Tools and Workflows for Data & Metadata Management of Complex Experiments

---

Building a Foundation for Reproducible & Collaborative Analysis in the Neurosciences

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades
einer Doktorin der Naturwissenschaften genehmigte Dissertation

vorgelegt von
Julia Sprenger
Master of Science in Physics

aus Stuttgart

Berichter: *Prof. Dr. Sonja Grün*
Prof. Dr. Björn Kampa

Tag der mündlichen Prüfung: 14. Februar 2020

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek verfügbar.

List of contributing papers and software projects

The presented thesis is based on the publications and software projects listed below.

Massively parallel multi-electrode recordings of macaque motor cortex during an instructed delayed reach-to-grasp task

by Thomas Brochier, Lyuba Zehl*, Yaoyao Hao, Margaux Duret, Julia Sprenger, Michael Denker, Sonja Grün, and Alexa Riehle*

Published in Scientific Data on April, 10th, 2018 (Brochier et al., 2018).

This publication forms the basis of Chapter 2 and Appendix A and contributed to Zehl (2018). The individual authors contributed to the following aspects of the publication as described by Zehl (2018): "Thomas Brochier designed, set up and performed the experiment and wrote the manuscript. Lyuba Zehl designed and performed the data and metadata management of the experiment, developed and implemented the data and metadata loading and pre-processing routines, wrote the manuscript and designed the corresponding figures. Yaoyao Hao performed the experiment, helped with technical issues of the experimental setup and provided valuable feedback for the manuscript. Margaux Duret was involved in setting up and performing the experiment and corresponding pre-processing steps, and provided valuable feedback for the manuscript. Julia Sprenger was involved in implementing experimental pre-processing steps, supported the implementation of the data and metadata loading routines, and provided valuable feedback for the manuscript. Michael Denker provided valuable feedback for the data and metadata management, was involved in implementing the data and metadata loading routines, and provided valuable feedback for the manuscript. Sonja Grün was involved in writing the manuscript and provided valuable feedback. Alexa Riehle was involved in setting up performing the experiment, performed the spike sorting and provided valuable feedback for the manuscript." In the following, Julia Sprenger took over the further development of the data and metadata pipeline that lead to this data publication and extended it to enable the potential release of additional datasets.

odMLtables: A user-friendly approach for managing metadata of neurophysiological experiments

by Julia Sprenger, Lyuba Zehl, Jana Pick, Michael Sonntag, Jan Grewe, Thomas Wachtler, Sonja Grün and Michael Denker

Published in Frontiers in Neuroinformatics on September, 27th, 2019 (Sprenger et al., 2019).

This publication forms the basis of Chapter 3 and contributed to Chapter 1. The individual authors contributed to the following aspects of the publication:

Julia Sprenger designed and developed the publicly available software including the graphical user interface, testing framework and documentation and contributed to the manuscript. Lyuba Zehl initialized the software project, supervised the software design and gave valuable feedback for the manuscript. Jana Pick designed and implemented an early version of the software. Michael Sonntag and Jan Grewe developed the underlying odML package, contributed to the manuscript and provided feedback to the manuscript. Thomas Wachtler and Sonja Grün gave valuable feedback on the manuscript. Michael Denker was involved in the software design and contributed to the manuscript.

The *Neo* Python Package[1]

The open-source software package *Neo* (Garcia, Guarino, et al., 2014) is the main focus of Chapter 4 with version 0.7.1 being considered here. Among other active *Neo* developers, Julia Sprenger contributed to the release versions 0.5.1, 0.5.2, 0.6.0, 0.7.0 in form of extending the software package to new formats (`NeuralynxIO`, `NestIO`), performance improvement and bug fixes for already supported formats (`BlackrockIO`), testing and feedback of writable formats (`NixIO`), conceptual contribution and feedback on the structural development of the data representation (`RawIO` mechanism, lazy loading, future versions of `ChannelIndex` mechanism), the design and supervision of the development of an extended annotation mechanism (`array_annotations`), development and support of utility functionality and community support. She also contributed to closely related projects like the nix-odML-converter[2].

[1] Neo, http://neuralensemble.org/neo, RRID:SCR_000634

[2] nix-odML-converter, https://pypi.org/project/nixodmlconverter

Using Elephant to construct reproducible analysis workflows of electrophysiological activity data from experiment and simulation

by Michael Denker, Alper Yegenoglu, Andrew P. Davison, Julia Sprenger, Danylo Ulianych, Sonja Grün, Elephant contributors.

Expected submission is end of 2019.

Julia Sprenger contributed tutorial material demonstrating the interaction between *Elephant*, *Neo*, *odML* and *odMLtables*. She developed a pilot structure for the integration of external spike sorting software into *Elephant*, implemented the spike field coherence and spike triggered average functionality and contributed to the maintenance of the software project. Parts of Chapters 4 and 6 will contribute to the this anticipated publication.

Additional related publications not discussed in this thesis:

1024-channel electrophysiological recordings during resting state in macaque visual cortex

by Xing Chen, Aitor Morales-Gregorio, Julia Sprenger, Sacha van Albada, Sonja Grün, Pieter Roelfsema

Expected submission is 2020.

Julia Sprenger supported the data release by supervising the development of the preprocessing and preparation of the datasets.

Summary

The scientific knowledge of mankind is based on the verification of hypotheses by carrying out experiments. As the construction and conduct of an experiment becomes increasingly complex more and more scientists are involved in a single project. In order to make the generated data easily accessible to all scientists and, at best, to the entire scientific community, it is essential to comprehensively document the circumstances of the data generation, as these contain essential information for later analysis and interpretation.

In this thesis, I present two complex neuroscience projects and the strategies, tools, and concepts that were used to comprehensively track, process, organize, and prepare the collected data for joint analysis. First, I describe the older of the two experiments and explain in detail the generation of data and metadata and the pipeline used for aggregating metadata. A hierarchical approach based on the open source software *odML* for metadata organization was implemented to capture the complex meta information of this project. I evaluate the design concepts and tools used and derive a general catalogue of requirements for scientific collaboration in complex projects. Also, I identify issues and requirements that were not yet addressed by this pipeline. There were, in particular, the difficulties in i) entering manual metadata and structuring the metadata collection, ii) combining metadata with the actual data, and iii) setting up the pipeline in a modular generic and transparent manner.

Guided by this analysis, I describe concept and tool implementations to address these identified issues. I developed a complementary tool (*odMLtables*) to i) facilitate the capture of metadata in a structured way and to ii) convert these easily into the hierarchical, standardized metadata format *odML*. *odMLtables* provides an interface between the easy-to-read tabular metadata representation in the formats commonly used in laboratory environments (`csv`/`xls`) and the hierarchically organized *odML* format based on `xml`, which is designed for a comprehensive collection of complex metadata records in an easily machine-readable manner.

Supplementing the coordinated capture of metadata, I contributed to and shaped the *Neo* toolbox for the standardized representation of electrophysiological data. This toolbox is a key component for electrophysiological data analysis as it integrates different proprietary and non-proprietary file formats and serves as a bridge between different file formats. I emphasize new features that simplify the process of data and metadata handling in the data acquisition workflow.

I introduce the concept of workflow management into the field of scientific data processing, based on the common Python-based *snakemake* package. For the second, more recent electrophysiological experiment, I designed and implemented the workflow for capturing and packaging metadata and data in a comprehensive form. Here I used the generic neuroscience information exchange format (*Nix*) for the user-friendly packaging of data sets including data and metadata in combined form.

Finally, I evaluate the improved workflow against the requirements of collaborative scientific work in complex projects. I establish general guidelines for conducting such experiments and workflows in a scientific environment. In conclusion, I present the next development steps for the presented workflow and potential avenues for deploying this prototype as a production prototype to a wider scientific community.

Zusammenfassung

Das wissenschaftliche Wissen der Menschheit basiert auf der Überprüfung von Hypothesen durch Experimente. Da der Aufbau und die Durchführung eines Experiments immer komplexer werden, werden immer mehr Wissenschaftler an einem einzigen Projekt beteiligt. Um die erzeugten Daten für alle Wissenschaftler und bestenfalls für die gesamte wissenschaftliche Gemeinschaft leicht zugänglich zu machen, ist es unerlässlich, die Umstände der Datengenerierung umfassend zu dokumentieren, da diese wesentliche Informationen für die spätere Analyse und Interpretation enthalten.

In dieser Arbeit stelle ich zwei komplexe neurowissenschaftliche Projekte und die Strategien, Werkzeuge und Konzepte vor, mit denen die gesammelten Daten umfassend verfolgt, verarbeitet, organisiert und für die gemeinsame Analyse vorbereitet wurden. Zunächst beschreibe ich das ältere der beiden Experimente und erkläre detailliert die Erzeugung von Daten und Metadaten sowie die Pipeline zur Aggregation von Metadaten. Um die komplexe Metainformation dieses Projekts zu erfassen, wurde ein hierarchischer Ansatz auf Basis der Open-Source-Software *odML* für die Metadatenorganisation implementiert. Ich evaluiere die verwendeten Designkonzepte und Werkzeuge und leite daraus einen allgemeinen Anforderungskatalog für die wissenschaftliche Zusammenarbeit in komplexen Projekten ab. Außerdem identifiziere ich Probleme und Anforderungen, die durch diese Pipeline noch nicht gelöst wurden. Insbesondere bestand die Schwierigkeit darin, i) manuelle Metadaten einzugeben und die Metadatenerfassung zu strukturieren, ii) Metadaten mit den eigentlichen Daten zu kombinieren und iii) die Pipeline modular generisch und transparent aufzubauen.

Anhand dieser Analyse beschreibe ich Konzept- und Tool-Implementierungen, um diese identifizierten Probleme anzugehen. Ich habe ein ergänzendes Werkzeug (*odMLtables*) entwickelt, um i) die strukturierte Erfassung von Metadaten zu erleichtern und ii) diese einfach in das hierarchische, standardisierte Metadatenformat *odML* zu konvertieren. *odMLtables* bietet eine Schnittstelle zwischen den leicht lesbaren tabellarischen Metadatenrepräsentation in den in Laborumgebungen gebräuchlichen Formaten (`csv`/`xls`) und dem hierarchisch organisierten *odML*-Format auf Basis von `xml`, das für eine umfassende Sammlung komplexer Metadatensätze in leicht maschinenlesbarer Form konzipiert ist.

Ergänzend zur koordinierten Erfassung von Metadaten habe ich die *Neo* Toolbox für die standardisierte Darstellung elektrophysiologischer Daten mitgestaltet. Diese Toolbox ist eine Schlüsselkomponente für die elektrophysiologische Datenanalyse, da sie verschiedene proprietäre und nicht-proprietäre Dateiformate integriert und als Brücke zwischen verschiedenen Dateiformaten dient. Ich betone neue Funktionen, die den Prozess des Daten- und Metadatenhandlings im Workflow der Datenerfassung vereinfachen.

Ich führe das Konzept des Workflow-Managements in den Bereich der wissenschaftlichen Datenverarbeitung ein, basierend auf dem gängigen Python-basierten *snakemake* Paket. Für das zweite, neuere elektrophysiologische Experiment habe ich den Workflow zur Erfassung und Verpackung von Metadaten und Daten in einer umfassenden Form konzipiert und implementiert. Hier habe ich das generische neurowissenschaftliche Informa-

tionsaustauschformat (*Nix*) für die benutzerfreundliche Verpackung von Datensätzen mit Daten und Metadaten in kombinierter Form verwendet.

Schließlich evaluiere ich den verbesserten Workflow anhand der Anforderungen an die wissenschaftliche Zusammenarbeit in komplexen Projekten. Ich erstelle allgemeine Richtlinien für die Durchführung solcher Experimente und Workflows in einem wissenschaftlichen Umfeld. Abschließend stelle ich die nächsten Entwicklungsschritte für den vorgestellten Workflow und mögliche Wege vor, diesen Prototyp als Serienprototyp einer breiteren wissenschaftlichen Gemeinschaft zur Verfügung zu stellen.

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

The execution of experiments has accompanied humanity throughout its evolution as a cornerstone in the expansion of its knowledge. In particular due to the ever-growing complexity of research, the long term documentation of experimental procedures has become as necessity for a sound exchange of scientific results. However, this is based on the possibility to record the experimental purpose, execution and findings in an exact, comprehensive manner and without bias of any kind. Nowadays, tedious manual scripture has largely been replaced by digital information, making information easier to be transferred, retrieved and duplicated. Therefore, modern scientific research relies mainly on the acquisition and storage of this digitized data. Although raw recording data can be easily stored and disseminated with modern technologies, interpretation of research data is not straightforward as datasets are highly diverse between, and often even within, scientific areas. This inhomogeneity depends highly on the respective field. In areas that require large experimental setups, such as particle physics or high field FMRI, there are only few data formats established by the community and companies that produce the corresponding tools, e.g., the `root` format (Brun and Rademakers, 1996) and the NIfTI file format. In other fields the diversity of data is larger as scientific methods and objectives require more diverse approaches. Unification would require large-scale coordination within the community and would imply additional overhead on the level of each experiment. A number of initiatives try to screen, collect and evaluate data and metadata approaches within and across communities. Some examples here are the Human Brain Project[1] (HBP), which develops a platform to gather data from all neuroscientific areas. The German national research data infrastructure[2] is a recent initiative for systematic and sustainable research data management within and across disciplines. However, often such a top-down approach focuses on the few common features of datasets, where it is typically already a challenge to define a set of basic minimal metadata to superficially describe a dataset. Much less attention is devoted to the problem of obtaining an in-depth description of a specific dataset that is consistent with similar data. In light of the complexity of such in-depth metadata providing standardized tools and workflows is a prerequisite for the implementation of sustainable

[1] HBP, https://www.humanbrainproject.eu/en/
[2] NFDI, https://www.dfg.de/foerderung/programme/nfdi/

data and metadata management on laboratory level. This permits the integration of data and metadata on a detailed level in a standardized structure, which can later be integrated in large scale infrastructure projects.

The diversity in data modalities and file formats promotes a heterogeneity in data analysis steps and tools used for the extraction of scientific findings. Almost 700 data analysis software tools[3] are registered with SciCrunch[4], a curated repository of scientific resources that includes tools, databases and core facilities with a focus on data analysis in biomedical research. Despite this large set of indexed software tools being searchable as well as citable, many scientific findings have been found to lack reproducibility (Ioannidis, 2005; Ioannidis, 2007; Baker, 2016; Eisner, 2018). One reason for this might be usage of custom code and the neglect of tracking processing and analysis circumstances (provenance tracking). Other reasons might be the inaccessibility of the original data, e.g., due to loss of the original files, deprecated formats or undocumented data selection criteria.

This issue has gained more attention in recent time and started a scientific debate about the need for reproducibility of scientific insights, resulting in a number of publications evaluating the reproducibility of published findings (Fig. 1.1). Within this debate, the terms replicability and repeatability (Plesser, 2018) have been used to describe different aspects of reproducibility. However, the definitions of all three terms are typically limited to a small scientific community, since the restrictions by the methods used (e.g., laboratory equipment versus human subjects versus scientific simulation software) do not permit a direct translation to other areas. This aggravates the communication on this topic across communities and potentially delays the development towards a more mature awareness of reproducibility in some scientific fields.

While reproducibility is a topic that emerged in the scientific literature already in 1990, especially in the neurosciences it has been relatively unattended and is only recently growing in actuality (Fig. 1.1). In addition to the terminology differences described above, the setting of the neurosciences between biology, engineering, physics, chemistry, computer sciences and mathematics might be another reason for this delay. This interdisciplinarity can demand additional communication between neuroscientists and thereby prevent the rigorous questioning of the reproducibility of findings. Another reason might be the relative young age of neuroscience compared to more established scientific fields. This comes with the delayed development of concepts, tools and standards, accompanied by a community awareness of the reproducibility subject.

To address the issue of reproducibility from the perspective of the availability of data, Wilkinson et al. (2016) defined the FAIR principles. These introduce guidelines for handling research data and metadata and are summarized in four key points. Re-

[3]697 results for a query of resource type = 'resource' and 'software resource' and 'data analysis software' and keyword='analysis', accessed on 26.08.2019

[4]SciCrunch, https://scicrunch.org/

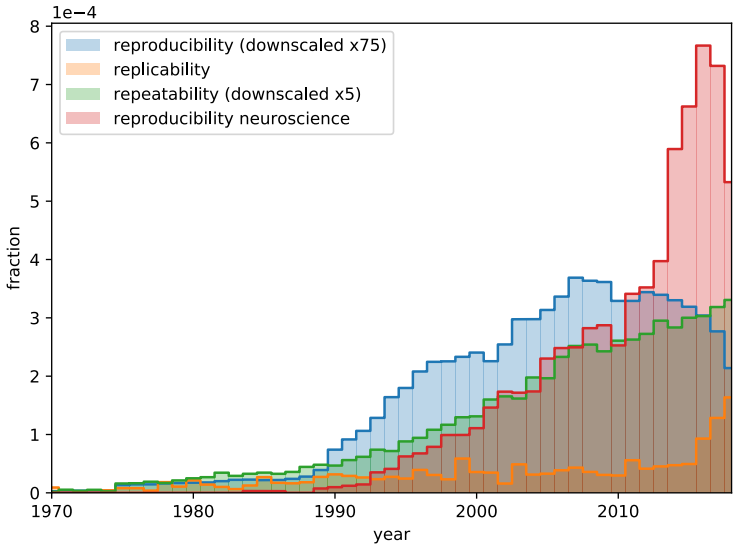


Figure 1.1: Reproducibility related publications. Depicted are publications registered by PubMed$^{\alpha}$ that are related to the keywords reproducibility, replicability, repeatability, and reproducibility in combination with neuroscience. Plotted is the fraction of matching publications per year with respect to the total number of registered publications of the same year. Some fractions related to individual keywords were down scaled for better visualization (see legend). Data were extracted using Corlan (2004). $^{\alpha}$ PubMed, https://www.ncbi.nlm.nih.gov/pubmed/

search data should be made Findable, Accessible, Interoperable and Reusable to be of sustainable value for the scientific community.

In this thesis we present and discuss approaches for data and metadata management in the context of these FAIR principles. We focus on efficient and robust handling of research data from its acquisition to analysis with the aim of easy implementation and the usability by individual scientists as well as laboratory-scale collaborations. The presented examples are set in the field of neuroscience, but many concepts, approaches and tools are of generic nature and can therefore be transferred to other scientific disciplines.

1.1 Data and metadata models

Standardization of data and metadata is a fundamental requirement for the usability of research data. This work is based on two common, generic models for data and metadata representation and storage. Both software projects are developed and maintained by the German Neuroinformatics Node[5] (G-Node), which is an organization that aims to improve the infrastructure for data access, storage and analysis with an emphasis on the field of electrophysiology. These tools form the basis for the data and metadata acquisition workflows presented in this thesis.

[5]G-Node, http://www.g-node.org/

1.1.1 Hierarchical metadata in the *odML* model

The open metadata Markup Language[6] (*odML*) is a versatile hierarchical framework for the representation and storage of metadata (Grewe, Wachtler, and Benda, 2011). While it was originally designed for electrophysiological metadata, its generic structure makes it also applicable to other scientific contexts.

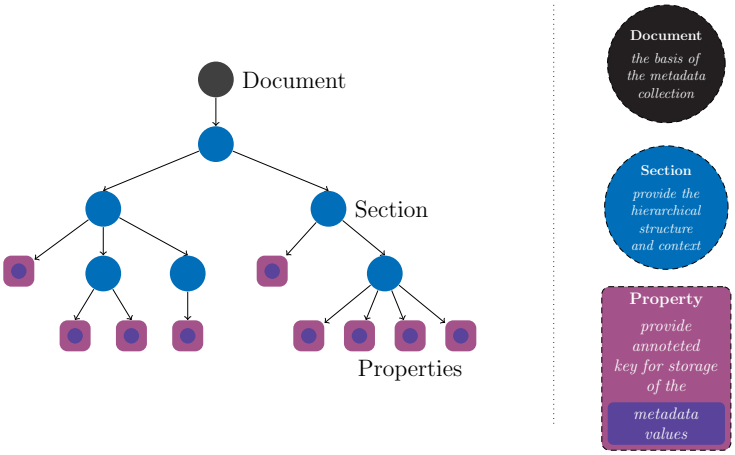


Figure 1.2: *odML* structure and objects. *odML* provides three objects for metadata organization: The *odML* `Document` forms the basis of a hierarchy for metadata storage. It can link to a number of *odML* `Sections`. `Sections` are used to build a hierarchical structure and to provide context and relation between metadata. Each `Section` can link to multiple `Properties`. These contain the actual metadata values accompanied by essential information providing the context for interpretation of the values.

The basic concept is to use a tree-like structure of `Sections` to store metadata as `Properties` (extended key-value pairs) in a common `Document` (Fig. 1.2). The `Document` captures information about the metadata collection: the author of the collection, the date of generation, a custom version specifier and a custom reference repository. The hierarchical structure of the collection is formed by `Sections` which can be concatenated to build the branches of a tree structure (Fig. 1.3). Here, each `Section` carries information about the subset of metadata it contains in form of a `Section name` providing a brief categorization, a `definition` extending on the category typically in form a complete sentence, and a `type` for grouping across `Sections`. Additionally, a `Section` can also point to an external `repository` or `reference` (e.g., a data base) or `link` to or `include` parts of other *odML* structures. The `Property name` provides the key corresponding to the stored metadata values. Each `Property` contains a list of value entries and gathers the corresponding metadata as its `Property` attributes. All context information provided by a `Property`, i.e. data type (`dtype`), physical `unit`, `uncertainty` and `value origin`, is common to all values stored within that `Property`. Similar to the `Section` also the `Property` can carry a human-readable `description` of the values contained and can also `reference` to an external location. The value of a `Property` can

[6] *odML*, https://github.com/G-Node/python-odml, RRID:SCR_001376

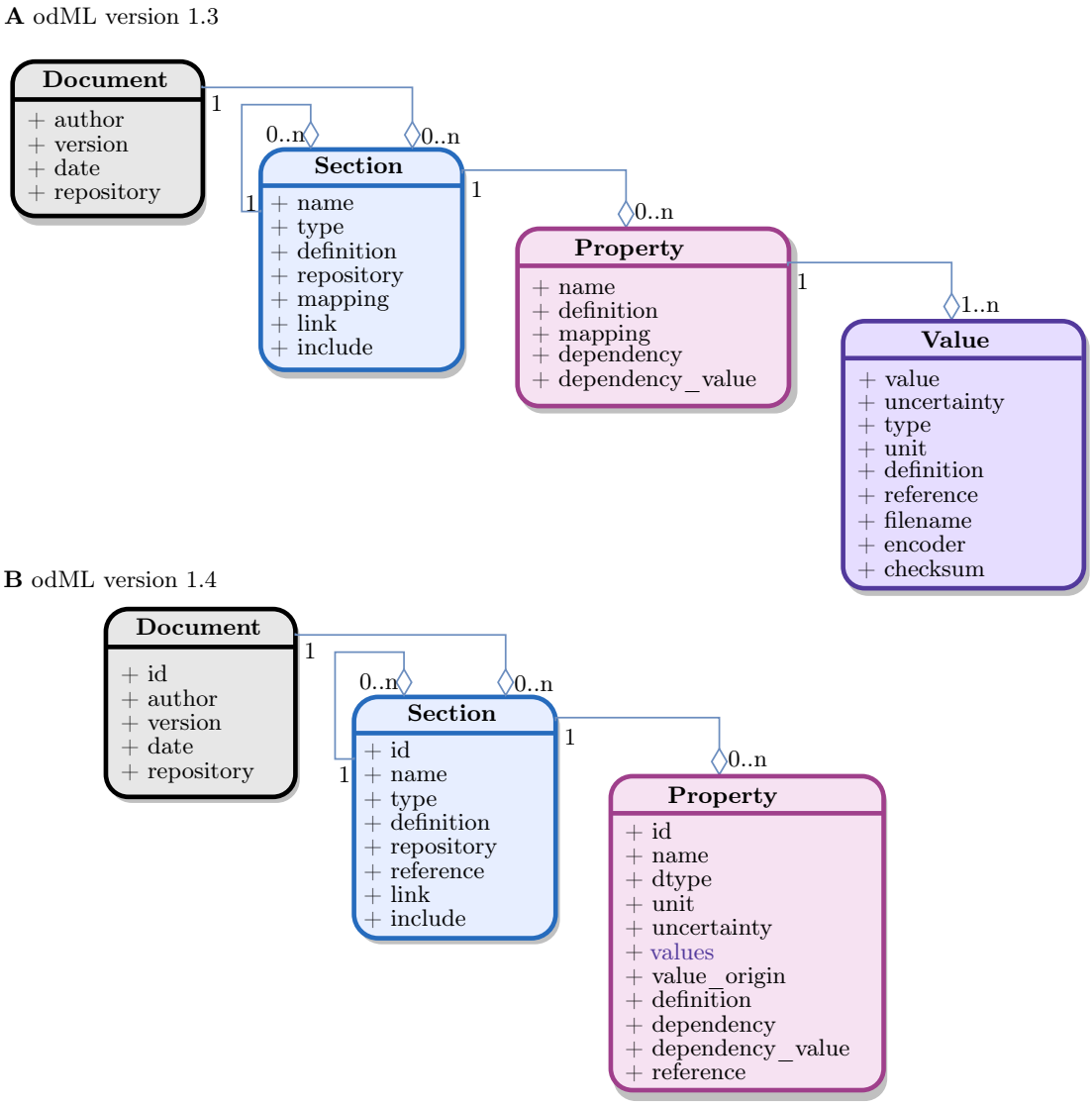


Figure 1.3: *odML* model versions. Illustrated are *odML* version 1.3 (A) and version 1.4 (B). Each box represents an entity defined by the data model and is color coded. Connections between entities are illustrated using the UML aggregation relation where a diamond denotes the target of a connection; the numbers at source and target denote the cardinality of each entity in the connection. `Documents`, `Sections` and `Properties` can link to multiple of their child attributes, whereas each object has exactly one parent objects, generating a branching, hierarchical structure. The `Document` additionally contains attributes to store the `author`, the `version`, the generation `date` and the corresponding `repository` reference. The `Section` is a container for its child `Sections` and `Properties`. The `Property` name acts as key associated to the actual metadata value stored. In *odML* version 1.3 the value information is stored in dedicated `Value` objects, whereas in in *odML* version 1.4 this feature is integrated into the `Property` object. Here, the `Property` provides supplementary essential information for the interpretation of the metadata value which was previously stored in the dedicated `Value` object in *odML* version 1.3. In *odML* version 1.4 each object has an identifier (`id`) for unique identification across files.

also depend on another `Property` (`dependency`) or another value (`dependency_value`). All *odML* objects carry a universally unique identifier for unique identification of odML entities even across unrelated files to ensure comprehensive provenance tracking. This permits the referencing and inclusion of *odML* objects across files and projects.

Based on the presented *odML* objects, we can design a small example structure for capturing metadata of an experiment involving a subject and a force recording device (Fig. 1.4). For example, the metadata can be grouped on a first level of `Sections` into hardware related or non-hardware related metadata. Here, this implies the generation of two first level `Sections`, one for the description of the subject and one for the description of the recording device. On the next level each of these groups can be separate more detailed aspects of the experiment. Here, we only track two aspects of the recording device: the upper limit of the force that can be recorded (`Property` with `name` 'Maximum Force') and the supported sampling rates of the device (`Property` with `name` 'Sampling Rates'). The corresponding value entries to the keys provided by the `Property names` are of type `list` as *odML* supports the capture of multiple values belonging to a `Property`. In this example four different sampling rates are supported, sharing the data type, unit, description and all additional attributes of the `Property`. The subject is described by two `Properties`, the species and its weight, which are accompanied by the required data type specification and an optional corresponding unit specification. Finally, we also track metadata about the training the subject in a separate subsection of the `Section` that describes the subject. Here, the training is solely defined by a start and end date, captured in two corresponding `Properties`.

