ occurs.

### 8.1.2 Using uncertainty analysis to explore models

When using an uncertainty analysis to explore model dynamics by choosing “arbitrary” parameter distributions, it is important to remember that the conclusions can be sensitive to the chosen parameter distributions. This dependence on the parameter distributions is illustrated in Figure 8.2. Here we examine how the total-order Sobol indices for the uncertain parameters of the reimplemented model by Tabak et al., 2011 change with a percentage-wise broadening of the parameter distributions. As seen from Figure 8.2, we get quite different sensitivities just by changing the percentage wise range of the distributions of the uncertain parameters.

Setting parameter distributions as percentage-wise distributions around a fixed value should generally be done with caution. For example, setting a percentage-
wise distribution when you have both an uncertain ion channel conductance and an uncertain ion channel reversal potential can be problematic. How should the percentage-wise distribution be created if the reversal potential is 0 mV? Additionally, by simply scaling the voltages of your model, you can change the parameter ranges. Another complication is that even if we do not know the parameter distributions, different types of parameters have different likely distributions in nature. For example, ion channel conductances tend to have a broader range of parameter distributions than reversal potentials. As such, using the same percentage wise values for the parameter distributions gives a wrong representation of the actual biological variability.

8.1.3 Missing feature evaluations

To be able to accurately calculate the uncertainty and sensitivity we require results for all (or most) of the model evaluations. The robustness towards missing evaluations depends on the method used for the uncertainty analysis. A problem that commonly occurs when calculating features of neuroscience models is that the features are not necessarily defined for all model evaluations. It is common to examine the spiking dynamics of neurons and thereby have features that define different aspects of action potentials. However, it is still a valid neuron evaluation if the model does not spike at all. Some features, such as the spike rate, are still defined (which in this case will be zero). The problem, however, is how features that are dependent on action potentials are defined. For example, what is the action potential depth when there are no action potentials? As such, valid model evaluations can often result in invalid feature evaluations. A solution to this problem is to redefine the problematic features so that they are defined even when there are no action potentials. How to redefine the new features depends on the problem to be examined.

We encountered this problem in both Paper II and III. In Paper II, a sufficiently large percentage of the parameter combinations resulted in action potentials (the lowest percentage was $\sim 91.5\%$). However, in the worst case in Paper III only around 48% of the parameter combinations resulted in action potentials. Due to the missing model evaluations, we chose to use a different set of features in Paper III than what we examined in Paper II. The new features were binary and were well defined in model evaluations without action potentials.

8.2 Data storage

As mentioned in chapter 5, the work in Paper IV does not try to solve the problem of missing data standards in neuroscience, however, it is a flexible specification that can be used once a standard has been agreed upon. The solution of storing data in a file-system hierarchy is already in use by other research groups, but no
common specification existed. Exdir provides such a specification, and thereby help enable data sharing and reproducible research.

Since Exdir uses a human-readable data format, folders and NumPy files (which are simple binary files), it is easy to get access to the content of an Exdir folder without using an implementation of Exdir. Another advantage of Exdir is the use of established standards. This makes the Exdir specification easier to implement for new languages, as most likely existing libraries can be used. It also makes Exdir more stable, as the standards used are well tested, reducing the number of possible bugs. However, it also means that Exdir is dependent on these projects and we might need to update Exdir when changes occur in its dependencies.

The requirements for data standards regularly change with the introduction of new experimental techniques. The flexibility of the Exdir specification hopefully makes Exdir able to accommodate future experimental techniques. The adoption of new specifications takes time, and new edge cases are expected to be found where Exdir needs to be improved. We hope that Exdir will be adopted by the community and that the community can help steer the development of Exdir in the most beneficial direction.

8.3 Education

Neuronify has been well received by the neuroscience community and is in active use in several neuroscience courses. To our delight, Neuronify has also been successfully used in research (Tewari et al., 2018). Although Neuronify has been a success and improves upon the problem of a lack of computational education, it does not solve the problem.

The lack of computational education is not limited to neuroscience but is also a problem in biology in general, where the computational aspect plays an ever more important role (Council, 2005; Committee on Undergraduate Biology Education to Prepare Research Scientists for the 21st Century et al., 2003; American Association for the Advancement of Science, 2009). Markowetz, 2017 even stated that: “All biology is computational biology”. To help solve the general lack of computational education in biology I have co-authored a new textbook titled “Introduction to analysis and modeling in biology with Python”. This textbook is used as curriculum in the course “BIOS1100 – Introduction to computational models for the biosciences”\textsuperscript{1} at the University of Oslo. This course has been completed and the textbook used, both in the fall of 2017 and 2018. Both the course and the textbook have received positive feedback.

There was a lack of suitable textbooks for such a course. Existing textbooks are generally either written for students that already have a background in biology

\textsuperscript{1}uio.no/studier/emner/matnat/ibv/BIOS1100
or programming. Our goal was to teach first-year biology students programming and a textbook at an introductory level in both topics was therefore desirable. The textbook I co-authored is project-based, and the students are introduced to various biological problems and their context. The students then use programming to solve these biological problems. The philosophy behind this textbook is just-in-time teaching, and programming concepts are only introduced when needed to solve a given biological problem. The focus is on the biological problems and the programming happens in the context of biology, which efficiently illustrates why programming is useful and what programming can be used for in biology.

8.4 Improvements to software since publication

The software developed in Paper I (Uncertainpy), IV (Exdir), and V (Neuronify) have all been improved since their time of publication.

Uncertainpy has received several improvements. The spike detection has been enhanced, and several options for how the spikes are detected have been added. After user feedback, an option to disable multiprocessing has been added. The documentation has been updated, the test suite has been expanded, and bug fixes have been made. A more complete summary of improvements can be found in appendix A.

Since publication, Exdir has received support for Python 2.7, and been added to PyPi\(^2\). The documentation has been expanded, the performance greatly increased, and several bug fixes have been made.

Neuronify has since publication received an upgraded desktop version with a visual overhaul. Some parameter settings have been improved and now show a preview of how the neuron responds to the changes in the parameters. Neuronify now supports uploading and downloading community created neural networks. Several bug fixes have also been made. Multicompartmental models and plastic synapses, and thereby networks that can learn, are in development.

8.5 Future prospects

As computational models in neuroscience become more complicated, and we get more and better parameter measurements, the need for uncertainty quantification and sensitivity analysis in neuroscience increases. There are several directions in which further work on uncertainty analysis can be taken.

\(^2\text{pypi.org}\)
8.5.1 Improve Uncertainpy

The most obvious future prospect is to continue to maintain and develop Uncertainpy. One area that can be enhanced is the interpolation performed. Uncertainpy interpolates the model or feature evaluations if they are irregular, in order to get the evaluations into the form required by the uncertainty analysis. At the moment there is only support for interpolation of one-dimensional model and feature evaluations (for example voltage traces), but higher dimensional interpolation could be implemented. Another likely enhancement is to improve the current screening method. The screening method available in Uncertainpy is not able to take interactions between parameters into account. As such, more advanced screening methods, for example, the Morris method of elementary effects, could be implemented. A third possible improvement is to add support for more features and models.

Various quality of life improvements could also be made. Currently, there is no support for restarting an uncertainty analysis after the model evaluations have been calculated. It would be practical to be able to load a set of previously calculated model evaluations, and then perform the uncertainty analysis from the loaded model evaluations. Additionally, the current default plots of Uncertainpy do not look as good as they could. The default plots can be enhanced to look more like the plots shown in Paper I, and the plotting class needs to be refactored.

The hope is that a community grows around Uncertainpy, which can help maintain and develop Uncertainpy. A community would also make it apparent which changes and new features that are the most wanted.

8.5.2 Further uncertainty analysis in neuroscience

Another project of interest would be to perform a methodological study of methods for uncertainty quantification and sensitivity analysis on neuroscience models. Since the methods for uncertainty quantification and sensitivity analysis are model dependent, it would be useful to know how the different methods perform for common neuroscience models. Additionally, we were unable to find any quantitative information on how robust the different methods are for missing model evaluations. It would be useful to examine this robustness.

It would also be interesting to find new projects to cooperate on and use Uncertainpy to perform uncertainty analysis of other models. Using Uncertainpy on new models helps to identify new functionality which can be added to the toolbox, as specific models often require special considerations.

Further uncertainty analysis of neuroscience models is likely to uncover new challenges. After gaining more experience, it would be useful to expand upon the section on “challenges of uncertainty analysis in neuroscience” (section 8.1). The aim would be to write a guide for uncertainty analysis in neuroscience, which outlines common pitfalls, the best methods to use and considerations to make.
Such a guide would help lower the barrier of entry for researchers to perform uncertainty analysis.

Another interesting project to undertake is to test how well Bayesian parameter estimation works on computational neuroscience models. Software for Bayesian parameter estimation exists, for example, the Python package PyMC3 (Salvatier et al., 2016), or PyStan\(^3\), the python interface for Stan (Carpenter et al., 2017). This project could be extended by coupling Bayesian parameter estimations to Uncertainpy or including it in Uncertainpy.

### 8.5.3 Data storage

Future improvements to Exdir would likely entail implementing support for features found in HDF5 that do not yet exist in Exdir, for example, support for soft and hard links. If there is interest, another potential task would be to implement Exdir in other languages, with C++ and Matlab as the most likely candidates. Adding support for parallelization would also be useful. Additionally, getting involved with the current standardization projects such as NWB would help enhance the visibility of Exdir, as well as being a great opportunity to find the areas where Exdir can be improved upon.

### 8.5.4 Education

In regards to Neuronify, the first step is to finish the development of plastic synapses and multi-compartment neuron models. Other neuron models can potentially also be implemented. The tutorial examples can also be improved. At the moment they rely too much on pure text when explaining what occurs, which is not the best pedagogical approach. A greater extension of Neuronify would be to incorporate game-design elements and make the exploration and building of neural networks game-like. Currently, Neuronify only gives the users a sandbox to play around in (including a few examples). Game elements that drive exploration would likely improve the learning outcome, as well as the motivation to use Neuronify. Planning even further ahead, other educational apps could be created for other physical and biological phenomena, not necessarily limited to neuroscience.

\(^3\)pystan.readthedocs.io


Committee on Undergraduate Biology Education to Prepare Research Scientists for the 21st Century, Board on Life Sciences, Division on Earth and Life Studies, National Research Council (2003) BIO2010: Transforming Undergradu-


Kidokoro Y. (1975) Spontaneous calcium action potentials in a clonal pituitary cell line and their relationship to prolactin secretion. Nature 258. DOI: 10.1038/253600a0.


Marelli S., Sudret B. (2013) UQLab: a framework for uncertainty quantification in MATLAB. In:


Author contributions

Paper I
Geir Halnes, Gaute T. Einevoll, Hans-Petter Langtangen, and I conceived of and designed the project. I designed, wrote, tested and documented the software, performed analysis of the example studies and wrote the first draft of the paper. Geir Halnes, Gaute T. Einevoll, and I revised the paper and wrote the final version.

Paper II
Geir Halnes and I conceived of and designed the project, and implemented the model. I performed the analysis, uncertainty quantification and exploration of the model and wrote the first draft of the paper. Geir Halnes, Trude M. Haug, Gaute T. Einevoll, Finn Arne Weltzien, Kjetil Hodne, and I revised the paper. Geir Halnes, Gaute T. Einevoll, and I wrote the final version.

Paper III
Geir Halnes, Trude M. Haug, Gaute T. Einevoll, Finn Arne Weltzien, and Kjetil Hodne conceived of and designed the project. All experiments were performed by Kjetil Hodne. Geir Halnes created and implemented the model. I performed the analysis and uncertainty quantification of the model. Geir Halnes wrote the first draft of the paper. Geir Halnes, Trude M. Haug, Gaute T. Einevoll, Finn Arne Weltzien, Kjetil Hodne, and I revised the paper and wrote the final version.

Paper IV
Svenn-Arne Dragly, Milad Hobbi Mobarhan, and Mikkel Lepperød conceived of and designed the project. Svenn-Arne Dragly, Milad Hobbi Mobarhan, Mikkel Lepperød, and I wrote the software, documentation, and the first draft of the paper. Svenn-Arne Dragly, Milad Hobbi Mobarhan, Mikkel Lepperød, Marianne
Fyhn, Torkel Hafting, Anders Malthe-Sørenssen, and I revised the paper and wrote the final version.

**Paper V**

Svenn-Arne Dragly conceived of the project. I wrote the software and first draft of the paper together with Svenn-Arne Dragly, Milad Hobbi Mobarhan, and Andreas Våvang Solbrå. Svenn-Arne Dragly, Milad Hobbi Mobarhan, Andreas Våvang Solbrå, Anders Hafreager, Anders Malthe-Sørenssen, Marianne Fyhn, Torkel Hafting, Gaute T. Einevoll, and I revised the paper and wrote the final version.
Part II

Papers
Paper I

Uncertainpy: A Python Toolbox for Uncertainty Quantification and Sensitivity Analysis in Computational Neuroscience
Uncertaintypy: A Python Toolbox for Uncertainty Quantification and Sensitivity Analysis in Computational Neuroscience

Simen Tennøe, Geir Halnes and Gaute T. Einevoll

1 Centre for Integrative Neuroplasticity, University of Oslo, Oslo, Norway, 2 Department of Informatics, University of Oslo, Oslo, Norway, 3 Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway, 4 Department of Physics, University of Oslo, Oslo, Norway

Computational models in neuroscience typically contain many parameters that are poorly constrained by experimental data. Uncertainty quantification and sensitivity analysis provide rigorous procedures to quantify how the model output depends on this parameter uncertainty. Unfortunately, the application of such methods is not yet standard within the field of neuroscience. Here we present Uncertaintypy, an open-source Python toolbox, tailored to perform uncertainty quantification and sensitivity analysis of neuroscience models. Uncertaintypy aims to make it quick and easy to get started with uncertainty analysis, without any need for detailed prior knowledge. The toolbox allows uncertainty quantification and sensitivity analysis to be performed on already existing models without needing to modify the model equations or model implementation. Uncertaintypy bases its analysis on polynomial chaos expansions, which are more efficient than the more standard Monte-Carlo based approaches. Uncertaintypy is tailored for neuroscience applications by its built-in capability for calculating characteristic features in the model output. The toolbox does not merely perform a point-to-point comparison of the “raw” model output (e.g., membrane voltage traces), but can also calculate the uncertainty and sensitivity of salient model response features such as spike timing, action potential width, average interspike interval, and other features relevant for various neural and neural network models. Uncertaintypy comes with several common models and features built in, and including custom models and new features is easy. The aim of the current paper is to present Uncertaintypy to the neuroscience community in a user-oriented manner. To demonstrate its broad applicability, we perform an uncertainty quantification and sensitivity analysis of three case studies relevant for neuroscience: the original Hodgkin-Huxley point-neuron model for action potential generation, a multi-compartmental model of a thalamic interneuron implemented in the NEURON simulator, and a sparsely connected recurrent network model implemented in the NEST simulator.

Keywords: uncertainty quantification, sensitivity analysis, features, polynomial chaos expansions, quasi-Monte Carlo method, software, computational modeling, Python
SIGNIFICANCE STATEMENT

A major challenge in computational neuroscience is to specify the often large number of parameters that define neuron and neural network models. Many of these parameters have an inherent variability, and some are even actively regulated and change with time. It is important to know how the uncertainty in the model parameters affects the model predictions. To address this need we here present Uncertainpy, an open-source Python toolbox tailored to perform uncertainty quantification and sensitivity analysis of neuroscience models.

1. INTRODUCTION

Computational modeling has become a useful tool for examining various phenomena in biology in general (Brodland, 2015) and neuroscience in particular (Koch and Segev, 1998; Dayan and Abbott, 2001; Sterratt et al., 2011). The field of neuroscience has seen the development of ever more complex models, and models now exist for large networks of biophysically detailed neurons (Izhikevich and Edelman, 2008; Merolla et al., 2014; Markram et al., 2015).

Computational models typically contain a number of parameters that for various reasons are uncertain. A typical example of an uncertain parameter in a neural model can be the conductance \( g_x \) of a fully open ion channel of a specific type \( x \). Despite the parameter uncertainty, it is common practice to construct models that are deterministic in the sense that single numerical values are assigned to each parameter.

Uncertainty quantification is a means to quantify the uncertainty in the model output that arises from uncertainty in the model parameters. Instead of assuming fixed model parameters as in a deterministic model (as illustrated in Figure 1A), one assigns a distribution of possible values to each model parameter. The uncertainty in the model parameters is then propagated through the model and gives rise to a distribution in the model output (as illustrated in Figure 1B).

Sensitivity analysis is tightly linked to uncertainty quantification and is the process of quantifying how much of the output uncertainty each parameter is responsible for (Saltelli, 2002b). A small change in a parameter the model is highly sensitive to, leads to a comparatively large change in the model output. Similarly, variations in a parameter the model has a low sensitivity to, result in comparatively small variations in the model output.

Given that most neuroscience models contain a variety of uncertain parameters, the need for systematic approaches to quantify what confidence we can have in the model output is pressing. The importance of uncertainty quantification and sensitivity analysis of computational models is well known in a wide variety of fields (Leamer, 1985; Beck, 1987; Turanyi and Turányi, 1990; Oberkampf et al., 2002; Sharp and Wood-Schultz, 2003; Marino et al., 2008; Najm, 2009; Rossa et al., 2011; Wang and Sheen, 2015; Yildirim and Karniadakis, 2015). Due to the prevalence of inherent variability in the parameters of biological systems, uncertainty quantification and sensitivity analysis are at least as important in neuroscience. Toward this end we have created Uncertainpy1, a Python toolbox for uncertainty quantification and sensitivity analysis, tailored toward neuroscience models.

The uncertainty in a model parameter may have many origins. It may be due to (i) measurement uncertainty or (ii) lack of experimental techniques that enable the parameter to be measured. The uncertainty can also be due to an inherent biological variability, meaning the value of a parameter can vary (iii) between neurons of the same species (Edelman and Gally, 2001; Hay et al., 2013), or (iv) dynamically within a single neuron due to plasticity or homeostatic mechanisms (Marder and Goaillard, 2006). Additionally, some models include parameters that are (v) phenomenological abstractions, and therefore do not represent directly measurable physical entities.

They might, for example, represent the combined effect of several physical processes. The above uncertainties can generally be divided into two main classes: aleatory uncertainties and epistemic uncertainties. Epistemic uncertainty reflects a lack of knowledge, and can in principle be reduced to zero by acquiring additional information. Aleatory uncertainty, on the other hand, is uncertainty due to inherent variability of the parameters. The importance of distinguishing between aleatory and epistemic uncertainties has evoked some debate (Ferson and Ginzburg, 1996; Hora, 1996; Oberkampf et al., 2002; Ferson et al., 2004; Kuireghian and Ditlevsen, 2009; Mullins et al., 2016), but the distinction is important for how to interpret the results of an uncertainty quantification. Parameters with epistemic uncertainties produce an uncertainty as to whether or not we have acquired the “correct” result, while parameters with aleatory uncertainties reflect the true variability of the system.

A common way to avoid addressing the uncertainty in measured parameters is to use the means of several experimental measurements. This can be problematic since the underlying distribution of a set of parameters can be poorly characterized by the mean and variance of each parameter (Golowasch et al., 2002). Additionally, during model construction, a subset of the uncertain parameters are commonly treated as free parameters. This means the parameters are tuned by the modeler to values that make the model output match a set of experimental constraints. An example is fitting an ion-channel conductance \( g_x \) so the membrane potential of a neuron model reproduces an experimentally measured voltage trace. Whatever method used, the tuning procedure does not guarantee a unique solution for the correct parameter set, since it is often the case that a wide range of different parameter combinations give rise to similar model behavior (Bhalla and Bower, 1993; Beer et al., 1999; Goldman et al., 2001; Golowasch et al., 2002; Prinz et al., 2004; Tobin, 2006; Halnes et al., 2007; Schulz et al., 2007; Taylor et al., 2009; Marder and Taylor, 2011).

When we have uncertain parameters, but nevertheless choose to use a single set of fixed parameter values, it is a priori difficult to assess to what degree we can trust the model result. Performing an uncertainty quantification enables us to properly take the effects of the uncertain parameters into account, and it quantifies what confidence we can have in the model output.

---

1https://github.com/simetenn/uncertainpy

---
An uncertainty quantification enables us to model the naturally occurring variation in the parameters of biological systems. It also increases our understanding of the model by quantifying how the uncertain parameters influence the model output. Additionally, performing an uncertainty quantification makes comparing two model outputs, as well as a model output and an experimental result, more informative. By knowing the distribution of the model output we can better quantify how similar (or different) the two model outputs, or model output and experimental result, are.

Performing a sensitivity analysis provides insight into how each parameter affects different aspects of the model, and it gives us a better understanding of the relationship between the parameters (and by extent the biological mechanisms) and the output of the model (Marino et al., 2008). A model-based sensitivity analysis can thus help to guide the experimental focus (Zi, 2011). Knowing how sensitive the model is to changes in each parameter, enables us to take special care to obtain accurate measures of parameters with a high sensitivity, while more crude measures are acceptable for parameters with a low sensitivity.

Sensitivity analysis is also useful in model reduction contexts and when performing parameter estimations (Degenring et al., 2004; Zi, 2011; Snowden et al., 2017). A parameter that the model has a low sensitivity to, can essentially be set to any fixed value (within the explored distribution), without greatly affecting the variance of the model output. In some cases, such an analysis can even justify leaving out entire mechanisms from a model. For example, if a single neuron model is insensitive to the conductance of a given ion channel $g_x$, this ion channel could possibly (but not necessarily) be removed from the model with only small changes to the model behavior.

Unfortunately, a generally accepted practice of uncertainty quantification and sensitivity analysis does not currently exist within the field of neuroscience, and models are commonly presented without including any form of uncertainty quantification or sensitivity analysis. When an effort is made in
that direction, it is still common to use rather simple, so-called One-At-A-Time methods, where one examines how much the model output changes when varying one parameter at a time (see e.g., De Schutter and Bower, 1994; Blot and Barbour, 2014; Kuchibhotla et al., 2017). Such approaches do not account for potential dependencies between the parameters, and thereby miss correlations within the often multi-dimensional parameter space (Borgonovo and Plischke, 2016). Other methods that have been applied are local methods, which are multi-dimensional, but confined to exploring small perturbations surrounding a single point in the parameter space (see e.g., Gutenkunst et al., 2007; Blomquist et al., 2009; O’Donnell et al., 2017). Such methods can thus not explore the effects of arbitrarily broad uncertainty distributions for the parameters.

Methods for uncertainty quantification and sensitivity analysis that take the entire parameter space into account are often called global methods (Borgonovo and Plischke, 2016; Babtie and Stumpf, 2017). Global methods are only occasionally used within the field of neuroscience (see e.g., Halnes et al., 2009; Torres Valderrama et al., 2015). The most well-known of the global methods is the (quasi-)Monte Carlo method, which relies on randomly sampling the parameter distributions, followed by calculating statistics from the resulting model outputs. The problem with the (quasi-)Monte Carlo method is that it is computationally very demanding, particularly for computationally expensive models. A means to obtain similar results in a much more efficient way, is provided by the recent mathematical framework of polynomial chaos expansions (Xiu and Hesthaven, 2005). Polynomial chaos expansions are used to approximate the model with a polynomial (as a surrogate model), on which the uncertainty and sensitivity analysis can be performed much more efficiently.

To lower the threshold for neuroscientists to perform uncertainty quantification and sensitivity analysis, we have created Uncertainpy, an open-source Python toolbox for efficient uncertainty quantification and sensitivity analysis. Uncertainpy aims to make it quick and easy to get started with uncertainty quantification and sensitivity analysis. Just a few lines of Python code are needed, without any need for detailed prior knowledge of uncertainty or sensitivity analysis. Uncertainpy implements both the quasi-Monte Carlo method and polynomial chaos expansions. The toolbox is model-independent and treats the model as a “black box,” meaning that uncertainty quantification can be performed on already existing models without needing to modify the model equations or model implementation.

Whereas its statistical methods are generally applicable, Uncertainpy is tailored for neuroscience applications by having a built-in capability for recognizing characteristic features in the model output. This means Uncertainpy does not merely perform a point-to-point comparison of the “raw” model output (e.g., a voltage trace). When applicable, Uncertainpy also recognizes and calculates the uncertainty in model response features, for example the spike timing and action-potential shape for neural models and firing rates and interspike intervals for neural networks.

To present Uncertainpy, we start this paper with an overview of the theory behind uncertainty quantification and sensitivity analysis in section 2, with a focus on the (quasi-)Monte Carlo method and polynomial chaos expansions. In section 3 we explain how to use Uncertainpy, and give details on how the uncertainty quantification and sensitivity analysis are implemented. In section 4 we illustrate the use of Uncertainpy by showing four different case studies where we perform uncertainty analysis of: (i) a cooling coffee-cup model (Newton’s law of cooling) to illustrate the uncertainty analysis on a conceptually simple model, (ii) the original Hodgkin-Huxley point-neuron model for action potential generation, (iii) a comprehensive multi-compartmental model of a thalamic interneuron, and (iv) a sparsely connected recurrent network model (Brunel network). The final section of section 4 gives a comparison of the performance, that is, numerical efficacy, of the quasi-Monte Carlo method and polynomial chaos expansions using the original Hodgkin-Huxley model as an example. We end with a discussion and some future prospects in section 5.

2. THEORY ON UNCERTAINTY QUANTIFICATION AND SENSITIVITY ANALYSIS

Uncertainty quantification and sensitivity analysis provide rigorous procedures to analyze and characterize the effects of parameter uncertainty on the output of a model. The methods for uncertainty quantification and sensitivity analysis can be divided into global and local methods. Local methods examine how the model output changes with small perturbations around a fixed point in the parameter space. Global methods, on the other hand, take the whole range of parameters into consideration.

The global methods can be divided into intrusive and non-intrusive methods. Intrusive methods require changes to the underlying model equations and are often challenging to implement. Models in neuroscience are often created with the use of advanced simulators such as NEST (Peyser et al., 2017) and NEURON (Hines and Carnevale, 1997). Modifying the underlying equations of models using such simulators is a complicated task best avoided. Non-intrusive methods, on the other hand, consider the model as a black box and can be applied to any model without needing to modify the model equations or model implementation. Global, non-intrusive methods are therefore the methods of choice in Uncertainpy. The uncertainty calculations in Uncertainpy are mainly based on the Python package Chaospy (Feinberg and Langtangen, 2015), which provides global, non-intrusive methods for uncertainty quantification and sensitivity analysis. Additionally, Uncertainpy uses the package SALib (Herman and Usher, 2017) to perform sensitivity analysis with the quasi-Monte Carlo method.

In this section, we go through the theory behind the methods for uncertainty quantification and sensitivity analysis used in Uncertainpy. We start by introducing the notation used in this paper (section 2.1). Next, we introduce the statistical measurements for uncertainty quantification (section 2.2) and sensitivity analysis (section 2.3). Further, we give an introduction to the (quasi-)Monte Carlo method (section 2.4) and polynomial chaos expansions (section 2.5), the two methods used to perform
the uncertainty analysis in Uncertainpy. We next explain how Uncertainpy handles cases with statistically dependent model parameters (section 2.6). Finally, we explain the concept and benefits of performing a feature-based analysis (section 2.7). We note that detailed insight into the theory of uncertainty quantification and sensitivity analysis is not a prerequisite for using Uncertainpy, so the more practically oriented reader may choose to skip this section, and go directly to the user guide in section 3.

2.1. Problem Definition
Consider a model \( U \) that depends on space \( x \) and time \( t \), has \( d \) uncertain input parameters \( Q = [Q_1, Q_2, \ldots, Q_d] \), and gives the output \( Y \):

\[
Y = U(x, t, Q).
\] (1)

The output \( Y \) can have any value within the output space \( \Omega_Y \) and has an unknown probability density function \( \rho_Y \). The goal of an uncertainty quantification is to describe the unknown \( \rho_Y \) through statistical metrics. We are only interested in the input and output of the model, and we ignore all details on the inner workings of the model. The model \( U \) is thus considered a black box and may represent any model, for example a spiking neuron model that returns a voltage trace, or a neural network model that returns a spike train.

We assume the model includes uncertain parameters that can be described by a multivariate probability density function \( \rho_Q \). Examples of parameters that can be uncertain in neuroscience are the conductance of a single ion channel or the synaptic weight between two types of neurons in a neural network. If the uncertain parameters are statistically independent, the multivariate probability density function \( \rho_Q \) can be given as separate univariate probability density functions \( \rho_{Q_i} \), one for each uncertain parameter \( Q_i \). The joint multivariate probability density function for the independent uncertain parameters is then:

\[
\rho_Q = \prod_{i=1}^{d} \rho_{Q_i}.
\] (2)

In cases where the uncertain input parameters are statistically dependent variables, the multivariate probability density function \( \rho_Q \) must be defined directly. It should be noted that with statistically dependent parameters we here mean that there is a dependence between the input parameters. When drawing parameters from the joint probability function, by drawing one parameter we influence the probability of drawing specific values for the other parameters. Thus, we do not refer to dependencies between how the input parameters affect the model output. We assume the probability density functions are known and are not here concerned with how they are determined. They may be the product of a series of measurements, a parameter estimation, or educated guesses.

2.2. Uncertainty Quantification
As mentioned, the goal of an uncertainty quantification is to describe the unknown distribution of the model output \( \rho_Y \) through statistical metrics. The two most common statistical metrics used in this context are the mean \( \mathbb{E} \) (also called the expectation value) and the variance \( \mathbb{V} \). The mean is defined as:

\[
\mathbb{E}[Y] = \int_{\Omega_Y} y \rho_Y(y)dy,
\] (3)

and tells us the expected value of the model output \( Y \). The variance is defined as:

\[
\mathbb{V}[Y] = \int_{\Omega_Y} (y - \mathbb{E}[Y])^2 \rho_Y(y)dy,
\] (4)

and tells us how much the output varies around the mean.

Another useful metric is the (100 · \( x \))-th percentile \( P_x \) of \( Y \), which defines a value below which 100 · \( x \) percent of the model outputs are located. For example, \( 5\% \) of the evaluations of a model will give an output lower than the 5th percentile. The (100 · \( x \))-th percentile is defined as:

\[
x = \int_{-\infty}^{P_x} \rho_Y(y)dy.
\] (5)

We can combine two percentiles to create a prediction interval \( I_x \), which is a range of values within which a 100 · \( x \) percentage of the outputs \( Y \) occur:

\[
I_x = \left[ P_{(x/2)}, P_{(1-x/2)} \right].
\] (6)

The 90% prediction interval gives us the interval within which 90% of the \( Y \) outcomes occur, which also means that 5% of the outcomes are above and 5% are below this interval.

2.3. Sensitivity Analysis
A sensitivity analysis quantifies how much of the uncertainty in the model output each uncertain parameter is responsible for. Several different sensitivity measures exist, for a review of methods for sensitivity analysis see Saltelli et al. (2007), Hamby (1994), and Zi (2011). Uncertainpy uses variance-based sensitivity analysis and computes the commonly considered Sobol sensitivity indices (Sobol, 1990). This sensitivity analysis is global, non-intrusive and allows the effects of interactions between parameters within the model to be studied (Zi, 2011). (Two parameters are said to interact if they have a non-additive effect on the output (Saltelli et al., 2007)).

The Sobol sensitivity indices quantify how much of the variance in the model output each uncertain parameter is responsible for. If a parameter has a low sensitivity index, variations in this parameter result in comparatively small variations in the final model output. Similarly, if a parameter has a high sensitivity index, a change in this parameter leads to a large change in the model output.

There are several types of Sobol indices. The first-order Sobol sensitivity index \( S_i \) measures the direct effect each parameter has on the variance of the model:

\[
S_i = \frac{\mathbb{V}[E[Y|Q_i]]}{\mathbb{V}[Y]}.
\] (7)
Here, \( \mathbb{E}[Y|Q_i] \) denotes the expected value of the output \( Y \) when the parameter \( Q_i \) is fixed. The first-order Sobol sensitivity index tells us the expected reduction in the variance of the model when we fix parameter \( Q_i \). The sum of the first-order Sobol sensitivity indices cannot exceed one, and is only equal to one if no interactions are present (Glen and Isaacs, 2012).

Higher order Sobol indices exist and give the sensitivity due to interactions between a parameter \( Q_i \) and various other parameters. It is customary to only calculate the first and total-order indices (Saltelli et al., 2010). The total Sobol sensitivity index \( S_{T,i} \) includes the sensitivity of both the first-order effects, as well as the sensitivity due to interactions between a given parameter \( Q_i \) and all combinations of the other parameters (Homma and Saltelli, 1996). It is defined as:

\[
S_{T,i} = 1 - \frac{\mathbb{V}[\mathbb{E}[Y|Q_{-i}]]}{\mathbb{V}[Y]},
\]

where \( Q_{-i} \) denotes all uncertain parameters except \( Q_i \). The sum of the total Sobol sensitivity indices is equal to or greater than one, and is only equal to one if there are no interactions between the parameters (Glen and Isaacs, 2012). When the goal is to use sensitivity analysis to fix parameters with low sensitivity, it is recommended to use the total-order Sobol indices.

We might want to compare Sobol indices across different features (introduced in section 2.7). This can be problematic when we have features with a different number of output dimensions. In the case of a zero-dimensional output, the Sobol indices are a single number and for a one-dimensional output we get Sobol indices for each point in time. To better be able to compare the Sobol indices across such features, we also calculate the average of the first-order Sobol indices \( S_{i} \), and total-order Sobol indices \( S_{T,i} \).

### 2.4. (Quasi-)Monte Carlo Method

A typical way to obtain the statistical metrics mentioned above is to use the (quasi-)Monte Carlo method. We give a brief overview of the Monte Carlo and quasi-Monte Carlo method here, for a more comprehensive review see Lemieux (2009).

The general idea behind the standard Monte Carlo method is quite simple. A set of parameters is randomly drawn from the joint multivariate probability density function \( \rho_Q \) of the parameters. The model is then evaluated for the sampled parameter set. This process is repeated thousands of times, and statistical metrics such as the mean and variance are computed from the resulting series of model outputs. The accuracy of the Monte Carlo method, and by extent the number of samples required, is independent of the number of uncertain parameters. Additionally, the Monte Carlo method makes no assumptions about the model. However, a limitation of the Monte Carlo method is that a very high number of model evaluations are required to get reliable statistics. If the model is computationally expensive, the Monte Carlo method may thus require insurmountable computer power.