This small example demonstrates how *odML* objects can be used to build a hierarchical metadata structure and group information in a logical way. The additional attributes of `Sections` and `Properties` provide contextual information for the plain metadata values and are essential for the interpretability of the metadata collection. The same concepts as presented here can be used to build full-sized metadata collections capturing metadata of complex experiments (see Chapter 3). For example in the presented example next steps could comprise the addition of more information about the subject in additional `Properties` to capture the age, gender, handedness, etc or add additional `Sections`, e.g., for describing details related to the recording data (recording date, filenames, etc) or preprocessing steps (filtering, offset removal, etc).

Model revisions The projects presented in this thesis rely on different versions of the *odML* library. Here we present the main differences between the *odML* version 1.3 to and the current *odML* version 1.4. In order to simplify the usage of the *odML* framework two major changes were introduced in *odML* version 1.4. The first change was the merging of `Value` and `Property` entities (compare Fig. 1.3A and B). Previously, each `Property` required at least one child `Value`. In version 1.4. this restriction is lifted, as `Properties` can contain an empty list of values. The merge of the two objects prevents value ambiguities within a `Property` and reduces the effective file size since the value dependent attributes ("unit", "uncertainty", "data type" and "reference") are defined

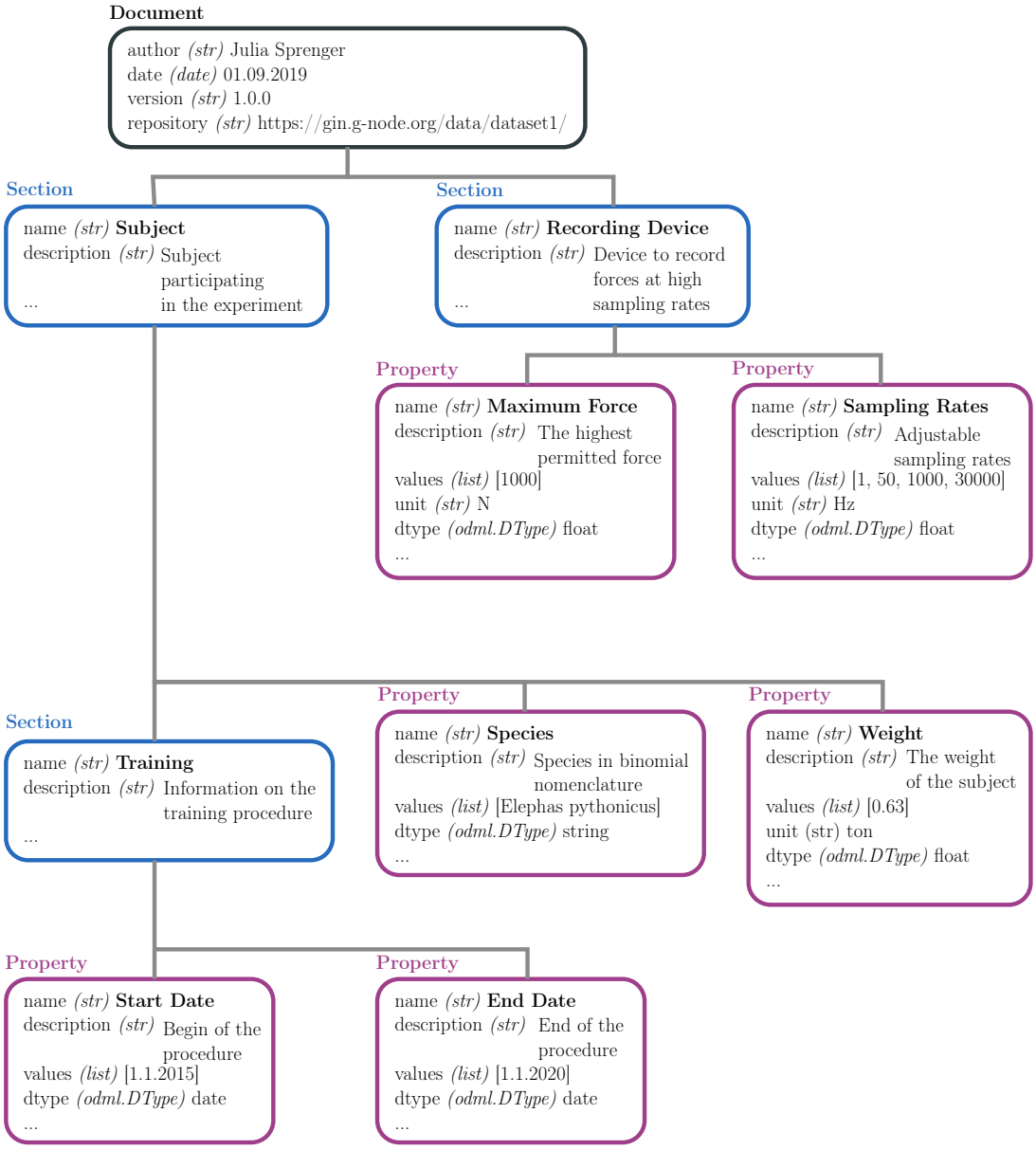


Figure 1.4: Example *odML* structure. The *odML* structure contains metadata from an experiment involving a subject that generates a force. The metadata related to this collection itself are denoted in the *odML* `Document`, e.g., the author. The two top-level `Sections` separate subject and recording setup. Here, the recording setup is characterized by two `Properties`: the maximum recordable force and the supported sampling rates, consisting of a list of values. The subject is characterized by its species and weight. The training information is described one level below in a `Section` named Training. Here, the training is characterized solely by a start and end date.

only once for a set of values. Second, *odML* entities now contain a universally unique identifier ("id"), an auto-generated identifier with extremely low collision probability. Compatibility for odML files using the old format version is ensured via automatized conversion functionality.

Additional features The *odML* core library provides an in-built mechanism to search and retrieve `Sections`, `Properties` or values within a `Document`. The need to consistently search for metadata entities across `Documents` from different sources led to the development of an export feature of *odML* metadata to the Resource Description Framework (RDF) format[7], a general and widely used storage format of graph databases. Multiple *odML* files exported to RDF can be loaded into any graph database supporting RDF and will be combined into a single graph. Moreover, while XML is the default storage format, *odML* additionally supports storing the metadata in the text based file formats JSON[8] and YAML[9]. JSON is a de-facto data exchange standard between web based and standalone computer applications. The support of JSON makes *odML* metadata more easily consumable in machine-only workflows through modern applications. Since both XML and JSON primarily aim at machine-readability, their structure is not easily readable by humans. *odML* also supports the export to the YAML file format to provide a human readable text format of the raw metadata files.

For visualization and manipulation of metadata files, *odML* comes with a native *odML* GUI (odml-ui[10]). The GUI provides a visual representation of the hierarchical structure for navigation and editing of individual metadata entries.

1.1.2 Generic data organization via the *Nix* model

The *Nix* model is a format to store and represent combined data and metadata in a common framework. For this six generic data objects are defined and combined with an *odML* based metadata structure. The *Nix* model is provided with a C++ reference implementation[11] and bindings for Java and *Matlab*. An independent Python implementation is provided[12] with version 1.5.0*b*3 being considered here.

The *Nix* model consists of six data and two metadata objects described in the following (Fig. 1.5). Data values are captured using `DataArrays` capable of describing any type of data that can be represented using a single or multidimensional array. In addition to the values, the `DataArray` also describes the physical properties of the stored data, e.g., the type of data, the physical unit and a human readable label. Additionally the data array is connected to `Dimension` objects that provide detailed information about each of the associated dimensions of the `DataArray` including a label, the physical unit, a sampling interval and offset. With these two objects *Nix* captures all required data for a meaningful visualization of the stored data values (e.g., see Fig. 1.6). In

[7] http://www.w3.org/TR/rdf-primer
[8] https://json.org
[9] https://yaml.org
[10] https://github.com/G-Node/odml-ui
[11] nixio *C++*, https://nixio.readthedocs.io, RRID:SCR_016196
[12] nixio / nixpy, Python, https://pypi.org/project/nixio/

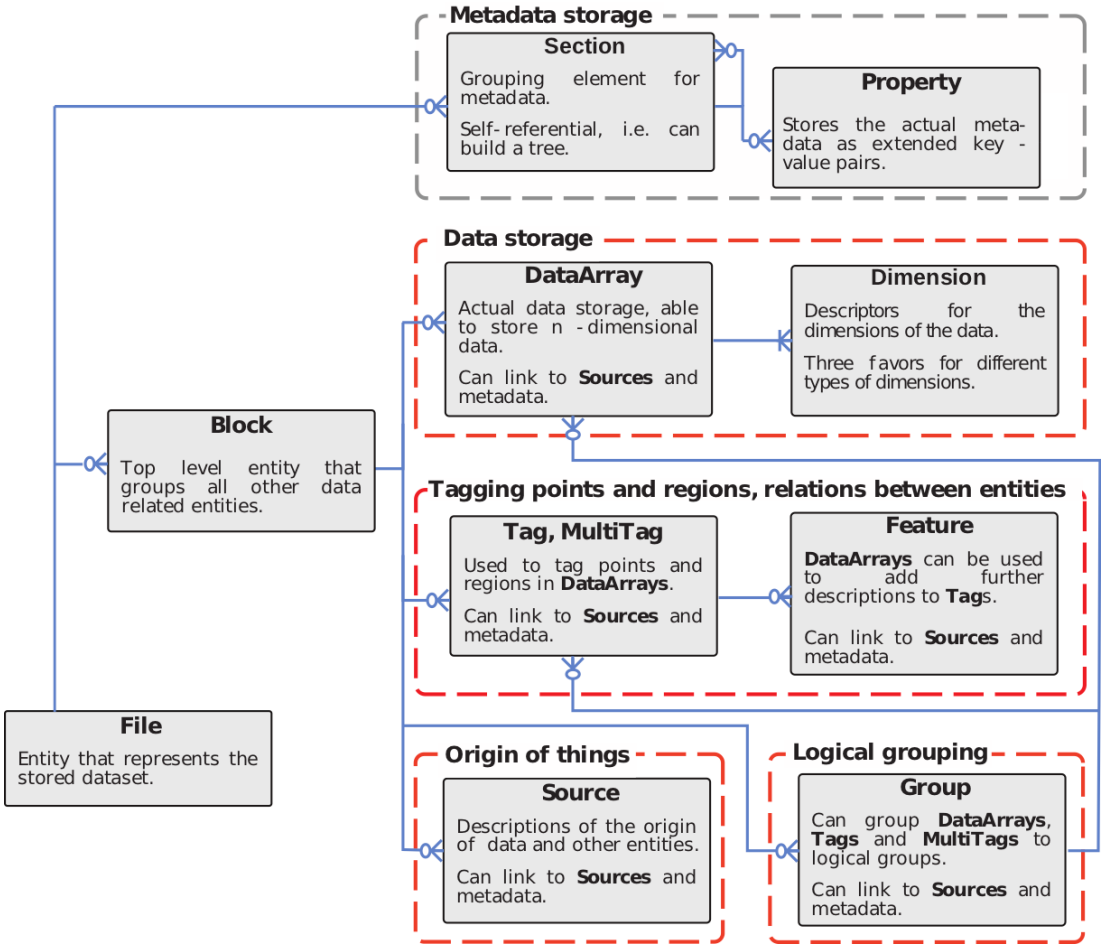


Figure 1.5: *Nix* model objects. The model consist of objects for storing data and metadata and relations between these. Metadata is captured in a *odML* based structure. Additionally, six objects are implemented to capture data and relation between these. The main data object (`DataArray`) stores multidimensional data and captures the physical attributes of the data using `Dimension` objects. `Tag`s and `Multitag`s are used to label a subset of the data contained in a `DataArray` and can provide supplementary information using `Feature` objects. Furthermore, `Group` objects can be used to represent logical relations between objects and `Source` objects provide background information about the origin of the data. `Block`s and `File`s represent a complete dataset and file, respectively. Each of the data objects (except `Dimension`s) can link to a corresponding metadata `Section` providing additional information specific to that data object. Figure adapted from *Nix* documentation (https://nixio.readthedocs.io/en/latest/data_model.html, accessed Aug 2019).

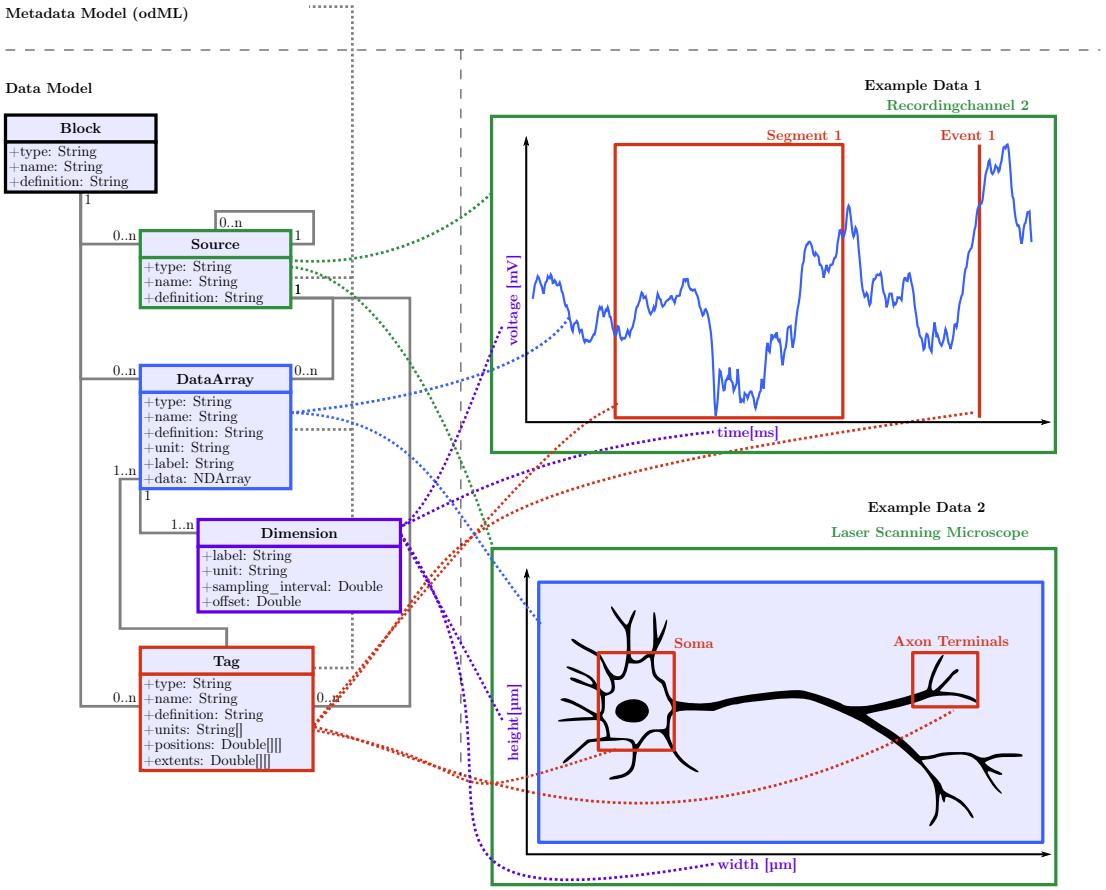


Figure 1.6: *Nix* model application examples. The model can capture different varieties of data, e.g., electrophysiological recording traces (example 1) and imaging data (example 2). Both signals types can be described by the same *Nix* object types. Figures adapted from *Nix* the documentation (https://github.com/G-Node/nix/wiki/The-Model).

addition, subsets of the data stored in a `DataArray` can be tagged via a `Tag` or `Multitag` object. These objects can be used to provide more information about a particular subset of values, e.g., mark the time points of stimulus presentation in a continuous recording signal (Fig. 1.6). Furthermore, `Source` objects can be used to track the origin of data, e.g. relate a downsampled signal to the original signal. `Group` objects can be used for logical grouping of other *Nix* data objects. All of these objects are coordinated via `Block` objects, which again are, together with the metadata objects, contained in a *Nix* `File` object that represents a complete dataset. The metadata objects used in the *Nix* framework are adopted from the *odML* framework. All data objects within *Nix*, except for the `Dimension` object, can link to a single `Section` in the metadata collection of the *Nix* `File`, which contains additional information about the data object. Depending on the declared types of the linked data and metadata object, this relation is interpreted uni- or bidirectional, i.e. the metadata `Section` provides details about the data object or the metadata object is additionally defined via the data object.

Nix is accompanied by detailed user-level documentation in form of an extensive

wiki[13] and online documentation[14] including tutorials and demos. Furthermore, the *Nix* model is natively integrated in the electrophysiology recording system *RELACS*, the *EEGbase*[15] a system for storage, management, sharing and retrieval of EEG data as well as *Neo*, a Python tool for standardized representation of electrophysiology data (Chapter 4).

1.2 Thesis overview

In Chapter 2 we describe two published datasets of a complex, electrophysiological experiment including an extensive metadata collection used in a collaborative setting. We describe the process of data and metadata preparation required for the data publication and discuss the pipeline used in this publication to identify strengths and shortcomings of the presented approach. In Chapter 3 we present *odMLtables*, a tool that facilitates the collection of metadata in the standardized *odML* format and that emerged from the implementation of the previously presented pipeline. We demonstrate the embedding of *odMLtables* in a real-world metadata application and highlight the features of the tool. Chapter 4 complements the previous section by introducing tools for standardized data representation and demonstrates their application in the context of data and metadata handling based on three example scenarios. Chapter 5 goes beyond the pipeline approach presented in Chapter 2 by introducing the concept of modern workflow management software for the efficient organization and structuring of scientific projects. We demonstrate the integration of the previously presented tools in a systematic fashion using modern workflow management software to coordinate the application of data and metadata software in a neuroscientific project. Finally, in Chapter 6 we discuss the presented approaches and provide an outlook on future development of the tools and concepts.

[13] *Nix* wiki, https://github.com/G-Node/nix/wiki
[14] *Nix* documentation, https://nixio.readthedocs.io
[15] EEGbase, http://eegdatabase.kiv.zcu.cz, RRID:nif-0000-08190

Chapter 2

Sharing data
- Publication of two complex electrophysiological datasets

Scientific progress is the result of scientific findings that build on each other. To validate such findings, typically hypotheses are proposed and tested against experimental data using statistical methods. The resulting finding and interpretation is communicated to other scientists via the publication of a manuscript. In recent years, however, the practice of publishing papers has been criticized for a lack of robustness. Attempts in a number of scientific fields including life sciences, among others, to reproduce published scientific findings failed to support the same conclusions (Baker, 2016; Fidler et al., 2017; Pashler and Wagenmakers, 2012; Ioannidis, 2005; Goodman and Greenland, 2007; Ioannidis, 2007; Open Science Collaboration, 2015). To qualitatively distinguish between different levels of reproducibility, a collection of terms has been applied: reproducibility, replicability, repeatability (Plesser, 2018; Drummond, 2009). However, the specific interpretation of each of these terms highly depends on the scientific field they are applied in. Repeatability, for example, can be defined as the re-performance of the same experimental study using identical components. However, for computer sciences repeating an experiment might be re-running of the same software simulation on the same machine using the same software packages. In consequence, the feasibility of repeating a study highly depends on the field. For example repeating a study by running the same code on the identical machine might be a realistic project, whereas running the identical experiment in psychology is not (Anderson et al., 2016). A widely accepted version of reproducibility terminology by Goodman and Greenland (2007) is therefore omitting the terms 'replicability' and 'repeatability' for simplification and defines instead three different types of reproducibility (from Plesser, 2018):

Methods reproducibility

'provide sufficient detail about procedures and data so that the same procedures could be exactly repeated.'

Results reproducibility

'obtain the same results from an independent study with procedures as closely matched to the original study as possible.'

Inferential reproducibility

'draw the same conclusions from either an independent replication of a study or a reanalysis of the original study.'

The definition of 'methods reproducibility' highlights a central aspect of scientific publications: the underlying data. Reproducing findings of publications is difficult for older publications (Vines et al., 2013; Rostami et al., 2017), which to a large part can be accounted for by the decreasing ability to retrieve of the original datasets. The obligation to provide research data to other scientists is a practice promoted by increasingly many funding organizations (e.g. NIH since 2003[1], EU Horizon 2020 Open Research Data Pilot[2] since 2014) and for publications in a number of journals (e.g. Springer Nature[3]), however, the readiness to comply is rather low (1/10 publications provides research data on request (Savage and Vickers, 2009)). To address the problem of methods reproducibility for original research articles, a number of new journals were put into place, which invite to publish replications of the original articles in computational science [4], economics[5], psychology[6] and neuroscience (Yeung, 2017), as well as original research data (e.g. ScientificData[7]).

To improve the situation of data availability, Wilkinson et al. (2016) defined the FAIR principles, a set of guidelines to achieve sustainable data and metadata management. The four key objectives are to make research data

F indable by generating unique and persistent identifiers for data and metadata files. This includes the generation of a comprehensive metadata collection, clearly linked to the data via the identifier and vice versa. This way, datasets and corresponding metadata can be registered and indexed and are findable via search queries.

A ccessible by making data and metadata retrievable via their unique identifier in a standardized way. The applied standards and protocols should be open, free and universal and the accessibility of either data or metadata should not depend on the availability of the other.

[1] https://grants.nih.gov/grants/policy/data_sharing/

[2] https://ec.europa.eu/research/participants/docs/h2020-funding-guide/cross-cutting-issues/open-access-data-management/data-management_en.htm

[3] https://www.springernature.com/gp/authors/research-data-policy/data-availability-statements/12330880

[4] ReScience, http://rescience.github.io/

[5] http://www.economics-ejournal.org/special-areas/replications-1

[6] https://www.apa.org/pubs/journals/xge/replication-articles

[7] ScientificData, https://www.nature.com/sdata/

I nteroperable by using standardized, broadly applicable tools and languages for knowledge representation. This includes using FAIR terminology and providing qualified references to other metadata.

R eusable by providing comprehensively described data and metadata including a clear license statement and detailed provenance information using community standards.

Complementing these guidelines, a number of sites have become available for hosting scientific data (Zenodo[8], figshare[9], Pangaea[10], BMC Research Notes[11], DataDryad[12], GIN[13], EuDat[14], Research Data Australia[15], see also Assante et al. (2016)) and publishing data descriptor papers (ScientificData[16], DataScience[17], see also Candela et al. (2015)). It has been shown that mandated data archiving upon publication highly improves data availability (Vines et al., 2013). However, a central prerequisite for the usability of the dataset is the existence of a comprehensive metadata collection (Ferguson et al., 2014; Parekh, Armañanzas, and Ascoli, 2015; Ascoli et al., 2017). Going one step further, Chen et al. (2019) show that making only the data (and metadata) available is not sufficient for reproducible science but rather, data also need to be accompanied by software packages used for further analysis. In the optimal case this encompasses a detailed, step-wise description of the motivation for an analysis workflow. Jomhari, Geiser, and Bin Anuar (2017) demonstrate that even for datasets surpassing commonly available hardware capabilities (e.g., due to memory limitations), e.g. from particle physics, making the effort of publishing the data is possible and can lead to successful reuse of the data.

The publication of a dataset adhering to the FAIR principles therefore involves a number of concepts, but also technical tools to enable this process. To demonstrate what this entails in practice, here we report on our initial efforts to establish such knowledge for electrophysiology data in the context of the workflows established for a concrete data publication, and discuss critically that implementation. We provide an example of two complex published datasets from the field of neuroscience (Brochier et al., 2018), demonstrating the richness of data and metadata involved in such an experiment and the concepts to prepare the data for analysis and publication. We present the pipeline used for metadata aggregation and discuss its strengths and shortcomings. Parts of the conceptual underpinnings of the pipeline are published in Zehl et al. (2016) where they serve as example cases for metadata handling practices. In the following we describe the published datasets based on Brochier et al. (2018).

[8] Zenodo, https://zenodo.org/
[9] figshare, https://figshare.com/
[10] Pangaea, https://www.pangaea.de
[11] BMC Research Notes, https://bmcresnotes.biomedcentral.com/ & https://bmcresnotes.biomedcentral.com/about/introducing-data-notes
[12] DataDryad, https://datadryad.org/
[13] GIN, https://gin.g-node.org/
[14] EuDat, https://eudat.eu/
[15] RDA, https://www.ands.org.au/online-services/research-data-australia/rda-registry
[16] ScientificData, https://www.nature.com/sdata/
[17] DataScience, https://datascience.codata.org/

In Brochier et al. (2018) we publish two complete high-dimensional electrophysiological datasets containing Utah Array recordings from monkey motor cortex. Recording subjects are two macaque monkeys (monkey L and N) which performed a complex instructed delayed reach-to-grasp task. Each recording contains continuously sampled data from 96 electrodes accompanied by the corresponding spiking activity. Spikes were extracted during online recording and offline sorted using the *Plexon Offline Sorter*[18]. Electrophysiological data are accompanied by a number of behavioral and control signals. Detailed metadata were aggregated and are provided in the *odML* format (see Chapter 1 and Section 2.3).

Complex datasets, as the two provided here (high-dimensional, multi-scale during complex behavior), are a challenge for performing reproducible analysis. Besides the often rather variable nature of the circumstances under which such data were recorded, the data additionally experience a number of often interactively performed preprocessing steps before they can be used in actual data analyses. Without a detailed knowledge about all these steps, the actual data analysis may be biased or strongly affected. In most cases, electrophysiological data available as open source are in this respect not sufficiently annotated and documented. For this reason, we provide here a comprehensive description of how and under which circumstances the datasets were recorded as well as a detailed description of preprocessing steps that need to be considered before performing analyses on the data. We additionally publish a machine-readable format of these metadata including our parameters and results of the described preprocessing steps. We are aware that, despite a high level of investment in this process, all provided information may not be sufficient for the reproducible analysis of such data. The reason is that reproducible workflows including the provenance trail are not yet established for electrophysiological neuroscience, especially not for such complex experiments as presented here.

2.1 Relevance to the field

The published datasets contain rare, multi-channel recordings from a complex electrophysiological setup. There are very few datasets of this type of experiment with this level of complexity available[19], wherefore we provide a valuable addition to the collection of openly available datasets. The datasets are interesting for multiple reasons and can be used in a variety of contexts:

- Public datasets are of high value for teaching and demonstration purposes. Especially for teaching, it is important to not only work with toy examples, but having real-world data at hand. This improves the teaching quality and provides a better insight into the field. For teaching advanced analysis methods that analyze the correlative properties of neuronal dynamics, corresponding parallel data is essential, such as the provided dataset.

[18] Plexon Offline Sorter, https://plexon.com/products/offline-sorter/

[19] published datasets from Utah array recordings in ScientificData: 1 (Brochier et al., 2018), CRCNS.org: 1 (only spiking activity); date of access: 30 August 2019

- For analysis of coordinated population activity of neural networks in the form of the local field (LFP) potential (Mitzdorf, 1985; Logothetis and Wandell, 2004; Einevoll et al., 2013), this datasets provides the opportunity to study LFP activity exhibiting wave activity across a spatially extended volume of cortex (Denker, Zehl, et al., 2018).

- The datasets contain spiking activity from approximately 100 neurons recorded simultaneously. This permits analysis of coordinated spiking activity, e.g., via correlation analysis techniques (Torre, Canova, et al., 2016; Torre, Quaglio, et al., 2016; Quaglio, Yegenoglu, et al., 2017; Quaglio, Rostami, et al., 2018).

- The availability of LFP as well as spiking data invites to study the relation between the two modalities, e.g., by relating synchrony in spiking activity to beta band power in the LFP (Denker, Roux, et al., 2011).

- In addition to spiking and LFP data, the dataset is also rich in behavioural aspects, since the monkey is performing a complex commonly investigated delayed reach-to-grasp task (Smeets, Kooij, and Brenner, 2019; Runnarong et al., 2019). These datasets provide the opportunity to relate the previously described neuronal analysis to the behaviour of the animal.

- The published datasets can serve as first application target for the development of new analysis methods for LFP and spiking activity. Especially needed are, e.g., analysis methods applicable to hundreds and more simultaneously recorded spike trains since recent technical development rapidly increases the number of electrodes that can be recorded from in parallel. Most currently available methods are already approaching limits with respect to their demands for compute time and memory with current datasets due to combinatorial explosion (Seo et al., 2015; Jun et al., 2017). An example of methods that overcome combinatorial issues are dimensionality reduction methods like Gaussian-Process Factor Analysis (GPFA) (Yu et al., 2009).

- The published datasets together with the detailed description of metadata sources can serve as a template for development of a comprehensive metadata tracking system for neuroscientific experiments. Parts of this development are presented in later sections of this manuscript.

- The correct identification of spikes based on a continuous recording trace (spike sorting) is a field of active research (Rey, Pedreira, and Quian Quiroga, 2015; Lefebvre, Yger, and Marre, 2016; Sukiban et al., 2019). For the evaluation of spike sorting algorithms, the availability of testing data is critical. With this dataset we provide raw recording traces, automatically extracted threshold crossing events as well as a manual spike sorting to compare new spike sorting algorithms to.

2.2 The experiment

The experiment is a delayed reach-to-grasp task performed by macaque monkeys. The basic components of the experiment are displayed in Fig. 2.1. The monkeys are trained to attend two cues coding for the grip type and force required to pull the object. After the second cue the monkey performs the movement, pulls the object and holds it to receive a reward. At the same time neuronal activity is recorded from motor and premotor areas of the cortex via a chronically implanted Utah Array.

The subjects of the published datasets are two Rhesus macaque monkeys (Macaca mulatta) which were trained to perform the delayed reach-to-grasp task prior to implantation of the recording arrays. Details on the two subjects are summarized in Table 2.1. Depending on the monkey character and learning abilities, the different instruction steps (accustoming to the experimenter, setup, different versions of the task) require several months of daily training. The training was completed when a monkey had an average success rate of 85% of the trials in the standard task setting.

The two monkeys differ in gender, active hand of the task and character, resulting in different behaviours observed during the recording task. As monkey N is rather calm and less motivated he is in general performing shorter sessions and less trials per session as monkey L, who is eager to work (see Table 2.1). A recording day typically consisted of multiple recording sessions, whereas each session lasts between 10 and 20 minutes resulting in an average working time per weekday of 1.5 hours. Weekends and holidays were usually excluded from training and recording.

For recording of neural activity, each monkey had a single Utah array[20] implanted in the motor cortex contralateral to the active hand. The Utah array is an electrode array with 100 electrodes arranged in a 10×10 grid. 96 of 100 of the iridium coated electrodes are used for recording (Fig. 2.2). The individual electrodes are 1.5mm long, 400μm apart and have an average pre-implantation impedance of 50kΩ. The electrode array is connected to a head stage via a fiber bundle connecting individual electrodes to the contact of the CerePort Connector, which is a connector external to the skin permitting the daily coupling of the implant to the recording setup. In addition to the array electrodes, one ground and two reference electrodes are implanted. During the implantation surgery the skull was opened and the dura removed to pneumatically insert the Utah array into the cortex. The dura and skull were closed, whereby the cables were arranged to lead to the connector which was attached to the skull on the other hemisphere. The Utah arrays were implanted a few millimeters anterior to the central sulcus and they were rotated before implantation to cover the arm/hand representation of the primary motor cortex as well as parts of premotor cortex (Fig. 2.2).

The task the monkeys had to perform involved grasping an object using one of two grip types: side (SG) or precision grip (PG). Detailed sketches of the different grips are depicted in Fig. 2.1A center and right panel. For both grip types the position of thumb (T), index (I) and middle (M) finger are indicated relative to the profile of the object. The force required to pull the object towards the monkey was either about

[20]Blackrock Microsystems, Salt Lake City, UT, USA

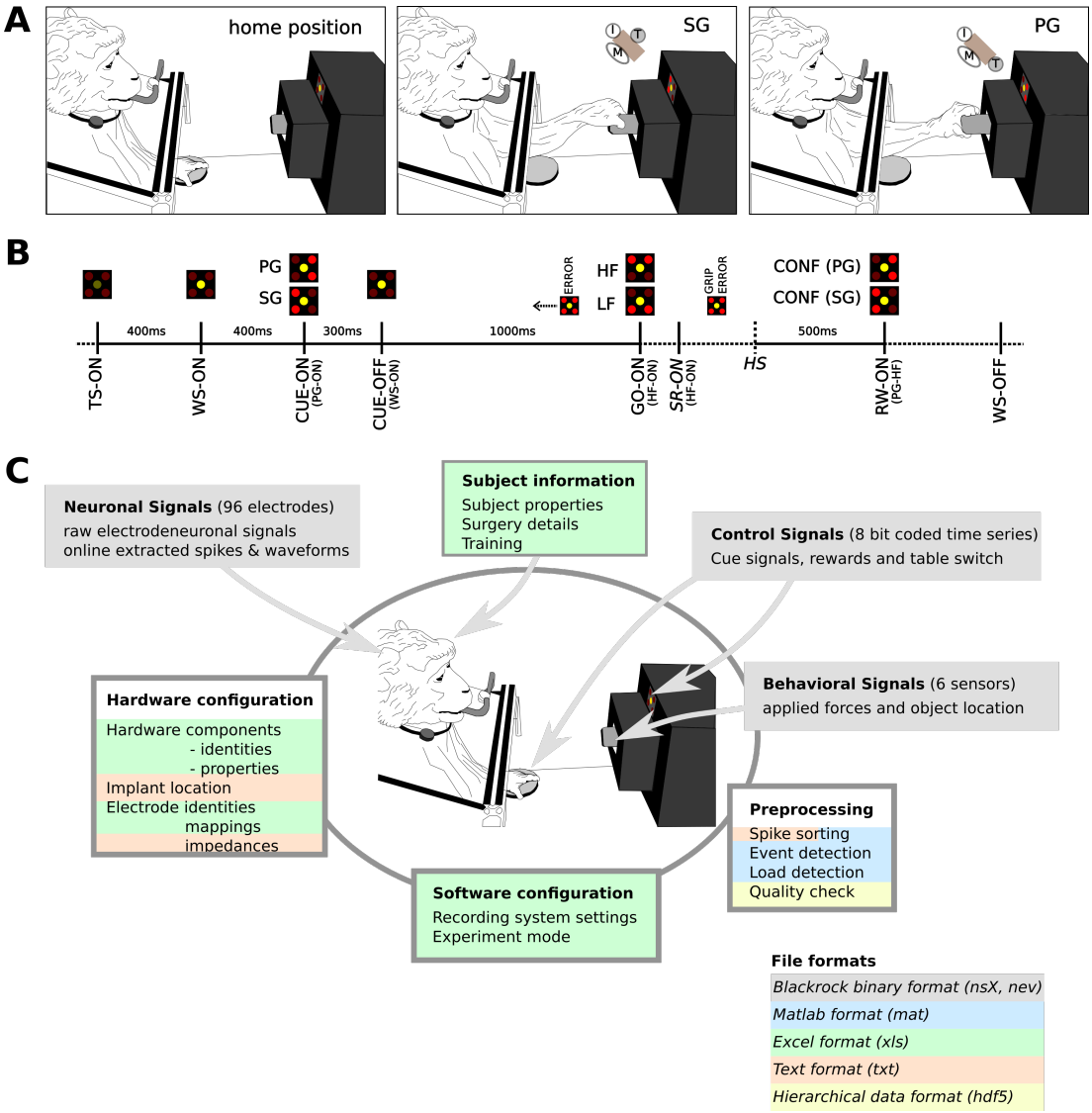


Figure 2.1: Components of the Reach-to-Grasp experiment. A) The monkey is located in a monkey chair, his active hand resting on the home switch (left panel). The monkey performs an instructed, delayed reach towards an object. Two grip conditions (side grip/precision grip (SG/PG, respectively)) can be instructed via visual cues. B) Visual cues are presented via five LEDs located at the height of the monkeys eyes. The monkey initialized a trial by holding the home switch with his active hand for 400 milliseconds, then the central LED is activated and stays active for the complete trial. Two complementary cues are presented with a delay of 1000 milliseconds, coding for the grip type and the force needed to move the object. Of the four corner LEDs, always two neighboring ones are activated. The two vertical arrangements code for the grip type and the horizontal ones for the force type. After the second cue (GO-ON), the monkey releases the home switch (SR-ON), reaches for the object (hold start, HS) and holds it for 500 milliseconds. Then a reward is delivered and the monkey can initiate the next trial by returning to the resting configuration and holding the home switch again. C) The data acquired during a recording session comprise neural activity signals, as well as behavioral and control signals. Metadata is captured in multiple formats, encompassing hard- and software components and settings as well as information about the subject and post-processing steps performed on the data. Figure modified from Brochier et al. (2018).

	monkey L		**monkey N**		**description**
Monkey					
gender	female		male		
birth date	15th March 2004		15th May 2008		
weight	5kg		7kg		
active hand	left		right		
character	eager to work quick efficient nervous		slow learner calm less motivated		
Utah array rotation	216°		239°		
Recording					
session name (*)	**l101210-001**		**i140703-001**		
session files	*.ccf	108.2 kB	*.ccf	187.1 kB	cerebus configuration digital events
	*.nev	287.7 MB	.nev	168.3 MB	unsorted spikes times spike waveforms digital events
	*-02.nev	287.7 MB	*-03.nev	168.3 MB	sorted spikes times spike waveforms
	*.ns2	8.5 MB	*.ns2	204.7 MB	analog signals of sensors LFP (only monkey N)
	*.ns5	4.1 GB	*.ns6	5.8 GB	raw neuronal signal
	*.odml	2.7 MB	*.odml	2.3 MB	metadata
recording start	Fr, 10th Dec 2010 10:50am		Th, 3rd Jul 2014 10:41am		
recording duration	11:49 min		16:43 min		
daily recording id	1		1		
number of recordings same day	9		3		
total daily recording time	01:28 h		00:51 h		
Performance					
recorded trials	204		160		
trials performance correct/error (grip error)	135	69 (12)	135	19 (16)	
trial types SG-LF/SG-HF	41	30	35	35	
trial types PG-LF/PG-HF	31	33	35	36	
Spike sorting					
# SUA	93		156		
# MUA	49		19		
# electrodes with SUA	65		78		
# electrodes with SUA or MUA	86		89		

Table 2.1: Overview of monkeys, recording session files and dates, spike sorting and monkey performance. The monkeys differ in gender, age, weight and active hand used for the task. Information is provided about individual recording files, sessions, performance of the monkey as well as single unit (SUA) and multi unit activity (MUA).

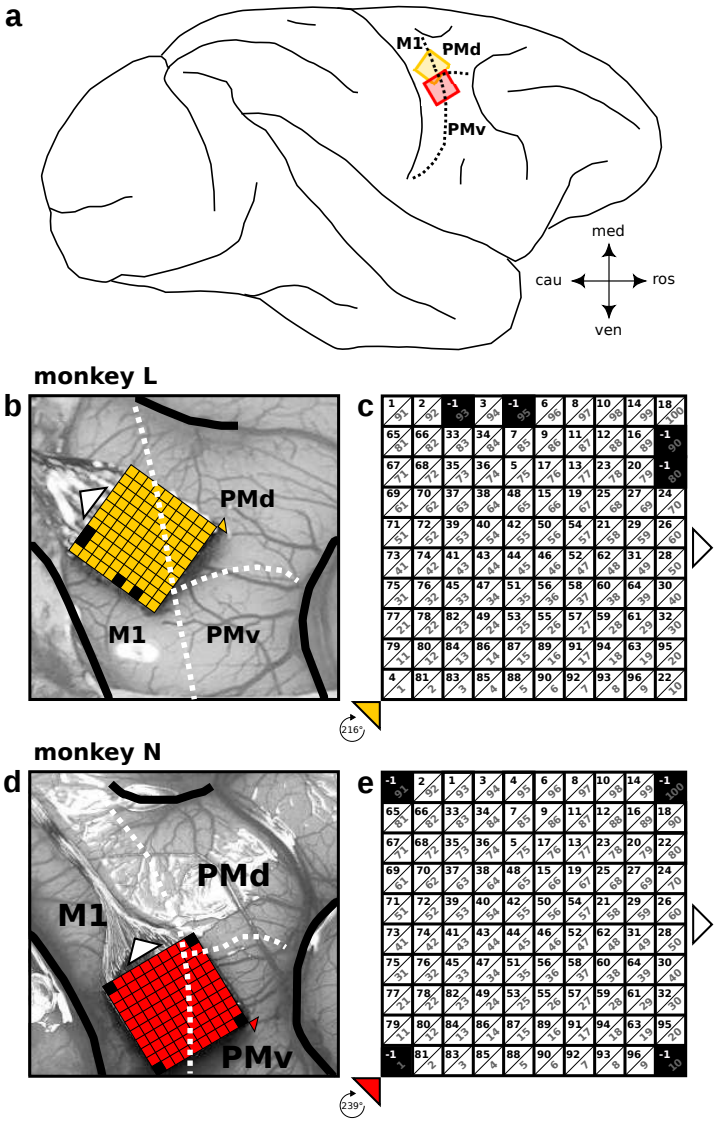


Figure 2.2: Implant locations of the Utah arrays. The figure displays the anatomical location of the Utah Array of both monkeys after implantation as well as the fabrication settings of each array provided by Blackrock. (a) Schematic drawing of a macaque cortex with implant location of the array of both monkeys. Both arrays were implanted along the central sulcus and overlapping the putative border (dotted line) between primary motor cortex (M1) and dorsal or ventral premotor cortex (PMd or PMv) of the right hemisphere. (b, d) Exact location of the array for each monkey in the close-up picture of the implantation site taken during the surgery (length of an array side is 4mm). The central sulcus, the arcuate sulcus and the superior precentral dimple are emphasized as thick black lines (to the left, right and top, respectively). (c, e) Scheme of each array in a default array orientation where the wire bundle (indicated with white triangles in (b-e)) to the connector points to the right. Each array scheme shows the 10-by-10 electrode grid with the electrode identification numbers (IDs) defined by Blackrock (black numbers) and the location of the non-active electrodes (indicated in black as $ID = -1$). Gray numbers show an alternative set of connector-aligned electrode IDs, assigned based on electrode location with respect to the connector of the array, which are more practical for data analysis and comparisons across monkeys. In order to best cover the arm/hand representation of the primary motor cortex, each array was rotated for the implantation. The center of rotation is indicated by a colored triangle (b-e), stating below (in c and e) the degree of rotation for each array.

1N (low force) or 2N (high force). In a trial the monkey was always instructed by the first cue (CUE) about the grip type (SG/PG) and with the second cue (GO) about the force level (LF/HF), resulting in 4 different trial types (SGLF, SGHF, PGLF, PGHF) possible (Fig. A.1). The second cue is presented with 1s delay after the first. In this time the monkey has to memorize the presented grip type and can prepare for the movement, which can be initialized after the second (GO) cue. Cues are presented by combinations of two of five illuminated LEDs. The encoding of the grip and force type can be seen in Fig. 2.1B (CUE-ON and GO-ON). The setup consisted of three main parts, the neural recording platform for acquisition of neuronal signals, the experimental apparatus providing the environment for performing the task and the behavioral control system controlling and coordinating the task procedure (Fig. A.2, see also Appendix A).

Differences between the datasets In principle, the same setup was used for both monkeys, however, small deviations exist which are highlighted in the corresponding Figs. 2.2, A.2 and A.3 in yellow (monkey L) and red (monkey N). The four main aspects are

Recording Arrays The datasets are recorded using different Utah arrays. This includes a different electrode mapping as well as a different implant location.

Connecting Hardware Usage of different hardware for connecting the implant with the data acquisition system. The two headstages (samtec/patient cable) influence the recording quality by their specific electrical properties.

Software versions and settings The recording software versions and setting differ, since an update was available between the recordings. This includes the usage of different file extensions for recordings of continuous data (ns5 in monkey L vs ns6 in monkey N) and additionally a downsampled and filtered version of the neural data being recorded in parallel in the ns2 format for monkey N, which was not available by the software for monkey L. Furthermore, the number of samples extracted online for each of the threshold crossing events was increased from 39 samples in monkey L to 48 samples in monkey N.

Experiment Control The LabView program controlling and monitoring the task was updated which leads to the usage of different binary coding for digital events (Table 2.2).

After the recordings, a number of preprocessing steps (pre in the sense of before the actual upcoming data analysis, but being the post-processing after the recording) were performed. This includes (i) the translation of the digital events from a bit code to a human-readable format, by putting the events in context of the expected executed trial event sequence, (ii) the offline detection of behavioral trial events and object load force from the analog signals, and (iii) the offline spike sorting.

In addition to the preprocessing steps that needed to be performed to gain more content of the raw data, some technical validations of the data also had to be conducted

decimal code	**(8-bit) binary code**								**trial interpretation**		
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0			monkey
65296	0	0	0	1	0	0	0	0	**TS-ON**		L,N
65280	0	0	0	0	0	0	0	0	**TS-OFF**		L
65344	0	1	0	0	0	0	0	0	**WS-ON**		L
65360	0	1	0	1	0	0	0	0			N
65360	0	1	0	1	0	0	0	0	**PG-ON**	**(CUE-ON)**	L
65365	0	1	0	1	0	1	0	1			N
65354	0	1	0	0	1	0	1	0	**SG-ON**		L
65370	0	1	0	1	1	0	1	0			N
65344	0	1	0	0	0	0	0	0	**CUE-OFF**		L
65360	0	1	0	1	0	0	0	0			N
65353	0	1	0	0	1	0	0	1	**LF-ON**	**(GO-ON)**	L
65369	0	1	0	1	1	0	0	1			N
65350	0	1	0	0	0	1	1	0	**HF-ON**		L
65366	0	1	0	1	0	1	1	0			N
65385	0	1	1	0	1	0	0	1	**SR**	(+LF)	L,N
65382	0	1	1	0	0	1	1	0		(+HF)	L,N
65509	1	1	1	0	0	1	0	1	**RW-ON**	(+CONF-PG)	L,N
65514	1	1	1	0	1	0	1	0		(+CONF-SG)	L,N
65376	0	1	1	0	0	0	0	0	**GO-OFF/RW-OFF**		L,N
65312	0	0	1	0	0	0	0	0	**STOP**		L,N
65280	0	0	0	0	0	0	0	0			L
65391	0	1	1	0	1	1	1	1	**ERROR**	(+switch)	L
65359	0	1	0	0	1	1	1	1			L
	pump	**LED**	**switch**	**TS**			**LED**				
		c			**bl**	**tr**	**tl**	**br**			

Table 2.2: Translation table of 8-bit binary to decimal event codes and their interpretation in a trial context. The 8-bit binary event code created by LabView states the activation (bit status 1) and deactivation (bit status 0) of the LEDs of the cue system (c:center, t:top, b:bottom, l:left, r:right), the table switch (switch), the reward pump (pump) or the trial start (TS) internally set by LabView. During each trial the (8-bit binary) status of these devices/settings (cf. bottom row) were sent from LabView to the NSP of the Cerebus DAQ system (Figure 2). There, the event codes were converted to a decimal code of the bit sequence assuming another byte with all bits set to 1 in front. The decimal event codes are found in the nev files with a time stamp. The correct interpretation of these events in context of a trial are here indicated in the second column from the right. Due to different versions of the LabView control program for monkey L and N the decimal codes for the same event may differ between the monkeys (cf. first column from the right). Some event codes (65381, 65386, 65390, 65440, 65504) are not listed here, because they do not have a concrete meaning and occur only sporadically in the nev file due to a mistake in the sampling of the digital events - they have to be ignored. Except for the "ERROR" codes, the event codes are sorted in sequential order from top to bottom with respect to the task, i.e. their order corresponds to the sequence found in the nev file in a reach-to-grasp trial. Note that some events are represented by the same decimal codes and are just differently interpreted due to their sequential occurrence in a trial (cf. TS-OFF and STOP, as well as WS-ON and CUE-OFF).

(see also Appendix D). These technical validations include the correction of the irregular alignment data files of the recording system and a general quality assessment of the data. In order to validate the quality of the recording, a series of algorithms were applied to the data. On the one hand the quality of the LFP signals was assessed per electrode and per trial by evaluating the variance of the corresponding signal in multiple frequency bands. On the other hand the quality of the offline sorted single units (Appendix B.3) was determined by a signal-to-noise measure. In addition, noise artifacts occurring simultaneously in the recorded spiking activity were detected and marked.

2.3 The metadata structure

All aggregated metadata for a single recording session are collected in a separate *odML* file (Table 2.1). Within an *odML* file, information is organized in a hierarchical fashion, following a schema that was defined and continuously refined during the course of the set up of the metadata processing pipeline. Fig. 2.3 shows an exemplary part of the metadata collection of the recording session of monkey L. The *odML* files contains the following eight top-level sections: *Project* (general information on the reach-to-grasp project), *Subject* (information on the monkey), *Setup* (details of the experimental apparatus), *Cerebus* (settings of the recording system), *UtahArray* (information on the multi-electrode array including spike sorting results and the corresponding quality assessment), *Headstage* (general settings), *Recording* (task settings, trial definitions with event time stamps and periods) and *PreProcessing* (results of LFP quality assessment and general information on the spike sorting procedure).

Each top-level `Section` contains a branch structure built from `Section`s to logically organize the information about the specific aspect of the experiment. For example, the *Subject* `Section` contains nine `Properties` (denoted by a folder icon), and multiple `Section`s beneath, among them for example a *Training* `Section` and *ArrayImplant* `Section` (denoted by a collapsible folder icon in Fig. 2.3). All metadata available describing the training of monkey L are the coach (Thomas Brochier), start and end date of the training period (June 2010 - September 2010) and a comment stating that the training was easy and fast. Larger numbers of metadata are stored for describing the recording array used for the recording. Here, the `Section` *UtahArray* contains general information about the hardware (material, dimensions, etc) and detailed information about each electrode is contained in the *Electrode* `Section`s. Each electrode `Section` contains information about the electrode identity and physical location, the impedance value and offline sorting results for data recorded here (not shown in Fig. 2.3). The metadata collection of monkey N has an analogous structure.

2.4 Data and metadata processing pipeline

The metadata information as described Section 2.3 was aggregated from multiple files and formats into a single file in the *odML* format (Section 1.1.1). This aggregation requires access to, and interpretation of a variety of files and formats, extraction of the

Figure 2.3: Schematic metadata collection of recording session l101210-001 as rendered by the GIN webinterface. `Properties` are displayed in the form of `<Property Name>: <List of Values> (<Property Description>)`. Individual `Sections` are unfolded via user interaction to display the underlying metadata structure. Figure modified from https://gin.g-node.org/INT/multielectrode_grasp/src/master/datasets/l101210-001.odml.

corresponding information and integration in the designated location a pre-designed *odML* structure. In the pipeline used here (Fig. 2.4), the construction of the *odML* structure of a particular recording session is strictly separated from the enrichment of the structure with metadata content. In a first step, an *odML* structure with default metadata entries is set up based on multiple Python scripts generating individual branches of the *odML* structure. Since the particular structure required for a recording session depends on the specifications of the recording, this process depends to some extend on the metadata to be added (see Fig. 2.4, red arrows). This interdependency requires the metadata to be loaded and partially interpreted prior to the initialization of the structured metadata collection. One example for such a dependency is the existence of spike sorted data after offline spike sorting. Here, first the source files need to be evaluated to extract if the particular session was already sorted to construct the metadata structure accordingly such that it can later accommodate potential spike sorting metadata.

After construction of the *odML* structure, metadata information is added via a set of custom enrichment functions, i.e., filling available metadata into the structure. This enrichment process is split into different functions that first extract information from the different specialized metadata source file formats and add this data in the *odML* structure. In the enrichment process, the location of the target `Property` in the structure is implemented by exploiting the filter functionality of *odML*. This way, filling the *odML* structure requires no detailed information regarding the *odML* hierarchy. This provides some degree of flexibility in dealing with variations in the *odML* structure, i.e. for different variations of the task or through the continuous improvement of the structure over the course of the experiment. However, given the large number of metadata values to fill, this is computationally more intense than accessing the target location directly via its hierarchy path. Finally, after building and enriching the metadata collection the *odML* file with more than 1300 `Section`s, 8000 `Properties` and about 10000 `Value`s is saved.

Setting up this pipeline required preparatory work on multiple levels: A) The manual preparation of the *odML* structure in form of *odML* templates in dedicated Python scripts. These scripts mostly comprise between 700 and 1700 lines of code and contain the generation of all *odML* `Section`s and `Properties` including default `Value`s, since `odML` version 1.3 used at that time requires a minimum of one `Value` per `Property` (see Section 1.1.1). B) The preparation of the metadata source files in the required format. This involved manual offline spike sorting using the *Plexon* spike sorter to generate a sorted `nev` file and spike sorting `mat` and text files containing the resulting metadata. For a single recording session this process requires a few working hours of a trained scientist. Additionally, xls sheets following the *odMLtables* standard needed to be set up containing information about the recording (sub)session, the monkey and the recording setup. Furthermore, a single text file providing the mapping spatial systems of electrode IDs used in the experiment (`brain_aligned_elids.txt`) and the results of three *Matlab* based preprocessing scripts for detection of behavioural events based on

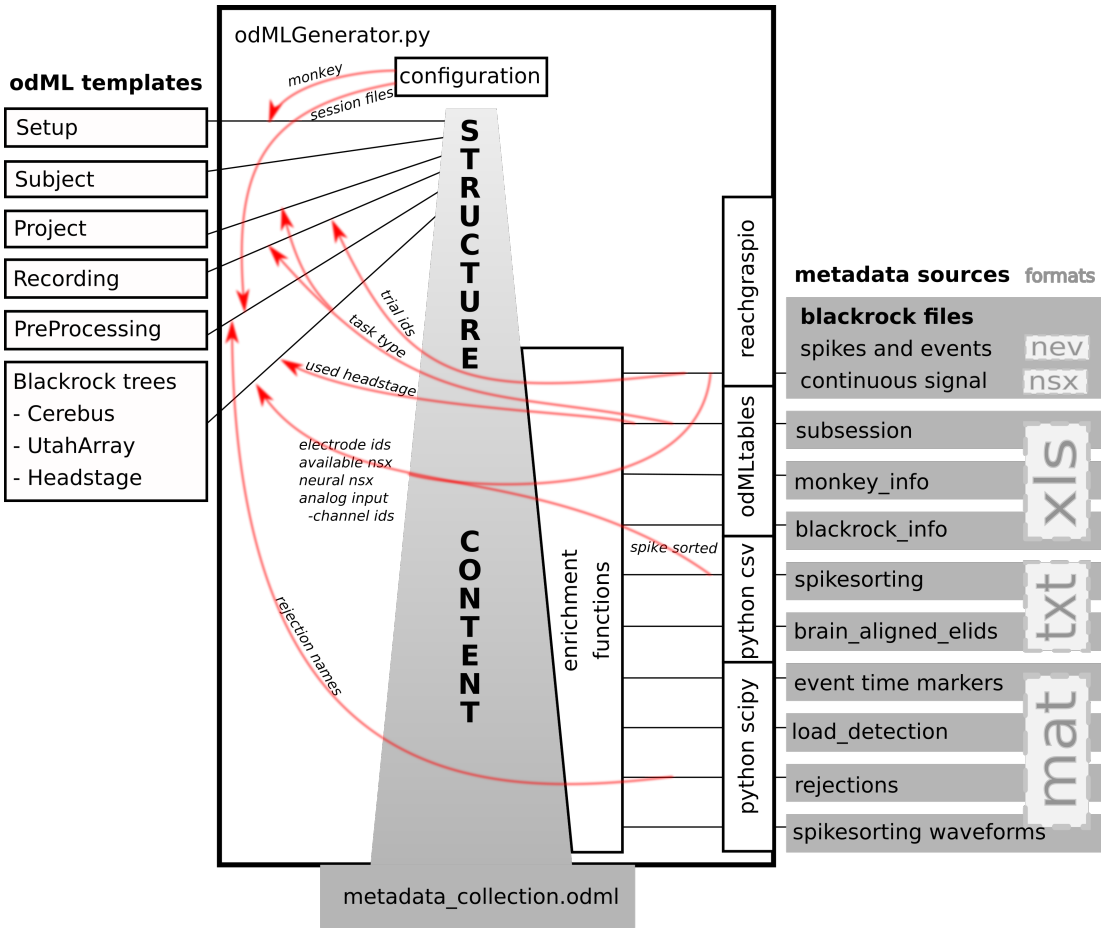


Figure 2.4: Schematic metadata aggregation pipeline used in Brochier et al. (2018). The *odMLGenerator* script (middle box) accesses `odML` templates (left box) and metadata source file in different formats (right box). Based on a configuration file, first the structure of the `odML` is set up relying on Python scripted `odML` templates. In a second step, the structure is enriched with information content from the metadata files. Information from these files is extracted using a variety of different tools (`reachgraspio`, *odMLtables*, the Python *csv* and *scipy* packages) and added to the pre-built *odML* structure via custom enrichment functions to generate the final metadata collection. Setting up the *odML* structure requires information about the metadata and the configuration (red arrows), which is extracted from the metadata files prior to the building of the *odML* structure.

continuous signals (event time markers and load detection) and the quality checks for continuous signals in multiple frequency ranges and electrodes (`rejections`) need to be provided. Most of these metadata source files are recording session specific and need to be generated after the recording was performed. Typically, these files are generated by the experimenter, however, due to the time required for this manual step and the amount of available data, not all of this information becomes available immediately, possibly only years after the experiment. C) The implementation of the central script to generate the resulting *odML* (Fig. 2.4 `odMLGenerator.py`) by combining the *odML* structure from the templates with the content of the metadata sources. Implementation of this script needs detailed knowledge about both aspects of the *odML* generation process to be able to combine the two.