The quasi-Monte Carlo method improves upon the standard Monte Carlo method by using variance-reduction techniques to reduce the number of model evaluations needed. This method is based on increasing the coverage of the sampled parameter space by distributing the samples more evenly. Fewer samples are then required to obtain a given accuracy. Instead of randomly selecting parameters from \( \rho_Q \), the samples are selected using a low-discrepancy sequence such as the Sobol sequence or Hammersley sequence (Hammersley, 1960; Sobol, 1967). The quasi-Monte Carlo method is faster than the Monte Carlo method, as long as the number of uncertain parameters is sufficiently small, and the model is sufficiently smooth (Lemieux, 2009).

Uncertainty allows the quasi-Monte Carlo method to be used to compute the statistical metrics. When this option is chosen, the metrics are computed as follows. With \( N_s \) model evaluations, which gives the results \( Y = \{Y_1, Y_2, \ldots, Y_{N_s}\} \), the mean is given by

\[
\mathbb{E}[Y] \approx \frac{1}{N_s} \sum_{i=1}^{N_s} Y_i,
\]

and the variance by

\[
\mathbb{V}[Y] \approx \frac{1}{N_s - 1} \sum_{i=1}^{N_s} (Y_i - \mathbb{E}[Y])^2.
\]

Prediction intervals are found by sorting the model evaluations \( Y \) in an ascending order, and then finding the \( (100 \cdot x/2) \)-th and \( (100 \cdot (1 - x/2)) \)-th percentiles. The Sobol indices can be calculated using Saltelli’s method (Saltelli, 2002a; Saltelli et al., 2010). The number of samples required by this method is:

\[
N_s = N(d + 2),
\]

where \( N \) is the number of samples required to get a given accuracy with the quasi-Monte Carlo method. This means that the number of samples required by both the Monte Carlo method and the quasi-Monte Carlo method for sensitivity analysis depends on the number of uncertain parameters. Due to how the samples are selected in Saltelli’s method, when selecting \( N \) samples for the uncertainty quantification (which give \( N_s = N \)), we get \( N_s = N(d + 2)/2 \) samples for the sensitivity analysis. The chosen number of samples \( N \) is effectively halved.

It should be noted that there is no guarantee that each set of sampled parameters will produce a valid model evaluation. For example, the spike width will not be defined for a model that produces no spikes. The (quasi-)Monte Carlo method is robust for such missing model results when performing an uncertainty quantification, as long as the number of valid model evaluations is relatively high. However, for the sensitivity analysis the (quasi-) Monte Carlo method using Saltelli’s approach requires that there are no missing model results. A suggested workaround (Herman and Usher, 2017) is to replace invalid model evaluations with the mean of the evaluations\(^2\). This workaround introduces an error depending on the number of missing evaluations but enables us to still calculate the Sobol indices. This workaround is used in Uncertainty.

\(^2\)https://github.com/SALib/SALib/issues/134
2.5. Polynomial Chaos Expansions

A recent mathematical framework for efficient uncertainty quantification and sensitivity analysis is that of polynomial chaos expansions (Xiu and Hesthaven, 2005). This method calculates the same statistical metrics as the (quasi-)Monte Carlo method but is typically much faster (Xiu and Hesthaven, 2005; Crestaux et al., 2009; Eck et al., 2016). For the Hodgkin-Huxley model, we find that polynomial chaos expansions require one to three orders of magnitude fewer model evaluations than the quasi-Monte Carlo method (see section 4.5). We here give a short review of polynomial chaos expansions, for a comprehensive review see Xiu (2010).

Polynomial chaos expansions are typically much faster than the (quasi-)Monte Carlo method as long as the number of uncertain parameters is relatively low, typically smaller than about 20 (Xiu and Hesthaven, 2005; Crestaux et al., 2009; Eck et al., 2016). This means polynomial chaos expansions require far fewer model evaluations than the (quasi-)Monte Carlo method to obtain the same accuracy. It is often the case that neuroscience models have fewer than about 20 parameters, and even for models with a higher number of uncertain parameters, polynomial chaos expansions can be used for selected subsets of the parameters.

The main limitation of polynomial chaos expansions is that the required number of model evaluations scales worse with an increasing number of uncertain parameters than the (quasi-)Monte Carlo method does. This is the reason why the (quasi-)Monte Carlo method becomes better at around 20 uncertain parameters. Another limitation of the polynomial chaos expansions is that the performance is reduced if the output has a non-smooth behavior with respect to the input parameters (Eck et al., 2016).

The exact gain in efficiency when using polynomial chaos expansions instead of the quasi-Monte Carlo method is problem dependent. However, Crestaux et al. (2009) examined three different benchmark problems with three, twelve, and five uncertain parameters. They found that the error in the polynomial chaos expansions converged as $N^{-6}$, $N^{-2}$, and between $N^{-1}$ and $N^{-3/4}$, respectively. In comparison, the error of the quasi-Monte Carlo method converged as $\sim N^{-3/4}$ for each of the problems. Polynomial chaos expansions thus have a much faster convergence for the first two benchmark problems, while the convergences were essentially the same for the last problem. The last benchmark problem was non-smooth, which led to the slower convergence of the polynomial chaos expansions. Still, even in the worst-case example considered in Crestaux et al. (2009), the convergence of the polynomial chaos expansions was essentially as good as for the quasi-Monte Carlo method.

The general idea behind polynomial chaos expansions is to approximate the model $U$ with a polynomial expansion $\hat{U}$:

$$U \approx \hat{U}(x, t, Q) = \sum_{n=0}^{N_p-1} c_n(x, t)\phi_n(Q),$$

where $\phi_n$ are polynomials, and $c_n$ are expansion coefficients. The number of expansion factors $N_p$ is given by

$$N_p = \left( \frac{d + p}{p} \right),$$

where $p$ is the polynomial order. The polynomials $\phi_n(Q)$ are chosen so they are orthogonal with respect to the probability density function $\rho_Q$ which ensures useful statistical properties.

When creating the polynomial chaos expansion, the first step is to find the orthogonal polynomials $\phi_n$. In Uncertainpy this is done using the so-called three-term recurrence relation (Xiu, 2010) if available, otherwise the discretized Stieltjes method (Stieltjes, 1884) is used. The next step is to estimate the expansion coefficients $c_n$. The non-intrusive methods for doing this can be divided into two classes, point-collocation methods and pseudo-spectral projection methods, both of which are implemented in Uncertainpy.

Point collocation is the default method used in Uncertainpy. This method is based on demanding that the polynomial approximation is equal to the model output evaluated at a set of collocation nodes drawn from the joint probability density function $\rho_Q$. This demand results in a set of linear equations for the polynomial coefficients $c_n$, which can be solved by the use of regression methods. The regression method used in Uncertainpy is Tikhonov regularization (Rifkin and Lippert, 2007). Hosder et al. (2007) recommends using $N_i = 2(N_p + 1)$ collocation nodes.

Pseudo-spectral projection methods are based on least squares minimization in the orthogonal polynomial space and calculate the expansion coefficients $c_n$ through numerical integration. The integration uses a quadrature scheme with weights and nodes, and the model is evaluated at these nodes. The number of samples is determined by the quadrature rule. The quadrature method used in Uncertainpy is Leja quadrature, with Smolyak sparse grids to reduce the number of required nodes (Smolyak, 1963; Narayan and Jakeman, 2014). Pseudo-spectral projection is only used in Uncertainpy when requested by the user.

Of these two methods, point collocation is robust toward invalid model evaluations as long as the number of remaining evaluations is high enough, while spectral projection is not (Eck et al., 2016).

Several of the statistical metrics of interest can be obtained directly from the polynomial chaos expansion $\hat{U}$. The mean is simply

$$E[Y] \approx c_0,$$

and the variance is

$$V[Y] \approx \sum_{n=1}^{N_p-1} \gamma_n c_n^2,$$

where $\gamma_n$ is a normalization factor defined as

$$\gamma_n = \mathbb{E}[\phi_n^2(Q)].$$
The first and total-order Sobol indices can also be calculated directly from the polynomial chaos expansion (Sudret, 2008; Crestaux et al., 2009). On the other hand, the percentiles (Equation 5), and thereby the prediction interval (Equation 6), must be estimated by using $\tilde{U}$ as a surrogate model and then performing the same procedure as for the (quasi-)Monte Carlo method.

### 2.6. Dependency Between Uncertain Parameters

One of the underlying assumptions when creating the polynomial chaos expansions is that the model parameters are independent. However, dependent parameters in neuroscience models are quite common (Achard and De Schutter, 2006). Fortunately, models containing dependent parameters can be analyzed with Uncertainpy by the aid of the Rosenblatt transformation from Chaospy (Rosenblatt, 1952; Feinberg and Langtangen, 2015). Briefly explained, the idea is to create a reformulated model $\tilde{U}(x, t, R)$ based on an independent parameter set $R$, and then perform polynomial chaos expansions on the reformulated model. The Rosenblatt transformation is used to construct the reformulated model so it gives the same output (and statistics) as the original model, i.e.:

$$\tilde{U}(x, t, R) = U(x, t, Q).$$

(17)

For more information on the use of the Rosenblatt transformation, see the Uncertainpy documentation⁴ or Feinberg and Langtangen (2015).

### 2.7. Feature-Based Analysis

When measuring the membrane potential of a neuron, the precise timing of action potentials often varies between recordings, even if the experimental conditions are the same. This behavior is typical for biological systems. Since the experimental data displays such variation, it is often meaningless and misleading to base the success of a computational model on a direct point-to-point comparison between a particular experimental recording and model output (Druckmann et al., 2007; Van Geit et al., 2008). A common modeling practice is therefore to have the model reproduce essential features of the experimentally observed dynamics, such as the action-potential shape or action-potential firing rate (Druckmann et al., 2007). Such features are typically more robust across different experimental measurements, or across different model simulations, than the raw data or raw model output itself, at least if sensible features have been chosen.

Uncertainpy takes this aspect of neural modeling into account and is constructed so that it can extract a set of features relevant for various common model types in neuroscience from the raw model output. Examples include the action potential shape in single neuron models and the average interspike interval in neural network models. Thus Uncertainpy performs an uncertainty quantification and sensitivity analysis not only on the raw model output but also on a set of relevant features selected by the user. Lists of the implemented features are given in section 3.4, and the value of a feature-based analysis is illustrated in two of the case studies (sections 5.3 and 5.4).

### 3. USER GUIDE FOR UNCERTAINPY

Uncertainpy is a Python toolbox, tailored to make uncertainty quantification and sensitivity analysis easily accessible to the computational neuroscience community. The toolbox is based on Python, since Python is a high level, open-source language in extensive and increasing use within the scientific community (Oliphant, 2007; Einevoll, 2009; Müller et al., 2015). Uncertainpy works with both Python 2 and 3, and utilizes the Python packages Chaospy (Feinberg and Langtangen, 2015) and SALib (Herman and Usher, 2017) to perform the uncertainty calculations. In this section, we present a guide on how to use Uncertainpy. We do not present an exhaustive overview, and only show the most commonly used classes, methods and method arguments. We refer to the online documentation⁴ for the most recent, complete documentation. A complete case study with code is shown in section 4.1.

Uncertainpy is easily installed by following the instructions in section 3.8. After installation, we get access to Uncertainpy by simply importing it:

```
import uncertainpy as un
```

Performing an uncertainty quantification and sensitivity analysis with Uncertainpy includes three main components:

1. The `model` we want to examine.
2. The `parameters` of the model.
3. Specifications of `features` in the model output.

The model and parameters are required components, while the feature specifications are optional. The three (or two) components are brought together in the `UncertaintyQuantification` class. This class performs the uncertainty calculations and is the main class the user interacts with. In this section, we explain how to use `UncertaintyQuantification` with the above components, and introduce a few additional utility classes.

### 3.1. The Uncertainty Quantification Class

The `UncertaintyQuantification` class is used to define the problem, perform the uncertainty quantification and sensitivity analysis, and save and visualize the results. Among others, `UncertaintyQuantification` takes the arguments:

```
UQ = un.UncertaintyQuantification(  # Required
    model=...,  
    parameters=...,  
    # Optional
    features=...  )
```

³http://uncertainpy.readthedocs.io/

⁴http://uncertainpy.readthedocs.io/
The model argument is either a Model instance (section 3.2) or a model function (section 3.2.2). The parameters argument is either a Parameters instance or a parameter dictionary (section 3.3). Lastly, the features argument is either a Features instance (section 3.4) or a list of feature functions (section 3.4.1). In general, using the class instances as arguments give more options, while using the corresponding functions are slightly easier. We go through how to use each of these classes and corresponding functions in the next three sections.

After the problem is set up, an uncertainty quantification and sensitivity analysis can be performed by using the UncertaintyQuantification.quantify method. Among others, quantify takes the optional arguments:

```python
data = UQ.quantify(
    method="pc"|"mc",
    pc_method="collocation"|"spectral",
    single=False
)
```

The method argument allows the user to choose whether Uncertainpy should use polynomial chaos expansions ("pc") or the quasi-Monte Carlo method ("mc") to calculate the relevant statistical metrics. If polynomial chaos expansions are chosen, pc_method further specifies whether point collocation ("collocation") or spectral projection ("spectral") methods are used to calculate the expansion coefficients. single specifies whether we perform the uncertainty quantification for a single parameter at the time, or consider all uncertain parameters at once. Performing the uncertainty quantification for one parameter at the time is a simple form of screening. The idea of such a screening is to use a computationally cheap method to reduce the number of uncertain parameters by setting the parameters that have the least effect on the model output to fixed values. We can then consider only the parameters with the greatest effect on the model output when performing the “full” uncertainty quantification and sensitivity analysis. This screening can be performed using both polynomial chaos expansions and the quasi-Monte Carlo method, but polynomial chaos expansions are almost always the faster choice. If nothing is specified, Uncertainpy by default uses polynomial chaos expansions based on point collocation with all uncertain parameters. The Rosenblatt transformation is automatically used if the input parameters are dependent.

The results from the uncertainty quantification are returned in data, as a Data object (see section 3.6). By default, the results are also automatically saved in a folder named data, and the figures are automatically plotted and saved in a folder named figures, both in the current directory. The returned Data object is therefore not necessarily needed.

As mentioned earlier, there is no guarantee that each set of sampled parameters produces a valid model or feature output. In such cases, Uncertainpy gives a warning which includes the number of runs that failed to return a valid output and performs the uncertainty quantification and sensitivity analysis using the reduced set of valid runs. However, if a large fraction of the simulations fail, the user could consider redefining the problem (e.g., by using narrower parameter distributions).

Polynomial chaos expansions are recommended as long as the number of uncertain parameters is small (typically < 20), as polynomial chaos expansions in these cases are much faster than the quasi-Monte Carlo method. Which of the polynomial chaos expansion methods to preferably use is problem dependent. In general, the pseudo-spectral method is faster than point collocation, but has a lower stability. We therefore recommend to use the point-collocation method.

The accuracy of the quasi-Monte Carlo method and polynomial chaos expansions is problem dependent and is determined by the chosen number of samples $N$, as well as the polynomial order $p$ for polynomial chaos expansions. It is therefore a good practice to examine if the results from the uncertainty quantification and sensitivity analysis have converged (Eck et al., 2016). A simple method for doing this is to increase or decrease the number of samples or polynomial order, or both, and examine the difference between the current and previous results. If the differences are small enough, we can be reasonably certain that we have an accurate result.

### 3.2. Models

In order to perform the uncertainty quantification and sensitivity analysis of a model, Uncertainpy needs to set the parameters of the model, run the model using those parameters, and receive the model output. Uncertainpy has built-in support for NEURON and NEST models, found in the NeuronModel (section 3.2.4) and NestModel (section 3.2.5) classes respectively. It should be noted that while Uncertainpy is tailored toward neuroscience, it is not restricted to neuroscience models. Uncertainpy can be used on any model that meets the criteria in this section. Below, we first explain how to create custom models, before we explain how to use NeuronModel and NestModel.

#### 3.2.1. The Model Class

Generally, models are created through the Model class. Among others, Model takes the argument run and the optional arguments interpolate, labels, postprocess and ignore.

```python
model = un.Model(
    run=example_model,
    interpolate=True,
    labels=["xlabel", "ylabel"],
    postprocess=example_postprocess,
    ignore=False
)
```

The run argument must be a Python function that runs a simulation on a specific model for a given set of model parameters and returns the simulation output. In this paper we call such a function a model function. If we set interpolate=True, Uncertainpy automatically interpolates the model output to a regular form, meaning each model evaluation has the same number of measurement points (most commonly time points). An irregular model, on the other hand, has a varying number of measurement points between different evaluations (the output...
is on an irregular form), a typical example is a model that uses adaptive time steps. The uncertainty quantification requires the model output to be on a regular form, and we must set `interpolate=True` for irregular models. labels allows the user to specify a list of labels to be used on the axes when plotting the results. The `postprocess` argument is a Python function used to post-process the model output if required. We will go into details on the requirements of the `postprocess` and model functions below. Finally, if `ignore=True` we do not perform an uncertainty quantification of the model output. This is used if we want to examine features of the model, but are not interested in the model result itself.

### 3.2.2. Defining a Model Function

As explained above, the `run` argument is a Python function that runs a simulation of a specific model for a given set of model parameters, and returns the simulation output. An example outline of a model function is:

```python
def example_model(parameter_1, parameter_2):
    # An algorithm for the model, 
    # or a script that runs an 
    # external model, using the 
    # given input parameters.
    # Returns the model output and 
    # model time along with the 
    # optional info object.
    return time, values, info
```

Such a model function has the following requirements:

1. **Input.** The model function takes a number of arguments which define the uncertain parameters of the model.
2. **Run the model.** The model must then be run using the parameters given as arguments.
3. **Output.** The model function must return at least two objects, the model time (or equivalent, if applicable) and model output. Additionally, any number of optional info objects can be returned. In Uncertainpy, we refer to the time object as `time`, the model output object as `values`, and the remaining objects as `info`.

   a. **Time** (`time`). time can be interpreted as the x-axis of the model. It is used when interpolating (see below), and when certain features are calculated. We can return `None` if the model has no time associated with it.

   b. **Model output** (`values`). The model output must either be regular (each model evaluation has the same number of measurement points), or it must be possible to interpolate or post-process the output (see section 3.2.3) to a regular form.

   c. **Additional info** (`info`). Some of the methods provided by Uncertainpy, such as the later defined model post-processing, feature pre-processing, and feature calculations, require additional information from the model (e.g., the time when a neuron receives an external stimulus). This information can be passed on as any number of additional `info` objects returned after `time` and `values`. We recommend using a single dictionary as `info` object, with key-value pairs for the information, to make debugging easier. Uncertainpy always uses a single dictionary as the `info` object. Certain features require specific keys to be present in this dictionary.

The model itself does not need to be implemented in Python. Any simulator can be used, as long as we can set the model parameters and retrieve the simulation output via Python. As a shortcut, we can pass a model function to the `model` argument in `UncertaintyQuantification`, instead of first having to create a `Model` instance.

### 3.2.3. Defining a Post-process Function

The `postprocess` function is used to post-process the model output before it is used in the uncertainty quantification. Post-processing does not change the model output sent to the feature calculations. This is useful if we need to transform the model output to a regular form for the uncertainty quantification, but still need to preserve the original model output to reliably detect the model features. Figure 2 illustrates how the objects returned by the model function are sent to both model `postprocess` and feature `preprocess` (see section 3.4).

An example outline of the `postprocess` function is:

```python
def example_postprocess(time, values, info):
    # Post-process the result to a 
    # regular form using time, values, 
    # and info returned by the model 
    # function.
    # Return the post-processed 
    # model output and time.
    return time_postprocessed, 
    values_postprocessed
```

The only time post-processing is required for Uncertainpy to work is when the model produces output that cannot be interpolated to a regular form by Uncertainpy. Post-processing is for example required for network models that give output in the form of spike trains, i.e., time values indicating when a given neuron fires. It should be noted that post-processing of spike trains is already implemented in Uncertainpy (see section 3.2.5). For most purposes, user-defined post-processing will not be necessary.

The requirements for the `postprocess` function are:

1. **Input.** The `postprocess` function must take the objects returned by the model function as input arguments.
2. **Post-processing.** The model time (`time`) and output (`values`) must be post-processed to a regular form, or to a form that can be interpolated to a regular form by Uncertainpy. If additional information is needed from the model, it can be passed along in the `info` object.
3. **Output.** The `postprocess` function must return two objects:
FIGURE 2 | Classes that affect the objects returned by the model. The Uncertainpy methods that use, change, and perform calculations on the objects returned by the model function \((\text{time, values, and the optional info})\). Functions associated with the model are in red while functions associated with features are in green.

(a) **Model time** \((\text{time\_postprocessed})\). The first object is the post-processed time (or equivalent) of the model. We can return \text{None} if the model has no time. Note that the automatic interpolation can only be performed if a post-processed time is returned (if an interpolation is required).

(b) **Model output** \((\text{values\_postprocessed})\). The second object is the post-processed model output.

3.2.4. NEURON Model Class

NEURON (Hines and Carnevale, 1997) is a widely used simulator for multi-compartmental neural models. Uncertainpy has support for NEURON models through the \text{NeuronModel} class, a subclass of \text{Model}. Among others, \text{NeuronModel} takes the arguments:

```python
model = un.NeuronModel(
    file="mosinit.hoc",
    path="path/to/neuron_model",
    interpolate=True,
    stimulus_start=1000,  # ms
    stimulus_end=1900  # ms
)
```

The \text{file} argument is the name of the hoc file that loads the NEURON model, which by default is \text{mosinit.hoc}. \text{path} is the path to the folder where the NEURON model is saved (the location of the \text{mosinit.hoc} file). \text{interpolate} indicates whether the NEURON model uses adaptive time steps and therefore should be interpolated. \text{stimulus\_start} and \text{stimulus\_end} denote the start and end time of any stimulus given to the neuron. \text{NeuronModel} loads the NEURON model from \text{file}, sets the parameters of the model, evaluates the model and returns the somatic membrane potential of the neuron (we record the voltage from the segment named "soma"). \text{NeuronModel} therefore does not require a model function to be defined. A case study of a NEURON model analyzed with Uncertainpy is found in section 4.3.

If changes are needed to the standard \text{NeuronModel}, such as measuring the voltage from other locations than the soma, the \text{Model} class with an appropriate model function could be used instead. Alternatively, \text{NeuronModel} can be subclassed and the existing methods customized as required. An example of the latter is shown in \text{uncertainpy/examples/bahl/}.

3.2.5. NEST Model Class

NEST (Peyser et al., 2017) is a simulator for large networks of spiking neurons. NEST models are supported through the \text{NestModel} class, another subclass of \text{Model}:

```python
model = un.NestModel(
    run=nest_model_function,
    ignore=False
)
```

Unlike \text{NeuronModel}, \text{NestModel} requires the model function to be specified through the \text{run} argument. The NEST model function has the same requirements as a regular model function, except it is restricted to return only two objects: the final simulation time (denoted \text{simulation\_end}), and a list of spike times for selected neurons in the network, which we refer to as spike trains (denoted \text{spiketrains}).

A spike train returned by a NEST model is a set of irregularly spaced time points where a neuron fired a spike. NEST models therefore require post-processing to make the model output regular. Such a post-processing is provided by the implemented \text{NestModel.postprocess} method, which converts a spike train to a list of zeros (no spike) and ones (a spike) for each
time step in the simulation. For example: If a NEST simulation returns the spike train \([0, 2, 3.5]\), it means the neuron fired three spikes occurring at \(t = 0, 2,\) and \(3.5\) ms, respectively. If the simulation has a time resolution of 0.5 ms and ends after 4 ms, NestModel.postprocess will return the post-processed spike train \([1, 0, 0, 0, 1, 0, 0, 1, 0]\), and the post-processed time array \([0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4]\). The final uncertainty quantification of a NEST network therefore predicts the probability for a spike to occur at any specific time point in the simulation. It should be noted that performing an uncertainty quantification of the post-processed NEST model output is computationally expensive. As such we recommend setting ignore=True as long as you are not interested in the uncertainty of the spike trains from the network. An Uncertainpy-based analysis of a NEST model is found in the case study in section 4.4.

### 3.3. Parameters of the Model

The parameters of a model are defined by two properties: They must have (i) a name and (ii) either a fixed value or a distribution. It is important that the name of a parameter is the same as the name given as the input argument in the model function. A parameter is considered uncertain if it is given a probability distribution, which are defined using Chaospy. 64 different univariate distributions are available in Chaospy, and Chaospy has support for easy creation of multivariate distributions. For a list of available distributions and detailed instructions on how to create probability distributions with Chaospy, see section 3.3 in Feinberg and Langtangen (2015).

The parameters are defined by the Parameters class. Parameters takes the argument parameters, which is a dictionary where the names of the parameters are the keys, and the fixed values or distributions of the parameters are the values. Here is an example of such a parameter dictionary with two parameters, where the first is named name_1 and has a uniform probability distribution in the interval \([8, 16]\), and the second is named name_2 and has a fixed value of 42:

```python
import chaospy as cp

parameters = {
    "name_1": cp.Uniform(8, 16),
    "name_2": 42
}
```

Parameters is now initialized as:

```python
parameters = un.Parameters(parameters=parameters)
```

As a shortcut, we can pass the above parameter dictionary to the parameters argument in UncertaintyQuantification, instead of first having to create a Parameters instance.

If the parameters do not have separate univariate probability distributions, but a joint multivariate probability distribution, the multivariate distribution can be set by giving Parameters the optional argument distribution:

```python
parameters = un.Parameters(parameters=parameters, distribution=multivariate)
```

### 3.4. Features

As discussed in section 2.7, it is often more meaningful to examine the uncertainty in salient features of the model output, than to base the analysis directly on a point-to-point comparison of the raw output data (e.g., a voltage trace). Upon user request, Uncertainpy can identify and extract features of the model output. If we give the features argument to UncertaintyQuantification, Uncertainpy will perform uncertainty quantification and sensitivity analysis of the given features, in addition to the analysis of the raw output data (if desired).

Three sets of features come predefined with Uncertainpy, SpikingFeatures, EfelFeatures, and NetworkFeatures. Each feature class contains a set of features tailored toward one specific type of neuroscience models. We first explain how to create custom features, before explaining how to use the built-in features.

Features are defined through the Features class:

```python
feature_functions = [example_feature]

features = un.Features(  
    features_to_run="example_feature",  
    preprocess=example_preprocess,  
    interpolate="example_feature"
)
```

The new_features argument is a list of Python functions that each calculates a specific feature, whereas features_to_run specifies which of the features to perform uncertainty quantification of. If nothing is specified, the uncertainty quantification is by default performed on all features (features_to_run="all"). preprocess is a Python function that performs common calculations for all features. interpolate is a list of features that are irregular. As with models, Uncertainpy automatically interpolates the output of these features to a regular form. Below we first go into detail on the requirements of a feature function, and then the requirements of a preprocess function.

### 3.4.1. Feature Functions

A feature is given as a Python function. The outline of such a feature function is:

```python
def example_feature(time, values, info):
    # Calculate the feature using
    # time, values, and info.
```
Feature functions have the following requirements:

1. **Input.** The feature function takes the objects returned by the model function as input, except when a preprocess function is used (see below). In those cases, the feature function instead takes the objects returned by the preprocess function as input. preprocess is normally not used.

2. **Feature calculation.** The feature function calculates the value of a feature from the data given in time, values and optional info objects. As previously mentioned, all built-in features in Uncertainpy, info is a dictionary containing required information as key-value pairs.

3. **Output.** The feature function must return two objects:
   
   (a) **Feature time** (time feature). The time (or equivalent) of the feature. We can return None instead for features where this is not relevant.
   
   (b) **Feature values** (values feature). The result of the feature calculation. As for the model output, the feature result must be regular, or able to be interpolated. If there are no feature result for a specific model evaluation (e.g., if the feature was spike width and there were no spikes), the feature function can return None. The specific feature evaluation is then discarded in the uncertainty calculations.

As with models, we can, as a shortcut, directly give a list of feature functions as the feature argument in UncertaintyQuantification, instead of first having to create a Features instance.

3.4.2. Feature Pre-processing

Some of the calculations needed to quantify features may overlap between different features. One example is finding the spike times from a voltage trace. The preprocess function is used to avoid having to perform the same calculations several times. An example outline of a preprocess function is:

```python
def preprocess(time, values, info):
    # Perform all common calculations using time, values, and info returned by the model function.

    # Return the pre-processed model output and info.
    return time_preprocessed, values_preprocessed, info
```

The requirements for a preprocess function are:

1. **Input.** A preprocess function takes the objects returned by the model function as input.

2. **Pre-processing.** The model output (time, values, and additional info objects) are used to perform all pre-process calculations.

3. **Output.** The preprocess function can return any number of objects as output. The returned pre-process objects are used as input arguments to the feature functions, so the two must be compatible.

Figure 2 illustrates how the objects returned by the model function are passed to preprocess, and the returned pre-process objects are used as input arguments in all feature functions. This pre-processing makes feature functions have different required input arguments depending on the feature class they are added to. As mentioned earlier, Uncertainpy comes with three built-in feature classes. These classes all take the new_features argument, so custom features can be added to each set of features. These feature classes all perform a pre-processing and therefore have different requirements for the input arguments of new feature functions. Additionally, certain features require specific keys to be present in the info dictionary. Each class has a reference_feature method that states the requirements for feature functions of that class in its docstring.

3.4.3. Spiking Features

Here we introduce the SpikingFeatures class, which contains a set of features relevant for models of single neurons that receive an external stimulus and respond by producing a series of action potentials, also called spikes. Many of these features require the start time and end time of the stimulus, which must be returned as info["stimulus_start"] and info["stimulus_end"] in the model function. info is then used as an additional input argument in the calculation of each feature. A set of spiking features is created by:

```python
features = SpikingFeatures()
```

SpikingFeatures implements a preprocess method, which locates spikes in the model output. This preprocess method can be customized; see the documentation on SpikingFeatures.

The features included in SpikingFeatures are briefly defined below. This set of features was taken from the previous work of Druckmann et al. (2007), with the addition of the number of action potentials during the stimulus period. We refer to the original publication for more detailed definitions.

1. nr_spikes - Number of action potentials (during stimulus period).
2. spike_rate - Action-potential firing rate (number of action potentials divided by stimulus duration).
3. time_before_first_spike - Time from stimulus onset to first elicited action potential.
4. accommodation_index - Accommodation index (normalized average difference in length of two consecutive interspike intervals).
5. average_AP_overshoot - Average action-potential peak voltage.
6. average_AHP_depth – Average afterhyperpolarization depth (average minimum voltage between action potentials).
7. average_AP_width – Average action-potential width taken at the midpoint between the onset and peak of the action potential.

The user may want to add custom features to the set of features in SpikingFeatures. The SpikingFeatures.preprocess method changes the input given to the feature functions, and as such each spiking feature function has the following input arguments:

1. The time array returned by the model simulation.
2. A Spikes object (spikes) which contain the spikes found in the model output.
3. An info dictionary with info["stimulus_start"] and info["stimulus_end"] set.

The Spikes object is the pre-processed version of the model output, used as a container for Spike objects. In turn, each Spike object contains information about a single spike. This information includes a brief voltage trace represented by a time and a voltage (V) array that only includes the selected spike. The information in Spikes is used to calculate each feature. As an example, let us create a feature that is the time at which the first spike in the voltage trace ends. Such a feature can be defined as follows:

```python
def first_spike_ends(time, spikes, info):
    # Get the first spike
    spike = spikes[0]
    # The last time point
    # in the spike
    values_feature = spike.t[-1]
    return None, values_feature
```

This feature may now be used as a feature function in the list given to the new_features argument.

From the set of both built-in and user-defined features, we may select subsets of features that we want to use in the analysis of a model. Let us say we are interested in how the model performs in terms of the three features: nr_spikes, average_AHP_depth and first_spike_ends. A spiking features object that calculates these features is created by:

```python
features_to_run = [
    "nr_spikes",
    "average_AHP_depth",
    "first_spike_ends"
]
features = un.SpikingFeatures(
    new_features=[first_spike_ends],
    features_to_run=features_to_run
)
```

### 3.4.4. eFEL Features

A more extensive set of features for single neuron voltage traces is found in the Electrophys Feature Extraction Library (eFEL) (Blue Brain Project, 2015). A set of eFEL spiking features is created by:

```python
features = EfelFeatures()
```

Uncertainpy has all features in the eFEL library in the EfelFeatures class. At the time of writing, eFEL contains 160 different features. Due to the high number of features, we do not list them here, but refer to the eFEL documentation for detailed definitions, or the Uncertainpy documentation for a list of the features. EfelFeatures is used in the same way as SpikingFeatures.

### 3.4.5. Network Features

The last set of features implemented in Uncertainpy is found in the NetworkFeatures class:

```python
features = NetworkFeatures()
```

This class contains a set of features relevant for the output of neural network models. These features are calculated using the Elephant Python package (NeuralEnsemble, 2017). The implemented features are:

1. average_firing_rate – Average firing rate (for a single recorded neuron).
2. instantaneous_rate – Instantaneous firing rate (averaged over all recorded neurons within a small time window).
3. average_isi – Average interspike interval (averaged over all recorded neurons).
4. cv – Coefficient of variation of the interspike interval (for a single recorded neuron).
5. average_cv – Average coefficient of variation of the interspike interval (averaged over all recorded neurons).
6. local_variation – Local variation (variability of interspike intervals for a single recorded neuron).
7. average_local_variation – Average local variation (variability of interspike intervals averaged over all recorded neurons).
8. fanofactor – Fanofactor (variability of spike trains).
9. victor_purpura_dist – Victor-Purpura distance (spike train dissimilarity between two recorded neurons).
10. van_rossum_dist – Van Rossum distance (spike train dissimilarity between two recorded neurons).
11. binned_isi – Histogram of the interspike intervals (for all recorded neurons).
12. corrcoef – Pairwise Pearson’s correlation coefficients (between the binned spike trains of two recorded neurons).
13. covariance – Covariance (between the binned spike trains of two recorded neurons).

A few of these network features can be customized; see the documentation on NetworkFeatures for a further explanation.

---

http://efel.readthedocs.io
The use of NetworkFeatures in Uncertainpy follows the same logic as the use of the other feature classes, and custom features can easily be included. As with SpikingFeatures, NetworkFeatures implements a preprocess method. This preprocess returns the following objects:

1. End time of the simulation (end_time).
2. A list of NEO (Garcia et al., 2014) spike trains (spiketrains).

Each feature function added to NetworkFeatures therefore requires these objects as input arguments. Note that the info object is not used.

### 3.5. Uncertainty Calculations in Uncertainpy

In this section, we describe how Uncertainpy performs the uncertainty calculations, as well as which options the user has to customize the calculations. Moreover, a detailed insight into this is not required to use Uncertainpy, as in most cases the default settings work fine. In addition to the customization options shown below, Uncertainpy has support for implementing entirely custom uncertainty-quantification and sensitivity-analysis methods. This is only recommended for expert users, as knowledge of both Uncertainpy and uncertainty quantification is needed. We do not go into detail here but refer to the Uncertainpy documentation for more information.