2.5 Data loading and enrichment with metadata

To benefit from a complete metadata collection in the *odML* format when loading the original data the metadata should be made easily accessible in combination with the data. This permits the user to access the complete information available during later processing and analysis steps, e.g. for selection of specific trial time periods or recording signals without artifacts. For this, it is essential to combine the information from the metadata collection with the data as the user otherwise needs to switch between different sources and structures for retrieving information. This would introduce additional, unnecessary dependencies in the processing and analysis code. We provide the `ReachGraspIO` Python class which extends the *Neo* data structure upon loading with metadata (Appendix E). The `ReachGraspIO` inherits from the *Neo* `BlackrockIO` and exploits its functionality to read data from `nsX`[21] and `nev` file formats (Fig. 2.6). It returns a *Neo* `Block` based on the original structure generated by the *Neo* `BlackrockIO`. The API to interact with the data is therefore the same when using directly the *Neo* `BlackrockIO` and the `ReachGraspIO`. For a detailed description of the *Neo* structure and `IO`s, see Chapter 4.

The application programming interface (API) of the `ReachGraspIO` is almost identical to the API of the `BlackrockIO`. The `ReachGraspIO` uses slightly different default parameters, which are adjusted to the datasets generated by the Reach-to-Grasp experiment. The usage of the `ReachGraspIO` is demonstrated in an example Python script, which loads the data and visualizes the recording signals at the time point of a trial start (Code Listings 2.1 and 2.2 and Fig. 2.5). Identification and selection of a trial start event in the recording is facilitated by using the `ReachGraspIO` instead of the `BlackrockIO` directly as the `ReachGraspIO` identifies individual trials while loading the data and labels the trial events in a human readable fashion.

The `ReachGraspIO` extends and annotates the *Neo* structure generated by the `BlackrockIO` in multiple ways: continuous recording signals are corrected for time shifts introduced by online filters, event times extracted from analog signals are added and a merged

[21] where nsX represents the set of extensions ns1, ns2, ns3, ns4, ns5 and ns6

```
68  # Open the session for reading
69  session = reachgraspio.ReachGraspIO(session_name, odml_directory=odml_dir)
70
71  # Read the first 300s of data (time series at 1000Hz (ns2) and 30kHz (ns6)
72  # scaled to units of voltage, sorted spike trains, spike waveforms and events)
73  # from electrode 62 of the recording session and return it as a Neo Block. The
74  # time shift of the ns2 signal (LFP) induced by the online filter is
75  # automatically corrected for by a heuristic factor stored in the metadata
76  # (correct_filter_shifts=True).
77  data_block = session.read_block(
78      nsx_to_load='all',
79      n_starts=None, n_stops=300 * pq.s,
80      channels=[62], units='all',
81      load_events=True, load_waveforms=True, scaling='voltage',
82      correct_filter_shifts=True)


86  data_segment = data_block.segments[0]


92  # Here, we construct one offline filtered LFP from each ns5 (monkey L) or ns6
93  # (monkey N) raw recording trace. For monkey N, this filtered LFP can be
94  # compared to the LFPs in the ns2 file (note that monkey L contains only
95  # behavioral signals in the ns2 file). Also, we assign telling names to each
96  # Neo AnalogSignal, which is used for plotting later on in this script.


99  filtered_anasig = []
100 # Loop through all AnalogSignal objects in the loaded data
101 for anasig in data_block.segments[0].analogsignals:
102     if anasig.annotations['nsx'] == 2:
103         # AnalogSignal is LFP from ns2
104         anasig.name = 'LFP (online filter, ns%i)' % anasig.annotations['nsx']
105     elif anasig.annotations['nsx'] in [5, 6]:
106         # AnalogSignal is raw signal from ns5 or ns6
107         anasig.name = 'raw (ns%i)' % anasig.annotations['nsx']
108
109         # Use the Elephant library to filter the analog signal
110         f_anasig = butter(
111                 anasig,
112                 highpass_freq=None,
113                 lowpass_freq=250 * pq.Hz,
114                 order=4)
115         f_anasig.name = 'LFP (offline filtered ns%i)' % \
116             anasig.annotations['nsx']
117         filtered_anasig.append(f_anasig)
118 # Attach all offline filtered LFPs to the segment of data
119 data_block.segments[0].analogsignals.extend(filtered_anasig)


123 # Construct analysis epochs
124 #
125 # In this step we extract and cut the data into time segments (termed analysis
126 # epochs) that we wish to analyze. We contrast these analysis epochs to the
127 # behavioral trials that are defined by the experiment as occurrence of a Trial
128 # Start (TS-ON) event in the experiment. Concretely, here our analysis epochs
129 # are constructed as a cutout of 25ms of data around the TS-ON event of all
130 # successful behavioral trials.


143 start_events = get_events(
144     data_segment,
145     properties={
146         'name': 'TrialEvents',
147         'trial_event_labels': 'TS-ON',
148         'performance_in_trial': session.performance_codes['correct_trial']})


159 pre = -10 * pq.ms
160 post = 15 * pq.ms
161 epoch = add_epoch(
162     data_segment,
163     event1=start_event, event2=None,
164     pre=pre, post=post,
165     attach_result=False,
166     name='analysis_epochs')


174 cut_trial_block = Block(name="data_cut_to_analysis_epochs")
175 cut_trial_block.segments = cut_segment_by_epoch(
176     data_segment, epoch, reset_time=True)
```

Code Listing 2.1: Example code for loading and processing of published data. Metadata annotated data is loaded (line 69-86) and a low pass filtered version of the original signals in generated (line 99-119). Trials are identified and signals are cut into the corresponding segments (line 123-176). Code extracted from Brochier et al. (2018), https://gin.g-node.org/doi/multielectrode_grasp/src/master/code/example.py
.

```
195 # Create figure
196 fig = plt.figure(facecolor='w')
197 time_unit = pq.CompoundUnit('1./30000*s')
198 amplitude_unit = pq.microvolt
199 nsx_colors = ['b', 'k', 'r']
200
201 # Loop through all analog signals and plot the signal in a color corresponding
202 # to its sampling frequency (i.e., originating from the ns2/ns5 or ns2/ns6).
203 for i, anasig in enumerate(trial_segment.analogsignals):
204         plt.plot(
205             anasig.times.rescale(time_unit),
206             anasig.squeeze().rescale(amplitude_unit),
207             label=anasig.name,
208             color=nsx_colors[i])
209
210 # Loop through all spike trains and plot the spike time, and overlapping the
211 # wave form of the spike used for spike sorting stored separately in the nev
212 # file.
213 for st in trial_segment.spiketrains:
214     color = np.random.rand(3,)
215     for spike_id, spike in enumerate(st):
216         # Plot spike times
217         plt.axvline(
218             spike.rescale(time_unit).magnitude,
219             color=color,
220             label='Unit ID %i' % st.annotations['unit_id'])
221         # Plot waveforms
222         waveform = st.waveforms[spike_id, 0, :]
223         waveform_times = np.arange(len(waveform))*time_unit + spike
224         plt.plot(
225             waveform_times.rescale(time_unit).magnitude,
226             waveform.rescale(amplitude_unit),
227             '--',
228             linewidth=2,
229             color=color,
230             zorder=0)
231
232 # Loop through all events
233 for event in trial_segment.events:
234     if event.name == 'TrialEvents':
235         for ev_id, ev in enumerate(event):
236                 plt.axvline(
237                     ev,
238                     alpha=0.2,
239                     linewidth=3,
240                     linestyle='dashed',
241                     label='event ' + event.annotations[
242                         'trial_event_labels'][ev_id])
243
244 # Finishing touches on the plot
245 plt.autoscale(enable=True, axis='x', tight=True)
246 plt.xlabel(time_unit.name)
247 plt.ylabel(amplitude_unit.name)
248 plt.legend(loc=4, fontsize=10)
249
250 # Save plot
251 fname = 'example_plot'
252 for file_format in ['eps', 'png', 'pdf']:
253     fig.savefig(fname + '.%s' % file_format, dpi=400, format=file_format)
```

Code Listing 2.2: Continuation of Code Listing 2.1. The plotting of the data belonging to the extracted trials for generation of Fig. 2.5. The figure is initialized and plotting parameters are defined (line 196-199). `AnalogSignals` are visualized (line 201-208) and `Spiketrains` are plotted together with the corresponding waveforms (line 213-230). `Events` are visualized as vertical lines, labelled in human readable way (line 233-242). Finally, the plot is completed and saved in three different formats (line 244-153). Code extracted from Brochier et al. (2018), https://gin.g-node.org/doi/multielectrode_grasp/src/master/code/example.py

.

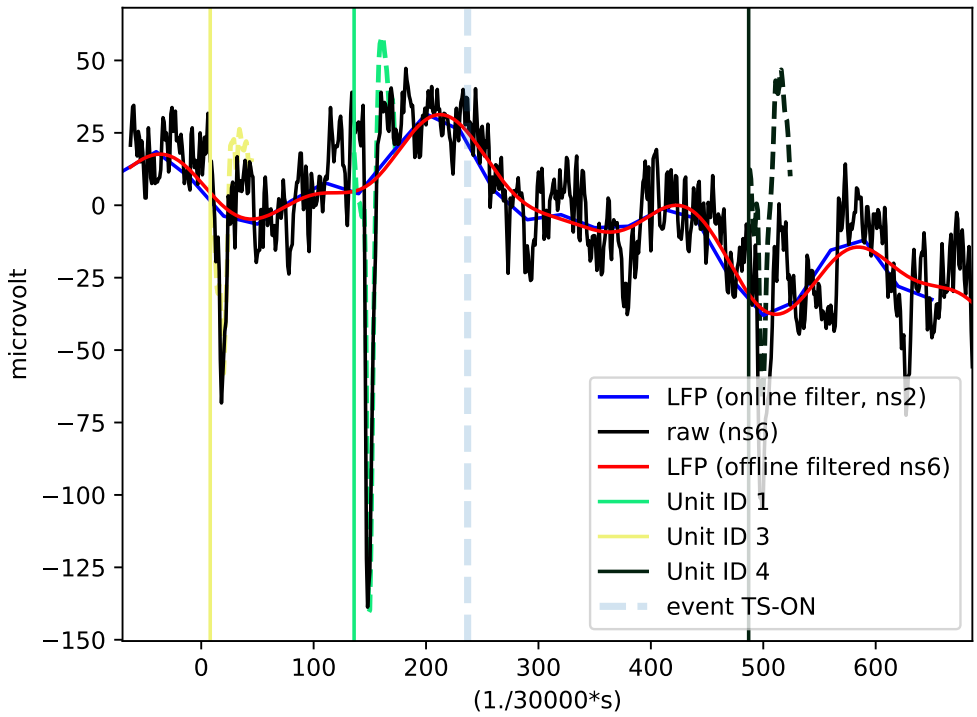


Figure 2.5: Example visualization of the published data. The code to generate the figure is shown in Code Listings 2.1 and 2.2. All recorded data from a single electrode are visualized: the raw, high resolution continuous recording signal (ns6, black), the online filtered continuous recording signal (Ns2, blue) as well as an offline filtered version based on the raw signal (red). In addition spikes times are marked corresponding to their *Unit* assignment and online extracted waveforms are plotted (yellow, green and black traces). Trial events are also visualized (dashed vertical line) and labelled in human readable way (TS-ON, gray).

representation of events from digital and analog sources is created (Fig. 2.6, red box). The first two of these extensions depend on metadata being provided from the metadata collection in *odML* format. Additional annotations based on information from the metadata collection are added to most of the *Neo* objects (Fig. 2.6, gray box). This includes basic metadata describing individual recording traces of `AnalogSignal`s, sorting information of `Unit`s, general metadata for `Segment` and `Block` objects, as well as information from the offline trial rejection. If no *odML* metadata collection is present, these extension and annotation steps are skipped. Some of these extension and annotation steps require also additional information, which is available within the `ReachGraspIO` as metadata lookup tables. These contain experiment specific code mappings which are essential for the interpretation of the data stored in the original `Blackrock` data files. This includes the mapping of bit-coded to human readable notation for events required to define a trial, event codes and equivalence groups, the mode in which the experiment is running (experimental conditions) as well as the performance codes standing for the outcome of a trial attempt (Fig. 2.6, green box). These metadata lookup tables are

used to translate the bit-based encoding of trial events into human readable versions. If an `odML` file is available, trial ids are extracted from there and also annotated. This interpretation of the events is a prerequisite for two further annotation steps: the identification of the task condition, i.e., the task paradigm used in the recording, which again uses a `ReachGraspIO` lookup table, as well as the annotation of rejected trials. If digital and analog events are present, these will be merged in to a single `Event` object to simplify the access to all events of recording session (Fig. 2.6, 'merge digital & analog events'). Finally the `ReachGraspIO` returns the modified `Block` which can be used in further data analysis and visualization scripts (e.g., see Code Listings 2.1 and 2.2) or in the generation of a metadata collection (see Fig. 2.4).

The original pipeline was implemented using Python 2 and *odML* version 1.3. The published datasets were updated in 2019 to be compatible with Python 3 and *odML* version 1.4. Old versions are still accessible via the published DOI[22] and the version control system of GIN[23].

2.6 Shortcomings of the *odML* generation pipeline

The presented data publication demonstrates the complexity of the datasets and the efforts required for preparation and release of the data. The complexity of the process is exemplified by the fact that only two of of almost 2000 recording sessions being performed in total with five monkeys were fully described at the time of publication. Here we list reasons we identified that complicate the preparation process:

Additional features Describing two datasets in a high degree of detail in a manuscript requires 23 pages of descriptor to provide all necessary details. Including additional datasets following the same experimental paradigm would expand the descriptor as additional features observed in the data need to be described. Extending the descriptor to include all peculiarities of all datasets will confuse the reader and therefore make the dataset less easily reusable. Moreover, even small additions or changes in the data and metadata processing may result in substantial changes to the pipeline design. An example for additional features which would require explicit description is an additional type of artifact, which is not described in Brochier et al. (2018) as these two datasets do not exhibit unintended gaps during the recording (Fig. 2.7). This type of artifact was first described in Sprenger (2014) and occurs when individual data packets are lost during data recording within the recording hardware. The loss of data packets does not cause an interruption of the recording and is not tracked in the recording file, but the recording files appear intact. Lost data packets can only be detected afterwards by comparing raw continuously recorded data with the corresponding threshold crossing events detected online during recording. After the occurrence of a lost data packet, these two signal are not aligned, as only the continuous signal is affected by lost data packages, and not the threshold crossing events.

[22] 10.12751/g-node.f83565, https://gin.g-node.org/doi/multielectrode_grasp
[23] GIN, https://web.gin.g-node.org/

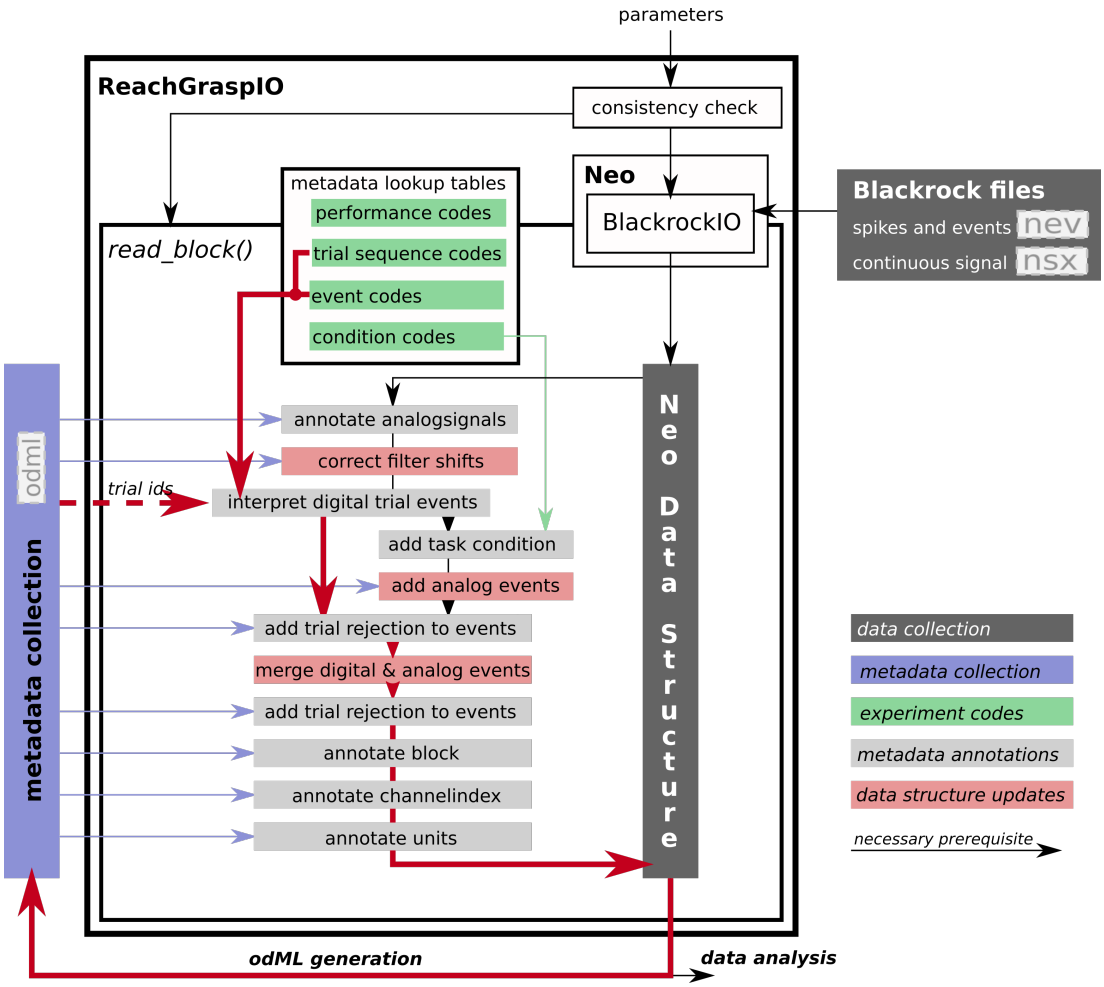


Figure 2.6: Schematic data loading pipeline used in Brochier et al. (2018). The `ReachGraspIO` (central box) utilizes the `BlackrockIO` of *Neo* to load the original recording data (top right). In a second step it enriches and extends the data structure returned by the `BlackrockIO` containing uninterpreted row data based on additional metadata information from two sources: a metadata collection in *odML* format (blue) and `ReachGraspIO` internal lookup tables (green) containing the essential information required for interpretation of the data from the original data files. The extension of the *Neo* structure includes correction of time shifts of the recording signals, addition of events which were extracted offline from behavioural signals and the merging of multiple event arrays containing meaningful, human readable events in the trail ('correct filter shifts', 'add analog events' and 'merge digital & analog events', respectively; red box). The enrichment of the *Neo* structure encompasses the addition of numerous annotations (gray box). If no metadata collection is found, these steps are skipped. Finally the extended *Neo* structure is returned and can be used for the generation of a metadata collection or further processing and analysis steps. This introduces a circular dependency between the `ReachGraspIO` and the metadata collection, as the first one can use the other if present to add trial ids as annotations to the *Neo* structure, but also the *odML* generation depends on the `ReachGraspIO` as essential metadata is annotated based on internal lookup tables (see also Fig. 2.4). Relations contributing to this circular dependency of metadata are marked by red arrows.

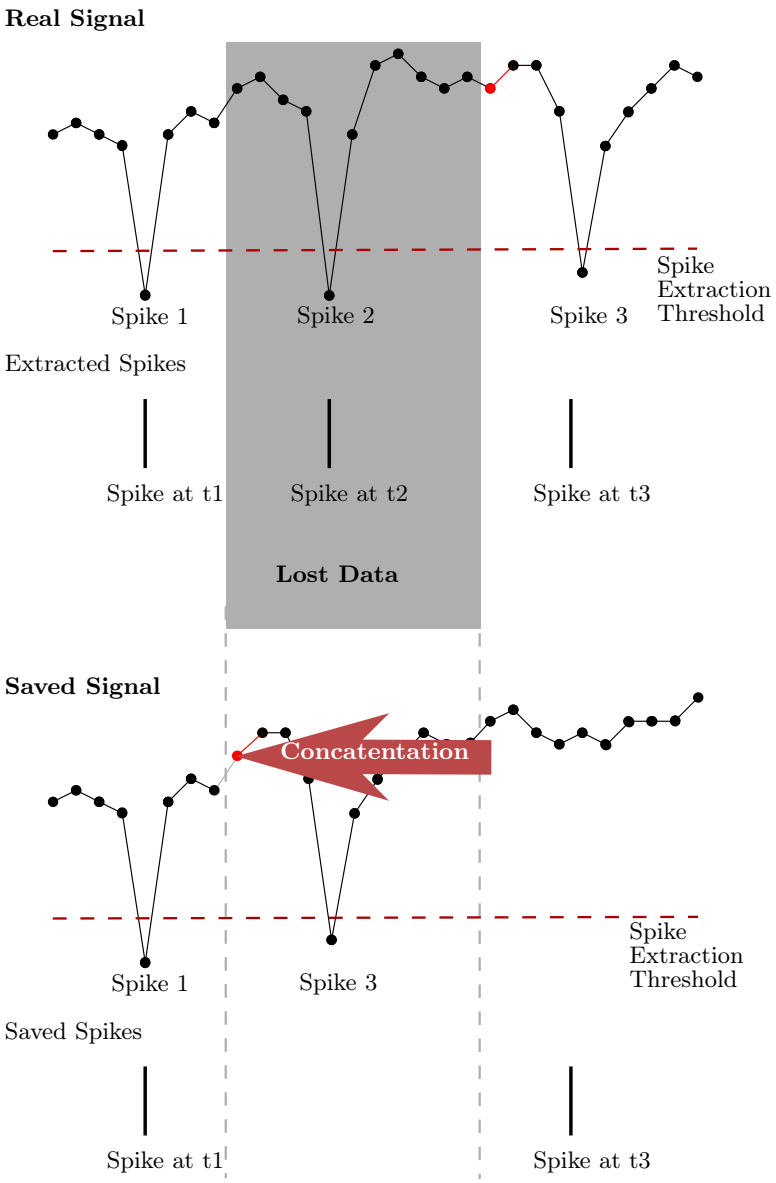


Figure 2.7: Origin of gaps in continuous recording data. Depicted are the recording signals as they are send by the Neural Signal Processor (NSP) ('Real Signal', top) and as they are received by the Central Suite computer (CSC) to be written to disc ('Saved Signal', bottom). For a visualization of the hardware components, see Fig. A.2. The spike times are extracted from the continuous signals by the NSP and the corresponding time of threshold crossing is attached to each spike. In case of a loss of data during the transfer to the CSC, the spike times are unaffected, only the corresponding spikes of the lost data are not present. For the continuous signal however, the loss of data leads to a concatenation of the remaining samples without considering the original time stamps of the data samples as these are not attached to the data packets (compare sample point marked in red). Therefore the continuous signal is shifted with respect to the extracted spikes starting from the beginning of the gap.

For a comprehensive check for gaps during the recording, the *odML* generation pipeline would need to be extended in multiple places: Firstly, an additional `Section` in the *odML* structure would be required to capture the potential details generated by a function scanning the original data files. This would require changes in the *odML* templates (Fig. 2.4, left box). Secondly, an additional function for gap detection by comparing events to continuous signals needs to be integrated in the *odML* generator. This requires extension of the *odML* generation code and would potentially increase the run time of the code considerably as the comparison for a full dataset can be compute intense and the *odML* generation process is not parallelized. Finally, since the coded detection of gaps can only identify the occurrence of a gap, without reliably characterizing the exact time point or exact extend of the gap, a manual inspection of the data with a detected gap is required after the first run of the *odML* generator. To reliably document the results of this manual inspection the integration of the results into the *odML* file is mandatory. This could be implemented by the manual generation of a secondary metadata source file, from which the corresponding metadata is extracted during the *odML* generation process and included in the metadata collection. However, the suggested procedure requires the execution of the complete *odML* generation pipeline twice, once for running the coded detection of gaps and once for integrating potential manually generated additional metadata describing the exact gap time and size. The suggested procedure would therefore more than double the run time of the complete *odML* generator for recording sessions with gaps, event though only a small fraction of the metadata was added.

Manual validation Preparing the data for data publication requires cross checking of all preprocessing steps (e.g., detection and correction of data packet loss) necessary for that particular set of data. However, in the present scenario, such cross-checks are neither automatized if possible, nor proceduralized. Instead they are performed once a specific dataset is selected for a given analysis, and only specific types of cross-checks relevant for the analysis are typically performed by the researcher.

Variability across datasets The pipeline for automatic metadata aggregation as presented here has been tested extensively for the two datasets published and the generated files have been manually checked. Running the exact same pipeline on different recording datasets from the same recording has been found to frequently fail, due to slight variations between recording sessions. An intuitive example for this is the set of event codes observed within a recording (Table 2.2). Here, the interpretation of the observed events codes is performed by the `ReachGraspIO` transforming combinations and sequences of bit-encoded events to human readable labels. Due to complex sequences of event codes that may arise, the interpretation of the bit codes in this experiment highly depend on the monkey covering scenarios not encountered for the two published datasets. Extending this to other monkeys and other recording sessions required i) extending the interpreting routine making the code more convolved and less readable and ii) extensive manual validations to ensure the correct interpretation of the bit encoded

events.

Availability of source files and pipeline complexity The preparation of source files for the metadata aggregation into a single *odML* file is laborious and requires thorough cross checking. Typically, there is no dedicated person responsible for this, but the task is shared between people involved in the experiments. This results in a diffusion of responsibility, as all scientists also have individual projects to care about often focused on a specific subset of the data. For the discussed experiment, not all source files are available for all recording sessions, therefore the generation of an *odML* file will not succeed for part of the recording sessions. This mainly affects sessions that are currently not used for analysis due to other quality exclusion criteria. Most importantly, the complexity of the *odML* generation leads to a situation where most scientists are not able to properly assist in the curation process.

In addition to the factors complicating the publication of a high volume of additional datasets from the same experiment as above, there are additional challenges to be taken into account when implementing a metadata aggregation pipeline of the complexity as presented here:

Multiple contributors In the case of multiple people involved in the curation process this also requires coordinated activity among all of them. Having individual copies of source files and running the metadata aggregation pipeline using different configurations will lead to various, potentially inconsistent metadata collections. Also the updates of metadata need to be tracked and communicated in a reproducible manner.

Robustness & Usability Special attention also needs to be invested making the execution of the pipeline (or minimally: the result of the execution) available to all collaborators, taking into account different needs and software (version) requirements of scientific projects. The pipeline therefore has to be robust to be run on different machines and different system environments and results need to be compatible with all analysis setups in use. For example for the published datasets accessing the data requires consistent `ReachGraspIO` and *Neo* versions.

Structure & Reusability The presented metadata pipeline separates the generation of the structural templates of the *odML* and their enrichment with metadata values. This separation is a good approach during the design of the metadata pipeline, as a general structure first needs to be established before implementing the enrichment. During the production phase of the *odML* generation, however, this separation results in a coupling between the two phases, as for the enrichment of the *odML* structure knowledge about the structure is required. During the runtime of an experimental series frequently also structural adjustments and extensions to the *odML* tree need to be performed as the experimental design was updated or the need to track additional metadata was recognised. The modification and extension of the metadata structure during the production

phase always requires changes on both ends of the *odML* generation process (structure building & enrichment), leading to an overly complex procedure to adjust the pipeline which is hard to maintain. Due to the separation of the structural templates and their enrichment with metadata only the template *odML* files are easy to extract from the presented pipeline for application in a different experiment. The enrichment of the structure is built-in the *odML* generation pipeline and can not be easily transferred to another metadata pipeline as this is code is specific to the metadata source files.

Linearity The presented pipeline for generating a metadata collection relies on the usage of the `ReachGraspIO` (Section 2.5) for a basic interpretation of recorded events using the internal metadata lookup tables. Hard coding metadata processing in this manner is a reasonable approach to make the data quickly usable while the experiment is still under development, but in a stable configuration this introduces cyclic dependencies. In the example presented, the `ReachGraspIO` relies on an (optional) *odML* file to provide metadata on the one side and on the other side the *odML* generation process depends on the usage of the `ReachGraspIO` (see Figs. 2.4 and 2.6). This introduces additional dependencies between software versions and metadata and convolutes the data loading from metadata interpretation and annotation aspect.

Portability The presented metadata pipeline relies on specific versions of software packages, whereas typically in an experimental environment the exact versions used are not necessarily well documented in a rigorous fashion. Therefore executing the pipeline on a different system,e.g., a high performance cluster, requires expert knowledge and additional effort to satisfy all version dependencies of the pipeline with respect to other software packages.

Consistency Working with the published data requires multiple compatible files to be present: the original data files, the metadata collection, and multiple software packages including multiple custom codes, in particular the `ReachGraspIO` and *Neo* in combination with *odML*. In other words, a researcher must use the same versions of all of these components as was used during the *odML* generation process. The version compatibility between all three aspects is typically only neglectfully documented and needlessly complicates the setup of a system to work with the data. Moreover, it locks the users into versions of libraries that may become outdated over time.

2.7 Requirements for maintainable and reproducible metadata management

Based on the experience from publishing the two datasets (Section 2.6) we identify essential requirements for maintainable and reproducible metadata management pipelines in complex, collaborative projects.

R1: Common terminologies To have a foundation for communication between collaboration partners it is important to agree on common terminologies. Within a scientific discipline, this might be given to a sufficient degree from mutual understanding, but as soon as scientists from multiple backgrounds need to interact, agreeing on common terms is an essential first step. Formalizing this in a documented way in the metadata collection is part of this process and documents the terminology also for future generations of scientists but also within the run-time of a single project.

R2: Structured machine & human readable metadata The metadata collection needs to be programmatically accessible to be used for data annotation and querying. However, metadata also needs to be human readable, for manual checks and scanning of metadata. This becomes of special importance in the context of laboratory notebooks and how automatically generated metadata collections can substitute the manual notebooks in the long term. Finally, if metadata are machine accessible this information can automatically be used during data processing and analysis (e.g., for labelling in data visualization). However, this approach only makes sense if the data is additionally human readable, as in the in the long run scientists inspect the processing and analysis results.

R3: Central data and metadata location As discussed in Section 2.6, in a collaborative environment a systematic organization of metadata is essential. Providing access to data and metadata via a central data storage is a straight forward solution.

R4: Version control To be able to communicate efficiently about data and metadata, version control can assist in collaborative curation, but also in making changes and addition to metadata visible. Using version control the usage of identical data and metadata versions by different scientists can be guaranteed as well as changes in the data and metadata collection can be tracked easily. This permits to track updates and changes on the data and metadata side and document these.

R5: Mostly automatic metadata compilation With the large amount of metadata accumulating it is a high priority to automate metadata aggregation as far as possible. In some cases this is not possible in all cases, e.g., when metadata are only available in hand-written form in laboratory notebooks. This type of metadata, however, is a very small fraction of metadata and has to be digitized manually in most cases.

R6: Extendable metadata workflow As discussed in Section 2.6 scientific metadata workflows need to be easily extendable to cope with latest analysis requirements and scientific findings.

R7: Reusability Scientific workflows should be (partially) reusable for related projects to simplify the setup of similar workflows and save time in implementing these.

R8: Standardized & reproducible preprocessing Standardizing preprocessing steps improves clarity and rigour of the preprocessing tools and contributes to the aspect of Reusability (`R7`) of the workflow. Making the individual preprocessing steps reproducible, e.g., by tracking used packages (provenance tracking) and documenting the code version helps making the whole workflow reproducible.

R9: Easy to access data and metadata for non-experts Data and metadata should be easy to use by non-experts of the preprocessing pipeline. In the ideal case an out-of-the-box solution can be presented to the user, who can load the data immediately without installing a series of software dependencies.

R10: Consistent data and metadata Data and metadata should always be consistent. This includes the guarantee that,e.g., preprocessing steps performed on the data should also be reflected in the metadata collection. The concept of version control and reproducibility aid the consistency assurance.

R11: Open source tools In as much as possible, the workflow should use open source software tools that provide their source code to the community. This has the advantage of validation of the correct functionality by other users and permits to fix potential errors. For community software projects also code corrections and enhancements can be suggested and will usually be reviewed and integrated into the tool. Open source tools are free of charge and therefore provide a suitable basis for scientific work to be independent of industry.

2.7.1 Evaluation of presented metadata pipeline

The pipeline used for metadata aggregation and access in Brochier et al. (2018) is described in Section 2.3. In Section 2.7 we identify eleven requirements for maintainable and reproducible metadata management. In the following we evaluate the presented pipeline with respect to these requirements. Table 2.3 summarizes the findings.

The presented metadata pipeline consists of two main components: the `odMLGenerator` for integrating the metadata in a single *odML* metadata collection and the `ReachGraspIO` for data access and integration with metadata. By utilizing *odML* for metadata storage the requirement of having common terminologies (`R1`) is fulfilled on a project level in this pipeline, since *odML* requires the specification and definition of terms within the collection which is then shared between collaborators. This approach also guarantees that the metadata collection is machine and human readable (`R2`), as the *odML* file is xml based and provides tools for user friendly visualization (*odMLtables* (Chapter 3), odML-UI[24], odml_view[25]).

The pipeline itself does not include the automatic storage of data or metadata or related scripts at a central location (`R3`). However, combining the pipeline manually with a central server or a common data hosting platforms is straightforward as data

[24]https://pypi.org/project/odML-UI/
[25]https://github.com/G-Node/python-odml/blob/master/odml/scripts/odml_view.py

Requirement	Brochier et al., 2018
R1: Common terminologies	in project
R2: Structured machine & human readable metadata	yes
R3: Central data and metadata location	no
R4: Version control	no
R5: Mostly automatic metadata compilation	manual initialization
R6: Extendable metadata workflow	minimal
R7: Reusability	partial
R8: Standardized & reproducible preprocessing	no
R9: Easy to access data and metadata for non-experts	partial
R10: Consistent data and metadata	partial
R11: Open source tools	mostly

Table 2.3: Overview of workflow features for Brochier et al. (2018) based on requirements for data and metadata workflows as defined in Section 2.6. The presented pipeline fulfills basic criteria regarding common terminology definition and structured and readable metadata and the usage of open source tools. Other criteria are partially or not met.

as well as metadata files are self contained. By using a central data hosting platform which supports version control, like GitHub[26], GitLab[27] or GIN[28], version control can be automatically introduced at the same time (`R4`). Alternative to using version control in combination with a remote server, a version control system can also be implemented locally, e.g. by using git[29], git-annex[30] or git-lfs[31] to version data locally.

The presented metadata pipeline is automated in the sense that it does not require direct manual interaction (`R5`). However, a fully automated workflow would include automatic initiation of the pipeline upon a change in the source files. This is not the case here, as there is no trigger mechanism included. Instead, the pipeline is triggered by the user and can be performed selectively on a subset of available files. The latter is bound to lead to a situation where the metadata collection of the complete experiment becomes inconsistent. Possible extensions including this would be the integration of centrally hosted code in combination with a web hook to a continuous integration service like Jenkins[32], Travis[33] or CircleCI[34]. In this way adding or modifying metadata sources or the code to generate *odML* files in a repository would automatically trigger the *odML* generation process.

The extendability (`R6`) of the presented pipeline is limited, as changes in the metadata always require code changes in multiple locations (see Section 2.6, 'Additional features', 'Variability across datasets', 'Structure & Reusability') and dependencies between structure and metadata need to be explicitly implemented in the `odML` generation

[26]https://github.com/
[27]https://about.gitlab.com/
[28]https://gin.g-node.org/
[29]https://git-scm.com/
[30]https://git-annex.branchable.com/
[31]https://git-lfs.github.com/
[32]https://jenkins.io/
[33]https://travis-ci.org/
[34]https://circleci.com/

process. This requires expertise knowledge about experiment, its metadata and the metadata pipeline and leads to convoluted code due to the monolithic design of the pipeline.

The separation between building the structure of the *odML* document and the enrichment with metadata makes parts of the structural templates usable for other experiments if they contain similar hardware, software or preprocessing components. However, the code for enriching these template structures is highly specific to the non-standardized formats of the metadata source files in this experiments. Therefore the reusability of the presented workflows for other experiments is limited to parts of the structural template of the *odML* document and a few generic routines (R7).

The compilation of the *odML* document in the presented pipeline is a single Python script that relies on a number of additional Python modules for the aggregation process. No provenance is captured in the process and the final *odML* file does not contain any information about its generation process (R8). In particular, this means that for the *odML* file it is impossible to match the version of the ReachGraspIO or the *odMLGenerator* and the content can not be verified. Possible extension to the pipeline could be the usage of a provenance tracking tool like Sumatra[35] or explicit tracking of file and package versions used in the pipeline.

For accessing the data in combination with the collected metadata in the *odML* format multiple software requirements need to be installed in specific versions: *Neo* for providing the data structure, *odML* for metadata handling, the experiment specific ReachGraspIO for combining the two as well as the original data files with a specific version of the *odML* file. All of these have version interdependencies, which need to be considered when setting up an analysis based on the published data. In particular, version requirements of the analysis process may conflict with that of the code to access the original data files, e.g. by requiring different *Neo* versions. Therefore installation of an analysis pipeline based on the published dataset requires some effort and time and can be demanding for non-experts (R9). Also, this may lead to analysis relying on outdated package versions and not benefiting from bug fixes and upgrades, as dependencies are restricted by the packages required for initial loading of the data.

In the presented workflow the consistency of data and metadata is not explicitly checked. Especially after the *odML* file generation, there is no mechanism to validate the metadata and data (R10). An indirect mechanism to ensure consistency of data and metadata files is to introduce provenance tracking (8) in combination with version control mechanisms (R4).

By using Python for the presented metadata pipeline in combination with the open-source software packages *odML*, *odMLtables* and *Neo* Brochier et al. (2018) rely strongly on open-source software (R11). Some exceptions are preprocessing steps, which are required to generate the metadata source files, e.g., spike sorting using *Plexon* and a custom event detection implemented in *Matlab*. Also the recording setup is based on *Windows* and closed-source recording software by Blackrock Microsystems, but in princi-

[35] https://pythonhosted.org/Sumatra/

ple alternative open-source projects for electrophysiology recordings (e.g., open ephys[36]) and spike sorting (e.g., tridesclous[37]) exist, but need to be evaluated in the context of this and future projects.

In summary, the presented pipeline constructs a metadata collection, which adheres partially to the FAIR principles, as it uses the standardized *odML* format and open source tools for metadata aggregation. However, the process of metadata aggregation presented here is complex and tedious, making the implementation of the FAIR principles labour-intensive. Additionally, the recording data are stored in proprietary formats which inherently only contain minimal metadata and lack interoperability. In the following we will present two projects for facilitated metadata acquisition (Chapter 3) and standardized data representation (Chapter 4).

[36]http://www.open-ephys.org/
[37]https://github.com/tridesclous/tridesclous

Chapter 3

Metadata management - The basis for reproducible science

3.1 Introduction to *odMLtables*

In recent years, the workflows involved in conducting and analyzing neurophysiological experiments have become increasingly complex (e.g. Coles, Carr, and Frey, 2008; Denker and Grün, 2016; Brochier et al., 2018). Several factors contribute to this development. Nowadays, a recording setup is usually comprised of several hardware and software components that are often produced by different companies, or might even be custom made. In addition, due to the technological progress in neuroscience during the last decades the task designs have become more and more sophisticated, as can be observed, for example, when considering experiments mimicking realistic, natural conditions. Neuronal or muscular signals (e.g., eye and arm movements) can be gathered in parallel from multiple optical or electrical recording sites (Nicolelis and Ribeiro, 2002; Verkhratsky, Krishtal, and Petersen, 2006; Obien et al., 2014) together with complex behavioral measures (Jacob et al., 2010; Maldonado et al., 2008; Vargas-Irwin et al., 2010; Schwarz et al., 2014). Moreover, these signals can be experimentally manipulated in intricate ways, e.g., via multidimensional natural stimuli (Geisler, 2008) or sophisticated optical or electrical stimulation methods (Deisseroth and Schnitzer, 2013; Miyamoto and Murayama, 2015). As a result, the amount of information required to fully describe all circumstances under which the experiment was conducted and data was recorded, here collectively referred to as 'metadata', has grown considerably at the same time. Therefore, metadata of neuroscientific studies are increasingly difficult to document and the implementation of specific software solutions to facilitate their management in daily routines involves a lot of time and effort (Zehl et al., 2016). Here the application of the FAIR principles can ameliorate the situation by providing guidelines for systematic and structured metadata handling and documentation. In the following we identify key points for the efficient and sustainable metadata documentation in the context of experiments and provide tools for addressing these issues.

The complexity of collecting metadata originates from two factors: Firstly, the growing heterogeneity of setup equipment alone makes it difficult to fully track the exact

circumstances under which the primary data were recorded and how the recorded signals were processed along an experimental recording session ("black box" effect, i.e., the difficulty to precisely relate inputs and outputs to the equipment). Secondly, the complexity of the signal types and manipulations using various tools within custom signal processing pipelines increases the effort needed for comprehensive metadata tracking across all parts of the recording system and all processing steps. In particular, the hardware components and software tools employed in these experimental setups typically do not provide a complete account of their metadata and store their output in non-standardized representations that impede gaining insights into the details of the recording process. Nonetheless, collecting and providing metadata of an experiment is a necessary step towards replicable experiments and therefore forms the basis for reproducible research (Tebaykin et al., 2017). In this regard, metadata have to be human readable in order to give users semantic access to the data, similar to a traditional lab book. However, only standardized, machine-readable metadata can be systematically reproduced during automatized analysis processes, which makes them a crucial ingredient for tracking the data provenance leading to a research publication.

A software approach to manage neuroscientific metadata is the *open metadata Markup Language* (odML) framework (Grewe, Wachtler, and Benda, 2011). It has been actively developed and extended during the last years to version 1.4, which is considered in the following. The implementation of the *odML* format is based on the generic eXtensible Markup Language (XML) and provides an application programming interface (API) for Python and Matlab. It is designed to organize metadata of arbitrary type into a standardized, hierarchically structured format that is both human and machine-readable. With this, it is possible to organize metadata originating from heterogeneous sources in a unified way and record them in an interoperable format. Providing metadata in such a standardized format along with the data files of an experiment facilitates the collaboration process between members of a scientific project, because metadata can be organized and made available to all members in a unified way, thus supporting rigour and reproducibility of data analysis through standardized and formalized access to the available metadata (Zehl et al., 2016).

Fig. 3.1 shows a generic representation of an example workflow that results in the generation of a metadata collection represented in odML. The starting point are collections of files containing various subsets of the metadata for individual recordings of an experiment (e.g., different recording days). The data in these files are often organized in different formats within a collection, and files and metadata between different collections may differ due to factors such as changes in the experiment. Therefore, it is possible and advisable to construct template structures for the metadata collection to enforce a systematic metadata structuring. However, in practical scenarios often custom scripts are required to populate these templates, e.g., to cover small variations between metadata collections when a certain piece of information is not present for a particular recording. In addition, the metadata collection must be manually enriched by information that is not digitally available in the first place. The outcome of this build

process are *odML* files for each recording, adhering to a uniform template structure. In a final step, these individual metadata collections may be merged into a single *odML* file in order to provide scientists with the ability to perform full metadata queries on the complete experiment. Zehl et al. (2016) provides a complete account of this workflow including practical examples.

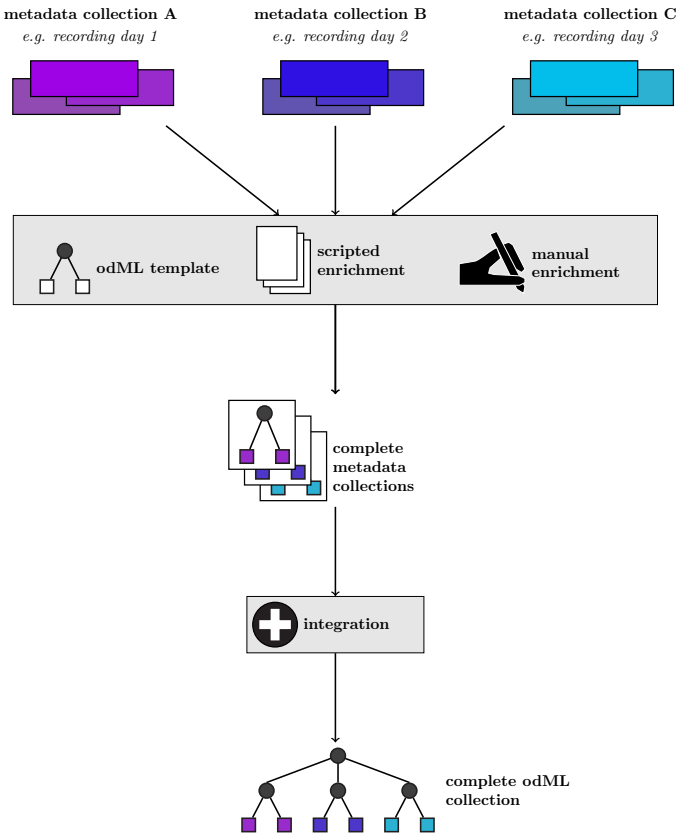


Figure 3.1: Generic workflow of generating metadata collections from source files using the *odML* framework. For a given metadata collection (top row, example metadata collections A-C), metadata are pooled from multiple files and enriched via manual entry (second row). These metadata are converted into individual collections via a scripted approach applying an *odML* template structure (third row). By further integrating individual multiple metadata collections (fourth row), a complete *odML* collection containing all recordings of a particular experiment can be created (bottom row).

Implementing and applying such a rigorous workflow as described in Fig. 3.1 requires programming skills by the scientist. However, metadata handling is often performed by several experimenters with varying computational expertise. Furthermore, extensive manual editing of the metadata files via the present graphical user interface (GUI) included in the *odML* framework tends to be cumbersome for large metadata trees due to their hierarchical, complex organization. While visualization of the hierarchical organization is suited for an overview of the general structure and relation of the metadata, finding or comparing particular values can be difficult if they are distributed in different branches of the hierarchy. Furthermore, editing of distributed entries is laborious,

because a hierarchical organization also requires navigation through the tree to access a particular entry. This makes this metadata management tool inefficient to use in an experimental laboratory where often (i) single particular entries need to be modified manually as the experiment progresses, (ii) a batch of similar entries need to be modified coherently as the data processing progresses. The combinations of all these factors results in many experimental laboratories frequently collecting metadata in flat tabular formats independent of an explicit, underlying hierarchical structure, using tools for generation and manipulation of tables that do not require programming expertise, are widely adopted, readily available and familiar to the experimenters.

Thus, for these purposes a flat tabular representation of the metadata appears to be suitable. It has the advantage of providing easier access to, and simpler visualization of, the metadata than a hierarchical format. Tabular representations of hierarchical structures are implemented in a number of generic software tools for `xml` representation[1]. However, these generic `xml` editors do not provide support for using `xml` to handle scientific metadata in a concise way.

We developed *odMLtables* as a Python package to complement the *odML* framework in simplifying working with, and in particular manually editing, the metadata stored in the hierarchical *odML* format. *odMLtables* facilitates the integration of the *odML* framework into the experimental workflow by converting between hierarchical *odML* and tabular representations in `xls` or `csv` format. These tabular formats are easily accessible via commonly known spreadsheet tools (e.g., *Microsoft Excel*, *LibreOffice Calc*) that enable neuroscientists to manually extend or edit the content of an *odML* metadata file. More importantly, the *odMLtables* package comprises a GUI that guides the user through all functional features of the tool. With this, it also opens access to the *odML* framework for scientists with little or no programming experience. All main functionality to interact with metadata files is directly accessible from the *odML* GUI since version 1.4.0. The software has benefited from the experiences gained in applying it in collaborative projects involving three different experiments collecting electrophysiological data: (i) cortical activity in macaque performing a visually-guided motor task (e.g., Denker, Zehl, et al., 2018; Brochier et al., 2018), (ii) cortical and hippocampal activity in a developmental study in mice (e.g., Bitzenhofer et al., 2017), and (iii) cortical activity in a category learning task in gerbils (e.g., Ohl, Scheich, and Freeman, 2001).

The embedding of *odMLtables* in a real world metadata management workflow is described in Zehl et al., 2016, resulting in a published dataset with detailed metadata descriptions in the *odML* format (Brochier et al., 2018). In both papers the focus resided on the concepts of metadata management and the detailed experiment description, whereas here we complement these studies by a technical tool to implement metadata capture. While the described experiment (an instructed, delayed reach to grasp task with multielectrode recordings from monkey motor cortex) is too complex to present the features and usage of odMLtables, the concepts presented here are abstracted from

[1] See, e.g., https://www.oxygenxml.com/xml_editor/xml_grid_editor.html or http://rustemsoft.com/xfox.aspx

these studies.

To optimally demonstrate the application of *odMLtables* we present seven minimalistic scenarios of practical metadata management using *odML* and odMLtables. Together these scenarios form a complete metadata workflow based on an exemplary multi-day experiment as commonly encountered in neurophysiology (cf. Fig. 3.1), but also other fields of science where data is aggregated in repetitive acquisition cycles (e.g., multiple days of measurements). Moreover, the scenarios are of sufficiently generic nature to transfer them to other situations where metadata information is collected. The first two scenarios demonstrate the first steps for setting up a new metadata workflow and daily metadata collection. Four scenarios deal with the ongoing metadata validation, enrichment and visualization. The last scenario introduces automation of metadata collection and management using *odML* and odMLtables.

In general, *odMLtables* facilitates access to sophisticated metadata management software *odML* for non-programmers and with that optimizes routine manual metadata acquisitions in any laboratory workflow. In addition, *odMLtables* can be used to create visually enhanced tabular overviews of complete or filtered metadata from any hierarchically structured *odML* files. For a scripted metadata approach a Python interface permits also programmers to benefit from *odMLtables* features.

3.2 Software description

odMLtables is a Python package that provides a set of functions for working with metadata descriptions in the *odML* metadata framework, with a particular focus on making these metadata easily accessible for users (Table 3.1). The key approach is to bring the typically complex, hierarchical structure of the *odML* format into a tabular and reduced representation, such that metadata can be more easily inspected or edited. Therefore, at its core, *odMLtables* provides functions to convert between the *odML* format and the corresponding tabular representation which can be represented in the *Microsoft Excel* (`xls`) or the generic comma separated value (`csv`) format (Fig. 3.2). Metadata converted to these tabular formats are accessible via widely used spreadsheet software (e.g., *Microsoft Excel*[2] or *LibreOffice Calc*[3]), such that users are able to intuitively view and edit the metadata. After editing, the metadata can be brought back to the standardized, hierarchical form defined by the *odML* framework (as illustrated in Fig. 3.2).

Next to the functionality of converting between *odML* and the tabular formats, *odMLtables* provides four additional capabilities that address common tasks when working with metadata collections:

[2]https://products.office.com/en-us/excel
[3]https://www.libreoffice.org/discover/calc/

Code version	1.0.0b3
Permanent link to code/repository	https://github.com/INM-6/python-odmltables
Documentation	https://odmltables.readthedocs.io
Support	https://github.com/INM-6/python-odmltables/issues
Programming Language	Python
Dependencies	odML, PyQt5
Research Resource Identifier (RRID)	SCR_016228
Legal Code License	BSD 3-Clause
Logo	

Table 3.1: Overview of *odMLtables* characteristics of the version considered here.

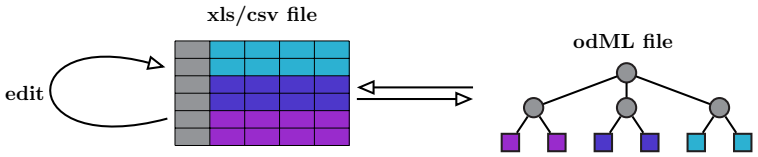


Figure 3.2: Minimal workflow for manually editing *odML* files via odMLtables. Metadata is manually edited in tabular form using spreadsheet software and stored in `xls` and `csv` formats (left). The minimal functionality of *odMLtables* is to convert between such tabular representations and the hierarchical *odML* structure (right). The hierarchical placement of individual metadata entries, i.e., the Sections of the *odML* tree, is encoded in a specific column of the table (gray boxes and circles), whereas the values and attributes of metadata entries, i.e., a Property represented as leaves of the *odML* tree, are stored in rows of the table (colored boxes).

- filtering (or reduction) of a metadata collection to a subset
- merging of two metadata collections
- generation of a basic *odML* structure to facilitate the design of a new metadata collection
- creating a tabular overview across multiple metadata entries within a metadata collection

The functionality of the *odMLtables* can be accessed in one of two ways. First, the API of the *odMLtables* complements the original Python *odML* API (Grewe, Wachtler, and Benda, 2011). As such, *odMLtables* simplifies the scripting of automated metadata extraction and aggregation tasks in an experiment. Second, *odMLtables* includes a GUI that enables non-programmers access to the large majority of functionality offered by the library. In this way, *odMLtables* can aid work with odML-based metadata collections in metadata workflows that do not include scripted processing stages.