#### 3.5.1. Quasi-Monte Carlo Method

To use the quasi-Monte Carlo method, we call `quantify` with the argument `method=“mc”`, and the optional argument `nr_mc_samples`:

```python
data = UQ.quantify(
    method=“mc”,
    nr_mc_samples=10**4
)
```

The quasi-Monte Carlo method quasi-randomly draws \( Ns = N(d + 2)/2 \) parameter samples, where \( N = nr\_mc\_samples \), and \( d \) is the number of uncertain parameters. This is the number of samples required by Saltelli’s method to calculate the Sobol indices. By default `nr_mc_samples` = 10000. These samples are drawn from a multivariate independent uniform distribution using Saltelli’s sampling scheme, implemented in the SALib library (Saltelli et al., 2010; Herman and Usher, 2017). We use the Rosenblatt transformation to transform the samples from this uniform distribution to the parameter distribution given by the user. This transformation enables us to use Saltelli’s sampling scheme for any parameter distribution.

The model is evaluated for each of these parameter samples, and features are calculated from each model evaluation (when applicable). To speed up the calculations, Uncertainpy uses the multiprocess Python package (McKerns et al., 2012) to perform this step in parallel. When model and feature calculations are done, Uncertainpy calculates the mean, variance, and 5th and 95th percentile (which gives the 90% prediction interval) for the model and each feature. This is done using a subset with \( N \) number of samples of the total set. We are unable to use the full set since not all samples are independent in Saltelli’s sampling scheme. The Sobol indices are calculated using Saltelli’s method and the complete set of samples. We use a modified version of the method in the SALib library, which is able to handle model evaluations with any number of dimensions.

Saltelli’s method requires all model and feature evaluations to return a valid result. When this is not the case we use the workaround\(^6\) suggested by Herman and Usher (2017), and replace invalid model and feature evaluations with the mean of that model or feature. This workaround introduces an error depending on the number of missing evaluations but enables us to still calculate the Sobol indices. If there are invalid model or feature evaluations, Uncertainpy gives a warning which includes the number of invalid evaluations.

#### 3.5.2. Polynomial Chaos Expansions

To use polynomial chaos expansions we use `quantify` with the argument `method=“pc”`, which takes a set of optional arguments (the specified values are the default):

```python
data = UQ.quantify(
    method=“pc”,
    pc_method=“collocation”,
    rosenblatt=“auto”,
    polynomial_order=4,
    nr_collocation_nodes=None,
    quadrature_order=None,
    nr_pc_mc_samples=10**4
)
```

As previously mentioned, Uncertainpy allows the user to select between point collocation (`pc_method=“collocation”`) and pseudo-spectral projections (`pc_method=“spectral”`). The goal of both these methods is to create separate polynomial chaos expansions \( \hat{U}_{\text{model/feature}} \) for the model and each feature. The first step of both methods is the same: Uncertainpy starts by creating the orthogonal polynomial \( \phi_n \) using \( \rho_Q \) and the three-term recurrence relation if available, otherwise the discretized Stieltjes method (Stieltjes, 1884) is used. By default, Uncertainpy uses a fourth order polynomial expansion, as recommended by Eck et al. (2016). The polynomial order \( p \) can be changed with the `polynomial_order` argument. The polynomial \( \phi_n \) is the same for the model and all features, since they have the same uncertain input parameters, and therefore the same \( \rho_Q \). Only the polynomial coefficients \( c_n \) differ between the model and each feature.

The two polynomial chaos methods differ in terms of how they calculate \( c_n \). For point collocation Uncertainpy uses \( N_t = 2(N_p + 1) \) collocation nodes, as recommended by Hosder et al. (2007), where \( N_p \) is the number of polynomial chaos expansion factors. The number of collocation nodes can be customized with `nr_collocation_nodes` (\( N_t \)), but the new number of nodes must be chosen carefully. The collocation nodes are sampled from \( \rho_Q \) using Hammersley sampling (Hammersley, 1960), also as recommended by Hosder et al. (2007). The model and features are calculated for each of the collocation nodes. As with the quasi-Monte Carlo method, this step is performed in parallel.

\(^6\)https://github.com/SALib/SALib/issues/134
The polynomial coefficients $c_n$ are calculated using the model and feature results, and Tikhonov regularization (Rifkin and Lippert, 2007).

For the pseudo-spectral projection, Uncertainpy chooses nodes and weights using a quadrature scheme, instead of choosing nodes from $RQ$. The quadrature scheme used is Leja quadrature with a Smolyak sparse grid (Smolyak, 1963; Narayan and Jakeman, 2014). The Leja quadrature is by default of order two greater than the polynomial order, but this can be changed with quadrature_order. The model and features are calculated for each of the quadrature nodes. As before, this step is performed in parallel. The polynomial coefficients $c_n$ are then calculated from the quadrature nodes, weights, and model and feature results.

When Uncertainpy has derived $\hat{U}$ for the model and features, it uses $\hat{U}$ to compute the mean, variance, first and total-order Sobol indices, as well as the average first and total-order Sobol indices. Finally, Uncertainpy uses $\hat{U}$ as a surrogate model and employs the quasi-Monte Carlo method with Hammersley sampling and $nr_{pc.mc.samples}=10^4$ samples to find the 5th and 95th percentiles.

If the model parameters have a dependent joint multivariate distribution, the Rosenblatt transformation is by default automatically used. This can be changed by setting rosenblatt=True to always use the Rosenblatt transform, or rosenblatt=False to never use the Rosenblatt transformation. Note that the latter gives an error if you have dependent parameters. To perform this transformation Uncertainpy chooses a multivariate independent normal distribution $\rho_R$, which is used instead of $\rho_Q$ to perform the polynomial chaos expansions. Both the point-collocation method and the pseudo-spectral method are performed as described above. The only difference is that we use $\rho_R$ instead of $\rho_Q$, and use the Rosenblatt transformation to transform the selected nodes from $R$ to $Q$, before they are used in the model evaluation.

3.6. Data Format

All results from the uncertainty quantification and sensitivity analysis are returned as a Data object, as well as being stored in UncertaintyQuantification.data. The Data class works similarly to a Python dictionary. The names of the model and features are the keys, while the values are DataFeature objects that store each statistical metric in Table 1 as attributes. Results can be saved and loaded through Data.save and Data.load, and are saved either as HDF5 files (Collette, 2013) or Exdir structures (Dragly et al., 2018). HDF5 files are used by default.

An example: If we have performed an uncertainty quantification of a spiking neuron model with the number of spikes as one of the features, we can load the results and get the variance of the number of spikes by:

```python
data = un.Data()
data.load("filename")
variance = data["nr_spikes"].variance
```

3.7. Visualization

Uncertainpy plots the results for all zero and one-dimensional statistical metrics, and some of the two-dimensional statistical metrics. An example of a zero-dimensional statistical metric is the mean of the membrane potential over time for a neural network (Figure 8). An example of a one-dimensional statistical metric is the mean of the membrane potential over time for a multi-compartmental neuron (Figure 4). Lastly, an example of a two-dimensional statistical metric is the mean of the pairwise Pearson's correlation coefficient of a neural network (Figure 9). These visualizations are intended as a quick way to get an overview of the results, and not to create publication-ready plots. Custom plots of the data can easily be created by retrieving the results from the Data class.

3.8. Technical Aspects

Uncertainpy comes with an extensive test suite that can be run with the test.py script. For information on how to use test.py, run:

```bash
$ python test.py --help
```

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Calculated values and statistical metrics, for the model and each feature stored in the Data class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculated statistical metric</td>
<td>Symbol</td>
</tr>
<tr>
<td>Model and feature evaluations</td>
<td>$U$</td>
</tr>
<tr>
<td>Model and feature times</td>
<td>$t$</td>
</tr>
<tr>
<td>Mean</td>
<td>$E$</td>
</tr>
<tr>
<td>Variance</td>
<td>$V$</td>
</tr>
<tr>
<td>5th percentile</td>
<td>$P_5$</td>
</tr>
<tr>
<td>95th percentile</td>
<td>$P_{95}$</td>
</tr>
<tr>
<td>First-order Sobol indices</td>
<td>$S$</td>
</tr>
<tr>
<td>Total-order Sobol indices</td>
<td>$S_T$</td>
</tr>
<tr>
<td>Average of the first Sobol indices</td>
<td>$\bar{S}$</td>
</tr>
<tr>
<td>Average of the total Sobol indices</td>
<td>$\bar{S}_T$</td>
</tr>
</tbody>
</table>
4. EXAMPLE APPLICATIONS

In the current section, we demonstrate how to use Uncertainpy by applying it to four different case studies: (i) a simple model for the temperature of a cooling coffee cup implemented in Python, (ii) the original Hodgkin-Huxley model implemented in Python, (iii) a multi-compartmental model of a thalamic interneuron implemented in NEURON, and (iv) a sparsely connected recurrent network model implemented in NEST. The codes for all four case studies are available in uncertainpy/examples/, which generates all results shown in this paper. All the case studies can be run on a regular workstation computer. Uncertainpy does not create publication-ready figures, so custom plots have been created for the case studies. The code for creating all figures in this paper is found in a Jupyter Notebook in uncertainpy/examples/paper_figures/.

For simplicity, uniform distributions were assumed for all parameter uncertainties in the example studies. Further, the results for the case studies are calculated using point collocation. For the examples shown we used the default polynomial order of $p = 4$, but also confirmed that the results converged by increasing the polynomial order to $p = 5$, which gave similar results (results not shown).

The case studies were run in a Docker\footnote{https://www.docker.com/} container with Python 3, created from the Dockerfile uncertainpy/.docker/Dockerfile_uncertainpy3. A similar Dockerfile is available for Python 2. The used version of Uncertainpy is 1.0.1, commit b7b3fa0, and Zenodo\footnote{https://zenodo.org/} DOI 10.5281/zenodo.1300336. We also used NEST 2.14.0, NEURON 7.5, and Chaospy commit 05fea24. A requirements file that specifies the version of all used Python packages is located in uncertainpy/examples/paper_figures/.

4.1. Cooling Coffee Cup

To give a simple, first demonstration of Uncertainpy, we perform an uncertainty quantification and sensitivity analysis of a hot cup of coffee that follows Newton’s law of cooling. We start with a model that has independent uncertain parameters, before we modify the model to have dependent parameters to show an example requiring the Rosenblatt transformation.

4.1.1. Cooling Coffee Cup With Independent Parameters

The temperature $T$ of the cooling coffee cup is given by:

$$\frac{dT(t)}{dt} = -\kappa(T(t) - T_{env}),$$

where $T_{env}$ is the temperature of the environment in units of $^{\circ}C$. $\kappa$ is a cooling constant in units of 1/min that is characteristic of the system and describes how fast the coffee cup radiates heat to the environment. We set the initial temperature to a fixed value, $T_0 = 95^{\circ}C$, and assume that $\kappa$ and $T_{env}$ are uncertain parameters characterized by the uniform probability distributions:

$$\rho_\kappa = \text{Uniform}(0.025, 0.075),$$
$$\rho_{T_{\text{env}}} = \text{Uniform}(15, 25).$$

The following code is available in uncertainpy/examples/coffee_cup/. We start by importing the packages required to perform the uncertainty quantification:

```python
import uncertainpy as un
import chaospy as cp
import numpy as np
from scipy.integrate import odeint
```

Next, we create the cooling coffee-cup model. To do this we define a Python function (coffee_cup) that takes the uncertain parameters $\kappa$ and $T_{\text{env}}$ as input arguments, solves Equation (18) by integration using scipy.integrate.odeint over 200 min, and returns the resulting time and temperature arrays.

```python
def coffee_cup(kappa, T_env):
    # Initial temperature and time array
    time = np.linspace(0, 200, 150) # Minutes
    T_0 = 95 # Celsius
    # The equation describing the model
    def f(T, time, kappa, T_env):
        return -kappa*(T - T_env)
    # Solving the equation by integration
    temperature = odeint(f, T_0, time, args=(kappa, T_env))[:, 0]
    # Return time and model output
    return time, temperature
```

We now use coffee_cup to create a Model object, and add labels:

```python
model = un.Model(
    run=coffee_cup,
    labels=['Time (min)', 'Temperature (C)'])
```

As previously mentioned, it is possible to use coffee_cup directly as the model argument in the UncertaintyQuantification class, however we would then be unable to specify the labels.

In the next step, we use Chaospy to assign distributions to the uncertain parameters $\kappa$ and $T_{env}$, and use these distributions to create a parameter dictionary:
# Create the distributions
kappa_dist = cp.Uniform(0.025, 0.075)
T_env_dist = cp.Uniform(15, 25)

# Define the parameter dictionary
parameters = {"kappa": kappa_dist,
              "T_env": T_env_dist}

We can now set up the UncertaintyQuantification:

```python
UQ = un.UncertaintyQuantification(model=model,
                                  parameters=parameters)
```

With that, we are ready to calculate the uncertainty and sensitivity of the model. We use polynomial chaos expansions with point collocation, the default options of quantify, and set the seed for the random number generator to allow for precise reproduction of the results:

```python
data = UQ.quantify(seed=10)
```

quantify calculates all statistical metrics discussed in sections 2.2 and 2.3, but here we only show the mean, standard deviation (square root of the variance), and 90% prediction interval (Figure 3A), and the first-order Sobol indices (Figure 3B). The reason we plot the standard deviation instead of the variance is to make it easier to compare it to the mean. As the mean (blue line) in Figure 3A shows, the cooling gives rise to an exponential decay in the temperature, toward the temperature of the environment $T_{env}$. From the sensitivity analysis (Figure 3B) we see that $T$ is most sensitive to $\kappa$ early in the simulation, and to $T_{env}$ toward the end of the simulation. This is as expected since $\kappa$ determines the rate of the cooling, while $T_{env}$ determines the final temperature. After about 150 min, the cooling is essentially completed, and the uncertainty in $T$ exclusively reflects the uncertainty of $T_{env}$.

4.1.2. Cooling Coffee Cup With Statistically Dependent Parameters

Uncertainpy can also perform uncertainty quantification and sensitivity analysis using polynomial chaos expansions on models with statistically dependent parameters. Here we use the cooling coffee-cup model to construct such an example. Let us parameterize the coffee cup differently:

$$\frac{dT}{dt} = -\alpha \kappa (T(t) - T_{env}).$$  (21)

In order for the model to describe the same cooling process as before, the new variables $\alpha$ and $\hat{\kappa}$ should be dependent, so that $\alpha \hat{\kappa} = \kappa$. We can achieve this by demanding that $\rho_{\kappa} = \rho_{\alpha}$ (note that $\rho_{\alpha}$ should not include 0) and otherwise define the problem following the same procedure as in the original case study. Since this gives us a dependent distribution, Uncertainpy automatically uses the Rosenblatt transformation.

In this case, the distribution we assign to $\alpha$ does not affect the end result, as the distribution for $\hat{\kappa}$ will be scaled accordingly. Using the Rosenblatt transformation, an uncertainty quantification and sensitivity analysis of the dependent coffee-cup model therefore return the same results as seen in Figure 3, where the role of the original $\kappa$ is taken over by $\hat{\kappa}$, while the sensitivity to the additional parameter $\alpha$ becomes strictly zero (we do not show the results here, but see the example in uncertainpy/examples/coffee_cup_dependent/).

4.2. Hodgkin-Huxley Model

From here on, we focus on case studies more relevant for neuroscience, starting with the original Hodgkin-Huxley model (Hodgkin and Huxley, 1952). An uncertainty analysis of this model has been performed previously (Torres Valderrama et al., 2015), and we here repeat a part of that study using Uncertainpy.

The original version of the Hodgkin-Huxley model has eleven parameters with the numerical values listed in Table 2.
As in the previous study, we assume each of these parameters has a uniform distribution in the range ±10% around their original value. We use uncertainty quantification and sensitivity analysis to explore how these parameter uncertainties affect the model output, i.e., the action potential response of the neural membrane potential to an external current injection.

As in the cooling coffee-cup example, we implement the Hodgkin-Huxley model as a Python function and use polynomial chaos expansions with point collocation to calculate the model output, i.e., the action potential response. Some of the findings confirm what we would expect from a general knowledge of action potential firing (see figure 3.12 in Sterratt et al., 2011 for an overview). For example, it is not surprising that the action potential peak potential is most sensitive to the Na⁺ reversal potential \(E_{Na}\), since this parameter is known to closely determine the peak potential. Nor is it surprising that \(g_{K}\) is the most important parameter during the downstroke of the action potential, since the essential role of the K⁺ channel is to repolarize the neuron.

A sensitivity analysis such as that in Figure 4B may serve to give an insight into how different mechanisms are responsible for different aspects of the neuronal response. Some of the findings show a much higher sensitivity to the leak current \(E_L\) and \(g_L\) which are important for determining the resting potential of the neuron.

Other parts of the analysis reveal some less intuitive relationships. For example, Figure 4B shows that the membrane potential during the upstroke of the action potential is most sensitive to \(g_K\). This may be surprising given that the Na⁺ channel (parameterized by \(g_{Na}\) and \(E_{Na}\)) is responsible for depolarizing the neuron. This indicates that the all-or-nothing response of the Na⁺ channel activation is rather robust, and that variance during the upstroke predominantly is due to the effects of the K⁺ channel on the timing of the action-potential onset. Another unexpected observation is that \(E_{Na}\) has a high sensitivity within a time window after the peak of the action potential. This indicates that the Na⁺ channel is not fully closed, and is involved in determining the potential at which the neuron lingers within this time window.

Another aspect of modeling where sensitivity analysis can be useful, is in exploring the dependence on initial conditions. When analyzing complex models, it is common to discard the initial part of the simulation from the analysis, i.e., one lets the model run for a time \(T\) before one starts to analyze its dynamics. The rationale behind this is that the model over time loses its dependence on (arbitrarily set) initial conditions of its dynamic variables, and reaches its inherent steady-state dynamics. In the example studied here, only the response for \(T > 5\) ms is analyzed. Figure 4B shows that the Hodgkin-Huxley model then has a negligible sensitivity to the initial membrane potential \(V_0\) and initial activation states of the Na⁺ channel \(m_0\) and \(h_0\), but maintains a sensitivity to the initial Na⁺ inactivation state \(h_0\) through most of the simulation. Such a dependence on the initial condition of a state variable is typically unwanted and indicates that the model should have had more time to settle in before its response was analyzed.
4.3. Multi-Compartmental Model of a Thalamic Interneuron

In the next case study, we illustrate how Uncertainpy can be used on models implemented in NEURON (Hines and Carnevale, 1997). For this study, we select a previously published model of a thalamic interneuron in the dorsal lateral geniculate nucleus (dLGN) of the thalamus (Halnes et al., 2011). Since the model is implemented in NEURON, the original model can be used directly with Uncertainpy by using the `NeuronModel` class. The code for this case study is found in `uncertainpy/examples/interneuron/`.

In the original modeling study, seven active ion channels were tuned (by trial and error) to capture the responses of thalamic interneurons to different current injections (Halnes et al., 2011). Here, we consider one of the stimulus conditions used in the original study, and examine how sensitive the interneuron response is to uncertain ion-channel conductances. The conductances in the original model are listed in Table 3, and we assume they have uniform distributions in the interval ±10% around their original values.

The uncertainty quantification of the membrane potential in the soma of the interneuron is seen in Figure 5A. The variance (or standard deviation) indicates that the neuronal response varies strongly between the different parameterizations. To illustrate the variety of response characteristics hiding in the statistics in Figure 5A, four selected example simulations are shown in Figure 5B, all obtained by drawing the uncertain parameters from intervals ±10% around their original values. In line with the discussion in section 2.7, the qualitative differences between the responses indicate that a feature-based analysis is more informative than a point-to-point comparison of the voltage traces.

Since we examine a spiking neuron model, we want to use the features in the `SpikingFeatures` class for the feature-based analysis. `SpikingFeatures` needs to know the start
and end times of the stimulus to be able to calculate certain features. When we initialize `NeuronModel` we therefore specify the `stimulus_start` (set to 1,000 ms) and `stimulus_end` (set to 1,900 ms) arguments. Additionally, the interneuron model uses adaptive time steps, meaning we have to use `interpolate=True` (which is the default option of `NeuronModel`). We also specify the path to the folder where the neuron model is stored (for this example, it is `path="interneuron_modelDB/"`). As before, we use polynomial chaos expansions with point collocation to compute the statistical metrics for the model output and all features.

**Figure 6** shows the sensitivity of the features in `SpikingFeatures` to the various ion-channel conductances (see section 3.4.3 for definitions of the features). For illustrative purposes, only the first-order Sobol indices are shown (although `Uncertainpy` by default calculates all statistical metrics from sections 2.2 and 2.3).

A feature-based sensitivity analysis such as in **Figure 6** gives valuable insight into the role of various biological mechanisms in determining the firing properties of a neuron. Some of the results confirm what we would expect from a general knowledge of neurodynamics. For example, it is not surprising that the spike rate (A), the number of action potentials elicited throughout the simulation (E), and the action-potential amplitude (F) are most sensitive to the Na⁺ channel conductance $g_{Na}$, given the well-established role of the Na⁺ channel in action-potential generation. Likewise, given the role of the K⁺ channel in repolarizing the neuron after an action potential, it is not surprising that the action-potential width (D) is predominantly sensitive to $g_{Kdr}$.

The third most important parameter identified in this sensitivity analysis is the T-type Ca²⁺ conductance $g_{CaT}$, known to be important for burst firing in thalamic interneurons (Zhu et al., 1999; Halnes et al., 2011; Allken et al., 2014). T-type Ca²⁺ channels are typically activated when the membrane potential makes a sudden step from low to high values, such as at the stimulus onset. Upon activation, T-type Ca²⁺ channels then evoke Ca²⁺ spikes which may act to boost the initial response of a neuron to an external stimulus. This explains why the timing of the first spike (C) has such a high sensitivity to $g_{CaT}$. Bursts are typically more pronounced under other stimulus conditions than the one used in the current simulations, but in some cases, the Ca²⁺ spike was large enough to evoke small, initial bursts of action potentials (see example simulations in **Figure 5B**, II–IV, where the initial responses are small bursts of two action potentials). The additional action potentials in neurons that elicit bursts serve to explain why the spike rate (A) and total number of action potentials (G) also are highly sensitive to $g_{CaT}$.

A perhaps less expected result is that the depth of the afterhyperpolarization (G) (voltage dip following an action potential) has such a low sensitivity to the two K⁺ channels ($g_{Kdr}$ and $g_{AHP}$) in the model, as these are the channels that have a direct effect on the hyperpolarization of the neuron. As for many of the features in **Figure 6**, there are complex interactions between several mechanisms and the limited analysis considered here can only hint at the possible underlying relationships. Part of the explanation may be that the afterhyperpolarization current ($g_{AHP}$) is Ca²⁺ activated, and is more limited by the availability of Ca²⁺ than by its own maximum conductance. This could serve to explain the high sensitivity to the Ca²⁺ channel $g_{CaT}$. Furthermore, the high sensitivity to $g_{Na}$ implies that the Na⁺ channel also is open during the down-stroke of the action potential, and counteracts the hyperpolarizing K⁺ currents.

As **Figure 6** indicates, the variances of the `SpikingFeatures` are predominantly explained by the three model parameters $g_{Na}$, $g_{Kdr}$ and $g_{CaT}$, with some contributions from $g_{CaL}$, $g_{AHP}$ and negligible impact from the remaining
parameters $g_h$ and $g_{CAN}$. However, one should be cautious about generalizing insights found in an unexhaustive analysis such as the one presented here. Firstly, the presented analysis explores the sensitivity to variations within a $\pm 10\%$ range around the original parameter values, and thus spans a relatively local region of the parameter space. Additionally, this choice of distributions is a rather arbitrary choice and is unlikely to capture the actual uncertainty distributions. In reality, the uncertainty or biological variability, or both, in some of the parameters may have very different distributions, and an analysis that takes this into account could yield different results. Secondly, the above analysis was limited to a single stimulus protocol (a positive current step pulse of moderate magnitude to the soma), and a different stimulus protocol could activate a different set of neural mechanisms. For example, $g_h$ denotes the conductance of a hyperpolarization-activated cation current, which would need a negative current injection to activate. It is therefore not surprising that our analysis shows zero sensitivity to this parameter.

Thirdly, the SpikingFeatures class contains a limited number of features, and other features (e.g., from the more comprehensive EfelFeatures class) can be sensitive to the parameters that were observed to be of less importance in the current example. We do not here consider additional features, stimulus protocols, or uncertainty distributions in the analysis, as the main purpose of this case study was to demonstrate the use of Uncertainpy on a detailed multi-compartmental model.

### 4.4. Recurrent Network of Integrate-and-Fire Neurons

In the last case study, we use Uncertainpy to perform a feature-based analysis of the sparsely connected recurrent network of integrate-and-fire neurons by Brunel (2000). We implement the Brunel network using NEST inside a Python function, and
create 10,000 excitatory and 2,500 inhibitory neurons, with properties as specified by Brunel (2000). Each neuron has 1,000 randomly chosen connections to excitatory neurons and 250 randomly chosen connections to inhibitory neurons (a connection probability of \( e = 0.1 \)). The weight of the excitatory synapses (amplitude of excitatory synaptic current) is \( J = 0.1 \text{ mV} \). We simulate the network for 1,000 ms, record the output from 20 of the excitatory neurons, and start the recording after 100 ms. The code for this case study is found in [uncertaintypy](https://examples/brunel/).

Three more parameters are needed to specify the Brunel model: (i) the external input rate \( (v_{\text{ext}}) \) relative to the threshold rate \( (v_{\text{thr}}) \) given as \( \eta = v_{\text{ext}}/v_{\text{thr}} \), (ii) the relative strength of the inhibitory synapses compared to the excitatory synapses \( g \), and (iii) the synaptic delay \( D \). Depending on these parameters, the Brunel network may be in several different activity states. For the current case study we limit our analysis to two of these states, the synchronous regular (SR) state, where the neurons are almost completely synchronized, and the asynchronous irregular (AI) state, where the neurons fire mostly independently at low rates.

We create two sets of model parameters, one for the SR state and one for the AI state. For each set we assume that the uncertainties of the parameters \( \eta, g \) and \( D \) are characterized by uniform probability distributions within the ranges shown in Table 4. The parameter ranges are chosen so that all parameter combinations within the set give model behavior corresponding to one of the states. Two selected model results representative of the network in both states are shown in Figure 7, which illustrate the differences between the two states. Figure 7 shows the recorded spike trains for the Brunel network in the SR state between 200 ms and 300 ms of the simulation. The results in this time window exemplifies network behavior during the entire simulation after spiking has started. Since the firing rate is very high in this state, only results for a limited time window are shown. Figure 7B shows the recorded spike trains for the Brunel network in the AI state for the entire simulation period.

We use the features in NetworkFeatures to examine features of the network dynamics. Of the 13 built-in network features in NetworkFeatures, we here only focus on two: the average interspike interval and the pairwise Pearson’s correlation coefficient. These features are well suited to highlight the differences between the AI and SR network states, and to investigate how the details of the network dynamics depend on the model parameters within each of the states. We perform an uncertainty quantification and sensitivity analysis of the model and all features for each of the network states using polynomial chaos with point collocation. As for the previous examples we used the default polynomial order of \( p = 4 \) which was observed to be sufficient to achieve convergence, that is, the results did not change much when increasing \( p \) beyond 4.

We also explored the alternative situation where the excitatory synaptic weight \( J \) was included as a fourth uncertain parameter (with a similar relative spread as for the other uncertain parameters in Table 4). Here we observed that at least \( p = 7 \) (using the default number of collocation nodes) was needed to obtain accurate results. This illustrates that the required polynomial order, and by extension the required number of samples \( N_{\text{p}} \), to get accurate results is problem dependent.

### 4.4.1. Average Interspike Interval

The average interspike interval is the average time it takes from a neuron produces a spike until it produces the next spike, averaged over all recorded neurons. The uncertainty quantification and sensitivity analysis of the average interspike interval of the Brunel network are shown in Figure 8. The average interspike interval is seen to differ strongly between the SR and AI states. In the high-firing SR state (Figure 8A), the mean of the average interspike interval is low, with a comparatively low standard deviation reflecting the synchronous firing in the network. We can observe this in Figure 7A, where the interspike intervals are short and do not vary much (i.e., very little standard deviation). In the comparatively low-firing AI state (Figure 8B), the mean of the average interspike interval is high, with a large standard deviation, reflecting the irregular firing in the network seen in Figure 7B.

The two states were also found to be different in terms of which parameters the average interspike interval is sensitive to. In the SR state the network is predominantly sensitive to the synaptic delay \( D \). This reflects that in this state the neurons get very strong synaptic inputs so that the firing rate is mainly determined by the delay. In the AI state, the network is more balanced and “variance-driven”, and the dynamics are to a large degree determined by the relative strength of the inhibitory synapses compared to the excitatory synapses \( g \) (Brunel, 2000). Thus the average interspike interval is observed in Figure 7B to, not surprisingly, be most sensitive to \( g \). In the AI state the average interspike interval is quite long (~60 ms) so that an uncertainty in the synaptic delay of a couple of milliseconds (cf. Table 4) has little influence. Thus very little sensitivity to \( D \) is observed in this state.

### 4.4.2. Correlation Coefficient

The pairwise Pearson’s correlation coefficient is a measure of how synchronous the spiking of a network is. This correlation coefficient measures the correlation between the spike trains of two neurons in the network. In Figure 9 we examine how this correlation depends on parameter uncertainties by plotting the mean, standard deviation, and first-order Sobol indices for the pairwise Pearson’s correlation coefficient in the SR and AI states.

As expected from examining Figure 7, the mean pairwise Pearson’s correlation coefficient in the SR state (Figure 9A) is
FIGURE 7 Example model results for the Brunel network. (A) The recorded spike train for the Brunel network in the synchronous regular state between 200 and 300 ms of the simulation. (B) The recorded spike trains for the Brunel network in the asynchronous irregular state for the entire simulation period. The network has 10,000 excitatory and 2,500 inhibitory neurons, with properties as specified by Brunel (2000). Each neuron has 1,000 randomly chosen connections to excitatory neurons and 250 randomly chosen connections to inhibitory neurons. We simulate the network for 1,000 ms, record the output from 20 of the excitatory neurons, and start the recording after 100 ms.

FIGURE 8 The average interspike interval for the Brunel network in the two states. Mean, standard deviation, 90% prediction interval, and first-order Sobol indices of the average interspike interval of the Brunel network in the synchronous regular state (A), and asynchronous irregular state (B). The 90% prediction interval is indicated by the 5th and 95th percentiles, i.e., 90% of the average spike intervals are between \( P_5 \) and \( P_{95} \).

much higher than in the AI state (Figure 9D). The first-order Sobol indices further show that the degree of synchronicity is by far most sensitive to the synaptic delay \( D \) when the network is in the SR state (Figure 9C), and most sensitive to the relative strength of inhibitory synapses \( g \) when the network is in the AI state (Figure 9F).

Thus, for both features investigated here (the average interspike interval and the mean pairwise Pearson’s correlation coefficient), the conclusions regarding model sensitivity are the same. The SR state of the Brunel network is most sensitive to the synaptic delay \( D \), while the AI state is most sensitive to the relative strength of inhibitory synapses \( g \).

4.5. Comparing the Quasi-Monte Carlo Method to Polynomial Chaos Expansions

To compare the efficiency of the polynomial chaos expansions and the quasi-Monte Carlo method, we calculate the errors of the uncertainty quantification for the Hodgkin-Huxley model (section 4.2) using a varying number of model evaluations. The
FIGURE 9 | The pairwise Pearson’s correlation coefficient for the Brunel network in the two states. Mean (A,D), standard deviation (B,E), and first-order Sobol indices (C,F) for the pairwise Pearson’s correlation coefficient of the Brunel network in the synchronous regular (A–C) and asynchronous irregular (D–F) states.

code for this comparison can be found in uncertainpy/examples/mc_vs_pc.

As efficiency measure we use the number of model evaluations \( N_s \), since model evaluation generally is the computationally most costly step. We examine two versions of the Hodgkin-Huxley model to see how the efficiency of the two methods varies with the number of uncertain parameters. We use a reduced model with the three maximum conductances \( \bar{g}_{Na} \), \( \bar{g}_K \), and \( \bar{g}_L \) as uncertain parameters, and a complete model where all eleven parameters are uncertain. As in section 4.2, we assume each of these parameters to have a uniform distribution in the range \( \pm 10\% \) around their original value. We use polynomial chaos expansions with the point-collocation method, where the number of evaluations equals the number of collocation nodes.

As error measure we use the average of the absolute relative error over time, which we simply will refer to as the error:

\[
\varepsilon_{X} = \frac{1}{T} \int \left| \frac{X - X_{\text{estimate}}}{X} \right| \, dt,
\]

where “estimate” indicates the results from either the quasi-Monte Carlo method or the polynomial chaos expansions. \( T \) is the total simulation time in the model, disregarding the first 5 ms. \( X \) is either the mean \( E[Y] \), variance \( V[Y] \), or first-order Sobol indices \( S_i \) averaged over all parameters \( i \).

Since an analytical solution for the Hodgkin-Huxley model is not available, we use the quasi-Monte Carlo method with 200,000 model evaluations to calculate the “exact” \( E[Y] \) and \( V[Y] \), and 100000(\( d + 2 \)) (where \( d \) is the number of uncertain parameters) model evaluations to calculate \( S_i \). The quasi-Monte Carlo method is based on random sampling, so we calculate the average error of 50 re-runs for the quasi-Monte Carlo method, to get a more precise result.

The error of the mean, variance, and first-order Sobol indices of the two methods for the two variants of the model are shown in Figure 10. We clearly see that the polynomial chaos expansions are much faster than the quasi-Monte Carlo method for both test cases, that is, much fewer model evaluations \( N_s \) are needed to achieve a certain error.

Figure 10 shows the error for the Hodgkin-Huxley model with three uncertain parameters. In this case, the quasi-Monte Carlo method requires more than 200 times as many model evaluations as the polynomial chaos expansions to calculate the mean with an error of \( \sim 10^{-5} \), and more than 2,500 times as many model evaluations to calculate the Sobol indices with an error of \( \sim 0.5 \).

Figure 10B shows the error for the Hodgkin-Huxley model with eleven uncertain parameters. By comparing with the results for three uncertain parameters, we observe that polynomial chaos expansions scale worse with the number of uncertain parameters than the quasi-Monte Carlo method. However, polynomial chaos expansions are still superior in regards to the required number of model evaluations. For the full Hodgkin-Huxley model, the quasi-Monte Carlo method needs more than ten times as many model evaluations as the polynomial chaos expansions to calculate the mean with an error of \( \sim 2 \cdot 10^{-5} \). For the first-order Sobol indices the quasi-Monte Carlo method gives an error of more than 30 even after 65,000 evaluations. In contrast, the polynomial chaos expansions give an error of 0.26 after only 2,732 model evaluations.

4.6. Additional Examples

Additional examples for uncertainty quantification of the Izikevich neuron (Izhikevich, 2003), a reduced layer 5 pyramidal cell (Bahl et al., 2012), and a Hodgkin-Huxley model with shifted
FIGURE 10 | The error of the mean, variance and (average) first-order Sobol indices for the quasi-Monte Carlo method (QMC) and polynomial chaos expansions (PC) used on the Hodgkin-Huxley model. The average of the absolute relative error over time of the mean (Equation 3), variance (Equation 4), and first-order Sobol indices (Equation 7) averaged over all parameters \( i \) of the Hodgkin-Huxley model with three (A) and eleven (B) uncertain parameters. The mean, variance and first-order Sobol indices are calculated using the quasi-Monte Carlo method with 50 re-runs, and polynomial chaos expansion with point collocation. The “exact” solutions are found using the quasi-Monte Carlo method with \( N_s = 200000 \) model evaluations to calculate the mean and variance, and \( N_s = 100000(d + 2) \) model evaluations where \( d \) is the number of uncertain parameters) to calculate the Sobol indices.

5. DISCUSSION

A major challenge with models in neuroscience is that they tend to contain several uncertain parameters whose values are critical for the model behavior. In this paper we have presented Uncertainpy, a Python toolbox which quantifies how uncertainty in model parameters translates into uncertainty in the model output and how sensitive the model output is to changes in individual model parameters. Uncertainpy is tailored for neuroscience applications by its built-in capability for recognizing features in the model output.

The key aim of Uncertainpy is to make it quick and easy for the user to get started with uncertainty quantification and sensitivity analysis, without any need for detailed prior knowledge of uncertainty analysis. Uncertainpy is applicable to a wide range of different model types, as illustrated in the example applications. These included an uncertainty quantification and sensitivity analysis of four different models: a simple cooling coffee-cup model (section 4.1), the original Hodgkin-Huxley model for generation of action potentials (section 4.2), a multi-compartmental NEURON model of a thalamic interneuron (section 4.3), and a NEST model of a sparsely connected recurrent (Brunel) network of integrate-and-fire neurons (section 4.4). These analyses were mainly performed to illustrate the use of Uncertainpy, but also revealed both expected and unexpected features of the example models. However, we did not put any effort into estimating realistic distributions for the parameter uncertainties. The conclusions should therefore be treated with caution; see result sections for a detailed discussion.

To our knowledge, Uncertainpy is the first toolbox to use polynomial chaos expansions to perform uncertainty quantification and sensitivity analysis in neuroscience. Compared to the (quasi-)Monte Carlo method, polynomial chaos expansions dramatically reduce the number of model evaluations needed to get reliable statistics when the number of uncertain parameters is relatively low, typically smaller than about 20 (Xiu and Hesthaven, 2005; Crestaux et al., 2009; Eck et al., 2016). This was also observed in the present study where we found that polynomial chaos expansions require one to three orders of magnitude fewer model evaluations than the quasi-Monte Carlo method when applied to the Hodgkin-Huxley model with three or eleven uncertain parameters. This gain in efficiency is especially important for models that require a long simulation time, where uncertainty quantification using the (quasi-)Monte Carlo method could require an unfeasible amount of computer time.

5.1. Application of Uncertainpy

Uncertainpy is a computationally efficient Python toolbox that enables uncertainty quantification and sensitivity analysis for computational models. It is tailored toward neuroscience applications by its built-in capability for calculating characteristic features of the model output. While Uncertainpy has a broad applicability, as demonstrated in this paper, certain limitations exist. The first, and perhaps most obvious, is that Uncertainpy does not deal with the problem of obtaining the distributions of the uncertain parameters.

It is also typically not obvious which model is best suited to describe a particular system. For example, when we construct...
a neural model we first have to decide which mechanisms (ion channels, ion pumps, synapses, network connectivity, etc.) to include in the model. Next, we select a set of mathematical equations that describe these mechanisms. Such choices are seldom trivial, and no methods for resolving this structural uncertainty aspect of modeling are included in Uncertainpy. Nevertheless, quantitative measures such as those obtained with Uncertainpy may still give valuable insight in the relationship between model parameters and model output, which can guide experimentalists toward focusing on accurately measuring the parameters most critical for the model output. Additionally, it can guide modelers by identifying mechanisms that can be sacrificed for model reduction purposes.

The accuracy of the quasi-Monte Carlo method and polynomial chaos expansions is problem dependent and is determined by the number of samples, as well as the polynomial order for polynomial chaos expansions. It is therefore a good practice to examine if the results from the uncertainty quantification and sensitivity analysis have converged (Eck et al., 2016). A simple method for checking the convergence is to change the number of samples or polynomial order, or both, and examine the differences between the results. We can be reasonably certain that the results are accurate once these differences are small enough.

5.2. Further Development of Uncertainpy

There are several ways that Uncertainpy can be further developed. If a model or features of a model are irregular, Uncertainpy performs an interpolation of the output to get the results on the regular form needed in the uncertainty quantification and sensitivity analysis. Currently, Uncertainpy only has support for interpolation of one-dimensional output (vectors), but this aspect can be improved.

The screening method available in Uncertainpy is unable to take interactions between parameters into account. More advanced screening methods able to do this exist (Morris, 1991; Campolongo et al., 2007) and could be implemented.

The built-in feature library in Uncertainpy can easily be expanded by adding additional features. The number of built-in simulators (at present NEST and NEURON) can also easily be extended. We encourage the users to add custom features and models through Github pull requests.

5.3. Outlook

In many fields of the physical sciences, the model parameters that go into simulations are known with high accuracy. For example, in quantum mechanical simulations of molecular systems, the masses of the nuclei and electrons, as well as the parameters describing their electrical interaction, are known so precisely that uncertainty in model parameters is not an issue (Marx and Hutter, 2009). This is not the case in computational biology in general, and in computational neuroscience in particular. Model parameters of biological systems often have an inherent variability and some may even be actively regulated and change with time. They can therefore not be precisely known. We thus consider uncertainty quantification and sensitivity analysis to be particularly important in computational biology.

Uncertainpy was developed with the aim of enabling such analysis, that is, to provide an easy-to-use tool for precise evaluation of the effect of uncertain model parameters on model predictions. Being an open-source Python toolbox, we hope that Uncertainpy can be further developed through a joint effort within the neuroscience community.

AUTHOR CONTRIBUTIONS

ST, GH, and GE conceived of and designed the project. ST designed, wrote, tested, and documented the software and performed analysis of the examples. ST, GH, and GE wrote and revised the paper.

FUNDING

This work was funded by the Research Council of Norway (DigiBrain, project no: 248828).

ACKNOWLEDGMENTS

We would like to acknowledge the vital contribution to the vision of this work by Hans-Petter Langtangen, who regretfully passed away before he could see the work completed. We would also like acknowledge the help from Jonathan Feinberg in teaching the basics of polynomial chaos expansions, as well as how to use Chaospy. Additionally, we would like to thank Svenn-Arne Dragly, Milad H. Mobaran, and Andreas Våvång Solbrå for valuable discussions and feedback.

REFERENCES


Paper II

[Re] Fast-Activating Voltage- and Calcium-Dependent Potassium (BK) Conductance Promotes Bursting in Pituitary Cells: A Dynamic Clamp Study
[Re] Fast-Activating Voltage- and Calcium-Dependent Potassium (BK) Conductance Promotes Bursting in Pituitary Cells: A Dynamic Clamp Study

Simen Tennøe1,2, Kjetil Hodne3, Trude M. Haug1, Finn-Arne Weltzien3, Gaute T. Einvell1,5, and Geir Halnes1,5

1 Centre for Integrative Neuroplasticity, University of Oslo, Oslo, Norway 2 Department of Informatics, University of Oslo, Oslo, Norway 3 Department of Basic Sciences and Aquatic Medicine, Norwegian University of Life Sciences, Campus Adamstuen, Norway 4 Institute of Oral Biology, University of Oslo, Oslo, Norway 5 Faculty of Science and Technology, Norwegian University of Life Sciences, Ås, Norway 6 Department of Physics, University of Oslo, Oslo, Norway

simenten@student.matnat.uio.no

A reference implementation of


Introduction

As part of the dynamic clamp study by Tabak et al. 2011 [6], a computational model was developed for the voltage dynamics of endocrine pituitary cells in rats. The model captured the spontaneous activity of these cells, including the generation of Ca2+-channel mediated spikes and pseudo-plateau bursts. As an important achievement, the model explained the paradoxical role that big conductance K+ (BK) channels had in prolonging spike duration and sometimes promoting burst firing in these cells [8], contrary to what one would expect from a hyperpolarizing current. The original model was implemented in XPP [2]. The code for the model was made available online at https://www.math.fsu.edu/~bertram/software/pituitary/JNS_11b.ode, while the code used in the analysis of the model outcome was not made available.

In the current paper, we have reimplemented the computational model by Tabak et al. [6] using the Python interface for the NEURON simulator [4], a widely used simulator for multicompartmental neurons. In addition, we have performed an uncertainty quantification and sensitivity analysis of the model using the Uncertainty Python package [7], version 1.1.4 (Zenodo: 10.5281/zenodo.1473453). The model implementation works with Python 2 and 3. The results in this paper were created using Python 3.7.0 within a Docker (https://www.docker.com/) environment.

The reimplemented model reproduced the characteristic firing patterns seen in the original publication, and we thus confirmed the original study. The sensitivity analysis further presented a systematic overview of the model in terms of how its characteristic response features depended on the various model parameters. Supporting the main conclusion from the original work, the sensitivity analysis showed that the bursting...
propensity of the model was highly sensitive to the BK conductance. However, the analysis also revealed that the bursting propensity was sensitive to additional parameters (conductances), and thus that BK is not the sole determinant for whether the cell is bursty.

Methods

When reimplementing the model by Tabak et al. [6] we followed the descriptions in the original publication, using the original implementation for verification purposes. We also had a brief communication with the original authors to obtain details on the analysis part of the model.

Model

The model by Tabak et al. [6] was defined by the equation:

\[
C \frac{dV}{dt} = -(I_{\text{Ca}} + I_K + I_{\text{BK}} + I_{\text{SK}} + I_{\text{leak}} + I_{\text{noise}}),
\]

where \(C\) is the membrane capacitance, \(V\) is the membrane potential, and \(I_X\) the current through a specific ion channel \(X\). The model included six different currents:

- \(I_{\text{Ca}}\) – Voltage gated Ca\(^{2+}\) current.
- \(I_K\) – Voltage gated K\(^+\) current.
- \(I_{\text{BK}}\) – Big conductance K\(^+\) current.
- \(I_{\text{SK}}\) – Small conductance K\(^+\) current.
- \(I_{\text{leak}}\) – Leak current.
- \(I_{\text{noise}}\) – Stochastic current representing channel noise.

A current through an ion channel \(X\) was given by the simplified relation:

\[
I_X = G_X Y_X (V - E_X),
\]

where \(G_X\) denotes the maximum ion channel conductance, and \(E_X\) denotes the reversal potential of the ion species conducted by channel \(X\). \(Y_X\) denotes an ion channel specific gating function, which was unity for \(I_{\text{leak}}\), an instantaneous function of \(V\) for \(I_{\text{Ca}}\), an instantaneous function of the cytosolic Ca\(^{2+}\) concentration for \(I_{\text{SK}}\), and a dynamic function of \(V\) and \(I\) for the remaining ion channels \(I_K\), \(I_{\text{BK}}\), and \(I_K\).

The original implementation used the total membrane capacitance (units \(\text{pF}\)) and total membrane conductances (units \(\text{nS}\)), while the NEURON simulator requires these entities to be specified per membrane area with units \(\mu\text{F/cm}^2\) and \(\text{S/cm}^2\), respectively. NEURON also requires that the membrane area is defined. To get the parameters on the form required by NEURON we defined an arbitrary membrane area \((A)\), and divided the capacitance \(C\) and ion channel conductances \(G_X\) by \(A\):

\[
g_{X, \text{NEURON}} = \frac{G_X}{A}, \quad C_{\text{NEURON}} = \frac{C}{A}.
\]

Combining Equation 1, 2 and 3 shows that the model is independent of the choice of \(A\):

\[
\frac{C}{A} \frac{dV}{dt} = - \left( \frac{G_X}{A} Y_X (V - E_X) + \ldots \right).
\]

The original model further included an equation for handling the intracellular Ca\(^{2+}\) concentration, which is relevant for the gating of SK channels:
\[
\frac{d[Ca]}{dt} = -f_c(\alpha [Ca]_e + k_c[Ca]),
\]

where \( f_c \) denotes the fraction of free Ca\(^{2+} \) in the cytoplasm, \( k_c \) denotes the extrusion rate, and the constant \( \alpha \) converts an incoming current to a molar concentration. \( \alpha \) was converted to NEURON units by taking:

\[
\alpha_{\text{NEURON}} = A\alpha. \tag{6}
\]

Combining Equation 3, 5 and 6 shows that this choice keeps the model independent of the choice of \( A \):

\[
\frac{d[Ca]}{dt} = -f_c(\alpha G_{Ca} A (V - E_{Ca}) + k_c[Ca]). \tag{7}
\]

We arbitrarily chose a cell body with a membrane area of \( \pi \cdot 10^{-6} \text{ cm}^2 \), i.e. with a diameter of 10 \( \mu \text{m} \). We used all equations from the original publication, substituting \( G_K \) with \( g_K,_{\text{NEURON}}, C \) with \( C_{\text{NEURON}}, \) and \( \alpha \) with \( \alpha_{\text{NEURON}} \). The parameter values from the original publication and the converted parameter values are summarized in Table 1. Parameters not listed in this table were kept unchanged from the original publication. To make the discussion and results easier to compare to the original publication, we will refer to the original conductance values through the rest of this paper.

The noise was added by using a current clamp that injected a random current at each time step in the simulation, as described by the original publication. Simulations with noise were run with a fixed time step of \( \text{dt} = 0.01 \text{ ms} \), which is the same time step used in the original publication. When performing the sensitivity analysis, the noise amplitude was set to zero (\( A_{\text{noise}} = 0 \)), and the simulations were run using adaptive time steps.

We found one discrepancy between the parameters listed in the original publication and the values found in the original source code. The maximum conductance of K\(^+ \) channels (\( G_K \)) was listed as 3.2 nS in the original publication, while the value used in the original source code was 3 nS. Both values were tested and \( G_K = 3 \) nS gave results most similar to the results in the original publication. We therefore decided to use \( G_K = 3 \) nS instead of the value listed in the original publication.

**Table 1**: The parameter values in Tabak et al. [6] that were converted from currents and capacitance to currents and capacitance per membrane area due to requirements by the NEURON simulator. The original model parameter values are denoted Tabak while the parameter values in the reimplemented model are denoted NEURON, with names as used in the model implementation.

<table>
<thead>
<tr>
<th>Tabak</th>
<th>Value</th>
<th>Unit</th>
<th>NEURON</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>10</td>
<td>pF</td>
<td></td>
<td>( h )</td>
<td>( 3.14 \cdot 10^{-6} \text{ cm}^2 )</td>
</tr>
<tr>
<td>( G_{Ca} )</td>
<td>2</td>
<td>nS</td>
<td>( g_{Ca} )</td>
<td>( 6.37 \cdot 10^{-4} \text{ S/cm}^2 )</td>
<td></td>
</tr>
<tr>
<td>( G_K )</td>
<td>3</td>
<td>nS</td>
<td>( g_K )</td>
<td>( 9.55 \cdot 10^{-4} \text{ S/cm}^2 )</td>
<td></td>
</tr>
<tr>
<td>( G_{BK} )</td>
<td>2</td>
<td>nS</td>
<td>( g_{BK} )</td>
<td>0</td>
<td>S/cm(^2 )</td>
</tr>
<tr>
<td>( G_{SK} )</td>
<td>3</td>
<td>nS</td>
<td>( g_{SK} )</td>
<td>( 6.37 \cdot 10^{-4} \text{ S/cm}^2 )</td>
<td></td>
</tr>
<tr>
<td>( G_{i} )</td>
<td>0.2</td>
<td>nS</td>
<td>( g_{i} )</td>
<td>( 6.37 \cdot 10^{-5} \text{ S/cm}^2 )</td>
<td></td>
</tr>
<tr>
<td>( \alpha )</td>
<td>0.0015</td>
<td>( \mu \text{M} / \text{FC}^2 )</td>
<td>( \alpha \text{pha} )</td>
<td>( 4.71 \cdot 10^{-3} \text{ mM} \cdot \text{cm}^2 / \mu \text{C} )</td>
<td></td>
</tr>
<tr>
<td>( A_{\text{noise}} )</td>
<td>4</td>
<td>pA</td>
<td>( \text{noise}_\text{amplitude} )</td>
<td>0.004</td>
<td>nA</td>
</tr>
</tbody>
</table>
Event detection

In the analysis, we ran the model for 60000 ms and discarded the first 10000 ms of the voltage trace to eliminate the transient initial response.

The first step of the model analysis was to detect events (spikes or bursts) in the model voltage trace. To do this, the voltage was normalized so that its minimum value was set to 0 and its maximum was set to 1. The start of an event was specified to be when the voltage crossed an onset threshold (defined to be 0.55), and the end of an event to be when it next descended below another, lower termination threshold (defined to be 0.45). An event includes the first point before it crossed the onset threshold and the first point after it descended below the termination threshold.

The difference in onset and termination threshold was necessary to prevent random fluctuations around the threshold (during upstroke or downstroke) to be considered as independent events. If the voltage trace started above the onset threshold, we discarded the first part of the voltage trace until we got below the termination threshold. Similarly, if an event did not fall below the termination threshold before the simulation ended, that event was discarded. Additionally, we required that events have an amplitude of at least 10 mV. This prevents the problem where the normalization step leads to detecting false events with an amplitude less than 1 mV in cases where the model does not generate any events and instead exhibits small (much less than 1 mV) fluctuations around a steady state.

We used Uncertainty to detect events, as the described threshold-detection algorithm is available to us by using the `uncertainty.Spikes` object with the arguments `normalize=True, trim=False` and `min_amplitude = 10`. Note that in Uncertainty, the end_threshold is given relative to the onset threshold, so to get a termination threshold = 0.45 we set end_threshold = -0.1.

An event was defined as a burst when its duration was longer than a given threshold (60 ms). The burstiness factor was defined as the fraction of the total number of events that were considered as bursts. All parameters used in the analysis are summarized in Table 2.

The description of the threshold-detection algorithm for detecting events (bursts or spikes) was incomplete in the original publication. We contacted the original authors, who were helpful in describing the threshold-detection algorithm, but who did not recall the exact numerical values of all threshold choices. The onset threshold, termination threshold and burst-duration threshold used (Table 2) were therefore set to the values we found to give the best agreement between our analysis outcome and that in the original publication.

Table 2: The parameters used in the analysis of the model.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation time</td>
<td>60000</td>
<td>ms</td>
</tr>
<tr>
<td>Discard</td>
<td>10000</td>
<td>ms</td>
</tr>
<tr>
<td>Event onset threshold</td>
<td>0.55</td>
<td></td>
</tr>
<tr>
<td>Event termination threshold</td>
<td>0.45</td>
<td></td>
</tr>
<tr>
<td>Burst threshold</td>
<td>60</td>
<td>ms</td>
</tr>
<tr>
<td>Minimum event amplitude</td>
<td>10</td>
<td>mV</td>
</tr>
</tbody>
</table>

Uncertainty quantification and sensitivity analysis

We used Uncertainty to further examine the model through an uncertainty quantification and sensitivity analysis. This enabled us to quantify how sensitive salient response properties of the model is to changes in the various parameters. In the sensitivity analysis, the four conductances $G_{Ca}$, $G_K$, $G_{SK}$, and $G_I$ where assigned uniform
distributions within ±50% of their original values. $G_{BK}$, which had no default value in the original model, was given a uniform distribution between 0 and 1 nS as this was the parameter range explored in the original study. We use polynomial chaos with the point collocation method (the default of Uncertainpy) and a polynomial order of eight. In the sensitivity analysis, we wanted all the variance in the simulation outcome to reflect parameter variations, and the random noise was therefore turned off by setting $A_{\text{noise}} = 0$.

We calculated the uncertainty and sensitivity of the five features of the model:

- Event rate, which is the event firing rate (named \texttt{spike\_rate} in Uncertainpy).
- Average event peak, which is the average event peak voltage (named \texttt{average\_AP\_overshoot} in Uncertainpy).
- Average AHP (afterhyperpolarization) depth, which is the average minimum voltage between events.
- Burstiness factor, the fraction of events with a duration longer than 60 ms.
- Average duration, the average duration of an event.

Some of these features are not defined for all parameter combinations (for example average AHP depth is not defined when there are no events). The point collocation method still gives reliable results, as long as the features are defined for a sufficiently large fraction of the parameter combinations (in our case the lowest was -91.5%) [1].

Some of the outcomes from the sensitivity analysis were unexpected and were explored further by varying selected parameters and documenting how these variations affected the average event duration and burstiness factor of the model. The parameters varied in this additional analysis were $G_{BK}$, $G_{SK}$, and $G_{K}$, all of which were varied within the range used in the uncertainty analysis.

\section*{Results}

We repeated all simulations and qualitatively reproduced Figure 1 and Figure 2 in Tabak et al. [6]. The remaining results in the original publication were experimental results, and therefore outside the scope of this reproduction.

The results shown in Figure 1 corresponds well to those in Figure 1 of the original publication. The original model and the reproduced version showed the same behavior when increasing $G_{BK}$. As random noise was added to the simulations, an exact replication could not be expected.

The results shown in Figure 2 corresponds well to those in Figure 2 of the original publication. The behavior we observe is similar to the behavior in the original publication. When increasing $G_{BK}$ the model went from having a low burstiness factor (between 0 and 0.1) to having a high burstiness factor (between 0.9 and 1). For any value of $G_{BK}$, the model evaluations tended to be either predominantly bursting (i.e. most events were bursts) or predominantly spiking (few events were bursts), so that the number of model evaluations with an intermediate burstiness factor between 0.1 and 0.9 (events changed between being bursts or not) was always low (less than 20 evaluations).

A small deviation was found between the current analysis and the original work. For the maximum value of $G_{BK}$, we got fewer model evaluations with low burstiness than in the original work. We do not know the precise cause of this difference, though we can speculate that it may be due to smaller differences in the performed analysis (e.g., onset and termination threshold definitions or implementations), or due to underlying differences between the NEURON and XPP implementations (e.g., NEURON uses backward Euler as the numerical integration scheme while the XPP implementation uses forward Euler).
Figure 1: Model predictions for the effect of various $G_{BK}$ conductances on burstiness. A-C Left, membrane potential of the model. Right, distribution of event durations in the time interval from 1 to 5 s (of the 50 s simulated). The grey line indicates the threshold for what is considered a spike and what is considered a burst, and BF denotes the burstiness factor. D The burstiness factor increased with $G_{BK}$. E The burstiness factor decreased with $\tau_{BK}$. 
Figure 2: Robustness of the burstiness of the model for three values of $G_{BK}$ when changing $G_{Ca}$, $G_{K}$, $G_{SK}$, and $G_{j}$ uniformly within $\pm 50\%$ of their original values. A For $G_{BK} \rightarrow 0$ nS, 67.5\% of the active models were spikers (burstiness factor $< 0.3$). B For $G_{BK} \rightarrow 0.5$ nS, 33.8\% were spikers. C For $G_{BK} \rightarrow 1$ nS, only 4.4\% were spikers.
Figure 3: Uncertainty quantification and sensitivity analysis of a selected set of response features of the model. A Event rate denotes the event firing rate. B Average event peak denotes is the average event peak voltage. C Average AHP (afterhyperpolarization) depth denotes the average minimum voltage between two consecutive events. D Burstiness factor denotes the fraction of events with duration longer than 60 ms. E Average duration denotes the average duration of the events.

Uncertainty quantification and sensitivity analysis

The uncertainty quantification and sensitivity analysis of the model is shown in Figure 3. The sensitivity was given as the total-order Sobol indices, which quantify how much of the variance of the model each parameter (accounting for all of its interactions with other parameters) is responsible for [5].

The sensitivity analysis showed that the spike rate was sensitive to almost all ion channel conductances, but most so to $G_K$ (Figure 3A). Such a role of the delayed rectifying $K^+$ channel in controlling the firing rate has been seen in other studies [3].

The event amplitude was mainly sensitive to $G_{Ca}$ (Figure 3B), which is not surprising given that the events are generated by $I_{Ca}$. However, it also had a relatively high sensitivity to $G_{BK}$, in line with what was found in the previous study [6].

The average afterhyperpolarization depth was in turn most sensitive to $G_{Ca}$ (Figure 3C). This may seem counterintuitive, as $I_{Ca}$ is not a hyperpolarizing current. However, $I_{Ca}$ is responsible for triggering all the three hyperpolarizing currents ($I_K$, $I_{BK}$ and $I_{SK}$) that generate the afterhyperpolarization depth. $I_K$ and $I_{BK}$ are activated by the voltage deflection caused by $I_{Ca}$, while $I_{SK}$ is activated by the $Ca^{2+}$ entering through $I_{Ca}$.

The burstiness factor of the model was mainly sensitive to $G_K$ and $G_{BK}$ (Figure 3D). The sensitivity to $G_{BK}$ confirms the findings in the original publication, i.e. that BK channels promote bursting. However, the large sensitivity to $G_K$ is a novel insight for the current study and indicates that also $G_K$ was important for determining if the model produced bursts or spikes. This observation is tightly related to the explanation for how BK can act as a burst promoter in the first place, which is contrary to what
one would expect from a hyperpolarizing current. The explanation, proposed by both Tabak et al. [6] and the experimental studies they were inspired by [8], was that $G_{BK}$ promoted bursting by reducing the peak amplitude of events (as reflected in Figure 3B), thereby preventing full activation of the otherwise more strongly hyperpolarizing delayed rectifier current ($I_K$). In this context, the sensitivity analysis simply shows that the indirect effect on $I_K$ obtained by varying $G_{BK}$ was smaller than the direct effect on $I_K$ obtained by varying $G_K$ (Figure 3D).

Surprisingly, the average event duration had a very low sensitivity to $G_{BK}$ (Figure 3E), and was instead most sensitive to $G_{SK}$. This was unexpected since the burstiness was highly sensitive to $G_{BK}$, and a burst was defined as an event exceeding a certain duration. An exploration of the counterintuitive relationship between Figure 3D and E is presented below.

**Parameter exploration**

To explore the relationship between the results in Figure 3D and E, we examined the effects of varying $G_{BK}$, $G_K$, and $G_{SK}$ on the burstiness and the average duration of events (Figure 4). It should be noted that this figure only shows how the model responds when changing two parameters at the time, so the higher-order interactions included in the total-order Sobol sensitivity indices are absent.

Figure 4A shows the regions in the $G_{BK}/G_K$ parameter plane where the model produced regular spikes (yellow) and bursts (green). For low ($< 2$ nS) values of $G_K$, the cell was bursting regardless of the value of $G_{BK}$. Hence, for low values of $G_K$, the burstiness of the cell was insensitive to $G_{BK}$. In comparison, a sufficiently large change in $G_K$ could switch the cell between a regular and bursty state for any (fixed) value of $G_{BK}$. These results thus fit well with the sensitivity analysis in Figure 3D, which showed that the burstiness was more sensitive to $G_K$ than to $G_{BK}$.

We next fixed $G_K$ at the default value 3 nS, and explored how it could be the case that burstiness was sensitive to $G_{BK}$ but not so much to $G_{SK}$, while event duration was sensitive to $G_{SK}$ but not so much to $G_{BK}$ (Figure 4B). As the figure shows, for $G_{BK} < 0.2$ nS the cell was always regularly spiking, while for $G_{BK} > 0.8$ nS, the cell was always bursting, regardless of the values of $G_{SK}$. In comparison, changing $G_{BK}$ (keeping $G_{SK}$ fixed) could always switch the cell between a regular and bursty state. In equivalence with the analysis of Figure 4A, this explains why the burstiness was less sensitive to $G_{SK}$ than $G_{BK}$. However, although changes in $G_{BK}$ more often led to changes in burstiness, the effects on the event duration was modest. That is, for most (fixed) values of $G_{SK} >$, a change in $G_{BK}$ could push the event duration from slightly below to slightly above the burst-duration threshold, but did not lead to larger changes in burstiness. Oppositely, reducing $G_{SK}$ to the lower values in the explored range resulted in burst durations of several thousands of milliseconds (as long as $G_{BK} > 0.2$ nS). Hence, while $G_{BK}$ was important for achieving a burst in the first place, $G_{SK}$ had a much larger impact on the duration of the burst. This explains the difference in sensitivity between the average duration and burstiness factor observed in Figure 3D and E.

**Conclusion**

We were able to qualitatively reproduce all the computational results in Tabak et al. [6]. By performing an uncertainty quantification and sensitivity analysis we confirmed the key conclusions in the original publication using a different simulator and different analysis methods, which provided additional insight into how different membrane mechanisms interact to produce the characteristic response features of the model. Overall, the reproduction effort went smooth, with a little help from the original authors in describing the threshold-detection algorithm used in the analysis of the model.
Figure 4: The average duration of events while varying $G_{NK}$ and either A $G_K$ or B $G_{SK}$. The areas in parameter space where the average duration of the events is longer than the burstiness factor threshold are in green, while the areas where the average duration is below this threshold are in yellow. Areas in blue produce no events and the average duration is then set to -1 for visualization purposes.
The original model now exists as a model using the Python interface for NEURON, which hopefully makes it accessible to a wider audience.

References


Paper III

BK channels have opposite effects on sodium versus calcium mediated action potentials in endocrine pituitary cells
BK channels have opposite effects on sodium versus calcium-mediated action potentials in endocrine pituitary cells.

Geir Halnes¹,², *, Simen Tennøe ²,³, Trude M. Haug ⁴, Gaute T. Einevoll¹,⁵, Finn-Arne Weltzien⁶, Kjetil Hodne⁶.

¹ Faculty for Science and Technology, Norwegian University of Life Sciences, Ås, Norway
² Centre for Integrative Neuroplasticity, University of Oslo, Oslo, Norway
³ Department of Informatics, University of Oslo, Oslo, Norway,
⁴ Institute of Oral Biology, University of Oslo, Oslo, Norway
⁵ Department of Physics, University of Oslo, Oslo, Norway
⁶ Department of Basic Sciences and Aquatic Medicine, Norwegian University of Life Sciences, Campus Adamstuen, Norway

* Corresponding author - geir.halnes@nmbu.no.

Abstract

Pituitary endocrine cells fire action potentials (APs) to regulate their cytosolic Ca²⁺ concentration and hormone secretion rate. Depending on animal species, cell type, and biological conditions, pituitary APs are generated either by TTX-sensitive Na⁺ currents ($I_{Na}$), high-voltage activated Ca²⁺ currents ($I_{Ca}$), or by a combination of the two. Previous computational models of pituitary cells have mainly been based on data from rats, where $I_{Na}$ is largely inactivated at the resting potential, and spontaneous APs are exclusively mediated by $I_{Ca}$. As a part of the previous modeling studies, a paradoxical role was identified for the big conductance K⁺ current ($I_{BK}$), which was found to prolong the duration of $I_{Ca}$-mediated APs, and sometimes give rise to pseudo-plateau bursts, contrary to what one would expect from a hyperpolarizing current. Unlike in rats, spontaneous $I_{Na}$-mediated APs are consistently seen in pituitary cells of several other animal species, including several species of fish. In the current work we develop the, to our knowledge, first computational model of a pituitary cell that fires $I_{Na}$-mediated APs. Although we constrain the model to experimental data from gonadotrope cells in the teleost fish medaka (Oryzias latipes), it may likely provide insights also into other pituitary cell types that fire $I_{Na}$-mediated APs. In the current work, we use the model to explore how the effect of $I_{BK}$ depends on the AP generating mechanisms of pituitary cells. We do this by comparing simulations on the medaka gonadotrope model (two versions thereof) with simulations on a previously developed model of a rat pituitary cell. Interestingly, we find that $I_{BK}$ has the opposite effect on APs in the two models, i.e. it reduces the duration of already fast $I_{Na}$-mediated APs in the medaka model, and prolongs the duration of already slow $I_{Ca}$-mediated APs in the rat model.

Author summary

Excitable cells elicit electrical pulses called action potentials (APs), which are generated and shaped by a combination of ion channels in the cell membrane. While neurons use APs for interneuronal communication and heart cells use them to generate heart-beats, pituitary cells use APs to regulate their cytosolic Ca²⁺ concentration, which in turn
controls their hormone secretion rate. The amount of Ca\(^{2+}\) that enters the pituitary cell during an AP depends strongly on how long it lasts, and it is therefore important to understand the mechanisms that control this. Depending on animal species and biological conditions, pituitary APs may be initiated either by Ca\(^{2+}\) channels or Na\(^{+}\) channels. Here, we explore the differences between the two scenarios by comparing simulations on two different computer models: (i) a previously developed model which fires Na\(^{+}\)-based APs, adapted to data from pituitary cells in rats, and (ii) a novel model that fires Ca\(^{2+}\)-based APs, adapted to data from pituitary cells in the fish medaka. Interestingly, we find that the role of big conductance K\(^{+}\) (BK) channels, which are known to affect the duration of the AP, are opposite in the two models, i.e., they act to prolong Ca\(^{2+}\)-based APs while they act to shorten Na\(^{+}\)-based APs.

1 Introduction

The electrodynamics of excitable cells is generated by a combination of ion channels in the plasma membrane, which are typically characterized by their voltage and/or Ca\(^{2+}\) dependence. While neurons primarily use action potentials as a means of interneuronal communication, and cardiac cells use them to generate heartbeats, the primary role of APs in endocrine pituitary cells is to regulate the cytosolic Ca\(^{2+}\) concentration, which in turn controls the hormone secretion rate in these cells [1]. Hormone secretion often occurs as a response to hormonal stimuli from the hypothalamus, peripheral endocrine glands, and other types of pituitary cells. However, many endocrine cells are also spontaneously active [1–10]. The spontaneous activity is partly a means to regulate the re-filling of intracellular Ca\(^{2+}\) stores, but in several cells also leads to a basal release of hormones. An understanding of the mechanisms regulating the electrodynamics of these cells is therefore fundamental for understanding their overall functioning.