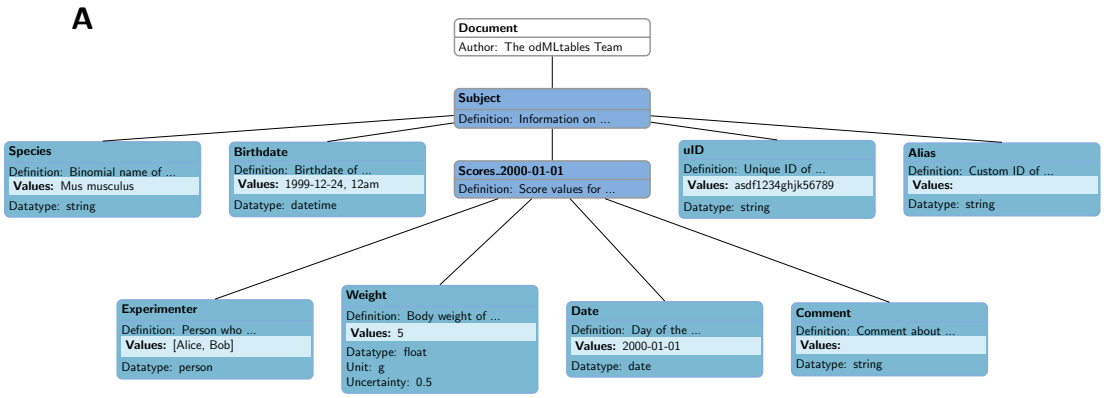
In the following, we describe in detail the structure of the hierarchical and tabular metadata representations, the main capabilities of *odMLtables* illustrated by means of the GUI, and its internal architecture.

3.2.1 Tabular representation of the *odML* format

odMLtables converts the hierarchical *odML* structure (Fig. 3.3A) into a specific tabular (flat) representation (Fig. 3.3B), stored either in the `xls` or `csv` format. In this format, each row corresponds to one particular value entry. The columns further describe the Property and Section each value belongs to, e.g. the Property name, the Section and Subsections the Property belongs to, the physical units, or the Property definition. The hierarchy of Sections in which a Property is located in the original *odML* structure is represented by a path construct, where individual Section names are delineated by the '/' character. For increased readability, repetitive information (i.e. identical information to the cell above) is optionally displayed only at the first instance (e.g., 'Path to Subject' entry ('/Subject') in row 3, 4 and 5 in Fig. 3.3B). By default, the column headers are predefined (Fig. 3.3B, first row), however, the header names can also be customized as long as a mapping between the predefined names and the custom names can be provided. The order of the columns of the table can be customized since the column header names are used to associate columns with attributes of the hierarchical *odML* structure. The *odML* Document attributes 'author', 'date', 'version' and 'repository' are handled separately and are placed in the top row of the tabular *odML* representation.

3.2.2 Software functionalities

odMLtables is a tool that provides 5 functionalities surrounding work in creating and accessing metadata collections. In the following, we describe the capabilities of these features, while their use is put into the context of a typical workflow in Section 3.3.



B

Document Information	author	The odMLtables Team	date		repository		version
Path to Section	Property Name	Value	Data Uncertainty	Data Unit	odML Data Type	Property Definition	Section Definition
/Subject	Species	Mus musculus			string	Binomial name of ...	Information on the investigated subject
	Birthdate	1999-12-24, 12am			datetime	Birthdate of the ...	
	uID	asdf1234ghjk56789			string	Unique ID of ...	
	Alias				string	Custom ID of the ...	
/Subject/Scores_2000-01-01	Experimenter	Alice			person	Person who perf...	Score values for health status estimation
		Bob					
	Weight	5	0.5	g	float	Body weight of the ...	
	Date	2000-01-01			date	Day of the ...	
	Comment				string	Comment about ...	

Figure 3.3: Mapping of an *odML* structure in (A) hierarchical metadata format to (B) tabular format. Individual attributes of the *odML* entities are represented in different columns in the tabular representation (e.g. 'Section Definition', 'Property Name', 'Data Uncertainty', compare color code). Document attributes ('author', 'date', 'repository' and 'version') are described separately in the first row of the tabular representation. The hierarchy of Sections is captured in an additional column ('Path to Section') describing the path between the *odML* Document and the current Section. Each metadata entry in the hierarchical format corresponds to a single row in the tabular format. Items of a list are treated as individual entries.

All main features of *odMLtables* are available via the *odMLtables* GUI (Fig. 3.4). Upon launching the application, it presents the user with 5 buttons, each leading to a series of dialogues (wizards) to perform a specific *odMLtables* functionality. For the more complex dialogues that include a large number of parameters to set, the GUI offers to save and load the dialog configuration to efficiently re-run a functionality with given parameters.

In addition to the functionality offered by the GUI, the Python programming interface of *odMLtables* offers additional features, most notably, the ability to customize the default values for *odML* data types. The default values can be displayed using a highlighted coloring scheme to indicate to the researcher that a Property currently contains a default value (for details, see the *odMLtables* documentation[4]).

The main features of *odMLtables* are described in detail below and are referred to as feature F1- F5:

F1: Convert between *odML* and table format. This function converts metadata collections between the representations in the different file formats odml, `xls`, and

[4]https://odmltables.readthedocs.io

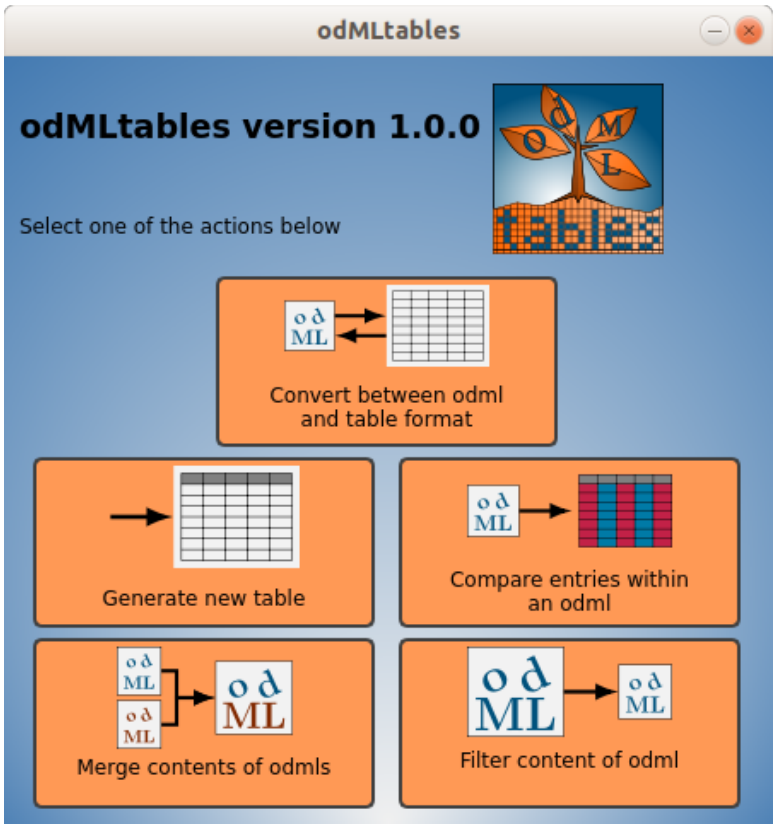


Figure 3.4: Main window of the *odMLtables* GUI. The interface gives access to the main functionalities available by the tool: Converting files from hierarchical to tabular (flat) representations, generating an empty generic *odML* table (template), comparing entries within a metadata collection, merging contents of two collections, and selecting a subset of a metadata collection (filtering). Each button starts a series of dialogues (wizards) that guide the user through the corresponding process.

`csv`. For the conversion to and from the tabular formats (`xls`/`csv`) a specific formatting of the table is required in order to interpret the table as hierarchical *odML* structure (see Section 3.2.1). Nevertheless, *odMLtables* allows for a certain degree of flexibility in order to give researchers the ability to design tabular formats to best fit their workflow. In particular, this encompasses the inclusion or removal of certain optional columns, the arrangement of columns, column headers, or the coloring scheme. Note however, that for the reverse conversion from a tabular format back to the *odML* format, these customizations need to be known (e.g. custom column names, see Section 3.3).

F2: Generate new metadata collection table. This function generates and saves an empty, generic (template) *odML* structure in the `xls` format. This generic structure provides a good starting point to design a metadata collection or template structure in a tabular format providing the required tabular structure for conversion to a hierarchical *odML* structure. Similar formatting options can be applied to the table as indicated above.

F3: Generate overview across entries within a metadata collection This function creates a chart listing multiple entries within a single metadata collection. It is

intended to develop overview sheets containing similar Properties, e.g. the animal weight at different ages. The generated table does not follow the tabular *odML* format and can therefore only be used for visualization and not for conversion into the hierarchical *odML* format. Using common spreadsheet software the comparison table can be saved as a figure and printed for usage in a laboratory notebook.

F4: Merge contents of two metadata collections This function allows to merge multiple files (*odML* format) into a single file. Here, by default, Sections, Properties and values are added to existing entities during merging. However, for values of coinciding Properties the option exists to overwrite values during the process of merging.

F5: Filter content of a metadata collection This function reduces the size of an *odML* file based on a filter mechanism, which can include multiple steps of filtering and custom filter functions to select only specific parts of an *odML* structure. The filter mechanism e.g. can extract all Properties containing no values to present the experimenter potential missing entries in the metadata collection.

3.2.3 Software architecture

In the following, we explain the internal structure of the *odMLtables* software. For a detailed description, see the function reference in the *odMLtables* documentation.

The core of *odMLtables* is the `OdmlTable` class, which provides the main functionality for loading and saving metadata collections in the different file formats. It implements basic operations on the loaded metadata independent of the file format they originate from. Within the class, metadata are internally represented as a list of dictionaries, where each dictionary corresponds to an *odML* Property. Functions that modify the metadata collection, like merging and filtering, act directly on this internal dictionary representation. The two tabular formats `xls` and `csv` require additional information regarding the table layout when being saved to disk, e.g., the color scheme. Therefore, two subclasses of the `OdmlTable` class (`OdmlXlsTable` and `OdmlCsvTable`) carry these additional output settings. Finally, a separate `CompareSectionTable` class implements the function for comparing Properties within one *odML* structure. As for the `OdmlTable` class, two specific subclasses for `xls` and `csv` output are defined to capture layout information (`CompareSectionXlsTable` and `CompareSectionCsvTable`).

One feature of *odMLtables* in generating `xls` files is to highlight a value entry if it corresponds to the default value of the corresponding Property's data type. However, the *odML* library itself does not specify such default values for all of its data types. Moreover, it is not mandatory, nor always desired, to specify a data type in the *odML* in all circumstances, e.g., when leaving a value empty. Therefore, *odMLtables* provides functionality to work with default values for data types in the `OdmlDtypes` class. It manages the data types, synonyms, default values, and value conversions. The class is used for entering default entries when loading empty values from a tabular representation, and for default value highlighting.

In addition to the core module, *odMLtables* provides a GUI that exposes most func-

tionality of the core module. The GUI is based on the PyQt5[5] framework and consists of a main window (Fig. 3.4) and five wizards (see Section 3.2.2). Each wizard inherits from the `OdmltablesWizard` class, which provides helper functions and error handling. The `Settings` class stores the current user settings for calls of *odMLtables* core functions, and provides functionality to save and restore user settings between different executions of the GUI.

3.3 Embedding *odMLtables* in data acquisition and analysis

While most scientists would agree that accurate records of the minute details of an experiment are the foundation of good scientific practice, in the everyday routine of an experimental electrophysiology lab it is difficult for the scientist to record, sort, and maintain the wealth of metadata information that accumulates during an experiment. While the *odML* format is suitable for storing metadata information from different sources, lacking to date is a set of tools that allows the scientist to create, manipulate and visualize the data stored in this format. In the following, we present commonly encountered scenarios involving metadata handling that originate from our collaborative work. These scenarios touch the issues of how to design the hierarchical structure to store and organize the metadata, how to practically enter metadata before, during or after the experiment, and how to create a comparison of rich metadata buried within the *odML* structure. It turns out that for each of these scenarios a flattened tabular representation of the metadata is a practical solution that feels intuitive to the user. In the following we demonstrate how to implement these scenarios that consist of combining operations in *odMLtables* and a spreadsheet program. All scenarios are also available as an interactive Jupyter Notebook[6] accessible via the *odMLtables* documentation[7].

Scenario 1: How to generate a metadata template without programming

In conducting animal experiments, a typical scenario where metadata are collected manually on a daily basis is the creation of an animal *score sheet*. Such score sheets record quantitative, and in part also qualitative, measures that are collected in order to document and judge the animal's health and state over the duration of the experiment. Often, these sheets are an obligatory piece of documentation of the experiment, such that only the availability of a defined workflow to create score sheets guarantees their consistency over multiple years and different experimenters. For example, for mouse experiments, typical measures are the body weight, water intake and breathing frequency, many of which can be used to assess the health of an animal, e.g., by calculating a health score for each mouse (Foltz and Ullman-Cullere, 1999; Burkholder et al., 2012). In Fig. 3.3 we depicted how metadata of a single, minimized score sheet can be integrated into an *odML* document containing collective information on a subject.

[5] https://wiki.python.org/moin/PyQt
[6] https://jupyter.org/
[7] https://odmltables.readthedocs.io/en/latest/tutorial.html

The measurements for such score sheets are typically easy to perform, and for this reason may be conducted by a number of different people in the lab. Therefore, the daily process must be simple, intuitive, and robust in order to be conducted by all members of the group. Collecting the information in a table format using common spreadsheet software tools, such as *Microsoft Excel* or *LibreOffice Calc*, satisfies these requirements.

To guarantee a consistent structure of such a score sheet, initially a template needs to be set up, i.e., a table containing the measures that are to be recorded on a single day. In order to accomplish this, as a first step we generate an empty template table using odMLtables. To improve the readability, we enter custom column names in *odMLtables* to create the table ('Section' instead of 'Path to Section', 'Measure' instead of 'Property Name', 'Unit' instead of 'Data Unit', and 'Type' instead of '*odML* Data Type'). Also we omit the attributes 'Section Definition', 'Property Definition' and 'Data Uncertainty' in the context of these example scenarios. As second step, using a spreadsheet, we design the metadata structure for a single score sheet as shown in Fig. 3.5. The value field for each entry can be either left empty or a default value can be entered. The latter case is interesting for values that are likely to be constant for the majority of experiments, e.g., the name of the experimenter. Since the colors of a table saved in the `xls` format are ignored when converting to the *odML* format, it is possible to use arbitrary color coding within the spreadsheet software to improve the readability of the table for the experimenters entering the values.

Section	Measure	Value	Unit	Type
/Subject/Scores_YYYY-MM-DD	Experimenter	Alice		person
	Weight		g	float
	Date			date
	Comment			string

Figure 3.5: Template score sheet. The template score sheet contains the measures required for each measurement day, including optional default values (here: "Alice" for "Experimenter" and "g" as unit for "Weight").

We designed the template table such that it matches the properties of the minimized score sheet section already depicted in Fig. 3.3. Notice that in the template the entry for column 'Section' already includes a parent section to reference the animal (cf., Fig. 3.3). This is convenient for defining the position of each score sheet in the *odML* hierarchy to simplify a later merging process (cf., scenario 2).

Scenario 2: Collecting daily observations in a common *odML* structure

Once the template from scenario 1 is complete, it is copied to a new file on each measurement day, and the copy is filled out by the person taking the measurements. To avoid that metadata are spread across multiple files and potentially multiple locations, we aim to gather the data from multiple days into a single *odML* file. To achieve this, we use *odMLtables* to first convert the individual `xls` file containing an individual score sheet into the *odML* format, and to subsequently merge these into a common *odML* structure spanning multiple recording days.

Specifically, the conversion from the `xls` to the *odML* format we use the *odMLtables* feature F1 (for details of *odML* features F1- F5, see Section 3.2.2). After the conversion, the current score sheet present in *odML* format is merged into the common *odML* document collecting the complete information of an animal using feature F4 on a daily basis. This extends the *odML* structure of the subject document by an additional Section each recording day. Note that this is possible because the first column of individual score sheets (Fig. 3.5) not only provides a unique Section name for each score sheet, but also indicates the location of the *odML* Section in the hierarchical structure of the subject document, (e.g., "Subject/Scores_2000-01-01"). The result is a single *odML* file containing measures collected on all recording days while the source files generated each day can be archived.

The metadata collection containing the merged score sheets of 2 recording days might look like the following:

```
Document "mouse-score-sheets"
└─ Section "Subject"
   ├─ Property "Species":  Mus musculus
   ├─ Property "Birthday":  1999-12-24 12:00:00
   ├─ Property "uID":  asdf1234ghjk56789
   ├─ Property "Alias":
   ├─ Section "Scores_2000-01-01"
   │  ├─ Property "Experimenter":  ["Alice", "Bob"]
   │  ├─ Property "Weight":  5g
   │  ├─ Property "Date":  2000-01-01
   │  └─ Property "Comment":
   └─ Section "Scores_2000-01-02"
      ├─ Property "Experimenter":  "Bob"
      ├─ Property "Weight":  5.5g
      ├─ Property "Date":  2000-01-02
      └─ Property "Comment":  "Small scratch at the right ear"
```

Scenario 3: Create a tabular representation of the *odML* file for better viewing using the color options

Once the recordings for a number of animals were performed and the corresponding metadata collection is completed, data and metadata should be shared among collaborators in a common repository. In order to get an overview of the data obtained across different animals, the metadata of each animal can be converted into the `xls` format to simplify the inspection of the associated metadata using spreadsheet software (cf., also, Fig. 3.3B). Here, *odMLtables* provides the option to use color coding and highlighting of default / missing values to improve the readability (Fig. 3.6).

Scenario 4: How to filter a subset of an *odML* file to edit it later on

As the common *odML* structure grows day by day it is of advantage to extract specific subsets of *odML* values of interest for visualization using the tabular format. Instead

Section	Measure	Value	Unit	Type
/Subject	Alias			string
/Subject/Scores_2000-01-01	Comment			string

Figure 3.6: Metadata collection filtered to show only Properties with an empty value. Missing values entries are highlighted in red by odMLtables.

of visualizing the whole metadata collection to periodically verify that all Properties are filled with a value, we can extract a subset of the collection and visualize only the relevant (e.g. empty fields) entries. For this, we use *odMLtables* feature F5 which can be used to generate an *odML* that contains only Properties without value information specified. We then convert this reduced *odML* into a tabular `xls` representation using *odMLtables* feature F1. The generated table, as shown in Fig. 3.6 indicating the two empty properties in the *odML* structure of scenario 2, can be visualized using spreadsheet software and, in case of values not being filled, these can be directly edited manually.

Scenario 5: Merging the edited subset back into the original structure

The enriched `xls` sheet generated in step 4 should now be merged back into the common *odML* structure. For this, we convert it back into the *odML* format and use the *odMLtables* merge feature F4 to replace the edited values in the common *odML* structure with the edited ones. Here, *odMLtables* merges the two *odML* files by extending the *odML* structure and appending metadata entries when the same Property is present in both files. However, when modifying already existing metadata entries in the filtered version this would result in duplication of entries. Therefore, *odMLtables* offers the possibility to overwrite already existing metadata entries when merging two *odML* structures. Note that a selective merge of a subset of metadata can be achieved by first filtering the file to be merged using feature F5 .

Scenario 6: Compare entries in the *odML* file for data screening and lab book usage

In addition to the complete metadata representation as presented in scenario 3, it is possible to generate a reduced overview table containing only plain values of selected Properties. This feature can be used to create a tabular display of Properties of interest (e.g., weight of a specific animal, experimenter who performed the experiment and comments regarding the measurement) in rows for the individual recordings (days) in columns. An example of such a table is given below:

	Scores_2000-01-01	Scores_2000-01-02
Date	2000-01-01	2000-01-02
Weight	5.0g	5.5g
Experimenter	Alice, ...	Bob
Comment	Blood sample was taken [...]	Small scratch at the right ear

This type of overview tables can also be printed and used as part of the mandatory documentation of the experiment in a written or printed lab book. This way, the recorded data only need to be documented once in a digital fashion and consistency between documentation and digitally available metadata is guaranteed.

Scenario 7: Automatized processing of metadata collections

After completion of an experiment covering many recording days, the processing steps presented in scenarios 1-6 can be performed in an automatized fashion on the complete metadata collection to generate a comprehensive metadata document and corresponding overviews. While it is possible to perform this action using the graphical user interface, an automated approach has the advantage that it can be repeatedly executed when one of the original files changes, e.g., by a retrospective update of metadata or loss of the generated metadata files. In addition an automated approach is more robust against errors introduced by the manual operation and can be at least partially reused for subsequent experiments.

By use of the *odML* library together with the *odMLtables* Python API, users have a rich collection of functions to manipulate and convert metadata stored in the *odML* format. In this specific example, we show an example script in Code Listing 3.1 that loads all daily animal score sheets, adds them to a common metadata structure and exports the final document into an overview and comparative `xls` sheet for visualization. The code demonstrates the metadata handling workflow by structuring it into a sequence of three generic functions, which can be of use in creating related workflows for different projects.

Improved handling and visualization of complex metadata structures

Up to now we demonstrated the basic mechanisms of *odMLtables* based on highly simplified examples presented above. In a real world example, however, metadata collections are inherently complex and corresponding metadata collections can easily encompass thousands of values. A publicly available example of this are electrophysiological recordings of macaque monkeys performing a reach to grasp task that include a rich metadata collection stored in the original *odML* files as well as the corresponding `xls` representation created by *odMLtables* (Brochier et al., 2018). We demonstrate the usage of *odMLtables* to select and visualize a subset of the complete metadata collection as well as generation of overview tables in an interactive Jupyter Notebook in the *odMLtables* documentation[8].

3.4 Discussion

We presented the *odMLtables* software, which facilitates the use of the *odML* metadata format in everyday experimental and data analysis work. In scenarios 1 to 7 we present the features of *odMLtables* within a simplified real world example, namely the definition

[8] https://odmltables.readthedocs.io/en/latest/tutorial.html

```
import os.path, glob
import odmltables as odt

def csv_to_odml(csv_file):
    """ Convert a score sheet from csv to odML format. """
    # initialize an OdmlTable object for handling metadata
    table = odt.OdmlTable()
    # specify headers used in the score sheet csv files (here: Section, Measure, Unit and Type)
    table.change_header(Path=1, PropertyName=2, Value=3, DataUnit=4, odmlDatatype=5)
    table.change_header_titles(Path='Section',PropertyName='Measure', DataUnit='Unit',
    ↪  odmlDatatype='Type')
    # load from csv format and save in odML format
    table.load_from_csv_table(csv_file)
    table.write2odml(csv_file[:-4] + '.odml')

def merge_odml_files(file1, file2, overwrite_values=False):
    """ Merge one odML file (file2) into another odML file (file1) """
    # load first odML file
    table1 = odt.OdmlTable(file1)
    # merge file2 into table1
    table1.merge(odt.OdmlTable(file2), overwrite_values=overwrite_values)
    # overwrite file1 with the merged score sheets
    table1.write2odml(file1)

def visualize_as_xls(odML_file):
    """ Generate an xls version of an odML file for visualization purposes """
    table = odt.OdmlXlsTable(odML_file)
    # optional: change the color options in the output table
    table.first_marked_style.fontcolor = 'dark_green'
    table.second_marked_style.fontcolor = 'dark_teal'
    table.highlight_defaults = True
    # write to xls format
    table.write2file(os.path.splitext(odML_file)[0] + '.xls')

def generate_overview(odML_file):
    """ Compare entries with same structure across an odML file """
    table = odt.compare_section_xls_table.CompareSectionXlsTable()
    table.load_from_file(odML_file)
    # specify all score sheet sections to be compared here
    table.choose_sections('Scores_2000-01-01', 'Scores_2000-01-02')
    # save to different odML file
    table.write2file(os.path.splitext(odML_file)[0] + '_overview.xls')

# extract all metadata files present in this metadata folder
folder = 'mymetadatacollection/'
source_files = sorted(glob.glob(folder + '/*.csv'))

# convert all source files
for source_file in source_files:
    csv_to_odml(source_file)
# merge score sheets into animal info document
for score_sheet in sorted(glob.glob(folder + '/score_sheet*.odml')):
    merge_odml_files(folder + '/animal_info.odml', score_sheet, overwrite_values=True)

# create visualization and comparison tables
visualize_as_xls(folder + '/animal_info.odml')
generate_overview(folder + '/animal_info.odml')
```

Code Listing 3.1: Program to assemble a target *odML* document covering metadata of multiple recording days by pooling information from multiple `csv` files and generate visualizations and overviews. Individual functions are automatizing functionalities presented in previous scenarios.

of an animal score sheet and with this the controlled routine collection of corresponding metadata. More specifically, we showed in scenario 1 the setup of a template for an animal score sheet in the `csv` format and its conversion to *odML* (F1, F2). In the next scenario, we used this template to routinely collect the animal's health measures and aggregated them in a single *odML* file per animal (F1, F4). Besides a simplification of metadata acquisition in the `csv` format, we showed in scenario 3 the benefits of a colored tabular representation for visual inspection of the collected score sheets (F1). In scenario 4, we demonstrated how supplements of metadata values can be easily added by extracting a the missing metadata entries from the complete collection (F5). Subsequently we demonstrated in scenario 5 the integration of the amended metadata back into the complete collection (F4). We generated a compressed overview table, summarizing the metadata from different routine collections in a concise, laboratory notebook suitable fashion in scenario 6 (F3). Finally, in scenario 7 we discussed the amortization of the workflow presented in the previous scenarios and provided code examples showcasing the *odMLtables* Python interface.

As *odMLtables* can be used by programmers and non-programmers alike, *odMLtables* simplifies the development of comprehensive metadata management in the scientific community by offering user-friendly interaction with the *odML* format. In this way, its usage is intended to improve reproducibility and replicability of experiments and to facilitate cooperative work, both within labs and across different laboratories. Complementing the model scenarios above, in Fig. 3.7 we summarize and generalize the use of individual components of *odMLtables* during the course of an entire experiment. Although the presented scenarios are set in a neuroscientific environment, *odML* and *odMLtables* are indifferent with respect to the scientific discipline and can therefore be used for metadata management in different contexts. In the following, however, we discuss specifically its embedding into a tools landscape developing in the field of electrophysiology.

Although, the real world workflow described in Brochier et al., 2018 and Zehl et al., 2016 as well as the minimalistic workflow presented here are all set in the field of animal experiments covering multiple days, *odML* as well as *odMLtables* can be used for metadata management in a broader context. For example in the context of experimental research the benefits of *odMLtables* can be seen at all stages of an experiment: from setting up a specific metadata structure in the preparatory phase, manual enrichment of the metadata collection during the experiment, through to the generation of overviews and summaries from metadata collections during data analysis. Also, for publicly available datasets with an *odML* metadata collection, *odMLtables* can be used to create a tabular representation of the *odML* files to quickly scan the metadata of the experiment.

3.4.1 Performance estimation

Since the release of the original version, *odML* has been used in various projects for storing metadata as they become available during data acquisition or analysis (e.g. in

the NIX[9] and RELACS[10] projects), as metadata schema in the EEGbase database[11] (see also Mouček et al., 2014), and as a part of the metadata data pipeline as described by Zehl et al., 2016 and Brochier et al., 2018. The advantage gained by comprehensive metadata management using *odML* can be demonstrated by a small example based on a published dataset (Brochier et al., 2018) for which detailed metadata are stored in the *odML* format. Accessing information about the number of neurons recorded on different electrodes contained in the *odML* files using common desktop hardware requires approximately 0.5 seconds for this dataset using the *odML* iteration and filter mechanism. Extracting the same information not from the *odML* metadata but from the original data files using the Python library *Neo* version 0.7.1[12] requires about 25 seconds using the *Neo* filter and annotation mechanism. Comparing these times, the usage of *odML* in this example gives a speedup of a factor 50. However, for a fair comparison also the time for *odML* generation needs to be taken into account, which for a dataset of this complexity is typically on the order of 10 minutes, considering that the generation process needs to read the data files and a number of associated files (Zehl et al., 2016), perform various quality checks or automated preprocessing checks. Comparing this conservative estimate of the generation time of the *odML* file, the access time using the *odML* format and the access time using the original data files shows that using the *odML* format pays off after 25 times of metadata access. This is a relatively small number of metadata accesses for a single dataset considering the relevance of metadata in multiple steps of the experiment, e.g. exploratory analysis and parameter scans in analyses runs, and collaborative work, where different people access the same metadata on different computers. In the latter setting using *odML* is also of advantage because the potentially large original data files might not be present on all computers of all collaborators, whereas *odML* files are much smaller in file size and can therefore be shared more easily, e.g., via a version control system like git[13].