While neuronal APs are are predominantly mediated by TTX-sensitive Na\(^{+}\) currents (\(I_{Na}\)), AP generation in endocrine cells depends strongly on high-voltage-activated Ca\(^{2+}\) currents (\(I_{Ca}\)), which in addition to their role in affecting the voltage dynamics of the cell, also are the main source of Ca\(^{2+}\) entry through the plasma membrane [3,11,12]. In some studies of endocrine cells, APs were exclusively mediated by \(I_{Ca}\), and the spontaneous membrane excitability was insensitive or nearly so to TTX [1,2,13–16]. In other studies, APs were evoked by a combination of \(I_{Ca}\) and Na\(^{+}\) currents [4,7,17,18]. The strong involvement of \(I_{Ca}\) could explain why pituitary APs typically last longer (typically some tens of milliseconds [8]) than neuronal APs (a few milliseconds), which are mainly mediated by \(I_{Na}\).

All endocrine cells express \(I_{Na}\) [8], and TTX sensitive APs can typically be triggered by current injections from hyperpolarized holding potentials even in cells where they are not elicited spontaneously [4,17,19,20]. The reason why the spontaneous activity is TTX insensitive is likely that a major fraction of \(I_{Na}\) is inactivated at the resting membrane potential [15,16]. The reason why this is not always the case, may be that the resting potentials vary greatly between different studies. Only for rat somatotropes, resting potentials ranging as wide as from −30 mV [13] to −80 mV [18] have been reported. In cells with hyperpolarized resting potentials (−80 mV), TTX was found to block single, brief action potentials, while action potentials of long duration and low amplitude persisted [18], indicating that both a \(I_{Ca}\) and \(I_{Na}\) component were present and that the two had different time-courses. However, the more typical resting potentials for rat pituitary cells lie in the range between −50 mV and −60 mV, and at these resting levels, \(I_{Na}\) tends to be inactivated and the spontaneous activity TTX insensitive (see reviews in [8,21]).

Computational models constructed to capture the essential activity of pituitary cells have predominantly relied on rat (or mouse) data, and have not included...
I\textsubscript{Na} [3,9,22–27]. When I\textsubscript{Na} was included in a recent modeling study, its role was mainly in modulating the firing patterns but was not essential for AP firing as such [28]. The main focus of these models was thus on exploring the interplay between the depolarizing I\textsubscript{Ca} and various K\textsuperscript{+} currents responsible for shaping the repolarization following an AP. The essentials of this interplay were nicely captured in a relatively simple computational model by Tabak et al. [9]. In this model, the AP upstroke was exclusively mediated by I\textsubscript{Ca}, while the repolarizing phase was mediated by three K\textsuperscript{+} currents \((I\textsubscript{K}, I\textsubscript{BK} \text{ and } I\textsubscript{SK})\). The model elicited spontaneous APs with a duration of about 40 ms, which is quite representative for what is seen experimentally in rat pituitary cells. In addition, the model was able to explain the important and paradoxical role that the big conductance Ca\textsuperscript{2+}-activated K\textsuperscript{+} current \((I\textsubscript{BK})\) has in rat endocrine cells, where it was found to make APs broader. The explanation to this broadening effect, which is the opposite of what one would expect from a hyperpolarizing current [23], was that \(I\textsubscript{BK}\) reduces the AP amplitude, and thereby prevents the (high-threshold) activation of the delayed rectifying K\textsuperscript{+} current \((I\textsubscript{K})\), which otherwise would become a more powerful hyperpolarizing current [9,23]. By inhibiting \(I\textsubscript{K}\), \(I\textsubscript{BK}\) indirectly becomes facilitating, and leads to broader APs and sometimes to voltage plateaus and so-called pseudo plateau bursts [8,23], which are believed to be more efficient than regular APs in evoking hormone secretion (see e.g. [29]). By manipulating the BK expression experimentally and in computational models, it was further shown that the difference between bursty endocrine cell types such as somatotropes and lactotropes, and regularly spiking cell types such as gonadotropes and corticotrophs, could be explained by the different cell types having different levels of \(I\textsubscript{BK}\) expression [9,28].

The modeling studies cited above were predominantly based on data from rats, and there are reasons to believe that teleost pituitary cells are different in terms of their dynamical properties. Firstly, TTX-sensitive spontaneous activity has been seen in goldfish resting at \(-60\) mV [4], and TTX-sensitive APs has been evoked from a holding potential as high as \(-50\) mV in pituitary cells in cod [7], suggesting that I\textsubscript{Na} may be more available in resting pituitary cells in fish [4]. Secondly, data from goldfish [4] and tilapia [5] indicate that the AP duration is shorter in fish (<10 ms) compared to rats (several tens of ms), and also this could indicate a stronger involvement of I\textsubscript{Na}. A third difference between fish and rat pituitary cells is in the role of I\textsubscript{BK}. I\textsubscript{BK} is almost absent in rat gonadotropes [23], and this was proposed as an explanation to why these cells tend to be less bursty than other pituitary cell types [1,9,28]. In contrast, I\textsubscript{BK} is highly expressed in medaka gonadotropes, but without making these cells bursty [12]. The indication that there are differences between rat and fish pituitary cells are supported by experiments presented in the current work, performed on gonadotrope cells in medaka. We show that these cells elicit brief spontaneous APs that (unlike spontaneous APs in rats) to a large degree are mediated by TTX sensitive Na\textsuperscript{+} currents \((I\textsubscript{Na})\). Furthermore, we show that I\textsubscript{BK} acts to make APs narrower in medaka gonadotropes, and thus have the opposite effect of what they do in rat pituitary cells. Since previous computational models based on rat data seem unsuited to describe the spontaneous activity of fish pituitary cells, we here present novel pituitary cell models constrained to data from luteinizing hormone-producing medaka gonadotropes. The main, and more general, aim of this study is to use computational modeling to explore the differences between pituitary cells that fire APs exclusively mediated by I\textsubscript{Ca} (like the previous rat models) and pituitary cells that fire APs that are predominantly mediated by I\textsubscript{Na} (like the here developed medaka models), with a special focus on the role that I\textsubscript{BK} has in the two cases. To do so, we compare simulations run on three different computational models:

1. RAT. The first model, which we refer to as RAT, is a reproduced version of the model by Tabak et al. [9].
2. MEDAKA 1. In the second model, which we refer to as MEDAKA 1, $I_{Ca}$ from RAT is replaced with a pair of depolarizing currents $I_{Na}$ and $I_{Ca}$ with kinetics constrained to voltage-clamp data from gonadotrope cells in medaka. In MEDAKA 1, all other currents were kept identical to that in RAT, as this allowed us to make a direct comparison between two models where only the AP generating mechanisms were different.

3. MEDAKA 2. In the third model, which we refer to as MEDAKA 2, $I_{Na}$ and $I_{Ca}$ were kept as in MEDAKA 1, but the remaining set of ionic currents were adjusted in order to obtain a model that better replicates the essential response features of gonadotrope cells in medaka.

By simulations, we show that MEDAKA 1 and MEDAKA 2 produce spontaneous APs that are faster than those in RAT, and thus more suited to describe the firing properties of fish pituitary cells. We further show that $I_{BK}$ has the opposite effect in the medaka models from what it has in RAT, and suggest that $I_{BK}$ acts as a mechanisms that makes slow ($I_{Ca}$-generated) pituitary APs broader, and fast ($I_{Na}$-generated) pituitary APs briefer. To our knowledge, MEDAKA 1 and MEDAKA 2 are the first computational models that describe $I_{Na}$-based APs in endocrine cells. Although the models were tailored to represent gonadotrope cells in medaka, we believe that they are of a more general value for improving our understanding of $I_{Na}$-based APs in the pituitary, which are elicited by several endocrine cell types and in several animal species, depending on biological conditions [4, 7, 17, 18, 30–32].

Results

Characteristic response patterns of medaka gonadotropes

The general electrophysiological properties of gonadotrope cells in medaka were assessed through a series of voltage clamp and current clamp experiments. Voltage clamp experiments used to develop kinetics models of $Na^+$ and $Ca^{2+}$ currents in MEDAKA 1 and MEDAKA 2 are presented in the Methods section. Here, we focus on the key properties of spontaneous APs as recorded by current clamp. Selected, representative experiments are shown in Fig 1.

Although variations were observed, the medaka gonadotropes typically had a resting potential around $-50 \text{ mV}$, which is within the range found previously for goldfish [4]and cod [10] gonadotropes. As for goldfish gonadotropes, the majority of medaka gonadotropes fired spontaneous APs, and with peak voltages slightly below 0 mV. The spontaneous APs were always regular spikes (i.e., not bursts) and typically had an average duration between 3 and 7 ms (blue traces in Fig 1). Similar brief AP waveforms have been seen in previous studies on fish [4, 5, 19], while the APs reported for rat gonadotrope cells are typically slower, i.e. from 10-100 ms [8]. The spontaneous AP activity was completely abolished by TTX application (Fig 1B), as has previously also been seen for goldfish somatotropes [19].

Finally, we explored how paxilline (an $I_{BK}$ blocker) affected the spontaneous activity of gonadotrope medaka cells. In the experiment shown in Fig 1D, paxilline increased the firing rate and slightly reduced the mean AP peak amplitude, but these effects were not seen consistently in experiments using paxilline application. However, in all experiments, paxilline application was followed by a small increase of the resting membrane potential (Fig 1C), and a broadening of the AP waveform (Fig 1D2-D3). Similar effects have been seen in goldfish somatotropes, where application of tetraethylammonium (a general blocker of $Ca^{2+}$ gated K+ currents) lead to broadening of APs [19]. The effect of $I_{BK}$ in goldfish and medaka gonadotropes is thus to make
Fig 1. Experimental voltage recordings. (A1-B1) Spontaneous AP firing in two selected cells. (A2-A3) Close-ups of selected APs from the cell in A1. (B2-B3) Close-ups of selected APs from the cell in A2. (C1) Spontaneous activity before (blue) and after (red) TTX application. (C2-C3) Close-ups of two selected events before (blue) and after (red) TTX application. (D1) Spontaneous activity before (blue) and after (red) paxilline application. (D2-D3) Close-ups of two selected events before (blue) and after (red) paxilline application. The firing rates in the various recordings were 0.64 Hz (A1), 0.57 Hz (B1), 1.22 Hz (C1, before TTX), 0.17 Hz (D1, before paxilline) and about 0.35 Hz (D1, after paxilline). AP durations varied between 3 and 7 ms, with mean durations of 3.7 ms (A1), 4.9 ms (B1), 3.7 ms (C1, before TTX). In (D1), average AP durations were 4.2 ms before paxilline, and 25 ms after paxilline. Mean AP peak values were $-0.4 \text{ mV}$ (A1), $-3.4 \text{ mV}$ (B1), $-5.1 \text{ mV}$ (C1, before TTX), $-3.1 \text{ mV}$ (D1, before paxilline), and $-6.4 \text{ mV}$ (D1, after paxilline). AP width was calculated at half max amplitude between $-50 \text{ mV}$ and AP peak. The experiments were performed on gonadotrope luteinizing hormone-producing cells in medaka. All depicted traces were corrected with a liquid junction potential of $-9 \text{ mV}$. The time indicated below each panel refers to the duration of the entire trace shown.

APs narrower, which is the opposite of what was found in rat pituitary cells, where $I_{BK}$ lead to broader APs and sometimes to burst-like activity [9,23].

Computational models of pituitary cells
In this work, we have compared simulations performed on three different pituitary cell models. The first model, RAT, is a replication of the previously published model by Tabak et al. [9]. This model was originally adapted to electrophysiological data from rat pituitary cells. As summarized by Eq 1, RAT contains a leakage current $I_{leak}$, three $K^+$...
currents $I_K$, $I_{BK}$ and $I_{SK}$, and a Ca$^{2+}$ current $I_{Ca}$. In addition, a gaussian noise stimulus was added in selected simulations (see Methods).

$$C_m \frac{dV}{dt} = -(I_{Ca'} + I_K + I_{BK} + I_{SK} + I_{leak} + I_{noise}).$$  \hfill (1)

The two other models are novel for this work and are the to date first pituitary cell models based on teleost data, and the first to elicit APs that are predominantly mediated by Na$^+$ currents. In these models, the Ca$^{2+}$ current $I_{Ca}$ from RAT was replaced by a pair of depolarizing currents, i.e., a novel Ca$^{2+}$ current ($I_{Ca}^{m}$) and a Na$^+$ current ($I_{Na}$), which were adapted to voltage clamp data from gonadotrope cells in medaka (see Methods). We present two versions of the medaka model, both of which are summarized by Eq 2:

$$C_m \frac{dV}{dt} = -(I_{Na} + I_{Ca}^{m} + I_K + I_{BK} + I_{SK} + I_{leak} + I_{noise}).$$  \hfill (2)

In the first version, MEDAKA 1, the three K$^+$ currents $I_K$, $I_{BK}$ and $I_{SK}$ were identical to those in RAT, so that only the depolarizing, AP generating mechanisms were different. In the second teleost model, MEDAKA 2, we adjusted $I_K$, $I_{BK}$ and $I_{SK}$ so that the model had an AP shape and AP firing rate that were in better agreement with the experimental data in Fig. 1. By comparing RAT to MEDAKA 1, we could then explore the difference between a model with Ca$^{2+}$ APs and one with Na$^+$/Ca$^{2+}$ APs with all other mechanisms being the same. By comparing RAT to MEDAKA 2, we could explore the difference two models that were more representative for experimental data from rat versus medaka. The differences between RAT, MEDAKA 1 and MEDAKA 2 are summarized in Table 1, which lists all the parameter values that were not identical in all three models, and Fig. 2 which shows the kinetics of all included ion channels.

For a full description of the models, we refer to the Methods section, but a brief overview is given here. $I_{Na}$ activated in the range between $-50$ mV and $-10$ mV, with half activation at $-32$ mV (Fig. 2A1), quite similar to what was previously found in goldfish gonadotropes [4]. $I_{Na}$ inactivated in the range between $-90$ mV and $-40$ mV, with half-inactivation at $-64$ mV, which was lower than in goldfish, where the half-inactivation was found to be around $-50$ mV [4]. With the activation kinetics adapted to medaka data, only 6% of the $I_{Na}$ was available at the typical resting potential of $-50$ mV. The fact that medaka still showed TTX-sensitive spontaneous activity thus suggests that $I_{Na}$ is very highly expressed in these cells.

Both $I_{Na}$ and $I_{Ca}^{m}$ had fast activation, $I_{Na}$ being slightly faster with a time constant of about 0.5 - 0.8 ms in the critical voltage range (Fig. 2A2), whereas $I_{Ca}$ had a time constant >1 ms in the critical voltage range (Fig. 2B2). $I_{Ca}^{m}$ activated in the range between $-40$ mV and $+10$ mV, with a half activation at 16 mV (red curve in Fig. 2B1). This activation curve was much steeper than $I_{Ca}$ in RAT (black curve in Fig. 2B1), which was based on data from rat lactotropes [33]. The high activation threshold suggests that $I_{Ca}^{m}$ is unsuitable for initiating spontaneous APs in medaka gonadotropes, making spontaneous activity critically dependent on $I_{Na}$.

The remaining currents ($I_K$, $I_{BK}$ and $I_{SK}$) were all adopted from the RAT model by Tabak et al. [9], using simplified kinetics descriptions with voltage independent time constants. As explained above, these were used in their original form in RAT and MEDAKA 1, but were adapted in MEDAKA 2.

**BK currents have opposite effects on Na$^+$- versus Ca$^{2+}$-mediated action potentials**

It has previously been shown that $I_{BK}$ acts to broaden APs and promote bursting in rat pituitary cells [9,23,28]. In contrast, a high $I_{BK}$ expression in medaka...
Fig 2. Ion channel kinetics (A1) $I_{Na}$ had three activation variables ($q$) and one inactivation variable ($h$). (B1) $I_{Ca}^{m}$ had two activation variables ($m$) in MEDAKA 1 and 2, and $I_{Ca}^{h}$ had one activation variable ($m$) in RAT. (C1) $I_{K}$ had one activation variable ($n$). (D1) $I_{BK}$, had one activation variable ($f$). (E1) $I_{SK}$ was $Ca^{2+}$ activated with one activation variable $s$. (A2-B2) Voltage dependent activation time constants were determined for $I_{Na}$ (A2) and $I_{Ca}^{m}$ (red curve in B2) used in MEDAKA 1 and MEDAKA 2. Voltage-independent activation-time constants were used for $I_{K}$ (C2), $I_{BK}$ (D2) and $I_{SK}$ (E2). (A-E) Black curves denote the RAT model, while red curves denote MEDAKA 2. MEDAKA 1 had the same kinetics as MEDAKA 2 for $I_{Na}$ and $I_{Ca}^{m}$ (A-B), and the same as RAT for the three K+ currents (C-E).

gonadotropes [12] does not make these cells bursty. On the contrary, the experiments in Fig. 1D showed that medaka gonadotropes became more bursty when $BK$ channels were blocked. We hypothesized that opposite effects of $BK$ channels in rats versus medaka are not due to the $BK$ channels being different, but rather due to (the same) $BK$ channels being activated in different ways in the two systems.

Here, we have used computational modeling to test this hypothesis, and have compared simulations on the models RAT, MEDAKA 1 and MEDAKA 2 for different levels of $I_{BK}$ expression (Fig. 3). Following the definition used by Tabak et al. [9], plateau-terminated broad APs of duration longer than 60 ms were defined as bursts (see Methods). When $BK$ was fully expressed in the models (i.e. had the values for $g_{BK}$ given in Table 1), MEDAKA 1 and MEDAKA 2 fired relatively narrow APs that were not classified as bursts, while RAT consistently fired bursts (Fig. 3A). A reduction of $g_{BK}$ lead to a broadening of the APs in MEDAKA 1 and MEDAKA 2, similar to what we saw in the experiments (Fig 2C), while it had the opposite effect on the APs fired by RAT. For an intermediate value of $g_{BK}$, MEDAKA 2 fired broader APs (but still no bursts), MEDAKA 1 became a consistent burster, while only about half of the AP-events in RAT were enduring enough to be classified as bursts (Fig. 3B). When $g_{BK}$ was set to zero, RAT became consistently non-bursty, while the two medaka models became consistent bursters (Fig. 3C).

As summarized in Fig. 3D, the relationship between $g_{BK}$ and the propensity for eliciting bursts in RAT was the opposite of that in the two medaka models. As RAT and MEDAKA 1 differed only in terms of their AP generating mechanisms, these findings suggest that $BK$ has opposing effects on $Ca^{2+}$ generated versus $Na^{+}$ generated APs.
Table 1. Model parameters differing between the models RAT, MEDAKA 1 and MEDAKA 2. $g_BK$ was varied between simulations, and had values between 0 and the (maximum) value given in the respective models. * $g_{Ca}^{in}$ had the units of a permeability (see Methods subsection titled ”Models” for explanation).
**A justification for using different values for $E_{leak}$ is given in the Methods-section.

<table>
<thead>
<tr>
<th>Units</th>
<th>RAT</th>
<th>MEDAKA 1</th>
<th>MEDAKA 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_{Ca}$</td>
<td>0.04</td>
<td>0</td>
<td>0 mS/cm²</td>
</tr>
<tr>
<td>$g_{Ca}^{in}$</td>
<td>0</td>
<td>0.2</td>
<td>0.2 cm/s*</td>
</tr>
<tr>
<td>$g_{Na}$</td>
<td>0</td>
<td>70</td>
<td>70 mS/cm²</td>
</tr>
<tr>
<td>$g_K$</td>
<td>0.96</td>
<td>0.96</td>
<td>1.3 mS/cm²</td>
</tr>
<tr>
<td>$g_BK$</td>
<td>0.32</td>
<td>0.32</td>
<td>0.96 mS/cm²</td>
</tr>
<tr>
<td>$\tau_K$</td>
<td>30</td>
<td>30</td>
<td>5 ms</td>
</tr>
<tr>
<td>$v_f$</td>
<td>-20</td>
<td>-20</td>
<td>-15 mV</td>
</tr>
<tr>
<td>$E_{leak}$</td>
<td>-50</td>
<td>-45</td>
<td>-45 mV</td>
</tr>
</tbody>
</table>

Why BK currents affect Na⁺ and Ca²⁺ spikes differently

The simulations in Fig. 3 had noise added to them. In order to explore why the same $I_{BK}$ could have such different effects in RAT versus MEDAKA 1 and MEDAKA 2, we ran the corresponding simulations without noise. In this way, we could ensure that all aspects of the simulated traces reflected membrane mechanisms (and not random fluctuations). In most of the noise-less simulations, all APs within a given spike train were identical (Fig. 4). The models were quite different in terms of their AP firing frequencies, but we here focus predominantly on how BK affected the shape of single APs.

The explanation of how BK can act to broaden APs in RAT was given in previous studies [9,23]. In brief, $I_{BK}$ acts to reduce the peak amplitude of an AP, as the simulations shown here also demonstrated (compare grey curves in e.g., Fig. 4A2 and E2)). This, in turn, affects $I_K$. Since $I_K$ activates at high voltage levels, the $I_{BK}$-reduced AP amplitude leads to less $I_K$ activation, and via that to a net reduction in the hyperpolarizing currents active during the AP downstroke. Accordingly, the downstroke occurs more slowly, and the AP event gets broader. In the case with full $g_{BK}$ expression, the interplay between the depolarizing $I_{Ca}$ and hyperpolarizing $I_{BK}$ in RAT also lead to transient oscillations during the downstroke (Fig. 4B2)).

To explain why $I_{BK}$ acted so differently in the two medaka models, we start by noting that the $I_{Na}$-mediated upstrokes of their APs were much faster than the $I_{Ca}$-mediated AP in RAT (blue and red curves have faster upstrokes than the grey curves in Fig. 4A2-E2)). $I_{BK}$, having a time constant of 5 ms in all models, therefore had less time to respond during the rapid AP upstrokes in MEDAKA 1 and MEDAKA 2, and its effect on the AP amplitude was, therefore, smaller than in RAT. Although $I_{BK}$ did reduce the AP amplitude in all models, the amplitude reduction when going from max $g_{BK}$ to zero $g_{BK}$ was by about 16 mV in RAT against only 8 mV and 7 mV in MEDAKA 1 and MEDAKA 2, respectively. Another consequence of the faster AP upstroke in the medaka models was that a major part of the action of $I_{BK}$ occurred after the AP peak, i.e., during the AP downstroke. In MEDAKA 1 the effect of $I_{BK}$ was thus shifted away from reducing the AP peak towards contributing to the AP downstroke, and, in this way, $I_{BK}$ acted to make AP events narrower.

For the case with full $g_{BK}$ expression (Fig. 4A), MEDAKA 1 fired APs with an unrealistically long duration and at a higher frequency than seen in the experimental data (Fig 1A-B). Even for full BK expression, the AP peaks in MEDAKA 1 were
Fig 3. Effects of $I_{BK}$ on bursting in rat versus medaka. The spontaneous activity of three pituitary cell models, RAT, MEDAKA 1 and MEDAKA 2, for different levels of BK expression. (A) Simulations performed with maximally expressed $g_{BK}$. (B) Simulations with $g_{BK}$ reduced to half of the maximum value. (C) Simulations with $g_{BK}$ set to zero. (A-C) Right panels show histograms of the distribution of AP durations in the respective models. The burstiness factor (BF) denotes the fraction of events that were counted as bursts, i.e., events with a duration exceeding 60 ms. (D) The relationship between $g_{BK}$ and the propensity for eliciting bursts in the three models. (A-D) The BK conductance is given relative to the maximum value $g_{BK}$ used in the respective models and was four times bigger in MEDAKA 2 than in RAT and MEDAKA 1. Simulations were otherwise the same as in Fig. 1 of the study by Tabak et al. [9], and a Gaussian noise stimulus was added.
**Fig 4.** Effects of $I_{BK}$ of AP shape in rats versus medaka models. (A1-E1) The spontaneous activity of three pituitary cell models, RAT, MEDAKA 1 and MEDAKA 2, for different levels of BK expression (no noise added to the simulations). (A2-E2) Close-ups of selected AP events. The events were aligned, i.e. $t = 0$ (in A2-E2) was defined independently for each trace (in A1-E1) to a point directly before the onset of the selected event. The BK conductance is given relative to the maximum value $g_{BK}$ used in the respective models and was four times bigger in MEDAKA 2 than in RAT and MEDAKA 1.
also responded to $I_{BK}$ blockage in a way that resembled that seen in experiments, i.e., $I_{BK}$ blockage lead to a broadening of the APs (Fig 1D). The resemblance with data was strongest for the simulations with partial blockage, when the AP plateaus underwent oscillations that presumably reflected an interplay between $I_{Na}$ and repolarizing currents activating/inactivating during the downstroke (red lines in Fig 9C2-D2). It is reasonable to assume that also in the experiments, the blockage of BK by paxilline was not complete. Despite differences between MEDAKA 1 and MEDAKA 2, the effect of $g_{BK}$ on the AP shape was similar in the two medaka models, and the opposite of that in RAT.

In summary, $I_{BK}$ had an inhibitory effect on $I_K$ by reducing the AP amplitude, and a collaborative effect with $I_K$ in mediating the AP downstroke. With fast AP upstrokes, as in RAT, the inhibitory effect of $I_{BK}$ on $I_K$ was stronger than the collaborative, while with slow AP upstrokes, as in MEDAKA 1 and MEDAKA 2, the collaborative effect was stronger than the inhibitory. In this way, $I_{BK}$ acted as a mechanism that reduced the duration of already brief (Na$^+$-mediated) APs, and prolonged the duration of already slow (Ca$^{2+}$-mediated) APs.

Membrane mechanisms responsible for burstiness

In order to explore in further detail how the various membrane mechanisms affected the AP firing in RAT, MEDAKA 1 and MEDAKA 2, we performed a feature-based sensitivity analysis of the three models. We then assigned the maximum conductances of all included currents uniform distributions within intervals $\pm 50\%$ of their default values (Table 1), and quantified the effect that this parameter variability had on selected aspects of the model output (see Methods). An exception was made for $g_{BK}$, which was assigned a uniform distribution between 0 and the maximum values given in Table 1), i.e., from fully available to fully blocked, motivated by the fact that $g_{BK}$ did not have a default value in RAT. We note the total order Sobol sensitivity indices considered in the current analysis reflects complex interactions between several nonlinear mechanisms, and that mechanistic interpretations therefore are difficult. Below, we have still attempted to extract the main picture that emerged from the analysis.

Three features of the model responses were considered: (i) I$^{\text{bursting}}$, (ii) I$^{\text{irregular}}$, and (iii) Isfiring. All these features were binary, meaning e.g., that I$^{\text{bursting}}$ was equal to 1 in a given simulation if it contained one or more bursts, and equal to 0 if not. The mean value of a feature (taken over all simulations) then represented the fraction of simulations that had this feature. For example, in RAT I$^{\text{bursting}}$ had a mean value of 0.57, I$^{\text{irregular}}$ had a mean value of 0.37, and I$^{\text{s firing}}$ had a mean value of 0.91. This means, respectively, that 57% of the parameterizations of RAT fired bursts, 37% fired regular APs, and 91% fired APs that were either bursts or regular spikes. We note that the mean values for the I$^{\text{bursting}}$ and I$^{\text{irregular}}$ features in RAT and MEDAKA 2 did not sum up exactly to the mean value for the AP firing feature. This was because some parameterizations of these two models fired both bursts and regular APs within the same simulation.

AP firing was seen quite robustly in RAT, which fired APs in 91 % of the sampled parameter combinations (mean value of 0.91 in Fig 5A3), while APs were fired in only 77 % and 48% of the parameterizations of MEDAKA 1 and MEDAKA 2, respectively (Fig 5B3-C3). In MEDAKA 2 there was thus AP activity in less than half of the parameterizations. This reflects that the default configuration had a resting potential only slightly above the AP generation threshold, so that any parameter sample that would make the cell slightly less excitable, would abolish its ability for AP generation.

Within the explored parameter range, RAT and MEDAKA 1 tended to be bursting, meaning that bursts were seen in a larger fraction of the simulations than regular spikes (mean values in Fig 5A1 and B1 are larger than in Fig 5A2 and B2). MEDAKA 2, on
Fig 5. Feature-based sensitivity analysis. Sensitivity of RAT (A), MEDAKA 1 (B) and MEDAKA 2 (C) to variations in the maximum ion channel conductances. The analysis summarizes a large number of simulations where the maximum conductances of all included ion channels were varied within intervals ±50% of their original values. An exception was made for $g_{BK}$, which was varied between 0 and 0.32 mS/cm² in (A) RAT and (B) MEDAKA 1, and between 0 and 1.26 mS/cm² in (C) MEDAKA 2. (A1-C1) $I_{bursting}$ denotes the fraction of simulations that elicited one or more bursts. (A2-C2) $I_{irregular}$ denotes the fraction of simulations that elicited one or more regular APs, and (A3-C3) $I_{firing}$ denotes the fraction of simulations that elicited any AP events at all. (A-C) Histograms depict the total order Sobol sensitivity indices (see Methods). The analysis was performed by aid of the recently developed toolbox Uncertainty [34] (see Methods for details). Simulations were run with no noise.

the contrary, was adapted to experimental data where AP firing under control conditions (Fig. 1A-B) was regular and with very narrow APs. MEDAKA 2 thus had a high propensity for firing regular APs and elicited bursts only in (14%) of the parameter combination (Fig 5C1-C2).

The total order Sobol indices (shown in histograms in Fig 5) quantify how much of the variability (between different simulations) in the response features that were explained by the variation of the different model parameters (i.e., maximum conductances). For example, $I_{firing}$ in MEDAKA 1 and MEDAKA 2 were most sensitive to $g_K$ and $g_{Na}$ (Fig 5B3 and C3)), meaning that these conductances were most important for whether the model was capable of generating AP events. This result was not surprising, since $I_K$, $I_{Na}$ (along with the leakage current, $I_L$) were the only currents with a nonzero activity level around rest, and thus were the ones that determined whether the resting potential was above the AP firing threshold (cf. Fig 2).

Having other AP generation mechanisms, $I_{firing}$ in RAT was instead sensitive...
predominantly to $g_{Ca}^r$ and $g_{SK}$ (Fig 5A3)). Due to the wide activation range of $I_{Ca}^r$, generating the APs in RAT, this model was in principle always above the AP firing threshold. The instances when AP firing was not seen in RAT then reflected lack of hyperpolarizing mechanisms, such as $g_{SK}$, that could repolarize the membrane potential after the AP peak.

In general, $I_{bursting}$ and $I_{regular}$ were not sensitive to the same parameters as $I_{firing}$. This implies that the mechanisms determining whether the cell fired an AP or not were not the same as those determining the duration of the AP once it had been fired. For example, $I_{firing}$ in MEDAKA 2 was nearly insensitive to $g_{BK}$, while $I_{bursting}$ and $I_{regular}$ spiking were highly sensitive to this parameter. A little simplified, we may say that $g_{K}$ and $g_{Na}$ thus determined whether MEDAKA 2 fired an AP (Fig 5C3), while $g_{BK}$ (and to some degree $g_{Ca}^r$) determined whether the AP became a burst or a regular spike (Fig 5C1-C2). In MEDAKA 1, the situation was more complex, and its $I_{bursting}$ and $I_{regular}$ were sensitive to almost all model parameters (Fig 5B1-B2).

In RAT, $I_{bursting}$ was most sensitive to $g_{K}$ and second most to $g_{BK}$ (Fig 5A1). The lower sensitivity to $g_{BK}$ compared to $g_{K}$ might seem surprising, given the previously proclaimed role of $g_{BK}$ as a burst promoter in this model [9]. However, we can interpret this finding in light of the above-established understanding of burst generation in RAT. We remember that $I_{BK}$ promoted bursting indirectly by inhibiting $I_{K}$, so that final determinant for whether an event became a burst was the magnitude (or rather the reduced magnitude) of $I_{K}$. The results from the sensitivity analysis then simply indicate the direct effect on $I_{K}$ obtained by a variation of $g_{K}$ was larger than the indirect effect on $I_{K}$ obtained by variation of $g_{BK}$.

We note that the three different feature sensitivities in Fig 5 were not independent. For example, if variations in a given parameter tended to switch the cell between bursting and not firing at all, $I_{bursting}$ and $I_{firing}$ would share a high sensitivity to this parameter. Likewise, if variations in a given parameter instead tended to switch the cell between bursting and regularly spiking, $I_{bursting}$ and $I_{regular}$ would share a high sensitivity to this parameter. In this regard, $g_{BK}$ was the parameter with the cleanest role as a switch between bursting and regular spiking, as in all models, the $I_{bursting}$ and $I_{regular}$ had a high sensitivity to $g_{BK}$, while $I_{firing}$ had a quite low sensitivity to $g_{BK}$.

A more general insight from the sensitivity analysis was that the propensity for AP firing, regular spiking and bursting in all three models depended on a complex interplay between several mechanisms. All models could be shifted from regularly spiking to bursty by changes, not only in $g_{BK}$ (as demonstrated in Fig 3 and Fig 4), but also in other conductances (such as $g_{K}$ or $g_{Ca}^r$). In all models, however, bursts were facilitated by $g_{Ca}^r$ and counteracted by $g_{K}$, while $g_{BK}$ was the more fascinating mechanism, having the opposite effect on the burstiness in RAT versus the two medaka models.

### Discussion

TTX-sensitive Na$^+$ currents ($I_{Na}$) are present in all pituitary cells, but are in many cases inactive during spontaneous activity [8]. Previous models of the electrical activity of pituitary cells have focused on conditions where $I_{Na}$ is of lesser importance, and where AP generation is predominantly mediated by high-voltage activated Ca$^{2+}$ currents [3,9,22-26]. To our knowledge, we have in the current work presented the first models that describe pituitary cells under conditions where AP generation is $I_{Na}$-mediated. The model was adapted to experimental data from gonadotrope luteinizing hormone-producing cells in medaka, whose spontaneous activity is highly $I_{Na}$-dependent. Voltage-clamp data was used to develop models for the activation...
kinetics for \( I_{Na} \) and \( I_{Ca} \) currents, and the firing properties of the model were further adapted to current-clamp data from spontaneously active cells (under control conditions, and after application of TTX and paxilline).

An important goal of this work was to perform a comparison between the response properties of pituitary cells that elicited \( I_{Ca} \)-mediated APs and those that elicited \( I_{Na} \)-mediated APs. For this, we used a model based on data from rat, which elicited \( I_{Ca} \)-mediated APs, and compared it to two models (MEDAKA 1 and MEDAKA 2) based on data from medaka, both of which elicited \( I_{Na} \)-mediated APs. In this context, MEDAKA 1 had the (methodological) advantage that all hyperpolarizing membrane mechanisms were kept identical to those in RAT, so that RAT and MEDAKA 1 differed only in terms of AP generation mechanisms. MEDAKA 2 was more strongly adapted to experimental data, and had the advantage that its firing properties were more representative for real medaka gonadotropes. The most interesting result that came out of this comparison, was that BK currents (\( I_{BK} \)) had a diametrically opposite role in terms of how they affect the AP shape in RAT versus the medaka models. When APs were generated by \( I_{Ca} \) (as in RAT), they were relatively slow, and \( I_{BK} \) then acted to broaden the AP events and promote bursting behavior [9, 23]. On the contrary, when APs predominantly were generated by \( I_{Na} \) (as in MEDAKA 1 and MEDAKA 2), they had a rapid upstroke, and \( I_{BK} \) then acted to make the AP narrower, and prevent bursting behavior (Fig 3). This suggests that \( I_{BK} \) acts as a mechanism that distinguishes between rapid and slow APs, and amplifies the difference between the two by narrowing down already narrow APs while broadening already broad APs.

It should be noted that the effect seen in medaka gonadotropes, i.e., that \( I_{BK} \) acted to reduce AP duration, is a commonly reported role for \( I_{BK} \) in many excitable cells [35–39], while the burst promoting effect that \( I_{BK} \) had in rat pituitary cells [9, 23] is less conventional. Furthermore, the role of \( I_{BK} \) as a burst promoter has not been found consistently in rat pituitary cells. In the study by Miranda et al. 2003, AP duration in rat pituitary cells was instead found to increase when \( I_{BK} \) was blocked with paxilline [38], i.e. similar to what we found for medaka gonadotropes (Fig 1D). The different effects of \( I_{BK} \) on spike duration observed in different laboratories [23, 35, 38] was addressed by Tabak et al. 2011 [9], who proposed possible explanations that could reconcile the conflicting results. One possible explanation could be that \( I_{BK} \) has a different role in terms of how BK channels are localized in various cells, and that BK channels that are co-localized with Ca\(^{2+}\) channels will respond rapidly to voltage fluctuations and promote bursting, while BK channels that are not co-localized with Ca\(^{2+}\) channels will react more slowly to voltage fluctuations and have the opposite effect [9]. A second possible explanation, also suggested by Tabak et al. 2011, was that \( I_{BK} \) might have different kinetic properties in different cells due to variations in their phosphorylation state [9]. A third explanation could be that different cells have different BK splice variants [40], or different regulatory sub-units.

The simulations presented in Fig 3 provide an alternative possible explanation to the conflicting conclusions regarding the role of \( I_{BK} \). The fact that \( I_{BK} \) has a different effect on the AP shape in different cells does not by necessity reflect a variation in \( I_{BK} \) expression or kinetics between the different cell types, as proposed by Tabak et al. 2011. As Fig 3 showed, the same model of \( I_{BK} \) could either prolong or reduce AP generation, depending on which membrane mechanisms (\( I_{Ca} \) or \( I_{Na} \)) that mediated the AP upstroke. Such differences in AP mechanisms could in principle explain the differences between the conflicting experiments on rat pituitary cells [23, 38]. In the experiment by Van Goor et al. 2001, where \( I_{BK} \) was found to prolong the AP duration, APs were predominantly mediated by \( I_{Ca} \) [9, 23]. In the experiment by Miranda et al. 2003, where \( I_{BK} \) was found to shorten the AP duration (i.e., that blocking \( I_{BK} \) lead to longer APs), it was reported that this only occurred under conditions in which short APs were
It is likely that the events described in that work as short APs, were \( I_{Na} \)-mediated APs, so that the differences between the two studies by Van Goor 2011 and Miranda 2003 are reflected by the differences between the simulations on RAT versus MEDAKA 1 and MEDAKA 2 in Fig 3.

Although MEDAKA 2 captured the essential firing properties of medaka gonadotropes, the agreement between model and data was not perfect. For example, the AP duration in MEDAKA 2 during control conditions (Fig 9C) was in the upper range of that seen in the experiments (Fig 1A-B), and we were not able to obtain briefer APs in the model without compromising the agreement between the experimental data and other model features, such as the AP peak amplitude, afterhyperpolarization, and response to \( I_{BK} \)-blockage. The conductances selected for MEDAKA 2 (Table 1) were thus a compromise made to obtain an acceptable match to several features simultaneously. The fact that we were not able to obtain a more accurate match between model and data likely reflect that some of the ion channels present in the model are imperfect representations of the ion channels present in the real cell. For example, the simplified kinetics schemes used for \( I_K \), \( I_{BK} \) and \( I_{SK} \) were adopted from a model of rat pituitary cells [9], and were not constrained to data from medaka gonadotropes. In addition, the biological cell is likely to contain a variety of additional ion channels [8] that were not included in the model.

To our knowledge, MEDAKA 1 and MEDAKA 2 are the first computational models that describe \( I_{Na} \)-based APs in endocrine cells. Although MEDAKA 2 was adapted to experimental recordings from gonadotrope luteinizing hormone-producing cells in medaka, we believe that the model can have a more general value. Different types of pituitary cells in several different species share many of the same membrane mechanisms [8]. In particular, \( I_{Na} \)-based APs are elicited by several pituitary endocrine cell types and in several animal species, depending on biological conditions [4, 7, 17, 18, 30–32]. It is thus likely that the response patterns of related cell types may be captured by up- or down-regulation of selected mechanisms included in MEDAKA 2. In that context, the sensitivity analysis presented in Fig 5 may give useful guidance in terms of which parameters that should be adjusted in order to obtain desired changes in the models firing properties.

**Methods**

**Experimental procedures**

The electrophysiological experiments were conducted using the patch-clamp technique on brain-pituitary slices from adult female medaka (as described in [41]). To record spontaneous action potentials and \( Ca^{2+} \) currents we used amphotericin B perforated patch configuration, while for \( Na^+ \) currents we used whole cell configuration. Extracellular (EC) solution used for recording spontaneous action potentials (current clamp) contained 134 mM NaCl, 2.9 mM KCl, 2 mM CaCl\(_2\), 1.2 mM MgCl\(_2\), 10 mM HEPES, 4.5 mM glucose. The solution was adjusted to a pH of 7.75 with NaOH (added drop-wise from 1M solution) and osmolality adjusted to 290 mOsm with mannitol before sterile filtration. Before use, the EC solution was added 0.1 % bovine serum albumin (BSA). For \( Na^+ \) current recordings (voltage clamp) we used a \( Ca^{2+} \) free and \( Na^+ \) fixed (140 mM) EC solution, pH adjusted with trizma base. In addition, 10 \( \mu \)M nifedipine, 2 mM 4-Aminopyridine (4-AP) and 4 mM Tetraethylammonium (TEA) was added to the EC solution just before the experiments. To record \( Ca^{2+} \) currents, we substituted NaCl with 120 mM choline-Cl and added 20 mM \( Ca^{2+} \), 2 mM 4-AP and 4 mM TEA. The patch pipettes were made from thick-walled borosilicate glass using a horizontal puller (P 100 from Sutter Instruments). The resistance of the patch pipettes was
was 4–5 MΩ for perforated patch recordings and 6–7 MΩ for whole-cell recordings. For recordings of spontaneous action potentials, the following intracellular (IC) electrode solution was added to the patch pipette: 120 mM KOH, 20 mM KCl, 10 mM HEPES, 20 mM Sucrose, and 0.2 mM EGTA. The pH was adjusted to 7.2 using CsH3NO4S (mes) acid, and the osmolality to 280 mOsm using sucrose. The solution was added 0.24 mg/ml amphotericin B to perforate the cell membrane (see [41] for details). In voltage clamp experiments the K+ was removed from the intracellular solution to isolate Na+ and Ca2+ currents. This was achieved by substituting KOH and KCl with 130 mM Cs-mes titrated to pH 7.2 with CsOH. The electrode was coupled to a Multiclamp 700B amplifier (Molecular Devices) and recorded signal was digitized (Digidata 1550 with humsilencer, Molecular Devices) at 10 KHz and filtered at one-third of the sampling rate. In selected experiments, voltage-gated Na+ channels were blocked using 5 μM TTX, and BK channels were blocked using 5 μM paxilline. Both drugs were dissolved in EC solution and applied using 20 kPa puff ejection through a 2 MΩ pipette, 30–40 μm from the target cell.

Under the experimental (voltage-clamp) conditions used for recording Na+ currents, and under the experimental current-clamp conditions, a liquid junction potential of about −9 mV was calculated and corrected for in the data shown in Fig 1, and in the kinetics model for $I_{Na}$ (Fig 2A). A liquid junction potential of about −15 mV was calculated for the experimental (voltage-clamp) conditions used for recording Ca2+ currents, and was corrected for in the kinetics model for $I_{Ca}^{m}$ (Fig 2B).

Models

As stated in the Results-section, RAT was described by the equation:

$$C_m \frac{dV}{dt} = -(I_{Ca}^m + I_K + I_{BK} + I_{SK} + I_{leak} + I_{noise}).$$  \hspace{1cm} (3)

and the two medaka models, MEDAKA 1 and MEDAKA 2, by:

$$C_m \frac{dV}{dt} = -(I_{Na} + I_{Ca}^m + I_K + I_{BK} + I_{SK} + I_{leak} + I_{noise}),$$ \hspace{1cm}\hspace{1cm} (4)

The passive membrane properties were the same in all models, with specific membrane capacitance $C_m = 3.2 \mu F/cm^2$, and a leak conductance 0.692 mS/cm2.

$$I_{leak} = g_{leak}(V - E_{leak}).$$ \hspace{1cm} (5)

RAT had a passive reversal potential of $E_{leak} = -50$ mV. In RAT, $I_{Ca}^m$ was quite active at the resting potential, and counteracted the hyperpolarizing effect of $I_K$ which was also had a non-zero activity level around the resting potential. Replacing $I_{Ca}^m$ with $I_{Ca}^m$ in MEDAKA 1 and MEDAKA 2 therefore removed a source of spontaneous depolarization and lead to an unwanted hyperpolarization of the resting potential. To compensate for this, $E_{leak}$ was adjusted to −45 mV in MEDAKA 1 and MEDAKA 2, which resulted in an effective resting potential around −50 mV, similar to that in RAT, and also to the experimental data in Fig 1.

Below, we give a detailed description of the active ion channels. The kinetics of all ion channels were summarized in Fig. 2, the model parameters that differed between the models were listed in Table 1, while new parameters will be defined when introduced.

$I_{Na}$ (MEDAKA 1 and MEDAKA 2) was modeled using the standard Hodgkin and Huxley-form [42]:

$$I_{Na} = g_{Na}q^3h(V - E_{Na}),$$ \hspace{1cm} (6)

with a reversal potential $E_{Na} = 50$ mV, and gating kinetics defined by:

$$\frac{dq}{dt} = \frac{q_{\infty} - q}{\tau_q}, \quad \frac{dh}{dt} = \frac{h_{\infty} - h}{\tau_h}. \hspace{1cm} (7)$$

$$ \frac{dV}{dt} = \left( \frac{I_{Na} - I_K - I_{BK} - I_{SK} - I_{leak}}{C_m} \right).$$
The steady-state activation and time constants ($q_\infty$, $h_\infty$, $\tau_q$ and $\tau_h$) were fitted to voltage-clamp data from medaka gonadotropes, as described below, in the subsection titled "Model for the voltage-gated Na\(^+\) channels".

$I_{Ca}^{m}$ (MEDAKA 1 and MEDAKA 2) was modelled using the Goldman-Hodgkin-Katz formalism, which accounts for dynamics effect on Ca\(^{2+}\) reversal potentials [43]:

\[
I_{Ca}^{m} = g_{Ca}m^2 \frac{F^2}{RT} V \left[ [Ca]_c - [Ca]_e \exp\left(\frac{-V}{RT}\right) \right] \left( 1 - \exp\left(\frac{-V}{RT}\right) \right),
\]

with

\[
\frac{dm}{dt} = \frac{m_\infty - m}{\tau_m}.
\]

Here, $R = 8.314$ J/(mol · K) is the gas constant, $F = 96485.3$ C/mol is the Faraday constant, $T$ is the temperature, which was set to 293.15 K in all simulations. $[Ca]_c$ and $[Ca]_e$ were the cytosolic and extracellular Ca\(^{2+}\) concentrations, respectively. The former was explicitly modelled (see below), while the latter was assumed to be constant at 2 mM. As Eq 8 shows, we used two activation variables $m$. This is typical for models of L-type Ca\(^{2+}\) channels (see e.g. [44–47]), which are the most abundantly expressed HVA channels in the cells studied here [11]. The steady-state activation and time constant ($m_\infty$ and $\tau_q$) were fitted to voltage-clamp data from medaka gonadotropes, as described below, in the subsection titled "Model for high-voltage activated Ca\(^{2+}\) channels". We note that $g_{Ca}$ in the Goldman-Hodgkin-Katz formalism (Eq 8) is not a conductance, but a permeability with units cm/s. (It is proportional to the conductance, and for simplicity, we have referred to it as a conductance in the text).

$I_{Ca}^{r}$ (RAT) was a simpler Ca\(^{2+}\)-channel model, with one gating variable and constant reversal potential [9]:

\[
I_{Ca}^{r} = g_{Ca}m_\infty (V - E_{Ca}),
\]

with instantaneous steady-state activation:

\[
m_\infty = \left[ 1 + \exp\left( (v_m - V) / s_m \right) \right]^{-1}.
\]

where $s_m$ = is a slope parameter and $v_m$ the potential at half-activation.

The directly rectifying K\(^+\) channel (all models) was modelled as

\[
I_K = g_K n (V - E_K),
\]

with reversal potential $E_K = -75$ mV, and a time dependent activation variable described by

\[
\frac{dn}{dt} = \frac{n_\infty - n}{\tau_K}.
\]

The constant (voltage-independent) time constant $\tau_K$ was 30 ms in RAT and MEDAKA 1, and 5 ms in MEDAKA 2, and the steady-state activation was described by:

\[
n_\infty = \left[ 1 + \exp\left( (v_n - V) / s_n \right) \right]^{-1}.
\]

with a slope parameter $s_n = 10$ mV, and half-activation $v_n = -5$ mV.

The BK-channel (all models) was modelled as

\[
I_{BK} = g_{BK} f (V - E_K),
\]

The activation kinetics was:

\[
\frac{df}{dt} = \frac{f_\infty - f}{\tau_{BK}}.
\]
with a constant (voltage-independent) activation time constant \( \tau_{BK} = 5 \text{ ms} \). The steady-state activation was given by:

\[
f_\infty = \left[1 + \exp((v_f - V)/s_m)\right]^{-1}.
\]

with a slope parameter \( s_m = 12 \text{ mV} \), and half-activation \( v_m \) with values \(-20 \text{ mV} \) in RAT and MEDAKA 1 and \(-15 \text{ mV} \) in MEDAKA 2. We note that BK was modeled as a voltage (and not \( \text{Ca}^{2+} \)) dependent current. The rationale behind this simplification was that BK activation depends on the \( \text{Ca}^{2+} \) concentration in highly localized \( \text{Ca}^{2+} \) nanodomains, where it is co-localized with, and depends on influx through, high-voltage activated \( \text{Ca}^{2+} \) channels. This influx is in turn largely determined by the membrane potential and reaches an equilibrium within microseconds [9], making the BK channel effectively voltage dependent.

Finally, the SK channel (all models) was modeled as:

\[
I_{SK} = g_{SK} s_\infty([\text{Ca}]) (V - E_K),
\]

with an instantaneous, \( \text{Ca}^{2+} \) dependent, steady-state activation:

\[
s_\infty([\text{Ca}]) = \frac{[\text{Ca}]^2}{[\text{Ca}]^2 + k_s^2}.
\]

where \([\text{Ca}]\) denotes the cytosolic \( \text{Ca}^{2+} \) concentration, and \( k_s \) was a half-activation concentration of 0.4 \( \mu \text{M} \).

\( I_{Na}^m \) and \( I_{SK} \) were dependent on the global cytosolic \( \text{Ca}^{2+} \) concentration. This was modelled as a simple extrusion mechanism, receiving a source through \( I_{mCa} \) (for MEDAKA 1 and MEDAKA 2) and \( I_{rCa} \) for RAT, and with a concentration dependent decay term assumed to capture the effects of various extrusion and buffering mechanisms:

\[
\frac{d[\text{Ca}]}{dt} = -f_c(\alpha I_{Ca} + k_c[\text{Ca}]).
\]

Here, \( f_c = 0.01 \) is the assumed fraction of free \( \text{Ca}^{2+} \) in the cytoplasm,

\[
\alpha = 0.004725 \text{ mM} \cdot \text{cm}^2/\mu\text{C} \text{ converts an incoming current to a molar concentration, and } k_c = 0.12 \text{ ms}^{-1} \text{ is the extrusion rate [9]. Due to the requirements from the NEURON simulator, the parameters from the original model by Tabak et al. [9] were converted from total currents/capacitance to currents/capacitance per membrane area, using a cell body with a membrane area of } \pi \cdot 10^{-6} \text{ cm}^2 \text{ (see https://github.com/ReScience/ReScience-submission/pull/53).}
\]

In the simulations in Fig 3, a noise input was added, described by

\[
I_{noise} = A_{noise} \eta/\sqrt{\Delta t},
\]

where the noise amplitude \( A_{noise} \) was 4 pA, and \( \eta \) was a random process drawn from a normal distribution [9].

Model for the voltage-gated Na\(^+\) channels

The steady-state values and time courses of the gating kinetics were determined using standard procedures (see e.g. [32, 42, 48, 49]), and was based on the experiments summarized in Fig 6. To determine activation, the cell was held at \(-60 \text{ mV} \) for an endured period, and then stepped to different holding potentials between \(-80 \) to \(100 \text{ mV} \) with 5 mV increments (Fig. 6A2), each for which the response current \( (I_{Na}) \) was recorded (Fig. 6A1). The inactivation properties of Na\(^+\) were investigated using stepwise pre-pulses (for 500 ms) between \(-90 \) and \(55 \text{ mV} \) with 5 mV increments before
recording the current at $-10$ mV (Fig. 6B2). The resulting Na$^+$ current then depended on the original holding potential (Fig. 6B1). Finally, the recovery time for the Na$^+$ current was explored by exposing the cell to a pair of square pulses (stepping from a holding potential of $-60$ mV to $-10$ mV for 10 ms) separated by a time interval $\Delta t$ (Fig. 6C2). The smallest $\Delta t$ was 10 ms, and after this, $\Delta t$ was increased with 100 ms in each trial. The cell responded to both pulses by eliciting Na$^+$-current spikes (Fig. 6C2). When $\Delta t$ was small, the peak voltage of the secondary spike was significantly reduced compared to the first spike, and a full recovery required a $\Delta t$ in the order of 1/2-1 s.

**Fig 6. Experimentally recorded Na$^+$ currents.** (A) Na$^+$ currents evoked by the activation protocol. (B) Na$^+$ currents evoked by the inactivation protocol. (C) Na$^+$ currents used to determine the time constant for recovery from inactivation. (A-C) Voltage protocols are shown below the recorded currents, and all panels show a series of experiments (traces). In (C), the cell was exposed to a pair of square 10 ms pulses arriving with various inter-pulse intervals ($\Delta t$). The first pulse always arrived after 40 ms (and coincided in all experiments), while each secondary spike represents a specific experiment (i.e., a specific $\Delta t$). The traces were normalized so that the first spike had a peak value of $-1$ (corresponding to approximately $-0.25$ pA).

**Steady-state activation and inactivation**

To determine steady-state activation, the peak current ($I_{\text{max}}$) was determined for each holding potential in Fig. 6A, and the maximum peak was observed at about $-10$ mV. For inactivation, the peak current ($I_{\text{max}}$) was recorded for each holding potential in Fig. 6B. In both cases, the maximal conductance ($g_{\text{max}}$) for each holding potential was computed by the equation:

$$g_{\text{max}} = \frac{I_{\text{max}}}{(V_{\text{hold}} - E_{Na})}.$$  \hspace{1cm} (22)

Under the experimental conditions, the intra- and extracellular Na$^+$ concentrations were 4 mM and 140 mM, respectively, and the temperature was 26 degrees Celsius, which gives a reversal potential ($E_{Na} = RT/F \cdot \ln([Na]_{ex})/\ln([Na]_{in})$) of 92 mV. The
estimates of $g_{\text{max}}$ for activation and inactivation are indicated by the markers ‘x’ in Fig. 7A and B, respectively, and markers ‘o’ indicate a secondary experiment. The dependency of $g_{\text{max}}$ on $V_{\text{hold}}$ was fitted by a Boltzmann curve:

$$f_{bs} = \frac{\bar{g}_{\text{max}}}{1 - \exp((V_\star - V_{\text{hold}})/k)^a},$$

(23)

where $\bar{g}_{\text{max}}$ corresponds to $g_{\text{max}}$ estimated for the largest peak in the entire data set (i.e. at about ~0.22 nA in Fig. 6A and ~0.18 nA in Fig. 6B). The factor $k$ determines the slope of the Boltzmann curve, the exponent $a$ corresponds to the number of activation or inactivation gates, and $V_\star$ determines the voltage range where the curve rises. When $a = 1$ (as for inactivation), $V_\star$ equals $V_{1/2}$, i.e. the voltage where $f_{bs}$ has reached its half maximum value. With a higher number of gates, $V_\star = V_{1/2} + k \cdot \ln(2^{1/a} - 1)$. Eq 23 gave a good fit for the steady state activation $m_{\text{inf}}^i$ (Fig 7A) and the steady state inactivation $h_{\text{inf}}$ (Fig 7B) with the parameter values for $a$, $k$ and $V_\star$ listed in Table 2.

### Time constants for activation and inactivation

With three opening gates ($q$) and one closing gate ($h$), the time constants for activation and inactivation were derived by fitting the function [42]

$$I_{\text{max}} = g_Na(V_{\text{hold}} - E_{Na})[1 - \exp(1 - t/\tau_q)^3(1 - \exp(1 - t/\tau_h))],$$

(24)

to the response curves in Fig. 6A1. For low step potentials (< ~40 mV), the response was too small and noisy to reveal any clear trend, and we were unable to obtain...
meaningful fits using the functional form of Eq. 22. For this reason, only the experiments with a step potential of $-40$ mV and higher were used when fitting the time constants. The fitting procedure resulted in a pair of time constants ($\tau_q$ and $\tau_h$) for each step potential in the protocol, as indicated by the data points ('x' and 'o') in Fig. 7C and D. The data points obtained by fitting Eq. 24 to the traces in Fig. 6A1 were sufficient to obtain a clear picture of the voltage dependence of the activation time constant ($\tau_q$), which had a peak value at $-24$ mV, i.e. within the voltage-range for which there was suitable data (Fig. 7C). The inactivation time constant ($\tau_h$) was, however, monotonously decreasing over the voltage range for which there was good data. We therefore needed additional data points for the voltage dependence of $\tau_h$ in the range $V < -40$ mV. Based on the insight from the recovery-experiments (Fig. 6C), we expected inactivation to be very slow at the resting potential and below. To account for this, we introduced the three additional data points marked by '*' in Fig. 7D, which assure a recovery time in the correct order of magnitude.

The data points for the time constants were fitted with curves on the functional form proposed by Traub et al. [50]:

$$
\tau = \left( \frac{p_1(p_2-V)}{\exp[(p_2-V)/p_3]-1} + \frac{p_4(V-p_5)}{\exp[(V-p_5)/p_6]-1} \right)^{-1}.
$$

(25)

Good fits to the data points were obtained with the parameter values in Table 2.

<table>
<thead>
<tr>
<th>parameter</th>
<th>activation value</th>
<th>inactivation value</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>0.038</td>
<td>0.040</td>
<td>$(\text{ms$\cdot\text{mV}$})^{-1}$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>-60.3</td>
<td>-32.4</td>
<td>mV</td>
</tr>
<tr>
<td>$p_3$</td>
<td>5.77</td>
<td>3.29</td>
<td>mV</td>
</tr>
<tr>
<td>$p_4$</td>
<td>0.135</td>
<td>2.65</td>
<td>$(\text{ms$\cdot\text{mV}$})^{-1}$</td>
</tr>
<tr>
<td>$p_5$</td>
<td>-26.17</td>
<td>-2145</td>
<td>mV</td>
</tr>
<tr>
<td>$p_6$</td>
<td>3e-5</td>
<td>139.3</td>
<td>mV</td>
</tr>
<tr>
<td>$V_*$</td>
<td>28.84</td>
<td>55.0</td>
<td>mV</td>
</tr>
<tr>
<td>$k$</td>
<td>4.55</td>
<td>5.07</td>
<td>mV</td>
</tr>
<tr>
<td>$a$</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$V_{lj}$</td>
<td>-9</td>
<td>-9</td>
<td>mV</td>
</tr>
</tbody>
</table>

Table 2. Parameters for Na$^+$ activation. The parameters $p_1$-$p_6$ are used together with Eq. 25 to yield the time constants for steady state activation and inactivation (in units of ms). The remaining parameters are used together with Eq. 23 to obtain the steady-state activation and inactivation functions. The curves obtained in this way describe the voltage dependence under experimental conditions, and was afterwards corrected by subtracting the liquid junction potential of $-9$ mV (see Fig 2A).

Model for high-voltage activated Ca$^{2+}$ channels

When estimating the steady-state values and time constant we followed procedures inspired from previous studies of L-type Ca$^{2+}$ channel activation, we did not use Eq 8, but used the simpler kinetics scheme $I_{Ca} = g_{HV}A_m^2(V - E_{Ca})$ (see e.g. [45]) assuming a constant reversal potential.

Steady-state activation

The steady-state value and time constant for $m$ were determined from the experiments summarized in Fig. 8A-B. To study steady-state activation, the cell was held at $-60$
mV for an endured period, and then stepped to different holding potentials, each for which the response current ($I_{Ca}$) was recorded (Fig. 8A). Due to the small cellular size, perforated patches was used for recording the Ca$^{2+}$ currents, and the recorded currents were small and noisy. As Fig. 8A shows, the $I_{Ca}$ responses did not follow a characteristic exponential curve towards steady state, as seen in many other experiments. Likely, this was due to $I_{Ca}$ comprising a complex of different HVA channels (e.g., P, Q, R, L-type) which have different activation kinetics [44–46,51–53]. In addition, some in some of the weaker responses $I_{Ca}$ even switched from an inward to an outward current, something that could indicate effects of ER release on the calcium reversal potential. Due to these complications, only the early part of the response was used, i.e., from stimulus onset and to the negative peak value in interval indicated by dashed vertical lines). Voltage-dependent deactivation of Ca$^{2+}$ currents (Fig. 8B) was examined by measuring the tail current that followed after a 5 ms step to 10 mV when returning to voltages between $-10$ mV and $-60$ mV. The deactivation protocol was used to provide additional data points for the activation time constants (see below).

![Graphs](http://dx.doi.org/10.1101/477976)

**Fig 8.** Fitted kinetics for the Ca$^{2+}$ current. (A) $I_{Ca}$ evoked by the activation protocol. (B) $I_{Ca}$ evoked by the deactivation protocol. (A-B) Voltage protocols are shown below the recorded currents, and all panels show a series of experiments (traces). The current-traces were low-pass filtered with a cutoff-frequency of 300 Hz. (C) Voltage dependence of steady-state activation, normalized so that the activation curve had a maximum value of 1 (assuming fully open channels). (D) Activation time constant. Red data points were estimated from the deactivation protocol (B), while blue data points were estimated using the activation protocol (A). Dashed lines in (C-D) denote the kinetics scheme when corrected for a liquid junction potential of $-15$ mV for the experimental conditions used for recording Ca$^{2+}$ currents.

The peak current ($I_{max}$) was recorded for each holding potential in Fig. 8A, and the maximum peak was observed at about 20 mV. By observations, ($I_{Ca}$) became an
outward current when step potentials were increased beyond 70 mV, and based on this we assumed a reversal potential of $E_{Ca} = 70$ mV. Similar to what we did for the Na$^+$ channel, the maximal conductance ($g_{max}$) for each holding potential was computed by the equation:

$$g_{max} = \frac{I_{max}}{(V_{hold} - E_{Ca})}. \quad (26)$$

The estimates of $g_{max}$ for activation are indicated by the crosses in Fig. 8C. The dependency of $g_{max}$ on $V_{hold}$ was then fitted by a Boltzmann curve (Eq 23), with $a = 2$ activation variables, and a good fit was obtained with values for $a$, $k$ and $V^*$ as in Table 3.

**Time constants for $I_{Ca}$ activation**

Assuming two opening gates ($m$), the time constant for $I_{Ca}$-activation was derived by fitting the function [42]:

$$I_{max} = g_{Ca}(V_{hold} - E_{Ca})[1 - \exp(1 - t/\tau_m)^2] \quad (27)$$

to the response curves in Fig. 8A and B. The activation protocol was used to determine $\tau_m$ at high step potentials (from −5 mV and upwards), where the response was not too small and noisy to reveal any clear trend (blue data points in Fig. 8D). The deactivation protocol was used to determine $\tau_m$ for lower step potentials (red data points in Fig. 8D). Like for the Na$^+$ channel, the voltage dependence of the time constants were fitted using the functional form in eq. 25. Good fits to the data points were obtained with the parameter values in Table 3.

<table>
<thead>
<tr>
<th>parameter</th>
<th>activation value</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>-0.128</td>
<td>(ms·mV)$^{-1}$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>-46.7</td>
<td>mV</td>
</tr>
<tr>
<td>$p_3$</td>
<td>19.0</td>
<td>mV</td>
</tr>
<tr>
<td>$p_4$</td>
<td>-101.54</td>
<td>(ms·mV)$^{-1}$</td>
</tr>
<tr>
<td>$p_5$</td>
<td>535.1</td>
<td>mV</td>
</tr>
<tr>
<td>$p_6$</td>
<td>-60.0</td>
<td>mV</td>
</tr>
<tr>
<td>$V^*$</td>
<td>-6.79</td>
<td>mV</td>
</tr>
<tr>
<td>$k$</td>
<td>6.57</td>
<td>mV</td>
</tr>
<tr>
<td>$a$</td>
<td>2</td>
<td>mV</td>
</tr>
<tr>
<td>$V_{lij}$</td>
<td>-15</td>
<td>mV</td>
</tr>
</tbody>
</table>

**Table 3. Parameters for Ca$^{2+}$ activation.** The parameters $p_1$-$p_6$ are used together with Eq. 25 to yield the time constants for steady state activation (in units of ms). The remaining parameters are used together with Eq. 23 to obtain the steady state activation function. The curves obtained in this way describe the voltage dependence under experimental conditions, and was afterwards corrected with a liquid junction potential of −15 mV (see Fig 2B).

**Adjustments of $I_K$, $I_{BK}$ and $I_{SK}$ made in MEDAKA 2**

In MEDAKA 2, $I_K$, $I_{BK}$ and $I_{SK}$ were adjusted in order to obtain a model whose firing pattern was in closer resemblance with the data, both in terms of the control conditions and under application of paxilline. The adjustments are briefly described below, using the simulations in Fig 9 as a reference.

Firstly, $I_{BK}$ activation was shifted by 5 mV relative to RAT and MEDAKA 1. This caused $I_{BK}$ activation to occur closer to the peak, it reduced the effect that blockage of
Fig 9. Firing properties of MEDAKA 2. The response properties of MEDAKA 2 were in good agreement with the experimental data in Fig 1. Under the control conditions, the AP firing rate was 0.7 Hz (A). Partial BK-blockage by paxilline made the cell bursty (B), and the burst shape depended on how large fraction of BK channels that were blocked (gray, yellow, pink, and green curves in C). Regular APs peaked at $-4.4\, \text{mV}$ (blue line in C) and had a duration (taken at half max amplitude between $-50\, \text{mV}$ and AP peak) of 6.4 ms. These values were within the range of peak and duration values observed in the data. As in the TTX-data (Fig 1C), blockage of $g_{\text{Na}}$ abolished AP generation completely (red line in A).

$I_{\text{BK}}$ had on the AP amplitude, and was necessary in order to obtain oscillations during the plateaus following APs (green, yellow and pink curves in Fig 9C).

Secondly, the kinetics of $I_{K}$ in RAT was based on data from rat lactotropes, where $I_{K}$ activation had a fast (3.7 ms) and a slow (30 ms) component [54]. The time constant $\tau_{K}$ in RAT was 30 ms, and thus reflected the slow component. For $I_{K}$ to have an effect on the repolarization of faster $I_{\text{Na}}$-mediated APs, $\tau_{K}$ was reduced to 5 ms in MEDAKA 2, a value closer to the fast component seen in the data [54].

Thirdly, with the maximum value of $g_{\text{BK}}$ in RAT or MEDAKA 1, APs were exceeded by a plateau. $I_{\text{Ca}}$-mediated plateau potentials have not been observed in goldfish [4] or tilapia [5], and was not seen during spontaneous firing in medaka (Fig. 1A-B). To remove the plateaus, $g_{K}$ and $g_{\text{BK}}$ were increased by factors 1.4 and 4, respectively, compared to RAT and MEDAKA 1, which gave MEDAKA 2 narrow regular APs under control conditions (blue curve in Fig 9C).

Fourthly, the faster repolarization obtained by increasing $g_{\text{BK}}$ and $g_{K}$ increased the firing rate of MEDAKA 2 to unrealistically high values. Similar effects of $I_{\text{BK}}$ on increasing firing rates have been seen in other systems [39]. A lower and more realistic firing rate was obtained by increasing $g_{\text{SK}}$ by a factor 3.

With the adjustments described above, MEDAKA 2 responded to partial $I_{\text{BK}}$ blockage in a way that resembled the experiments with paxilline application (compare Fig 9C and Fig 1D).

Software

Experimental current-clamp data (Fig 1), experimental voltage-clamp data (Figs 1, 6, 7, and 8)) and fitted ion-channel kinetics (Fig 2) were plotted using Matlab.
RAT, MEDAKA 1 and MEDAKA 2 were all implemented using the Python interface for the NEURON simulator [55]. MEDAKA 1 and MEDAKA 2 are original for this work, while RAT was based on a previous model [9].

All simulations (used in Figures 3, 4, 5 and 9) were run for 60,000 ms (although briefer intervals were shown in the figures). The first 10,000 ms were discarded to eliminate initial transients, while the remaining 50,000 ms were used in the uncertainty analysis (Fig 5) and to calculate the burstiness factor (Fig 3). Simulations with noise (Fig 3) were run using a fixed time step \( dt = 0.25 \) ms, while simulations without noise were run using adaptive time stepping provided by the NEURON simulator.

The sensitivity analysis in Fig 5 was performed by aid of the Python-based toolbox Uncertainpy [34]. The features considered (Bursting, Regular spiking, and AP firing as defined below) were custom made for the analysis in the current work. Uncertainpy was run using polynomial chaos with the point collocation method (the default of Uncertainpy) and a polynomial order of five. The sensitivity analysis was based on calculating Sobol indices. Only the total order Sobol indices were presented in this work. A total order Sobol index quantifies the sensitivity of a feature to a given parameter, accounting for all higher order co-interactions between the parameter and all other parameters (see [56] or the brief overview in Appendix B of [57]).