3.4.2 *odMLtables* as conversion tool

One may argue that extending the existing *odML* editor to support a flattened view on the metadata is a more direct and efficient way to implement tabular representations, as opposed to a converter (such as odMLtables) between formats. However, such a solution has direct implications on (i) the maintainability of the tool, (ii) its adoption by the community, and (iii) its interoperability in the heterogeneous types of workflows typically encountered in data acquisition. Regarding (i), the development of graphical editors for tabular data is a time-consuming endeavor and leads to a complex code base that is difficult to maintain. This is even more true in a scientific environment, where software maintenance is often left to persons who are not expert in GUI programming and design patterns for graphical applications. Regarding (ii), spreadsheet software is

[9] https://github.com/G-Node/nix, RRID:SCR_016196
[10] https://github.com/relacs/relacs
[11] http://eeg2.kiv.zcu.cz:8080/home-page?1, RRID:nif-0000-08190
[12] https://github.com/NeuralEnsemble/python-neo/releases/tag/0.7.1
[13] https://git-scm.com

already commonly used in laboratory environments to track metadata, and experimental scientists are used to efficiently use these tools in their daily routine. Therefore, integrating such software in a digitized workflow, rather than proposing an entirely new user-facing tool, is bound to lower the threshold for adoption in a laboratory. Finally, regarding (iii), data acquisition workflows in an experimental environment are often subject to constraints set by the individual formats in which metadata are generated by the components of the experimental setup. Tabular representations, and in particular those stored in the `csv` format, represent *per-se* one of the most commonly encountered and most simple formats to exchange data. Indeed, the capability to read `csv` data files is provided by the standard libraries of many programming languages, in particular those commonly used in data analysis and scientific computing, such as Python, Matlab, or R. Therefore, being able to convert between human readable tabular metadata generated automatically by various metadata sources of the experiment and their joint representation in a hierarchical *odML* metadata collection is helpful in creating a metadata acquisition workflow that is interoperable with the various components of the experiment. Combining such workflows with version control systems, such as git, to store the hierarchical or tabular metadata representations is a viable option to enable collaborative creation of metadata records, in particular when considering the text-based `csv` or and *odML* formats.

3.4.3 Relation to electronic laboratory notebooks

One particular case where flexible interoperability is in demand are electronic laboratory notebooks (ELNs) which are available from a large range of manufacturers and are becoming increasingly utilized by laboratories (Kwok and Kanza, 2018). Their design is actively being researched in the process of digitizing the research process (Samantha Kanza et al., 2017). ELNs are software tools originally designed to replace the hand-written lab book used in experimental sciences to document experiments, outcomes and analyses by providing a method to electronically enter such metadata in a digitally signed and potentially encrypted fashion that ensures protection from falsification. Some ELNs go beyond this functionality by integrating tightly with laboratory inventory management systems (LIMS) or analysis pipelines (comparisons of selected ELNs can be found in (Rubacha, Rattan, and Hosselet, 2011) and various web resources [14] [15] [16]). One major advantage of ELNs that store hard metadata (Grewe, Wachtler, and Benda, 2011) in form of key-value pairs is that they can be directly digitally accessed in analysis scripts, rather than having to manually copy the information from the hand-written lab book (Zehl et al., 2016). While for some disciplines specialized lab notebook software packages have been developed (Kwok and Kanza, 2018) that are aware of community standards for storing such metadata, most of these packages come with their own format for storing data that can only be accessed via file export functionality or specific APIs. In some disciplines this may be of little importance, since

[14] https://datamanagement.hms.harvard.edu/electronic-lab-notebooks
[15] https://www.labfolder.com/electronic-lab-notebook-eln-research-guide
[16] https://www.gurdon.cam.ac.uk/institute-life/computing/elnguidance

either the metadata records stored in the ELN are not required in the analysis process, or the metadata are captured using a domain-specific ELN that is integrated with functionality to directly perform the analysis steps from within the ELN. Nonetheless, other disciplines, such as neurophysiology, require detailed metadata available in an environment suitable for performing complex, exploratory analysis protocols that go beyond the capabilities of currently available ELNs. Here, *odML* is a potential candidate for implementing such features. In absence of a global standard to record metadata, `csv` represents one of the de-facto standards to export metadata from ELNs in a universal format. For this reason, the conversion to *odML* via *odMLtables* provides access to metadata recorded with ELNs for external analysis pipelines that rely on hierarchically structured metadata collections. The same holds true for the reverse direction, where metadata generated by tools building on the *odML* specifications can be imported into an ELN. For example, the feature of *odMLtables* to create tabular overviews of the metadata (feature F3, see Section 3.2.2) would allow to generate current overview tables in terms of animal score sheets as `csv` that could be directly (and assuming the ELN has an API, even automatically) integrated into the documentation of an experiment contained within an ELN, assuming only basic `csv` import capabilities.

Beyond ELNs, labs increasingly resort to institution-wide databases to manage and record their research activities, and, in some cases, even the data as such. Depending on the architecture, some systems are likely to implement data imports using tabular schemata. One example of such a tool implementing database and processing functionality is DataJoint[17] as a tool to assist in ingesting, combining and analyzing heterogeneous data in a relational database (Yatsenko et al., 2015). It is easy to populate a DataJoint database using tabular data, as described in detail in the accompanying online documentation. For example, one may extract a subset of the metadata in form of a comparison table using the *odMLtables* feature F3, and then incorporate this table into a larger DataJoint database spanning all experiments using a generic function for populating from `csv` tables. In such a fashion, *odMLtables* presents a gateway to integrate structured metadata by the diverse tools used in a laboratory to organize the record keeping of an experiment.

3.4.4 Outlook

The current version of *odMLtables* provides a set of core functions that were identified as necessary in co-designing various data and metadata acquisition workflows in collaboration with multiple laboratories spanning different types of experiments and data modalities. Nevertheless, a number of additional features are envisioned as a result of feedback stemming from these collaborations to extend the range of applications for the tool and enhance its flexibility for heterogeneous metadata workflows. In addition, feature requests are welcome on the project's issue system on github. One next step will be to extend the capability to create tabular comparisons (feature F3) across metadata stores in multiple files. This would give researchers the option to query for metadata

[17] https://datajoint.io/, RRID:SCR_014543

that are distributed over several, even differently structured, *odML* files. For example, in chronic recordings of brain activity accumulated over the course of multiple months, researchers may decide to generate a single *odML* file per recording day, and may want to utilize such a functionality to compare the number of trials and other performance measures across the entire recording period.

A second planned feature addition to odMLtables, related to the previous aspect, is the ability to create complete tabular representations (i.e., feature F1) across multiple *odML* files, and vice versa. To this end, one may implement an additional column next to the *odML* path and Property name that indicates the file in which a certain metadata entry is found. As an example application, one may consider a complex experiment where metadata originating from different parts of the experiment are stored in separate *odML* files, but a large overview table is desired for manually browsing the metadata. While this is already possible by merging (F4) individual files and then converting (F1) the table, the information about the origin of metadata in the original file structure is lost.

A third feature addition to *odMLtables* is the automatic generation of Python code based on the steps the user performs in the graphical user interface. For example, this may yield the Python code to perform a certain filter operation designed in the GUI. This would simplify the automation of metadata processing without specific knowledge about the *odMLtables* API.

For communicating the structure of a complex metadata collection to new collaborators neither tabular nor hierarchical views have been found to be efficient. For this, a graphical representation of the metadata structure is likely to be more useful, especially for large metadata collections. For this reason, a fourth addition to *odMLtables* would be to introduce a common graphical representation as new output conversion format.

Lastly, as a fifth feature addition, *odMLtables* could assist scientists in defining the links between data and metadata in an experiment. Typically, several metadata are accumulated from various sources in an experiment that are directly related to one particular part of the data, and in fact, may be crucial in performing data analysis. For example, the signal recorded from a particular electrode may contain the impedance as measured by the manufacturer as well as noise estimated from a pre-processing step. Due to the heterogeneity of experiments and metadata descriptions, it is currently not feasible to establish these connections between data and metadata automatically, e.g., using a predefined mapping based on Property values. Instead, the mapping is carried out manually by implementing customized code that annotates data with metadata during the loading process. Even when data can be loaded via standardized data framework (e.g., Neo[18], see also Garcia, Guarino, et al., 2014), the annotation of data objects with metadata taken from a standardized metadata collection (e.g., odML), has to be performed independently (see Fig. 3.7). This complicates the process of reading data, and is not transparent to an external user. A possible way of reducing the implementation effort to create experiment-specific annotation of data with metadata, would be to

[18]https://github.com/NeuralEnsemble/python-neo, RRID:SCR_000634

store the relations between data and metadata directly in the metadata structure. For example, using odMLtables, we suggest to add supplementary fields to the table that directly link blocks of metadata to specific data, e.g., to channels with a certain channel ID or to events with specific IDs. In this way, compact objects containing both data and selected metadata could be loaded using a single, generic loading routine. Moreover, by providing *odMLtables* with a feature to export to NIX[19], e.g. using the odML-NIX conversion tool[20], as an additional output file format that combines odML-like metadata with primary data (Stoewer et al., 2014), such that combined data/metadata objects could be easily serialized to disk.

Validation of user generated input is implemented on the level of odML: When saving or loading an *odML* file via *odMLtables* or any other method, the *odML* structure is checked for basic integrity (e.g. consistency of data types and values). It is intended to support custom, user defined validations in future releases. That is, users will be provided with the means of defining own valuations to check for specific required Sections, Properties or Values and combinations thereof. These additional validations will be directly stored within the *odML* files. They can be applied to ensure metadata consistency even if the file is handled on a different host or by a different person. For example users would be able to define specific Values as required for a particular Property or make sure a Section tree with an experiment-specific content is present before the file can be saved.

In recent years, the scientific community has begun to recognize the need for developing workflows that enable rigorous data management not only to ensure reproducibility, but also to expedite research through efficient data sharing among scientists. The principles governing corresponding data management practices are summarized under the FAIR (Findable, Accessible, Interoperable, Reusable) principles (Wilkinson et al., 2016). The requirement to make data globally findable has lead to the emergence of multiple resources commonly grouped under the term "Knowledge Graph", referring to a graph-like linkage of metadata through an appropriate ontologies, for example, as done in the Knowledge Graph of the Human Brain Project[21]. The resource description format, RDF[22], is a semantic web technology that provides one possible standard interface to populate such metadata graphs (cf., Fig. 3.7). The complexity of creating RDF descriptions from scratch can be simplified by exploiting the functionality of *odML* to export RDF schemata from *odML* files. In this context, *odMLtables* can be incorporated as a bridge to support researchers in easily entering predefined metadata schemata to expose their data records in large-scale Knowledge Graph infrastructures.

odMLtables is actively developed and a comprehensive documentation including a tutorial is available for release versions on ReadtheDocs[23] and the latest version can be obtained from GitHub[24]. Future developments of *odMLtables* include the ongoing

[19]http://g-node.github.io/nix, RRID:SCR_016196
[20]https://github.com/G-Node/nix-odML-converter
[21]https://www.humanbrainproject.eu/en/explore-the-brain
[22]https://www.w3.org/RDF
[23]https://odmltables.readthedocs.io/en/latest
[24]https://github.com/INM-6/python-odmltables/

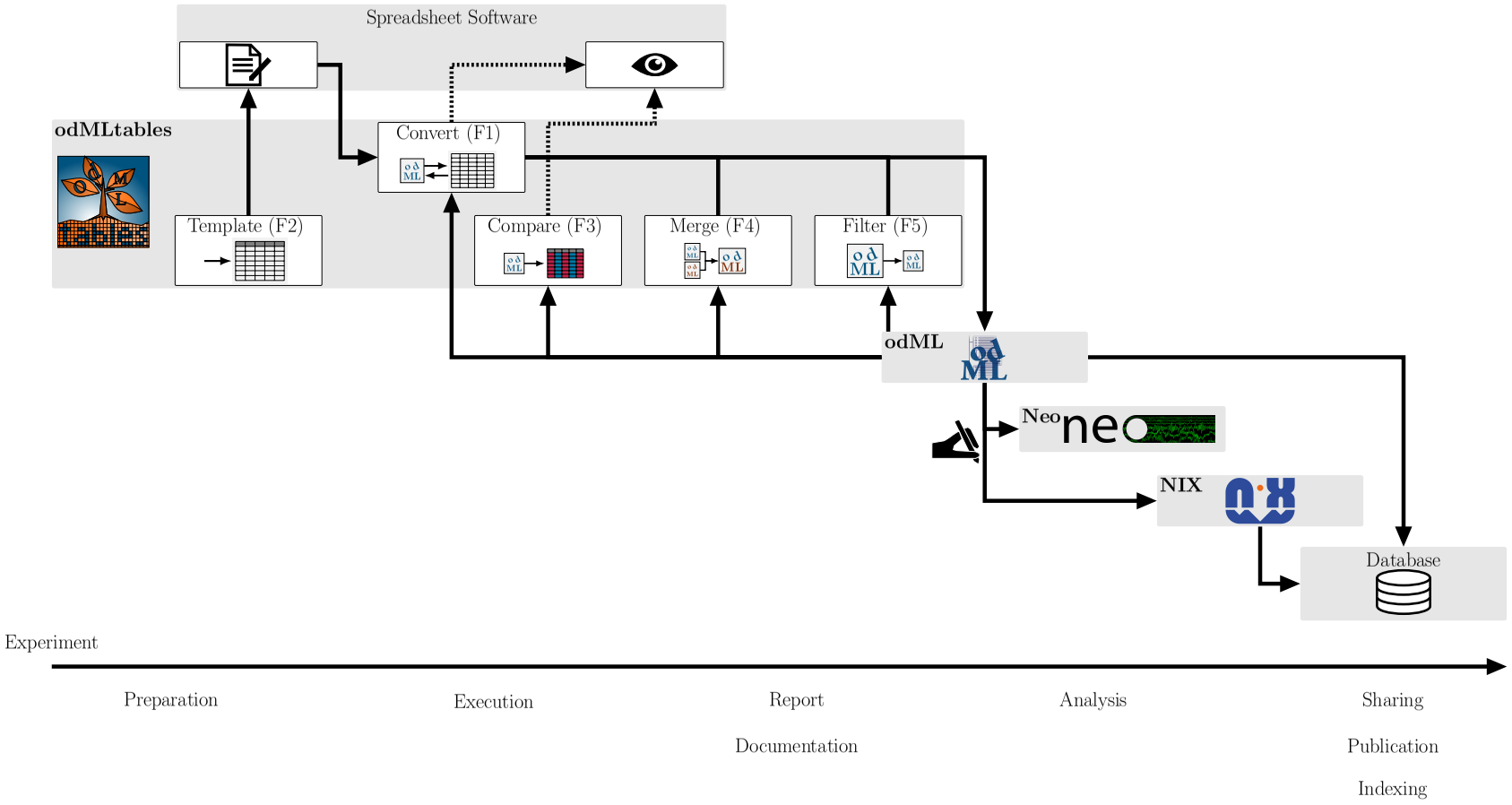


Figure 3.7: Integrating *odMLtables* and other software tools in the different stages of an experiment from preparation to publication. During the preparation of an experiment *odMLtables* in combination with spreadsheet software is used to develop an experiment specific structure of the metadata collection (templates, F2). During the execution and documentation of the experiment, *odMLtables* converts (F1) between the tabular and *odML* representations. The compare functionality (F3) is used to generate overviews of *odML* Properties across different Sections of a metadata collection. The filter (F5) and merge (F4) functionalities are used to create and merge subsets of *odML* collections, respectively. For analysis and sharing, data can be represented using the *Neo* framework and annotated with metadata from the *odML* metadata collection using custom scripts. This combined representation can be saved in a single format using the NIX framework, e.g., to share of data and metadata via a database. In parallel, metadata collections can be incorporated in databases, for example using an export of the *odML* to the RDF standard.

embedding of *odMLtables* in different neuroscientific data and metadata aggregation workflows, and, as a long term prospect, *odMLtables* is planned to become a fully integrated component of the *odML* and NIX libraries.

By permitting the easy access, modification and visualization of metadata collections based on the *odML*, *odMLtables* facilities the implementation of the FAIR principles for scientific data management and stewardship (Wilkinson et al., 2016) in an experimental environment. Additionally it promotes the usage of standardized formats, driving the community further towards the application of the FAIR principles.

Chapter 4

Standardized data representations - Making data usable

For widely accepted and utilized and interoperable data storage and usage, as suggested by the FAIR principles for scientific data management (Wilkinson et al., 2016), it is necessary to agree on common file formats for storage on disc as well as data representations in memory. The agreement on data standards can develop in a i) community driven way by groups of users adopting a format and therefore forming the basis for a general spread of the format or ii) industry defined, by companies or organizations defining a standard. In the context of data storage and representation there are numerous standards already defined and maintained. For example the World Wide Web Consortium[1] is an international community, that is coordinating the development and maintenance of open standards to ensure the long-term growth of the internet. This includes e.g. the widely spread scalable vector graphics (svg) format and the Hypertext Markup and Extensible Markup Languages (html, xml, respectively). These standards were developed in a community driven way, and are still continuously improved by working groups[2] (e.g. the svg2 format[3]).

Standards as they are defined by the World Wide Web Consortium affect millions of users and computer systems. Therefore, the issue of standardization forms a foundation for a working system with this large number of participants. On smaller scales, however, the topic of standardization is not as pressing as individual members are typically able to develop a system, which meets their requirements and often does not require interaction with other systems. At the same time the complexity and specificity of the data and metadata to be represented increases since more detailed information needs to be captured and tracked in a consistent manner. This leads to a an unproportionally high effort required when interacting with other members of the community since the locally established implicit agreements on data storage and handling need to be communicated in addition to the actual data content.

One way even smaller communities can benefit from standardized data representa-

[1] W3C, https://www.w3.org/
[2] W3C working groups, https://www.w3.org/Consortium/activities
[3] svg2, https://www.w3.org/TR/SVG2/

tions from an early stage on is the introduction of standards from the recording system manufacturing side. This situation is more likely to occur if only very few experimental systems exist (e.g. particle physics and the `root` format (Brun and Rademakers, 1996)) or only few manufacturers produce experimental setups, which simplifies the coordination between these (e.g. medical imaging and the Digital Imaging and COmmunications in Medicine (DICOM) standard[4]). Based on this, the neuroimaging community managed to extend the standard to coherently organize metadata with data in the Brain Imaging Data Structure[5] (Gorgolewski et al., 2016).

Standards can emerge from a community or can be established by dominating entities, e.g. in case of a monopoly for hardware production. However, also the development of tools used within and across communities is tightly linked the definition of standards, as these form a prerequisite for interacting with the data. This interaction works both ways: the establishment of standards is important for the development of tools to enable them to get the most information out of the datasets, but also tools can influence the establishment of standards especially in small communities by favouring one format over another and therefore influencing the community preferences. The development of tools applicable across communities requires the prior adoption of global standards, as community internal evolved agreements are typically too diverse to be transferred to other communities to form a basis for a common set of tools.

A prime example for the delayed recognition for the need of standardization is the development of data formats in the field of electrophysiology. Here specific requirements are needed when recording electrophysiology datasets: The file formats need to be able to support writing of large amounts of data in a streaming fashion, they have to cope with the custom data structure generated by the recording system and they need to document the parameters used during the recording as minimal metadata. This, together with a number of companies developing ready-to-use solutions for experimentalists led to a multitude of file formats (see also Fig. 4.2). All systems permit experimentalists to record data without investing years into the development of a recording system but also typically provide only very few supported output file formats. This again restricts the number of easily applicable analysis methods, as the vendor may provide tools for basic visualization, processing and simple analysis the data, but these are typically not easily extendable nor do they provide a simple programming interface for the implementation of custom analyses. This unnecessarily restricts the scientific questions which can be answered with a particular dataset based on the producer of the hardware system.

A project introducing a common output file format for electrophysiology recording systems is Neuroshare [6]. Version 1.0 was released in 2003 and is a C based library relying on direct-link libraries for the integration with recording systems and providing scripts for the integration in a *Matlab* environment. However, neuroshare was only taken up by few vendors of electrophysiology setups, as these are required to provide integration

[4]DICOM, https://www.dicomstandard.org/
[5]BIDS, https://bids.neuroimaging.io/
[6]Neuroshare, http://neuroshare.org/

routines for their acquisition system. Additionally, Neuroshare was mainly developed for Windows based systems, making it complicated to use in combination with other operating systems. In 2010 a Python implementation of Neuroshare was introduced[7], which is currently not maintained.

Since then, multiple projects attempt to tackle this problem from different angles: The Neurodata Without Borders: Neurophysiology[8] (NWB:N) (Teeters et al., 2015) project attempts to define an additional, more generic file format standard aiming to replace a multitude of existing formats. This project was launched in 2014 and a second version was released in 2019 (Rübel et al., 2019). The project involved multiple scientific laboratories, funding agencies as well as industry partners and provides a fixed set of structures to describe common data encountered in this collaboration. The *NWB:N* format is not supported by major recording setup manufacturers and therefore no file format generated by common electrophysiological recording setups. Instead primary recording data need to be converted into the *NWB:N* format. Another project tackling the same problem is the Python package *Neo*. *Neo* is a spin-off of the *Neurotools* toolbox, an attempt to set up a common file format for neurophysiology setups based on Microsoft dynamic-link libraries. It was initiated in 2009 and in contrast to NWB provides a standardized in memory data representation for electrophysiology data without introducing another file format. Therefore it bridges the gap between available electrophysiology file formats and forms the basis for a number of visualization, preprocessing and analysis tools. By interfacing to a large number of file formats *Neo* is the ideal tool for implementing flexible data management workflows independent of the particular file format of the original data files. In addition, it provides also flexibility in the range of software tools that can be used for further data processing as it interfaces with various tools that cover diverse requirements of data processing, visualization and analysis. In the following we will introduce *Neo* in more detail and provide example scripts for application of *Neo* for handling electrophysiology datasets.

4.1 The *Neo* Python package

Neo[9] (Garcia, Guarino, et al., 2014) is an open-source Python package for representing electrophysiology data in working memory. It offers interfaces for reading various electrophysiological proprietary and open file formats and represents the data in a generic way (Figs. 4.1 and 4.2). Thus it forms the bases for a number of open software tools: The electrophysiology analysis toolkit[10] for analysis of spiking activity and local field potentials, OpenElectrophy[11], SpykeViewer[12] and Ephyviewer[13] for visualization,

[7] python-neuroshare, https://github.com/G-Node/python-neuroshare
[8] *NWB:N*, https://www.nwb.org/
[9] *Neo*, http://neuralensemble.org/neo, RRID:SCR_000634
[10] Elephant, http://neuralensemble.org/elephant, RRID:SCR_003833
[11] OpenElectrophy, http://neuralensemble.org/OpenElectrophy, RRID:SCR_000819
[12] SpykeViewer, https://spyke-viewer.readthedocs.io
[13] Ephyviewer, https://ephyviewer.readthedocs.io

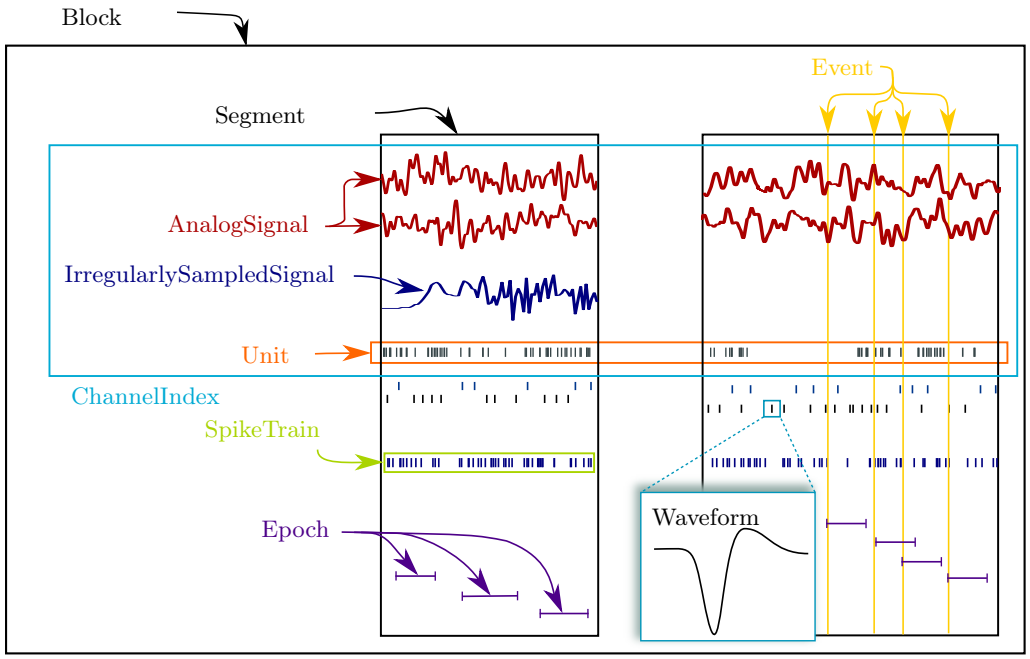


Figure 4.1: *Neo* 0.7 object structure. Figure modified from https://github.com/neuralensemble/python-neo.

Tridesclous[14] for online and offline spike sorting, NeoAnalysis[15] (B. Zhang, Dai, and T. Zhang, 2017) for rudimentary visualization and analysis, NetworkUnit [16] for validation testing of spiking networks. Related packages are *NiBabel*[17], a comparable package for neuroimaging file formats and *MNE*[18] an Python package for MEG and EEG analysis and visualization.

The two main features of *Neo* are 1) the interfacing to many different file formats, by providing reading capability for numerous proprietary formats and writing capability to selected open formats and 2) the standardized representation of electrophysiology data as a basis for further visualization and analysis steps. Using these features of *Neo* is typically used either as conversion tool from specialized to more generic formats or as run time data representation for further processing.

4.1.1 Feature updates and current development

The *Neo* version 0.3 was released in 2014 (Garcia, Guarino, et al., 2014). Since then the software has been extended to be compatible with more data formats, the object model has been revised for better usability and the implementation has been improved for performance. In the following we describe the enhancements introduced between version 0.3 (Fig. 4.3) and version 0.7 (Fig. 4.4).