The models (RAT, MEDAKA 1 and MEDAKA 2), and the code for generating Figures 3, 4, 5 are available for download (doi:10.5281/zenodo.1491552).

Definitions

Below, the various metrics used throughout this article are defined.

- **AP width**: In control conditions, AP width was defined as the time between the upstroke and downstroke crossings of the voltage midways between \(-50\) mV and the peak potential.

- **Event duration**: A metric proposed by [9], tailored to capture duration of longer events such as bursts. The voltage trace was first normalized so that the minimum voltage was set to 0 and the maximum voltage to 1. The start of an event was defined when the voltage crossed an onset threshold 0.55, and the end of the event was defined when the voltage then crossed a termination threshold 0.45. The duration of the event was the time from onset to termination.

- **Burst**: An AP event with a duration of 60 ms or more was classified as a burst.

- **Burstiness Factor**: The fraction of AP events within a given simulation that were bursts. This metric was most relevant in simulations with noise input (Fig 3). In simulations without noise added, all APs within a given simulation tended to be close to identical, so that the burstiness factor was either 0 or 1 (although there were exceptions).

- **Isbursting**: In the sensitivity analysis (Fig 5), the feature Isbursting was a binary variable that was 1 for simulations that contained bursts, and 0 for simulations that did not.

- **Isregular**: In the sensitivity analysis, the feature Isregular was a binary variable that was 1 for simulations that contained regular APs, and 0 for simulations that did not.
- **Isfiring:** In the sensitivity analysis, the feature *Isfiring* was a binary variable that was 1 for simulations that contained any APs (regular or bursts) and 0 for simulations that did not.

**References**


44. Kay AR, Wong RK. Calcium current activation kinetics in isolated pyramidal neurones of the Ca1 region of the mature guinea-pig hippocampus. J Physiol. 1987;392:603–616.


Experimental Directory Structure (Exdir): An Alternative to HDF5 Without Introducing a New File Format
Experimental Directory Structure (Exdir): An Alternative to HDF5 Without Introducing a New File Format

Svenn-Arne Dragly1,2*, Milad Hobbi Mobarhan1,3†, Mikkel E. Lepperød1,4†, Simen Tønnoe1,5, Marianne Fyhn1,3†, Torkel Hafting1,4 and Anders Malthe-Sørenssen1,2

1 Centre for Integrative Neuroplasticity, University of Oslo, Oslo, Norway, 2 Department of Physics, University of Oslo, Oslo, Norway, 3 Department of Biosciences, University of Oslo, Oslo, Norway, 4 Institute of Basic Medical Sciences, University of Oslo, Oslo, Norway, 5 Department of Informatics, University of Oslo, Oslo, Norway

Natural sciences generate an increasing amount of data in a wide range of formats developed by different research groups and commercial companies. At the same time there is a growing desire to share data along with publications in order to enable reproducible research. Open formats have publicly available specifications which facilitate data sharing and reproducible research. Hierarchical Data Format 5 (HDF5) is a popular open format widely used in neuroscience, often as a foundation for other, more specialized formats. However, drawbacks related to HDF5’s complex specification have initiated a discussion for an improved replacement. We propose a novel alternative, the Experimental Directory Structure (Exdir), an open specification for data storage in experimental pipelines which amends drawbacks associated with HDF5 while retaining its advantages. HDF5 stores data and metadata in a hierarchy within a complex binary file which, among other things, is not human-readable, not optimal for version control systems, and lacks support for easy access to raw data from external applications. Exdir, on the other hand, uses file system directories to represent the hierarchy, with metadata stored in human-readable YAML files, datasets stored in binary NumPy files, and raw data stored directly in subdirectories. Furthermore, storing data in multiple files makes it easier to track for version control systems. Exdir is not a file format in itself, but a specification for organizing files in a directory structure. Exdir uses the same abstractions as HDF5 and is compatible with the HDF5 Abstract Data Model. Several research groups are already using data stored in a directory hierarchy as an alternative to HDF5, but no common standard exists. This complicates and limits the opportunity for data sharing and development of common tools for reading, writing, and analyzing data. Exdir facilitates improved data storage, data sharing, reproducible research, and novel insight from interdisciplinary collaboration. With the publication of Exdir, we invite the scientific community to join the development to create an open specification that will serve as many needs as possible and as a foundation for open access to and exchange of data.

Keywords: file format, data storage, data management, analysis, Python
SIGNIFICANCE STATEMENT

An alternative storage solution that improves on certain drawbacks of Hierarchical Data Format 5 (HDF5) is to use directories in the file system to define a hierarchy, and store data in binary files, and metadata in text files. While this strategy can be deployed in various ways by research groups, no common standard for such a storage solution exists. Experimental Directory Structure (Exdir) is a proposal to standardize this storage solution. We envision the establishment of such a standard and present Exdir to the community as a starting point.

1. INTRODUCTION

Technology development is continuously driving science to new discoveries. In neuroscience, advancements in genetic tools, recording technology, and computer power have paved the avenue to reveal the underlyings of the healthy and diseased brain. Modern neuroscience usually involves recordings and perturbation at many levels, generating a range of data including imaging, electrophysiology, behaviors, perturbations, and molecular biology. Publication of raw data is acknowledged as critical to enable reproducible research and global large-scale collaborative projects and metadata analyses (Nelson, 2009). However, data from different acquisition systems come in a multitude of data formats that need to be readable for all relevant analysis software and stored for long-term archival. Acquisition systems often use proprietary and specialized formats tailored to data produced by specific types of equipment or software. However, these specialized formats have little applicability outside their intended purpose, making them inaccessible for extended use. In contrast, general-purpose formats can store data for multiple types of equipment and software. When based on open standards, general-purpose formats facilitate data sharing.

Hierarchical Data Format 5 (HDF5) (The HDF Group, 1997-2018) is a popular and open general-purpose format capable of storing many large and annotated datasets in a hierarchical structure within a single file. HDF5 is the basis of many formats in neuroscience, including the recent collaborative format, Neurodata Without Borders (NWB) (Teeters et al., 2015). However, issues with HDF5 have recently surfaced in the neuroscience community (Rossant, 2016a), but to the best of our knowledge no formal specification has been introduced. The lack of such a specification limits collaboration through data sharing, and inhibits development of analysis tools. Exdir represents the introduction of a specification that enables novel insight from interdisciplinary collaboration by facilitating reproducible research through improved data storage and sharing. With the publication of Exdir, we invite the scientific community to join the development to create an open specification that will serve as many needs as possible.

2. EXISTING ALTERNATIVES

2.1. Hierarchical Data Format (HDF5)

The HDF5 format holds many advantages over alternative data formats (see e.g., Teeters et al., 2015). However, the HDF5 format also has crucial disadvantages, such as described in Greenfield et al. (2015). In the list below, we have summarized the limitations and challenges from Greenfield et al. (2015) that are most relevant for scientific use along with some additional drawbacks which are addressed with Exdir:

1. Metadata is stored in a binary format which makes it unreadable without tools that read HDF5 files. This also makes the metadata unavailable for text-based command line tools.
2. The specification for HDF5 files is large and complex and there is only one de-facto implementation of HDF5 in C that most HDF5-libraries use. Because of the complex specification, this.

1https://support.hdfgroup.org/HDF5/doc/
2https://github.com/CINPLA/exdir/
implementation is hard to improve by external contributors. Furthermore, the dependency on one large implementation makes it hard to write software which reads and writes HDF5 files in ways that have not been anticipated by the implementation developers.

3. Like all data formats, HDF5 files are susceptible to data corruption. However, because HDF5 stores all data and metadata in a single file, data corruption in one part of an HDF5 file has a chance of corrupting the entire file.

4. Attributes in HDF5 do not support deeply nested structures, like JSON data, YAML data, or Python dictionaries.

5. External version control systems such as Git\(^4\) and incremental backup systems do not work optimally with HDF5 files because all datasets and metadata are stored in a single binary file. This makes it appear as if the entire file has changed when changes are made to a single dataset.

6. Comparing files in binary formats like HDF5 requires specialized tools. However, text-based formats have a wide range of tools that allow line-by-line comparisons, such as `diff` (MacKenzie et al., 2015), and `wdiff`\(^3\), or the graphical tools `meld`\(^5\) and `kdiff3` (Eibl, 2002–2007).

7. Deleting datasets in HDF5 files only removes a reference to the data, while the data remains on disk (except if the dataset is the last remaining object in a page allocated at the end of the file)\(^6\).

8. Raw data from acquisition or analysis is hard to access from external applications when stored inside an HDF5 file. An alternative is to organize raw data in a separate hierarchy outside the HDF5 file, which allows the raw data to be detached from the internal hierarchy and inconvenient to annotate.

### 2.2. Other Formats

Greenfield et al. (2015) propose a new format (Advanced Scientific Data Format, ASDF) to address many of the above problems. Similar to Exdir, ASDF also embraces YAML for metadata, but it also stores and organizes binary data in the same YAML file. Storing the data in one file has the same increased risk of data corruption as HDF5 and makes it harder for version control systems to keep track of incremental changes. Furthermore, ASDF does not provide a convenient way to store raw data in the internal hierarchy.

Some specifications, such as the Brain Imaging Data Structure (BIDS) (Gorgolewski et al., 2016), also approach the above problems by using the file system to define the data hierarchy, which is similar to the solution we propose with Exdir. However, these specifications often serve only the purpose of one or few particular scientific fields, such as neuroscience.

Exdir is not restricted to data from one scientific field and could be used as an alternative where the flexibility of HDF5 is currently enjoyed. Because Exdir has the same abstract data model as HDF5, it should be fairly easy to transition from HDF5 to Exdir for formats based on HDF5 also in other fields, such as geosciences (NetCDF4, Rew et al., 2006; Unidata, 2018), medical imaging (MINC, Vincent et al., 2004), and neutron, X-ray and muon science (NeXus, Könecke et al., 2015)\(^7\).

In Table 1, some of the commonly used open formats in neuroscience are listed. Some of these formats are discussed by Teeters et al. (2015) where they also introduce Neurodata Without Borders (NWB), a format recently developed in an attempt to unify cellular-based neurophysiology data and break down barriers for data sharing. Many of these formats, including NWB, are based on HDF5 and therefore share the same advantages and disadvantages as HDF5. Because Exdir is compatible with the abstract data model of HDF5, these formats could be based on Exdir in the future.

### 2.3. Requirements of a New Specification

We share many of the requirements reviewed in detail by Greenfield et al. (2015) for the ASDF format. To meet the challenges, a data format should:

1. Have an intrinsic hierarchical structure.
2. Be human-readable.
3. Be based on existing standards.
4. Be easy to extend.
5. Have efficient mechanisms to update data.
6. Have support for both text and binary data.

In addition to the above mentioned requirements, we want Exdir to:

7. Minimize the risks and consequences of data corruption.
8. Have a simple, yet flexible specification.
9. Be flexible to data modifications.
10. Be easy to use in ways that have not been anticipated by the authors.
11. Be based on the same abstractions as HDF5 to make it easy to port HDF5-based solutions.
12. Provide a convenient way to store raw data in the same hierarchy.

None of the existing formats known to the authors fulfill all of the mentioned requirements.

### 3. STANDARDS USED IN EXDIR

To fulfill the requirements stated in section 2.3, we propose a new specification, Exdir, which is based on a standardized directory structure and established open-source file formats. The structure follows the abstract data model used in HDF5, but Exdir uses regular file system directories to define the object hierarchy, and stores datasets, attributes, and corresponding metadata in separate files.

Exdir uses YAML files to store metadata and attributes. YAML is a human-readable and human-writable format that supports data types such as strings, numbers, lists, and key-value pairs. Furthermore, libraries for YAML support exist for most major

---

\(^1\)https://git-scm.org

\(^2\)https://support.hdfgroup.org/HDF5/docNewFeatures/FileSpace/RFC-Paged_Aggregation.pdf

\(^3\)https://www.gnu.org/software/wdiff/manual/wdiff.html

\(^4\)http://git-scm.org

\(^5\)http://git-scm.org/software/wdiff/manual/wdiff.html

\(^6\)http://git-scm.org/software/wdiff/manual/wdiff.html

\(^7\)https://support.hdfgroup.org/HDF5/users5.html
TABLE 1 | Overview of commonly used open formats in neuroscience.

<table>
<thead>
<tr>
<th>Name</th>
<th>HDF5</th>
<th>Notes</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>NWB</td>
<td>Yes</td>
<td></td>
<td>Teeters et al., 2015</td>
</tr>
<tr>
<td>Kwik</td>
<td>Yes</td>
<td></td>
<td>Kadir et al., 2014; Rossant et al., 2015</td>
</tr>
<tr>
<td>BRAINformat</td>
<td>Yes</td>
<td></td>
<td>Rubel et al., 2015</td>
</tr>
<tr>
<td>Open Ephys</td>
<td>Yes/No</td>
<td>Binary format specifically designed for electrophysiological data, HDF5 optional.</td>
<td>Siegle et al., 2015</td>
</tr>
<tr>
<td>NeuroShare</td>
<td>No</td>
<td>API to access binary formats and a binary format specifically designed for electrophysiological data.</td>
<td>neuroshare.org</td>
</tr>
<tr>
<td>Neo</td>
<td>N/A</td>
<td>In-memory data format for Python. Uses different formats for file storage.</td>
<td>Garcia et al., 2014</td>
</tr>
<tr>
<td>CARMEN NDF</td>
<td>Yes</td>
<td>Specifically designed for neuroscience. Stores hierarchical structure and metadata in XML files. Data is stored in MATLAB .mat files, which are technically HDF5 files.</td>
<td>carmen.org.uk</td>
</tr>
<tr>
<td>Nix</td>
<td>Yes</td>
<td>Adds a layer on top of the abstract data model that standardizes annotation of data. Directory-based backend in development.</td>
<td>Stoewer et al., 2014</td>
</tr>
<tr>
<td>odML</td>
<td>No</td>
<td>Only applies to metadata.</td>
<td>Grewe et al., 2011</td>
</tr>
<tr>
<td>NSDF</td>
<td>Yes</td>
<td>Format for neuroscience simulation data</td>
<td>Ray et al., 2016</td>
</tr>
</tbody>
</table>

YAML files in Exdir are based on version 1.2 of the YAML specification, but with some additional restrictions that are added because not all features of the full YAML specification are necessary for storing attributes and metadata in Exdir. The restrictions are made to make the format simpler and easier to parse for humans, which we believe improves data sharing. Further, the format should also be easier to parse programmatically, which could open up for the implementation of more efficient parsers in the future. Although some of the restrictions may be removed in a future version of Exdir, we want to start out with a strict subset of YAML and extend only when a clear need is identified for more advanced features. The restricted subset of YAML used in Exdir is compatible with the full YAML 1.2 specification.

The restrictions added to attribute and metadata files in Exdir are listed below. References to individual sections of the YAML 1.2 specification are shown in parentheses:

1. Only tags from the Failsafe, JSON, and Core schemas (sections 10.1–10.3) are allowed, which means that the supported types in YAML files in Exdir are: map, sequence, string, null, boolean, integer, and floating point.
2. Directives must not be used (section 6.8).
3. Node properties must not be used (section 6.9), which also means that explicit or application-specific tags must not be used and anchors must not be used.
4. Complex mapping keys must not be used (sections 2.2 and 8.2.2), which also means that the complex mapping key indicator, “?” is not allowed.
5. String values must not be used in plain style (section 7.3.3), which also means that string values must be enclosed in double or single quotes. However, keys are encouraged to be in plain style (more on key naming convention below).
6. Flow style must not be used for maps or sequences (section 7.4), which means that curly or square braces must not appear outside of string values.
7. Empty keys must not be used (section 7.4)
8. Block scalar styles must not be used (section 8.1), which also means that multi-line strings should be enclosed in double or single quotes.

Writing files with these restrictions may not always be easily done with existing YAML libraries in all programming languages.

---

8http://yaml.org/
8https://github.com/ewiger/yamlmatlab
8yaml.org/spec/1.2/spec.html
We have therefore chosen to read files with the full YAML 1.2 specification in our Python reference implementation and will only issue warnings whenever we encounter YAML files that do not adhere to the above restrictions. We recommend developers of Exdir libraries in other languages to do the same. The intention of the restrictions is not to introduce yet another file format, which could limit the adoption of Exdir, but to ease the transition to a stricter subset of YAML that is easier to parse by both humans and computers.

We also recommend users to adhere to a strict naming convention for keys:

1. Keys should only contain ASCII letters (a–z and A–Z), numbers (0–9), underscores (_), and hyphens (-).
2. Keys should not be surrounded by quotation marks.

This is not part of the Exdir specification because it would conflict with the more relaxed naming rules for keys in HDF5, which in turn could complicate a transition from HDF5 to Exdir in projects where such keys are used. If the first recommendation is broken and custom characters are introduced, we recommend also to break the second rule and add quotes around the key name in question.

JSON was also considered as a format for metadata and attributes, but was rejected because it requires additional tokens such as curly braces to delimit objects and commas to separate key-value maps. While JSON files are human-readable and widely supported, we find the additional tokens to make the files harder to write and maintain manually. The additional tokens play an important role when JSON is used as a serialization format to stream data over network protocols, where they are used to verify that all objects are complete and that the transmission was not interrupted. This is however of limited use in Exdir because we are primarily concerned with creating, editing, storing, and transferring data in bulk, rather than streaming serialized data.

It can be interesting to note that one of the improvements in YAML 1.2 over YAML 1.1 was to make YAML a superset of JSON. However, by introducing our above restrictions, we essentially end up with a subset of YAML that is no longer compatible with JSON. This is a consequence of our emphasis on human readability and simplicity over compatibility with JSON. Further, our restrictions also remove YAML features used to represent arbitrary native data structures, such as explicit tags. This is something we believe is necessary to improve the readability of the attribute files in Exdir. To store complex data structures, we encourage the use of human-readable structures in the attributes. For instance, physical quantities can be stored as a map with the attribute files in Exdir. To store complex data structures, we recommend also to break the second rule and add quotes around the key name in question.

When accessing large Exdir File objects, one can easily retrieve and share subtrees of the main hierarchy by copying the corresponding directories. This reduces memory, CPU, and server-communication costs by keeping the size of data handled to a minimum. When sharing Exdir data with others, one can use readily available compression file formats such as .zip or .tar.gz. Alternatively, the Exdir file can be converted into a HDF5 file, which can be used to exchange data with others (see section 6.3).

In the future, it could be possible to extend the Exdir specification to support additional standard data formats in addition to the NumPy format. It will for instance be of interest to add support for tabular data with named columns, such as CSV. This has however been postponed to a future version because such a format needs to be carefully evaluated based on interoperability, numerical precision, and more. We would also like to receive feedback from the wider scientific community about their needs for storing tabular data before reaching a conclusion.

In order to maintain the simplicity of Exdir and the reference implementation, data consistency verification is not built into Exdir. We envision the use of dedicated software for versioning and consistency, such as git and git-annex. For instance, we currently take care to assure that two processes do not modify the same objects simultaneously. Support for locking and parallel read/write operations to the same objects is planned for a future version of Exdir.

As each dataset is stored in its own directory, the risk of data corruption is reduced. If one dataset is corrupted, it is unlikely to affect the other files in a directory. This separation also makes Exdir avoid the problem of data remaining after deletion in HDF5 and taking up space. Deleting a dataset in Exdir immediately frees up disk space.

11https://docs.scipy.org/doc/numpy/neps/npy-format.html
12https://github.com/kwikteam/npy-matlab
13https://github.com/rogersce/cnpy
14https://git-scm.com
15https://git-annex.branchable.com/
plugins (see section 5.2) can be developed to use version control systems like git to track each object and their checksums in an Exdir directory. This will make it possible to detect when files have changed independently. This also allows Exdir to be combined with git-based systems like GIN, which are tailored toward cloud-based handling of large datasets (Garbers et al., 2017).

4. BASIC STRUCTURE OF EXDIR DIRECTORIES

Exdir has four types of objects, File, Group, Dataset, and Raw, where each is represented as a directory in the file system. Raw is a type of object that is not present in the original HDF5 abstract data model. Metadata for each of these objects is stored in a file named exdir.yaml. All objects can have attributes stored in an optional file named attributes.yaml. Figure 1 shows an example structure of an Exdir File, and a summary of specifications of the data format is shown in Table 2.

4.1. Metadata, Attributes, and Data Files

Metadata for each object is stored in the exdir.yaml file in the object's directory. This file defines that the current directory is an Exdir object, and contains information about the Exdir version and object type. For example, this is the exdir.yaml file of a dataset:

```yaml
exdir:
  version: 1
  type: "dataset"
```

The object type can either be file, group, dataset, or raw. The exdir.yaml file is optional for Raw objects.

User-defined attributes of an Exdir object are stored in that object's directory in the attributes.yaml file. Attributes are stored as key–value pairs, which can be nested:

```yaml
location:
  room: 123
  building: "A"
creator: "James"
equipment: ...
```

Binary data of a Dataset is stored in the NumPy format in a file named data.npy in the Dataset object's directory.

4.2. File, Group, and Dataset Names

Because Exdir stores File, Group, and Dataset objects as directories in the file system, special care has to be taken to adhere to the different filename rules on major operating systems. While file and directory names are case-insensitive in Microsoft Windows, they are case-sensitive on most Linux file systems. If two datasets exist in the same directory with the same name, but different case, e.g., Name and name, then transferring the Exdir directory from a Linux system to a Windows system will result in a conflict.

Datasets and groups at the top level of any Group or File must have unique, case-insensitive names. However, Exdir is case-aware and case-preserving when querying and storing objects, which means that objects must be referenced with the exact case when queried by name.

4.3. File

The File object is the root (top level) object of an Exdir hierarchy. Every directory below a File in the directory hierarchy is part of that File. A File cannot contain other File objects. The metadata of the File is stored in exdir.yaml, and optional attributes in attributes.yaml.

4.4. Group

Inside the File, multiple objects may be stored, among them Group objects. Group objects may also contain any number of other Group objects, Raw objects, and Dataset objects. Group objects are stored as directories in the file system with metadata stored in exdir.yaml, and optional attributes in attributes.yaml. File objects are a specialization of a Group object.

4.5. Dataset

Dataset objects are for storing data. Dataset objects are stored as directories with metadata in the exdir.yaml file, and user-defined attributes in an optional attributes.yaml file. The data within a Dataset is stored in a binary NumPy file named data.npy, and thereby follows the specifications of the NumPy format.

4.6. Raw

Raw objects are used to store data in other formats than the NumPy format. While the user may store any type of data in the a Raw directory it is encouraged to use Dataset objects if possible. For Raw directories the exdir.yaml file is optional. Further, attributes are stored in the optional attributes.yaml file. There is no similar concept to Raw objects in HDF5.

5. REFERENCE IMPLEMENTATION IN PYTHON

We have created a reference implementation of the Exdir specification in Python. This implementation is hosted on Github and is publicly available with an open-source license. It can easily be installed with Anaconda16.

The reference implementation of Exdir owes its relative simplicity to being based on existing formats, and to having a hierarchy based on regular file system directories. It is implemented using the open-source NumPy and PyYAML17 libraries, and is designed to be compatible with the popular HDF5 library, h5py. The compatibility should ease the transition from h5py to Exdir.

The class hierarchy of the reference implementation is shown in Figure 2. The Raw, Group, and Dataset classes inherit from Object, which contains their common methods. The File class is a subclass of Group and they share many of

---

16https://anaconda.org/cinpla/exdir
17http://pyyaml.org
FIGURE 1 | Overview of an example Exdir directory. File, Group, and Dataset refer to objects in Exdir, and are stored as directories in the file system. These objects are equivalent to the same objects in the HDF5 abstract data model. Raw is specific to Exdir and is a regular directory containing arbitrary data files. Inside each directory, there is a file named exdir.yaml with information about the object type and Exdir version. Each object may contain an attributes.yaml file containing user-defined attributes. Inside the Dataset directory is a file named data.npy that contains the data of the dataset stored in the NumPy binary format.

5.1. Overview of the Exdir API in Python
In this section we give a quick overview of the Exdir Python API. An Exdir File is created as follows:

```python
>>> import exdir
>>> f = exdir.File("mytestfile.exdir")
```
TABLE 2 | Exdir format structure.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Contains</th>
<th>Required</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Root object</td>
<td>Group</td>
<td>Raw</td>
<td>exdir.yaml attributes.yaml</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dataset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group</td>
<td>Intermediate directory</td>
<td>Group</td>
<td>Raw</td>
<td>exdir.yaml attributes.yaml</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dataset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataset</td>
<td>Data</td>
<td></td>
<td>exdir.yaml data.npy attributes.yaml</td>
<td></td>
</tr>
<tr>
<td>Raw</td>
<td>Arbitrary data files</td>
<td></td>
<td></td>
<td>exdir.yaml attributes.yaml</td>
</tr>
</tbody>
</table>

FIGURE 2 | Exdir reference implementation class hierarchy.

The File object points to the root directory in the Exdir directory structure. To create a Dataset inside the root directory (or other Group objects) the create_dataset() method can be used:

```python
>>> dset = f.create_dataset("my_data", (100,), dtype="i")
```

Exdir Dataset objects are not NumPy arrays, but behave similarly. They have both a shape and a data type:

```python
>>> dset.shape
(100,)
>>> dset.dtype
dtype('int32')
```

Dataset objects support array-style slicing, which can be used to read and write data to the Dataset:

```python
>>> import numpy as np
>>> dset[...] = np.arange(100)
```

In addition, Dataset objects can also be created from the data directly:

```python
>>> dset2 = f.create_dataset("my_data2", data=np.arange(100))
>>> dset2[0:100:10]
memmap([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Exdir uses NumPy’s memory mapping feature (memmap) to access segments of larger datasets on disk, without reading the entire file into memory. Furthermore, Exdir supports all the operations supported by memmap, including fancy indexing:

```python
>>> dset[dset[:] > 90]
array([91, 92, 93, 94, 95, 96, 97, 98, 99], dtype=int32)
```

A Group can be created using create_group():

```python
>>> grp = f.create_group("subgroup")
```

As with File objects, a Dataset is created inside a Group by using the create_dataset() method:

```python
>>> dset3 = grp.create_dataset("another_dataset", (50,), dtype="f")
```

Group objects support most of the Python dictionary-style interface. You retrieve objects in the file using the item-retrieval syntax:

```python
>>> dset3 = f["subgroup/another_dataset"]
```

As shown above the name of objects follows the hierarchy of the POSIX standard with /-separators. To retrieve the name of any object in an Exdir directory one can use:

```python
>>> dset3.name
'/subgroup/another_dataset'
```

Iterating over a File or a Group provides the names of their members:

```python
>>> for name in f:
...     print(name)
my_data2
subgroup
```

Containership testing also uses names:

```python
>>> "my_data" in f
True
>>> "other_data" in f
False
```

Group objects have the methods: keys(), values(), items(), iter(), and get().

All File objects, Group objects, and Dataset objects can have attributes. Attributes are accessed through the attrs property, which implements a dictionary interface:

```python
>>> dset.attrs["temperature"] = 99.5
>>> dset.attrs["temperature"]
99.5
```
>>> 'temperature' in dset.attrs
True

Unlike HDF5 and h5py, Exdir supports dictionaries as attributes:

```python
dset.attrs["my_attribute"] = {"key1": 
"value1", 
"key2": "value2"}
dset.attrs.items()
```

```
dict_items([('my_attribute ', 
{ 'key1': 
'verue1', 
'key2': 'value2')})
```

After the above commands, the Exdir directory structure becomes:

```
mytestfile.exdir
|-- exdir.yaml
|-- my_data
  | |-- attributes.yaml
  | `-- data.npy
  |-- exdir.yaml
|-- my_data2
  | |-- data.npy
  | `-- exdir.yaml
|-- subgroup
  |-- another_dataset
  |   | |-- data.npy
  |   | `-- exdir.yaml
  `-- exdir.yaml
```

### 5.2. Exdir Plugins

The functionality of Exdir can be extended with plugins. These allow modifying the behavior of Exdir when enabled. For instance, dataset and attribute plugins can perform pre- and post-processing of data during reading and writing operations. Note that plugins do not change the underlying specifications of Exdir.

Plugins are intended to perform verification of data consistency, and to provide convenient mapping from general in-memory objects to objects that can be stored in the Exdir format and back again. Some plugins are provided in the `exdir.plugins` module, while new plugins can be defined by Exdir users or package developers.

One of the built-in plugins provides experimental support for units using the quantities package (Dale, 2017):

```python
>>> import exdir
>>> import exdir.plugins.quantities
>>> import quantities as pq
>>> f = exdir.File("test.exdir", plugins=[exdir.plugins.quantities])
>>> q = np.array([1,2,3])*pq.mV
>>> dset_q = f.create_dataset("quantities_array", data=q)
>>> dset_q[:, ]
array([1., 2., 3.]) * mV
```

As shown in the above example, a plugin is enabled when creating a `File` object by passing the plugin to the `plugins` argument.

To create a custom plugin, one of the handler classes in `exdir.plugin_interface` must be inherited. The abstract handler classes are named after the object type you want to create a handler for. In this example we have a simplified `Quantity` class, which only contains a magnitude and a corresponding unit:

```python
>>> class Quantity:
...   def _ _init_ _(self, magnitude, unit):
...     self.magnitude = magnitude
...     self.unit = unit
```

Below, we create a plugin that enables us to directly use a `Quantity` object as a `Dataset` in Exdir. We do this by inheriting from `exdir.plugin_interface.Dataset` and overloading `prepare_write` and `prepare_read`:

```python
>>> import exdir
>>> class DatasetQuantity(exdir.plugin_interface.Dataset):
...   def prepare_write(self, dataset_data):
...     magnitude = dataset_data.data.
...     unit = dataset_data.data.unit
...     dataset_data.data = magnitude
...     dataset_data.attrs = {"unit": unit}
...     return dataset_data
...   def prepare_read(self, dataset_data):
...     unit = dataset_data.attrs["unit"]
...     magnitude = dataset_data.data
...     dataset_data.data = Quantity(magnitude, unit)
...     return dataset_data
```

The overloaded functions take `dataset_data` as an argument. This has the data, `attrs`, and `meta` properties. The property `attrs` is a dictionary with optional attributes, while `meta` is a dictionary with information about the plugin. In `prepare_write`, the magnitude and unit of the data is translated to a value (numeric or numpy.ndarray) and an attribute (dictionary-like) that then can be written to file. `prepare_read` receives the data from the NumPy file and the attributes from the YAML file, and uses these to reconstruct a `Quantity` object.

We create a plugin that uses this handler as follows:

```python
>>> my_plugin = exdir.plugin_interface.Plugin(
...   name="dataset_quantity",
...   dataset_plugins=[DatasetQuantity()]
... )
```

The plugin is enabled when opening a `File` by passing it to the `plugins` parameter:

```python
>>> f = exdir.File("test.exdir", plugins=[my_plugin])
```
>>> dset = f.create_dataset("test", data=Quantity(1.5, "meter"))

5.3. Converting From Using HDF5 to Exdir
As can be seen from Table 1, many common formats in neuroscience are based on HDF5. Since Exdir follows the abstract data model of HDF5, it is easy to switch from HDF5 to Exdir, and these formats should be able to support both HDF5 and exdir as backends. Often, the only change needed to transition from h5py to Exdir will be to switch from:

```python
import h5py
f = h5py.File("filename.hdf5", "w")
```

To the following:

```python
import exdir
f = exdir.File("filename.exdir", "w")
```

In most cases, the rest of the code can be left unchanged.

A few operators in h5py are missing in the reference implementation and will eventually be added. Furthermore, HDF5 has support for linking of objects, which is currently not part of the Exdir specification and will be added in the future. Finally, the reference implementation currently does not support parallel read/write operations on single subjects. A future plugin is planned to provide such support.

5.4. Reading and Writing to Exdir in Other Languages
It is simple to load and edit Exdir objects in languages with existing NumPy and YAML libraries, such as in MATLAB. Here we show how to read Exdir objects with MATLAB after writing them with Python.

Assume that we have written an Exdir file with the following Python script:

```python
import exdir
f = exdir.File("matlab-test.exdir")
g = f.require_group("group_1")
d = g.require_dataset("dataset_1", data=np.arange(3))
d.attrs["unit"] = 'ms'
d.attrs["trials"] = 1234
d.attrs["frequency"] = 1.23
```

Then, in order to load the dataset as a vector with its corresponding attributes as a struct, one first has to add the path to npy-matlab and yamlmatlab with

```python
addpath(genpath('/path/to/npy-matlab'))
addpath(genpath('/path/to/yamlmatlab'))
```

The data can be loaded into memory with the following code:

```python
data_path = 'matlab-test.exdir/group_1/dataset_1/data.npy';
attrs_path = 'matlab-test.exdir/group_1/dataset_1/attributes.yaml';
dataset = readNPY(data_path)
attributes = yaml.ReadYaml(attrs_path)
```

This results in the following output:

```
data = [0 1 2]
attributes =
trials: 1234
frequency: 1.2300
unit: 'ms'
```

Editing the dataset and attributes is similarly easy:

```python
attributes.name = 'Martin';
yaml.WriteYaml(attrs_path, attributes);
dataset(1:end) = 0;
writeNPY(dataset, data_path);
```

6. TOOLS FOR EXDIR
The Exdir command line interface and the Exdir browser are tools created to make it easier to work with Exdir data.

6.1. Exdir Command Line Interface
Exdir-cli is a simple command line interface for browsing Exdir directories and to create Exdir File objects and Group objects. Listing the content of an Exdir File is done in the command line by the following:

```
$ exdir list mytestfile.exdir

```

Listing the contents of a Dataset is done by the following:

```
$ exdir show dataset
```

6.2. Exdir browser
Exdir browser is a graphical user interface for viewing and editing Exdir directories written in C++ using the open-source Qt application framework (see Figure 3). The browser can be

19 https://github.com/kwikteam/npy-matlab
20 https://github.com/ewiger/yamlmatlab
21 https://www.qt.io/
Dragly et al. Experimental Directory Structure (Exdir)

FIGURE 3 | Screenshot of the Exdir browser.

installed on Linux, macOS, and Windows through Anaconda22 or from source 23.

After opening an Exdir directory, the Exdir browser shows a hierarchical tree of all the objects in that directory. Information about each object is shown when selected and attributes of all objects may be edited. Group objects can be expanded to show their child objects, similar to directories on the file system. When selecting a dataset, the contents is shown in a 2D table. If the dataset is three dimensional, you can select the slice.

6.3. HDF5/Exdir Converter

In order to allow users to convert existing HDF5 files to Exdir, or the other way around, we have created a simple conversion tool24 which can be used from the command line. Converting from HDF5 to Exdir is done using hdf2exdir:

```bash
$ hdf2exdir "filename.hdf5" # creates 
"filename.exdir"
```

And converting from Exdir to HDF5 is done using exdir2hdf:

```bash
$ exdir2hdf "filename.exdir" # creates 
"filename.hdf5"
```

This converter is currently in development, but the main functionality is already implemented, including converting Group, Dataset, and Attribute objects.

22https://anaconda.org/cinpla/exdir-browser
23https://github.com/CINPLA/exdir-browser
24https://github.com/CINPLA/hdf5-exdir-converter

7. PERFORMANCE

As with other formats, the performance of Exdir is limited by the file system and underlying hardware. In general, data readability has been prioritized over performance in Exdir, but we are improving the performance where possible.

We have performed benchmarks for some common operations and compared the Exdir reference implementation to the h5py Python library. The benchmarks can be explored as a Jupyter notebook in the source code repository (see README.md for details) or online using Binder25. This notebook also contains examples that illustrate how the individual benchmarks can be profiled to identify performance bottlenecks.

The results are listed in Table 3. These results were found by performing the benchmarks on a desktop computer running Linux and a laptop computer running Microsoft Windows (see the table caption for details).

As can be seen from “Add 200 attributes (one by one)” in Table 3, adding 200 attributes one by one is slow in Exdir compared to h5py. This is because each written attribute results in a complete rewrite of the “attributes.yaml” file. The performance might be improved by caching the changes and flushing them to the file at regular intervals, but we have chosen

to postpone the addition of such features to keep the current implementation simple. However, as is shown in "Add 200 attributes (single operation)" in Table 3, it is possible to emulate this behavior by first adding the same attributes to a Python dictionary and then assign them to the attrs property of an object. Adding many attributes in a single operation with Exdir is faster than adding them one by one with h5py. It should be noted that is only possible in Exdir, and not supported by h5py.

Further, manipulation of metadata in Exdir has an added benefit over HDF5 on networked file systems if the file system downloads and uploads entire files when they are modified. Metadata in Exdir is stored in separate files, and only these files need to be downloaded, while the rest of the dataset can remain on the server. This is in contrast to HDF5 where the entire file may have to be downloaded.

Reading and writing large continuous data in Exdir is about as fast as with h5py on Linux, and faster on Windows. This is also the case for reading and writing to parts of a dataset. However, HDF5 supports storing chunked data, which is a feature missing in Exdir, and in these cases, HDF5 is likely to outperform Exdir when reading and writing binary data.

Creating many empty objects is slower with Exdir than with h5py, as shown in the “Create 5000 groups (thorough validation)” benchmark in Table 3. Profiling this example on Linux shows that most of the time in Exdir is spent on low-level file system operations, such as file existence. However, the performance on Windows is almost as bad as with thorough validation enabled on Linux. Profiling this benchmark on Windows shows that much of the time is spent on on low-level file system operations, such as nt.open, nt.mkdir, and nt.stat. It therefore seems unlikely that performance can be improved much in this case.

In summary, the performance of Exdir is mostly limited by the performance of the file system and the performance of the YAML and NumPy libraries. Exdir performs worse than h5py with many individual operations on attributes, but performs better if the individual operations are accumulated into a single operation. Exdir performs worse than h5py with many small objects, which means that HDF5 is a better alternative for use cases where many small objects need to be written with high performance. However, when writing large datasets, Exdir performs similarly or better than h5py.

8. DISCUSSION

We have proposed a new specification, Exdir, that puts the abstractions of HDF5 on top of a hierarchical directory structure. Exdir gives the same flexibility as HDF5, but with the advantages of a simpler specification, human-readable metadata, and applicability of established tools. Further, the hierarchy and metadata can be modified manually without tools specific to Exdir, while the data is accessible by existing libraries for common languages. This makes Exdir a possible replacement for HDF5 in computational and experimental data pipelines.

We have presented a reference implementation in Python, a command-line client, and a graphical browser that are all open source and available on GitHub. Together, these tools will hopefully make it easy for other researchers to explore the specification and provide valuable feedback. Because Exdir is based on the established NumPy and YAML formats, we expect APIs for other languages to be fairly easy to implement.

The reference implementation has an extensive test suite and has been thoroughly tested, although the format is still under development. The flexibility of the format gives many possibilities for future development. Exdir includes the concept of plugins, which makes it easy to extend implementations with new functionality without adding more complexity to the specification.
Since different strategies for data storage are already in use, but no formal standard exists, we believe Exdir provides an opportunity for increased data sharing and development of tools that can be shared across multiple disciplines. We hope Exdir can lay the foundation for a standardization of such strategies, and contribute to the general discussion on data storage in science.

AUTHOR CONTRIBUTIONS

S-AD, MH, and ML conceived of and designed the project; S-AD, MH, ML, and ST wrote software, documentation, and the paper. All authors contributed to revising the paper and approved of the final version.

REFERENCES


Conflict of Interest Statement: S-AD is employed part-time by The Qt Company. The Qt Company develops the Qt application framework, which the authors used to create the Exdir Browser.

The other authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Copyright © 2018 Dragly, Hobbi Mobaharian, Lepperød, Ternæs, Fyns, Hafting and Malthe-Sørenssen. This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY). The use, distribution or reproduction in other forums is permitted, provided the original author(s) and the copyright owner are credited and that the original publication in this journal is cited, in accordance with accepted academic practice. No use, distribution or reproduction is permitted which does not comply with these terms.

FUNDING

This work was funded by the Norwegian Research Council (Grant No. 248828, 250259 to MF and Grant No. 231248 to TH) and by the University of Oslo.

ACKNOWLEDGMENTS

The development of Exdir owes a great deal to other standardization efforts in science in general and neuroscience in particular, among them the contributors to HDF5, NumPy, YAML, PyYAML, ruamel-yaml, SciPy, KlustaKwik, NeuroEnv, and Neuroscience Without Borders. We would also like to thank Thomas Wachtler and Daniel K. Wójcik for valuable feedback.
Neuronify: An Educational Simulator for Neural Circuits
Neuronify: An Educational Simulator for Neural Circuits

Svenn-Arne Dragly,1,2 Milad Hobbi Mobarhan,1,3 Andreas Våvang Solbrå,1,2 Simen Tønnes,1,4 Anders Hafreager,1,2 Anders Malthe-Sørenssen,1,2 Marianne Fyhn,1,3 Torkel Hafting,1,5 and Gaute T. Einevoll1,2,6

DOI: http://dx.doi.org/10.1523/ENEURO.0022-17.2017

Abstract

Educational software (apps) can improve science education by providing an interactive way of learning about complicated topics that are hard to explain with text and static illustrations. However, few educational apps are available for simulation of neural networks. Here, we describe an educational app, Neuronify, allowing the user to easily create and explore neural networks in a plug-and-play simulation environment. The user can pick network elements with adjustable parameters from a menu, i.e., synaptically connected neurons modelled as integrate-and-fire neurons and various stimulators (current sources, spike generators, visual, and touch) and recording devices (voltmeter, spike detector, and loudspeaker). We aim to provide a low entry point to simulation-based neuroscience by allowing students with no programming experience to create and simulate neural networks. To facilitate the use of Neuronify in teaching, a set of premade common network motifs is provided, performing functions such as input summation, gain control by inhibition, and detection of direction of stimulus movement. Neuronify is developed in C++ and QML using the cross-platform application framework Qt and runs on smart phones (Android, iOS) and tablet computers as well personal computers (Windows, Mac, Linux).

Key words: app; modeling; neural networks; software; teaching

Methods

History, Teaching, and Public Awareness

Significance Statement

Neuronify, a new educational software application (app) providing an interactive way of learning about neural networks, is described. Neuronify allows students with no programming experience to easily build and explore networks in a plug-and-play manner picking network elements (neurons, stimulators, recording devices) from a menu. The app is based on the commonly used integrate-and-fire type model neuron and has adjustable neuronal and synaptic parameters. To facilitate teaching, Neuronify comes with premade network motifs performing functions such as input summation, gain control by inhibition, and detection of direction of stimulus movement. Neuronify is available from http://ovilab.net/neuronify for smart phones (Android, iOS), tablet computers, and personal computers (Windows, Mac, Linux).

Introduction

Over the past decades, simulation and modeling of neurons have become essential tools in neuroscience.

Although modern software continues to make modeling more accessible (Gleeson et al., 2007), some programming experience is often required. This makes it difficult
for students to explore computational models early in their education. However, educational software applications (apps) allow interaction with computational models without knowledge of programming.

Educational apps have become more common in science, and while such apps exist in many fields, few intuitive and accessible apps have been made for simulating neural networks both on smartphones, tablet computers, and personal computers. Previous efforts have, however, made computational models more accessible. Neurons in Action (Stuart, 2008; Moore and Stuart, 2017) and Neurone (Ali et al., 2017) are examples of interactive tutorials for exploring properties of excitable membranes and neurons, while Emergent (Aisa et al., 2008), SimBrain (Tosi and Yoshimi, 2016), and SpineCreator (Cope et al., 2017) are examples of graphical applications where students can design and analyze neural networks on personal desktop computers.

We have developed an educational app, Neuronify, that goes beyond previous educational tools in that it enables students to easily create neural networks in a plug-and-play manner and is available also on smartphones and tablet computers. It thus provides a new low-threshold entry point to simulation-based neuroscience for students with no programming experience.

With Neuronify, we aim to improve teaching of neural networks and circuits to neuroscience students, by a combination of demonstrating existing circuits, challenging the user with exercises and allowing the user to explore the environment freely. The app makes it easy to teach about complicated network behavior such as direction selectivity based on lateral inhibition (Barlow and Levick, 1965; Fried et al., 2005), where stimuli moving in the non-preferred direction is prevented from inducing firing in the output neuron by a strong, temporally coordinated, inhibitory input volley set up by a tailor-made neural circuit. In the teaching of neuroscience courses, lateral inhibition is one of many examples of networks that are hard to explain with static illustrations. By live visualization of the network, it is possible to explain the process thoroughly by showing how the process works in slow motion. We show how this example is implemented in Neuronify in the Results section (Direction-selective network).

To build and explore neural networks in the app, you drag and drop neurons onto the app’s workspace. The neurons are connected by pulling synapses between them. Once connected, the neurons send a signal to each other whenever they spike. Neurons can also be driven by current sources, spikes generator, and touch and visual input provided via the smartphone, tablet, or computer peripherals. The neurons can be probed by various type of sensors such as voltimeters and spike detectors, and the latter can be forwarded to the loudspeaker. A step-by-step illustration on building a simple circuit is shown in Figure 1. The user can explore how changing the properties of a single cell leads to changes in entire networks. Additionally, the app comes with several premade simulations of common neural network motifs performing functions such as input summation, gain control by inhibition, and detection of direction of stimulus movement. Neuronify runs on smartphones (Android, iOS), tablet computers, and personal computers (Windows, Mac, Linux).

The article is structured as follows. We describe the circuit elements, the integrate-and-fire model, and go into detail on how the app is implemented in Materials and Methods. Then, we show some examples on how Neuronify can be used in Results. Lastly, we make some concluding remarks and discuss future prospects in Discussion.

**Materials and Methods**

In this section, we present the Neuronify workspace, the available circuit elements, and technical aspects.

**Workshop**

The workspace acts as a canvas and is where we expect the user to spend the most time in Neuronify. A simple example network is shown in Figure 2. The circuit elements can be dragged into the workspace and connected to each other.

The workspace is overlaid by a toolbar that contains buttons that activates the (from top to bottom) main menu, the creation menu, the playback controls, and the properties panel. All these menus are seen in Figure 3.

The main menu (Fig. 3A) is where the user can choose between a new simulation, existing simulations, or save and load own simulations. The creation menu (Fig. 3B) is where all the items are found. To add the items to the workspace, the user drags them from the creation menu and drops them onto the workspace. The different items are described in the subsections below.

The playback menu (Fig. 3C) allows the user to change the playback speed of the simulation. It ranges from ~5 ms simulated per 1 s in real time to 50 ms simulated per 1 s in real time. No matter which playback speed is chosen, the temporal resolution of the simulation, however, remains the same. This means that an increase in the playback speed results in a higher computational load for the device running the app.

The properties panel (Fig. 3D) is used to modify the properties of items and connections. This includes properties such as cell membrane resistance, current-source output, and synaptic delay, to name a few. Neurons can also be assigned labels that are used to identify them in other contexts, such as in the voltmeter plot labels (see below, Voltmeter).

**Neurons**

In the current version of Neuronify, two types of integrate-and-fire neurons are available: leaky and adaptive. Further, both types can be either excitatory or inhibitory.

**Leaky integrate-and-fire neurons**

The integrate-and-fire model (see, e.g., Sterratt et al., 2011) is the most commonly used spiking neuron model...
and is a standard part of the curriculum in neuroscience courses with a computational component. It has been demonstrated to be very useful for understanding how neurons process information (Burkitt, 2006). Each neuron is modeled as a point neuron, i.e., the soma and dendrites are assumed to be equipotential. The membrane potential describes the state of the neuron. Without any external input, the membrane potential decays like an RC electric circuit toward the resting membrane potential $V_r$, which is why the neuron is called "leaky."

A spike (action potential) is generated when the membrane potential reaches the threshold potential $V_{\text{thres}}$. When the neuron generates a spike, the membrane potential is reset to its initial potential $V_{\text{reset}}$, which is often defined to be equal to the resting potential $V_r$. After the spike, the membrane potential is fixed to $V_{\text{reset}}$ for an absolute refractory period $\tau_r$. Otherwise, the dynamics of the neuron’s membrane potential is described as (Burkitt, 2006):

$$C_m \frac{dV}{dt} = I_{\text{leak}} + I_{\text{syn}} + I_{\text{inj}}.$$  \hspace{1cm} (1)

Here, $C_m$ is the membrane capacitance, $I_{\text{leak}}$ is the current that drives the decay toward the resting potential, $I_{\text{syn}}$ is the sum of synaptic input currents, and $I_{\text{inj}}$ is the sum of injected currents.

With no synaptic inputs or injected currents, Equation 1 is defined to be equivalent to the equation for an electrical circuit with a resistor and capacitor in parallel (RC circuit). The leak current is therefore defined as:

$$I_{\text{leak}} = -\frac{1}{R_m}(V - V_r).$$  \hspace{1cm} (2)

Here, $V_r$ is the resting potential and $R_m$ is the resistance of the membrane. The membrane time constant is given by $\tau_m = R_m C_m$. Note that both $R_m$ and $C_m$ are assumed to be constant.

In Neuronify, the membrane resistance ($R_m$), membrane capacitance ($C_m$), resting potential ($V_r$), reset potential ($V_{\text{reset}}$), firing threshold ($V_{\text{thres}}$), refractory period ($\tau_r$), and synapse type (excitatory or inhibitory) can be changed in the properties panel. A figure of the leaky neuron spiking is shown in Figure 4A.

Adaptive leaky integrate-and-fire neurons
In many neurons, the firing rate decreases when they receive a sustained input. The standard leaky integrate-and-fire model is not able to reproduce such behavior...
but can easily be extended to incorporate adaptation (Brette and Gerstner, 2005). Here, this is done by adding an additional hyperpolarizing current \( I_{\text{adapt}} \) to Equation 1. The adaptive conductance of this current is incremented by an amount \( \frac{\Delta g_{\text{adapt}}}{\tau_{\text{adapt}}} \), whenever the neuron spikes (Koch, 1999; Latham et al., 2000). In between spikes, the adaptive conductance decays with a time constant \( \tau_{\text{adapt}} \):

\[
\frac{dg_{\text{adapt}}}{dt} = -\frac{g_{\text{adapt}}}{\tau_{\text{adapt}}},
\]

\[
I_{\text{adapt}} = g_{\text{adapt}}(V - V_r).
\]

As the activity in a neuron increases, the adaptive current will also increase due to the growing adapting conductance, making it harder for the cell to fire.

The adaptation time constant (\( \tau_{\text{adapt}} \)), adaptation conductance (\( g_{\text{adapt}} \)), and the synapse type (excitatory or inhibitory) can be changed in the properties panel. The spiking of an adaptive neuron receiving a regular spiking input is seen in Figure 4B.

**Synapses**

The synaptic input to an integrate-and-fire neuron can be modeled in at least two ways: as a conductance-based synapse or a current-based synapse (Sterratt et al., 2011). With a conductance-based synapse model where the current depends on the difference between the membrane potentials and the reversal potential of the synapse, the maximum current is limited. For current-based synapses, there are no such inherent limitations, and the neuron’s membrane potential may increase or decrease without limits. The current-based synapse makes the model easier to analyze and faster to simulate. In Neuronify, connecting two neurons will create a current-based synapse. Current-based synapses are also created when connecting regular spike generators, irregular spike generators, or visual inputs to neurons.

The time course of a synaptic input current is described by a decaying exponential function:

\[
t_{\text{syn}} = \begin{cases} 
I_{\text{syn}} \exp \left( -\frac{t - t_s}{\tau_{\text{syn}}} \right) & \text{for } t \geq t_s \\
0 & \text{for } t < t_s.
\end{cases}
\]

Here, \( I_{\text{syn}} \) is the maximum current (Sterratt et al., 2011).

The maximum current, the synaptic time constant, and the signal delay can be adjusted in the properties panel. Since with current based synapses we risk that the membrane potential goes far beyond the reversal potentials of
the involved ions, we limit the membrane potential to be within the range -90 to 60 mV. These are the reversal potentials for K⁺ and Na⁺, respectively. These limits can be modified or disabled by the user.

**Neuron activators**

Neuronify comes with several neuron activators that can be used to drive neural circuits, including DC and AC current generators, regular spike generators, and irregular (random) spike generators. Input from the user can be used in the form of touch or visual input.

**DC and AC current sources**

The DC current source is an item that, when connected to a neuron, injects constant current into the neuron. The AC current source injects an alternating current with the form of a sine wave. The amount of injected current can be adjusted by the user. The frequency of the sine-wave current can also be adjusted for the AC source.

**Regular spike generators**

The regular spike generator produces spikes with a constant firing rate. Connected neurons will experience these spikes as if they were received as synaptic input from a regularly firing neuron. Connecting a regular spike generator to a neuron creates a current-based synapse, with properties that can be modified as described above in Synapses. The generator can produce both excitatory and inhibitory output, i.e., mimicking afferent inputs both from excitatory and inhibitory neurons.

**Irregular spike generators**

The irregular spike generator produces a train of randomly timed spikes with an average firing rate specified by the user. The spikes follow a homogeneous Poisson process (Dayan and Abbott, 2005). For every time step of the simulator, there is a constant probability that the generator will produce a spike. As for the regular spike generator, the generator can produce excitatory or inhibitory spikes, i.e., mimicking afferent inputs both from excitatory and inhibitory neurons. The synaptic connection is the same as for the regular spike generator.

**Touch activator**

A touch activator makes connected neurons fire when activated. On mobile devices with a touch screen, the sensor is activated by touching it. On desktop versions of the app, the sensor is activated by left-clicking on it with the mouse.

**Visual input**

Visual input is a spike generator based on visual input from a camera connected to the user's device. This mimics a neuron with a visual receptive field (Dayan and Abbott, 2005; Mallot, 2013). There are three types of receptive fields implemented in Neuronify. (1) Rectangular
edge-detecting. This edge-detecting receptive field consists of two adjacent rectangular ON and OFF regions of the same size. The orientation of the ON and OFF region can be adjusted. This field is shown in Figure 5A. (2) Circular center-surround. The field is defined as the difference of two Gaussian functions, a type of receptive field found in the retina and lateral geniculate nucleus (Rodieck and Stone, 1965; Hoffmann et al., 1972). The center type (ON-center or OFF-center) can be set in the setting menu. This field is shown in Figure 5B. (3) Orientation-selective. The field is defined as a Gabor function, a type of receptive field found in the primary visual cortex (Hubel and Wiesel, 1962). The orientation of the field can be adjusted in the setting menu. This field is shown in Figure 5C.

In reality, receptive field neurons have a temporal component such that the response depends not only on the present visual stimulus, but also the stimulus in the recent past (Dayan and Abbott, 2005). In Neuronify, however, the visual input item currently depends only on the instantaneous input.

Sensors
To measure the activity of neurons, Neuronify provides several measurement items: a voltmeter, a spike detector, a firing-rate plot, and a loudspeaker.
**Technical aspects**

Neuronify is developed using the cross-platform framework Qt (Qt developers, 2016) and is written in a combination of C++, QML, and Javascript. C++ is a programming language suitable for high-performance computations, while QML is a programming language for defining visual items in a graphical user interface.

In the following section, we will briefly discuss how to install Neuronify and the implementation details of the app. While this is a brief introduction, detailed information can be found online.

**Installation**

Neuronify is available to download for multiple platforms. The app can be found in the app stores for Android and iOS. For Ubuntu, Neuronify is available as a download in Ubuntu Software. For Windows and Mac, Neuronify is available as a zip file and a dmg image, respectively. While the app is only supported on the above platforms, it should compile and run on any platform supported by the Qt framework. This includes a number of desktop and mobile platforms, in addition to embedded devices. For installation on other platforms or if you intend to make modifications to the source code, please see the next section about building from source.

**Building from source**

Neuronify is open-source software, allowing users to download the source code and make changes to the app. For details about the open-source license, please see the LICENSE file that comes with your copy of the source code. The source code is made available online at http://ovilab.net/neuronify. To obtain the source code, you may either clone the repository using git or download the most recent release as a zip file.

Up-to-date installation instructions can be found in the README.md file in your copy of the source code. Neuronify requires a recent version of Qt to be installed. As of writing, the source code is compatible with Qt 5.7. Once Qt is installed, the file neuronify.pro can be opened in Qt Creator, from which it can be built and run.

**Architecture**

Neuronify has a main engine named GraphEngine. This manages all the items in the simulation and is defined in the C++ class of the same name. The neurons and other items are structured within GraphEngine as nodes in a graph, hence the name. Each connection (or synapse) is handled as an edge in this graph. The behavior of an item or edge is defined by its implementation of certain functions. The most notable functions are stepped, fired, and receivedFire. These functions can be overloaded in either C++ or QML for new items. This flexibility allows for fast prototyping of items in QML while the final implementation can be written in C++ for improved performance.

In addition to fast prototyping, we have made this choice of architecture to allow for a future collaborative feature where the user can share custom items and neuron models with each other. This will be a feature in a future version of Neuronify.

The GraphEngine class is written in C++ and keeps track of all the nodes and edges in the simulation. The nodes are items such as neurons and synaptic connections, while the edges are synapses connecting the items. The GraphEngine class is responsible for moving the simulation forward by calling on all nodes and edges to do a time step. This updating solves the coupled ordinary differential equations for all the cells and synapses. If a cell fires during the time step, it reports this to the GraphEngine, which passes this information to any connected cells in the next time step.

The visual representation of items is defined in QML, a programming language made specifically for the Qt application framework. QML is declarative, which means that the programmer defines logical expressions rather than a sequence of operations. This makes it a good choice for programming graphical items and prototyping neuron models. Dynamic items are defined by their engine. They are implemented by defining the onFired, onStepped, and onReceivedFire signals.

The RegularSpikeGenerator is an example of such a dynamic item which is implemented in QML. It generates a spike with a constant interval, much like a metronome. We defined onStepped to sum up the time since last firing to zero. Below is a simplified definition of the RegularSpikeGenerator engine in QML:

```cpp
NodeEngine {
    property real rate:
    property real timeSinceFiring
}`"
//here we have omitted functions and
//properties for initialization and saving
onStepped: {
  timeSinceFiring += dt
  if(timeSinceFiring > 1.0/rate) {
    fire()
    timeSinceFiring = 0.0
  }
}

While most items are best defined by these functions
directly, Neuron objects share many common properties
and are therefore possible to define using a specialized
engine named NeuronEngine. This engine can have Cur-
tent objects as children. The current property of each
Current object is summed by the engine at each time step.
This sum, together with the synaptic and injected currents,
defines the total current over the neuron's membrane. The
NeuronEngine automatically controls firing by keeping track
of the neuron's voltage. Whenever the voltage goes above

Figure 6. Example of how Neuronify can be used to create interactive illustrations for neuroscience courses. This is a reproduction
of figure 8.5 in Sterratt et al., 2011. The example shows how different levels of current injection into a neuron model results in different
firing rates. Note that this example uses an artificial resting potential of 0 mV.
the firing threshold, the neuron will fire. In addition, the NeuronEngine adds any synaptic input current to the user-defined currents. A user that wants to implement a custom neuron model therefore only needs to define the currents of this model. Below is an example of a QML implementation of a NeuronEngine that defines a leak current:

```qml
NeuronEngine {
    id: engine
    Current {
        id: leakCurrent
        property real resistance: 100.0e6 // ohm
        onStepped: {
            var Em = neuronEngine.restingPotential
            var V = neuronEngine.voltage
            var R = resistance
            var I = -1.0/R * (V - Em)
            current = I
        }
    }
}
```

Figure 7. Example illustrating integration of synaptic inputs. In the upper circuit, the output neuron only receives input from a single presynaptic neuron. This input alone is not sufficient to make the output neuron spike. In the lower circuit, the output neuron instead receives input from three presynaptic neurons. This makes the neuron fire, thus illustrating how a neuron effectively integrates the synaptic input it receives to produce spikes. In the app, this example uses touch sensors instead of a current source for a more interactive illustration of this behavior.

**File format**

The saved simulations are stored in the JSON file format. This allows the use of the JavaScript functions `JSON.stringify()` and `JSON.parse()` to serialize and deserialize the items, respectively. Because the `JSON.stringify()` function would include all properties of a QML item, although not all are interesting to save, we have included a custom class called `PropertyGroup` that contains QML aliases for all the properties to save. This is stored in a list named `savedProperties` on each item. When we iterate all the items that are to be saved, we find all `PropertyGroup` instances in `savedProperties` and run `JSON.stringify()` on these. This turned out to be a very powerful way to add saved properties for new items.

To enable the above defined neuron for saving, we need to add the resistance property. The other properties that already exist on `NeuronEngine` are already enabled for saving by default. We add the resistance property in the following way:

```qml
NeuronEngine {
    id: engine
    savedProperties: PropertyGroup {
        property alias resistance: leakCurrent.resistance
    }
    Current {
        id: leakCurrent
    }
}
```
Once saved, all nodes and edges of the GraphEngine are gathered in the JSON file. In addition, the current version of the file format is saved to ensure the file is read back correctly if the file format has changed. The main structure of a saved file looks like this:

```json
{
  "fileFormatVersion": 3,
  "edges": [
    ...
  ],
  "nodes": [
    ...
  ],
  "workspace": {
    ...
  }
}
```

Here, we have omitted the contents of nodes and edges and the workspace properties for brevity.

**Results**

Here, we present some examples of neural network motifs that can be created with Neuronify. The below four examples can be found in Neuronify together with other premade simulations.

**Textbook example of spike threshold**

The large variety of networks that can be built with Neuronify opens up the possibility to use the app in neuroscience courses. One possible use is as an interactive alternative or addition to traditional illustrations. To illustrate this ability of Neuronify, we have reproduced figure 8.5 from Sterratt et al. (2011).

This example demonstrates how different levels of current injected into a neuron produce different behavior and firing rates. As shown in Figure 6, there are three cases in this example, one which results in no firing, one with a low firing rate, and one with a higher firing rate. It is observed that the level of current injection must be sufficiently high to bring the membrane potential to the firing threshold, otherwise the cell will not fire at all. For currents above the...
Figure 9. Example of direction-selective network. This example illustrates a direction-selective feedforward network based on one-sided lateral inhibitory connections. The upper row of touch inputs are connected to the input neurons. These are both connected to the relay neurons and the inhibitory neurons. Each inhibitory neuron inhibit the relay neuron positioned immediately to the right in the network. The relay neurons are connected to the output neuron. The effect of the inhibition is that the network only responds to input where the touch sensors are pressed sequentially from right to left but not in the opposite direction.
threshold for firing, increased levels of current injection will result in higher firing rates.

The benefit of an interactive example when teaching is that the student, at will, can adjust the level of current injection and properties of the neuron model to explore how this changes the dynamics. These changes are presented in real time to the user, which is better than static illustrations or even figures produced with computational tools where the results are only available once the simulation is completed. With Neuronify, the results are instead immediately accessible to the user.

Integration of synaptic inputs
Most neurons receive synapses from many neurons and require more than one synaptic input to reach threshold and fire. This summation, i.e., integration, of synaptic inputs determines the firing of neurons, and the principle is illustrated by the example in Figure 7.

Feedback inhibition
Feedback inhibition is a key network motif that, for example, may provide gain control in brain circuits. An example of a network with feedback inhibition is shown in Figure 8. Here, a DC current source delivers a constant current to the excitatory neuron labeled “Input”. This neuron is connected to the excitatory neuron A, which again is connected to the neuron labeled “Output”. The output neuron is further connected to the inhibitory neuron B, which finally inhibits neuron A. The overall result is reduced activity both in neuron A and in the output neuron in comparison to the input neuron.

Direction-selective network
Direction-selective neurons are common in the visual system (Cruz-Martin et al., 2014; Liu, 2015) and are expectedly involved in motion detection. One way to create networks with direction-selective neurons is to use feedforward inhibitory connections with lateral connections in one direction only. An example is shown in Figure 9. Here, we have a linear array of input neurons that receive inputs from touch sensors and converge onto a single output neuron through “one-sided” lateral inhibition. The output neuron will respond to a sequential set of touch signals from right to left, but not in the opposite direction. This stems from the network design where the inhibitory neurons provide feedforward inhibition only to the relay neuron placed to the right in the network. Thus, if the sequential touch signal goes from left to right, the relay neurons will already be inhibited when the excitation arrives from the input layer. Therefore, no relay-neuron spikes, and consequently, no spikes in the output neuron, will be generated. However, with a touch sequence from right to left, the inhibition arrives too late to prevent the firing of the relay neurons and the output neuron.

Discussion
In this article, we have presented Neuronify, an educational app that provides an interactive way of learning about neurons and neural networks. In Neuronify the user can add neurons, current sources, spike generators, and sensory input devices to the workspace. This makes it possible for students to create and explore their own neural networks without the need for programming. Students can build intuition for complicated circuits and behavior of neural networks. Neuronify should be a useful tool in many neuroscience courses because a large number of phenomena and networks can be demonstrated with the app. An additional use of Neuronify is as a proof-of-concept software, where the user easily can test the behavior of a simple network before implementing a more complex version in a suitable tool.

We plan to introduce more features in Neuronify in the future. One obvious candidate is synaptic plasticity. In short-term synaptic plasticity (Tsodyks and Markram, 1997), the synaptic efficacy is transiently changed, with typical time constants of less than a second, depending on the detailed pattern of afferent spike trains. In long-term plasticity, long-lasting changes in the strength of synaptic connections are induced, either long-term potentiation (LTP) (Bliss and Lomo, 1973) or long-term depression (LTD) (Ito, 1989). Inclusion of such plasticity into Neuronify would require detection of specific firing patterns and a modification of the synaptic strengths according to specific rules when various spike patterns are detected (Sterratt et al., 2011). While the inclusion of LTP and LTD would be particularly exciting as it would allow the user to create networks that can “learn,” a challenge would lie in visualising synaptic dynamics intuitively. It must be easy for the user to see the change in synaptic strength in addition to the change in network behavior.

New types of neurons can be implemented to explore a wide class of networks such as Izhikevich neurons (Izhikevich, 2003), adaptive-exponential integrate-and-fire neurons (Brette and Gerstner, 2005), or Hodgkin-Huxley type models (Sterratt et al., 2011).

Note that the list of possible new features to include in Neuronify is not exhaustive, nor a guarantee that they will be implemented. Exactly which features will be implemented depends on the feedback we receive.

Online sharing of user-generated networks and items is planned for a future version of Neuronify to foster a community of Neuronify users. This could inspire creativity and allow users to easily search and find networks of interest. It would also be a place where networks specific to a neuroscience course could be uploaded and shared with students.

We are hopeful that Neuronify can be a valuable tool in neuroscience courses around the world and even inspire the creation of other educational tools in neuroscience.

Note Added in Proof: The text “Neuronify is available from http://ovilab.net/neuronify” was accidentally left out of the significance statement in the version of this article that was published on-line on March 9, 2017, as an Early Release Article. Further, a link to the source code was accidentally left out of the “Building from source” section. This has since been corrected.

References
Part III

Appendix
Improvements to Uncertainpy since publication

Uncertainpy has undergone several revisions since Paper I was published and we list most of these improvements here.

The spike detection in the SpikingFeatures class has been improved, and several new options for how the spikes are detected have been added. An option to trim back the extent of a Spike to a given threshold (using the new Spikes.trim() method) now exists. The voltage trace can be normalized to be between zero and one through the keyword normalize. Additionally, options to set the minimum height (min_amplitude) and duration (min_duration) for what should be considered a spike have been added. The spike detection now supports different start and end thresholds using the keywords threshold and end_threshold. This improved spike detection was required in order to replicate the conclusions of Tabak et al., 2011 in Paper II.

The first spike is now discarded if it starts above the start threshold. Additionally, spikes start at the time step before they pass the threshold, and extend to the time step after they fall below the threshold. Support for concatenating two Spikes and printing spikes to screen (Spike.__add__() and Spike.__str__() now exists. A method to print Spikes to screen (Spikes.__str__() has been implemented. Additionally, the method Spikes.plot_voltage() has been added. Spikes.plot_voltage() plots the voltage trace and marks the peak of each spike found.

Several other improvements have also been made. An option for performing the uncertainty quantification and sensitivity analysis for a single uncertain parameter at the time have been added. The NeuronModel class can now load NEURON models from Python files, and not just .hoc files. After user feedback,
an option to disable multiprocessing has been implemented in order to make it easier to use Uncertainpy with computationally costly models that already are parallelized. A script to start a Docker environment with Uncertainpy and all dependencies installed has been added. The `PlotUncertainty` class has been refactored to remove the use of deprecated Matplotlib methods and settings (for Matplotlib 3.0.0).

Lastly, an option of setting `Model(..., **model_kwargs)` when creating a `Model` object has been added, where `**model_kwargs` are any number of arguments which are passed to the model function when the model function is run.

Several bug fixes have also been made:

- Fixed bug where logging to file made Uncertainpy freeze.
- `NeuronModel` is now able to handle empty Python paths, and the default option is an empty path.
- Fixed bug that prevented feature evaluations from being plotted when `Model(ignored=True)`.
- Corrected the default units for the “Spike rate” label when plotting.
- Missing ticks and labels when plotting have been restored.
- We are now using the correct number of zeros in the names of one-dimensional evaluation plots.

New tests have been added to the test suite for each new method added, as well as for most of the use cases where the bugs occurred. The tests also run using the newest version of NEURON (7.6) and NEST (2.16.0). We have made general improvements to the documentation and the documentation has been updated to reflect all the changes above.