[14]Tridesclous, https://tridesclous.readthedocs.io
[15]NeoAnalysis, https://github.com/neoanalysis/NeoAnalysis
[16]NetworkUnit, https://github.com/INM-6/NetworkUnit, RRID:SCR_016543
[17]NiBabel, https://nipy.org/nibabel, RRID:SCR_002498
[18]MNE, https://martinos.org/mne, RRID:SCR_005972

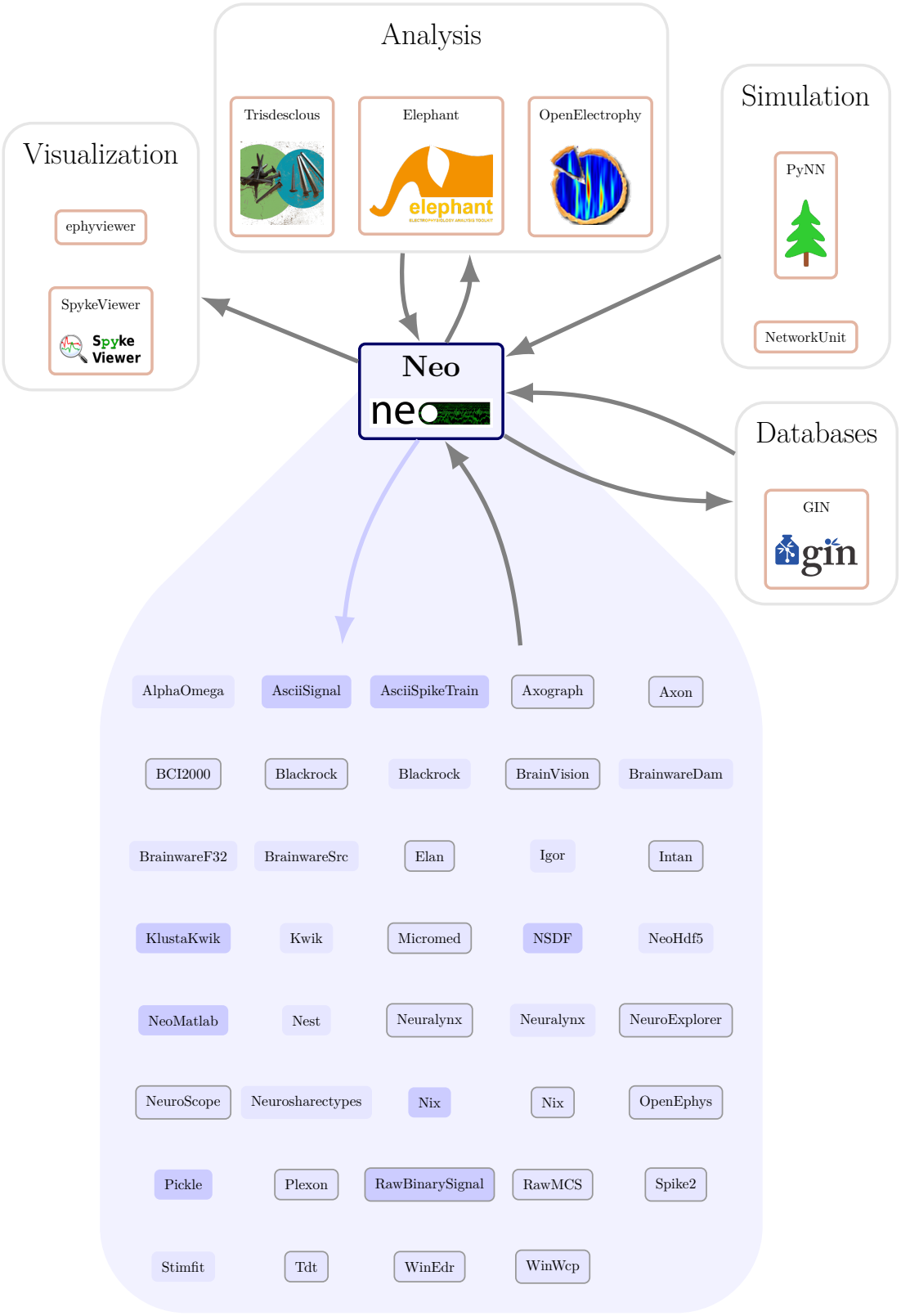


Figure 4.2: *Neo* embedding. *Neo* 0.7. supports a number of file formats for reading (light blue) and writing (dark blue). Many of the supported formats can be read in a improved fashion, permitting for more efficient memory usage (black frames). Neo provides an interface for many advanced tools for visualization, simulation, analysis and data storage.

Neo 0.3 architecture

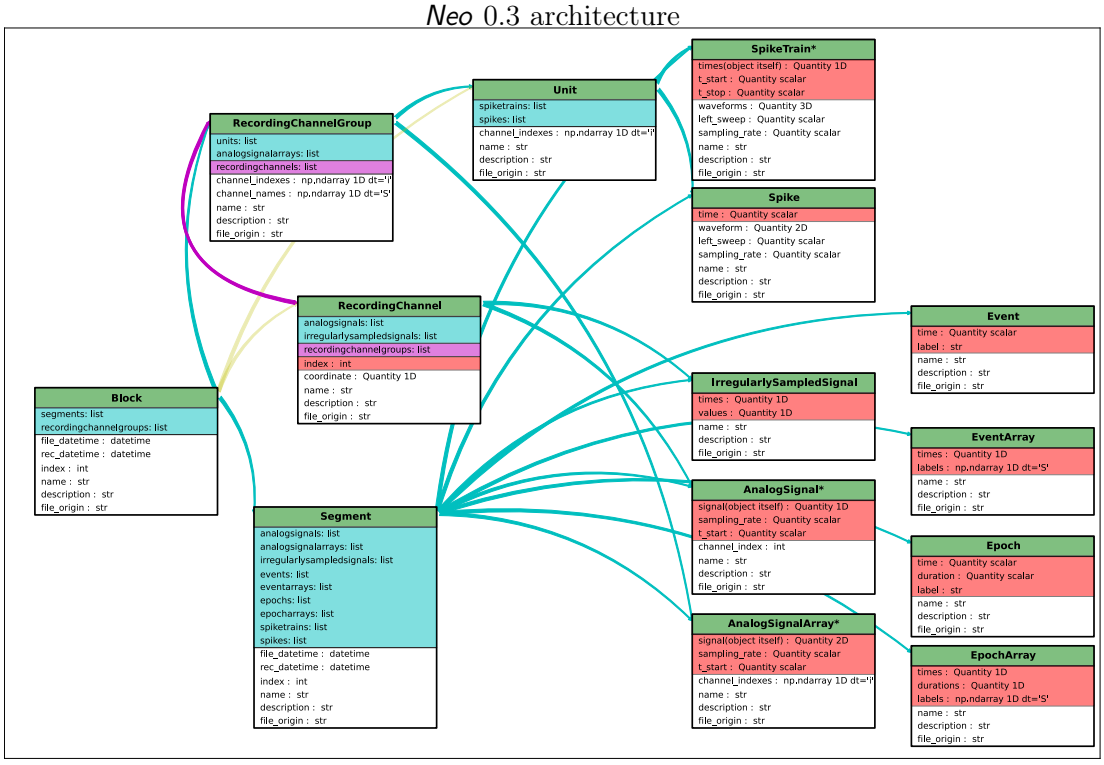


Figure 4.3: *Neo* 0.3 architecture. *Neo* represents electrophysiology signals in data objects such as `AnalogSignal`s, `AnalogSignalArray`s, `IrregularlySampledSignal`s, `SpikeTrain`s and `Spike`s, whereas the latter two optionally include information about waveform data for each spike. Additional supplementary information describing the timing during the recording can be provided using `Event`s and `EventArray`s or `Epoch`s and `EpochArray`s to mark time points or durations during the recording, respectively. All above described data objects are put into relation by container objects, such as `Segment`s (grouping all data objects simultaneous in time), `Unit`s (grouping `SpikeTrain`s and `Spike`s across time), and `RecordingChannel`s and `RecordingChannelGroup`s (grouping `AnalogSignal`s `IrregularlySampledSignal`s and `Unit`s, `AnalogSignalArray`s and `RecordingChannel`s, respectively). The top level container is a `Block` linking to `Segment`s and `RecordingChannelGroup`s. Figure from https://neo.readthedocs.io/en/0.3.3.

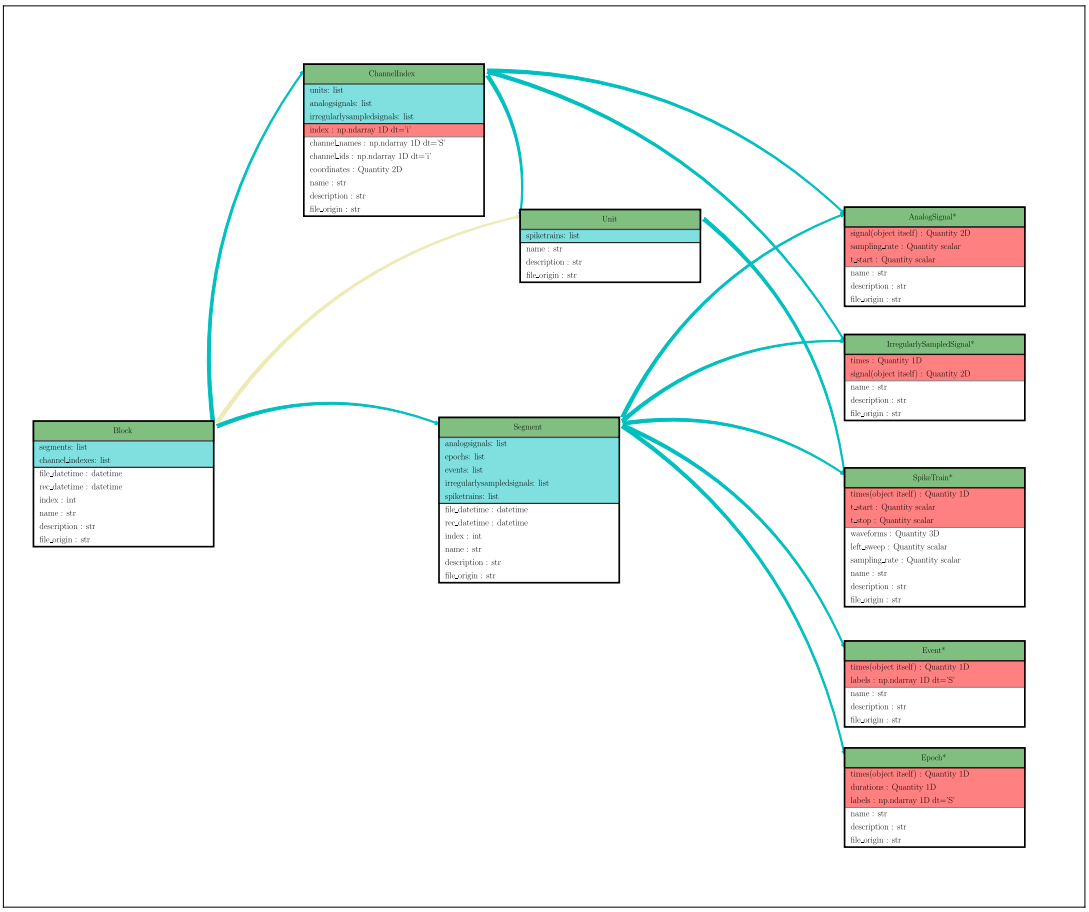


Figure 4.4: *Neo* 0.7 architecture. *Neo* represents electrophysiology signals in data objects such as `AnalogSignal`s, `IrregularlySampledSignal`s and `SpikeTrain`s optionally including information about waveform data for each spike. Additional supplementary information describing the timing during the recording can be provided using `Event`s or `Epoch`s to mark time points or durations during the recording, respectively. All above described data objects are put into relation by container objects, such as `Segment`s (grouping all data objects simultaneous in time), `Unit`s (grouping `SpikeTrain`s across time) and `ChannelIndex`es (grouping `AnalogSignal`s `IrregularlySampledSignal`s and `Unit`s). The top level container is a `Block` linking to `Segment`s and `ChannelIndex`es. Figure modified from https://github.com/neuralensemble/python-neo.

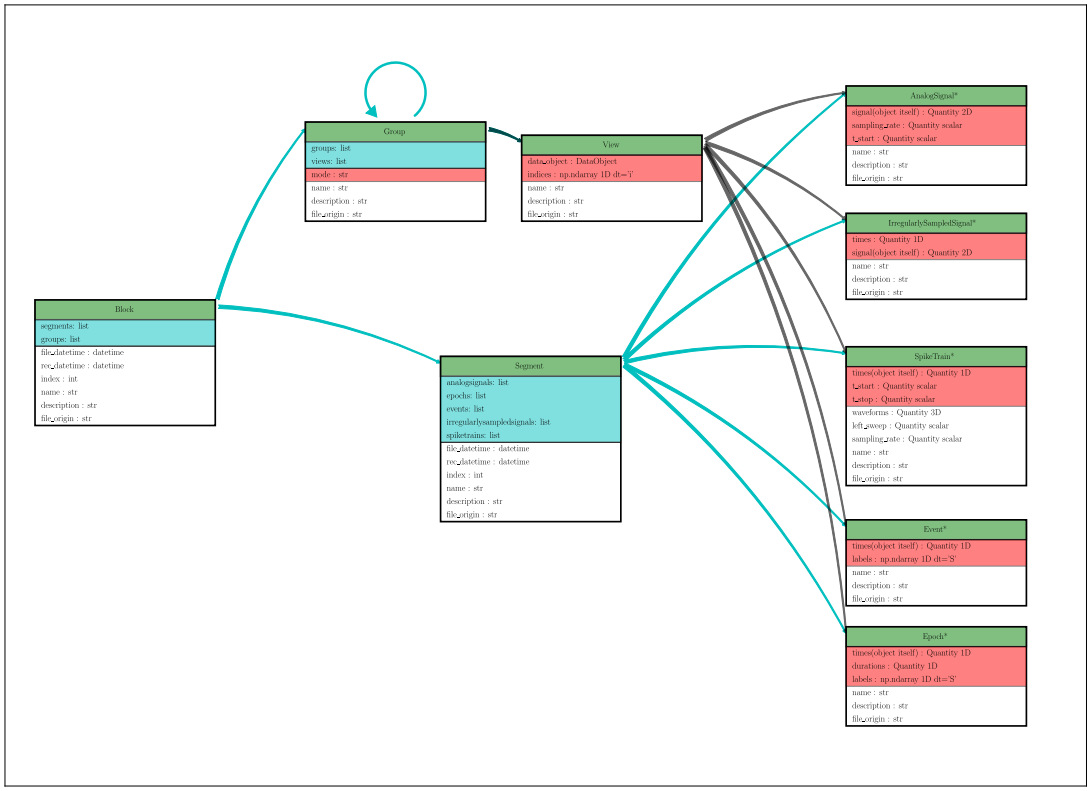


Figure 4.5: Proposed *Neo* architecture. The proposed *Neo* architecture preserves all objects from *Neo* version 0.7 except for `ChannelIndex`es and `Unit`s. These are replaced by `Group` and `View` objects, which a more generic, but still customizable way of organizing data objects. `View` objects can mask data objects by linking to a subset of the contained data (e.g. a single trace of an `AnalogSignal`. This linking is unidirectional, preventing complex dependencies involving data objects. `Group`s are capable of linking to any kind of data objects or `View` of data objects. The required specificity is provided by the different `mode`s of a `Group` object. These limit to the connected objects to a specific type and number, wherefore a `Group` can e.g. be used instead of a `Unit` object. `Group`s can also link to other `Group` objects to provide higher level organization of the data.

Interfaces to file formats *Neo* 0.7 is supporting additional file formats for reading, such as Axograph, OpenEphys, Stimfit, Kwik, Nix, Igor, Nest, Neuralynx, NSDF and BCI2000. The capabilities for reading the Axon, Blackrock, Brainvision, Brainware, Elphy, Intan Matlab structures, Neuroshare, Plexon, Spike2, Tdt, NeuroExplorer, Neuralynx, Igor, Elan, Micromed, RawMCS, WinWCP formats have been improved. Reading and writing capabilities have been improved for Nix and Pickle formats. PyNNText and PyNNNumpy formats are no longer supported. A new code design for readers has been implemented and the majority of readers has adjusted accordingly to enable improved loading performance and loading of subsets of data (RawIO implementation).

Object structure and usability The code has been modularized for more flexibility and maintainability, and a large number of unittests have been added. The object structure has been restructured for user friendliness and to boost performance by implementing sets of similar data entities in single objects instead of using individual data objects for each data entity (removal of dedicated array versions of data classes). A new relational container object `Channel_Index` was introduced to simplify the representation of logical relations between data objects replacing `RecordingChannel` and `RecordingChannelGroup` objects. Consistent deep copy functionality has been added for all data objects and additional internal consistency checks have been added. A new type of custom annotation mechanism has been added, which is designed to capture custom annotations in the same dimension as the data (array annotations). For the installation additional option were introduced, depending on the required file formats which need to be supported. The code style has been adjusted to follow the PEP8 guideline[19](*PEP 8 – Style Guide for Python Code* 2019). Support for Python 2.6 was dropped and consistent support for Python 3 was introduced.

Outlook Practical application of *Neo* confirmed an improved usability for version 0.7. The described data objects facilitate data access and performance and the combination of `Block` and `Segment` objects as container objects provides easy to use access to the data. However, the concept of `ChannelIndex` objects is covering too many aspects of relations between data at once: 1) grouping data objects, 2) masking data objects (selection of a subset of data within a data object) and 3) annotating individual samples within data objects. The last aspect has been moved to the individual data objects, by introducing array annotations.

For future versions, the splitting of `ChannelIndex` objects into two separate objects (`Group`, `View`) responsible for grouping and masking is planned. A `Group` object will be able to link different types of data objects, depending on its configuration. For example a `Group` object resembling a physical electrode will be able to link to a single `AnalogSignal` and multiple `Unit` objects. A `View` object can be used to refer to a subset of the data stored in a data object (e.g. a single recording trace within an `AnalogSignal`). This view can be used instead of a data object in any relation and will provide utility functionality to provide a sliced version of the actual data object.

[19]Python Enhancement Proposal 8, https://www.python.org/dev/peps/pep-0008

Another topic of discussion is the linking between *Neo* objects. Up to the current version 0.7 all links between *Neo* objects can be established bidirectional. However, the bidirectionality of the linking is not inherently guaranteed, since the generation of a link does not automatically generate a backward link between *Neo* objects. Introducing automatic bidirectional linking would guarantee bidirectional linking, but might complicate the set up of a *Neo* structure e.g. when reading a data file. There are different approaches possible to circumvent these problems: i) The use of only unidirectional links from higher level to lower level objects (top to bottom). This approach would still provide most of the functionality commonly used. ii) The implementation of a validation framework, which can on request check if a provided *Neo* object structure is fully linked including consistent bidirectional links. A suggested model implementing the first of the two suggestions is presented in Fig. 4.5. Here links from `Views` and `Groups` towards data objects are unidirectional, therefore preventing cyclic links across the complete *Neo* structure.

Spiking activity of individual neurons can not only be recorded using sharp electrodes, but also from multielectrode array recordings (MEA) and calcium image recordings (Kelly et al., 2007; Shew, Bellay, and Plenz, 2010). To support the usage of *Neo* also for imaging data an extension of *Neo* by two additional object types is planned: the `ImageSequence` object will capture sequences of regularly sampled images and is therefore closely related to the `AnalogSignal` as it contains the same type of data, but also captures the spatial relation between different pixels (traces). A second object relevant for image handling is a `RegionOfInterest` object, which is used to spatially mask a specific part of a stack of images. The `RegionOfInterest` can implemented as special case of a `View`. Support for `ImageSequences` also requires capabilities to read imaging data formats, which will be added subsequently.

4.1.2 *Neo* object structure

Neo objects can be separated into two types: data objects, describing basic recording data in combination with minimal metadata and container objects, providing the structural framework for the relation between the data objects (Fig. 4.1). In general, all *Neo* objects have three optional arguments to provide custom information about the captured data: 1) The `name` attribute can be used to label the object and can be used for simple data filtering and selection. 2) The `description` attribute is intended to provide a human readable, detailed, 1-2 sentence description for the data contained / grouped by the *Neo* object. The `file_origin` is can be used to describe the origin of the data, e.g. the original recording filename or simulation script. In addition to `name`, `description` and `file_origin`, all *Neo* objects can capture additional custom information in form of an `annotation` dictionary. This dictionary can contain arbitrary data in all basic Python data types as well as `datetime`, `date`, `time`, `timedelta` in arbitrary structures build from lists, dictionaries, tuples or *Numpy* arrays without any restrictions on the shape of these objects.

Data objects Data objects are based on *Numpy* arrays (Walt, Colbert, and Varoquaux, 2011) for efficient computation on large datasets. In addition, *Neo* objects are aware of physical quantities by using the *Quantities* package (Dale, 2019) (Fig. 4.1).

Neo provides data objects to capture regularly as well as irregularly sampled continuous signals in `AnalogSignal` and `IrregularlySampledSignal` objects, respectively. Both objects rely on a 2 dimensional *Quantities* (*Numpy*) array capturing the basic data signal, whereas the first dimension describes the time and the second dimension different signal traces. For `IrregularlySampledSignal`s time information is captured in a second separate *Quantities* array, sharing the first dimension with the signal array. For `AnalogSignal`s this in implemented in a more compact fashion by storing only the sampling rate (`sampling_rate`) and the starting time point of the recording (`t_start`). A *Quantities* array containing the time values corresponding to the data point can be generated on request via the `times` attribute of the `AnalogSignal`. For time series data *Neo* provides a `SpikeTrain` object, capturing the data in a `times` attribute. Additionally the start and stop times of the data acquisition are essential for the interpretation of the data, these are provided as mandatory `t_start` and `t_stop` scalar *Quantities* parameters. Optionally, a `SpikeTrain` object can also capture snippets of regularly sampled continuous signal around each time point in the waveform attribute. This links to a 3 dimensional array, capturing the time, spike ID and recording channel dimensions of the waveforms. The `SpikeTrain` attribute `sampling_rate` is used to capture the corresponding sampling information as in the `AnalogSignal` case. To relate the waveform snippets with the time series data the *Quantities* scalar `left_sweep` defines the constant offset between the time series data and the corresponding waveform snippet. A different type of time series data is non-neuronal time series which describe specific time points or durations during the recording of neuronal activity. These might be control signals of the experiment, e.g. trial start times, behavioral events of the subject, e.g. a the time in which a button was pressed. For the description of time series data *Neo* provides `Event` objects, capturing the data in a one dimensional *Quantities* array (`times` attribute) together with a string array of the same shape (`labels` attribute) providing labels to the individual time points. To represent extended periods of time, *Neo* offers `Epoch` objects having the same attributes as `Event`s, and in addition a one dimensional *Quantities* array of durations with the same shape as the main time series data.

All of the above mentioned data objects consist of a main *Quantities*-wrapped *Numpy* array with one or more dimensions. One of the latest features introduced in *Neo* is to implement a second kind of annotation mechanisms on this. The annotations of an object always refer to the object as a whole. However, in many use cases annotations have been used to provide details about the individual data samples by containing arrays, which share some dimensions with the main data samples. For example `AnalogSignals` which contain more than one recording trace are frequently annotated with list providing detailed information about the individual recording channels, e.g. the identity of the channel or the particular filter settings. However, when modifying the shape of the original data, these annotations can not be updated in an automatic fashion since they

were user defined and didn't follow a fixed schema. Since *Neo* version 0.7.0 all data objects have an additional feature `array_annotation`s, which fills this gap by providing an annotation mechanism for capturing sample based annotations, i.e. annotation entries with the same length as the main data dimension. These `array_annotations` are automatically adjusted when the shape of the main data is modified, e.g. by slicing the data in the time axis or extracting a single signal trace from an `AnalogSignal`.

The `Neo` 0.7 release also features a standardized way of optionally loading specific parts of the data on request. This is of advantage when dealing with datasets which are large in comparison to the available memory. The new `lazy` feature permits to generate the complete *Neo* structure (Fig. 4.4), but substituting all data objects with proxy objects, which feature the same attributes and links as classic data objects, but do not contain the actual data. For accessing the data a `load` mechanism is provided which loads the requested parts of the data and provides them in a separate classic data object not linked to the main *Neo* structure. Using the `lazy` mechanism large datasets can be processed chunk wise without requiring large amounts of memory.

Container Objects *Neo* container objects provide the structural relations between *Neo* data objects. The base object for a dataset is a `Block` object, containing everything related to the dataset. The `Block` object can link to a number of `Segments` and `ChannelIndex` objects which can be used to organize data objects according to their timing, spatial relation or custom grouping aspects, e.g. grouping the data objects by the signal quality of the contained data. `Segment` objects are intended for grouping objects which share the same time frame, e.g. simultaneously recorded `AnalogSignal`s and `SpikeTrain`s (Fig. 4.4). `SpikeTrain`s from different `Segment`s that are considered coming from the same source (neuron) can be linked across `Segments` via `Unit` objects. A `Unit` object again can be grouped together with `AnalogSignal`s and `IrregularlySampledSignal`s in a `ChannelIndex` object. In addition to the grouping functionality, a `ChannelIndex` object also provides an additional labeling functionality for child `AnalogSignal`s via the `channel_ids` and `channel_names` attributes. These consist of one dimensional integer and string arrays labeling the individual traces of the attached continuous signals. Via the mandatory attribute `index` a selection of the traces within the linked continuous signals can be done. However, this requires a consistent ordering of recording traces within all linked continuous signals of a `ChannelIndex`. All container objects provide utility functions to facilitate access and selection of data objects. This is implemented in the form of a `filter` method, which returns a list of *Neo* objects based on a combination of object type, attribute and annotation constraints.

4.2 *Neo* usage examples

In the following we demonstrate in three practical examples how *Neo* can be used to load data from different file formats in a memory efficient manner, access and select data, annotate and filter data according to custom metadata added and save data in an open source format.

4.2.1 Loading & visualization

Loading data in *Neo* is implemented in two stages: First initialization of the reader (IO) and second reading of the *Neo* structure. For readers implemented in the standardized manner (Fig. 4.2, black frame) you need to provide the filename of the dataset to load. This will generate an `io` object providing functionality to load the data into a *Neo* structure. Depending on the file format either a *Neo* `Block` or a *Neo* `Segment` can be loaded using the `read_block` or `read_segment` method of the `io` object.

In this example, the published dataset described in Chapter 2 is used for demonstrating the loading of electrophysiology data into the *Neo* structure (Code Listing 4.1). Data were previously downloaded from GIN[20] and continuous as well as sorted spiking data are loaded using the *Neo* `BlackrockIO` class. This IO provides standardized access to the data, permitting lazy loading of *Neo* objects (cf. Code Listing 4.1 line 10f). Here the *Neo* filter functionality is used to select all `SpikeTrain`s which have an annotation key `'channel_id'` and the corresponding value of the user requested channel index (`selected_channel`, line 13). In the next step, the corresponding `AnalogSignal` trace is extracted by finding the `AnalogSignal` with a corresponding entry in the `array_annotation`s with key `channel_ids` and extracting the `id` of the corresponding trace (line 15-18). Finally, the analog and spiking data of 10 seconds of recording are loaded into memory via the `load` mechanism of the respective data objects and returned by the `load_single_channel_data` function (line 19-22).

The visualization of a single `AnalogSignal` trace together with spiking activity from multiple `SpikeTrain`s is implemented in the `plot_data` function. This requires the corresponding data objects as input as well as a location to store the final scalable vector graphics plot (line 25). Here, *Matplotlib* (Hunter, 2007) is exploited to visualize the electrophysiological data (line 30 & 34-38). Correct scaling of the signals and automatic generation of axis labels is ensured by the inherent use of the *Quantities* package within *Neo* (line 28-31, 34-37). The resulting plot is exported to the scalable vector graphics (svg) format for storage in a flexible, memory efficient and scalable manner. Finally, user specific setting are extracted from command line parameters and both functions are executed sequentially to generate a visualization of a single recording trace and corresponding spiking activity.

4.2.2 Annotation of data with metadata from *odML*

Annotations are a key feature of the generic *Neo* structure to provide the necessary customizations to be used for a specific dataset. Standard annotations are a feature of all *Neo* objects and can be accessed via the `annotation` attribute. In addition, *Neo* data objects also provide a more specific type of annotation mechanism, namely `array_annotations` which act in the same manner as regular `annotations` but are directly coupled to the dimension of the underlying data. This permits to store metadata which are linked to individual data entries and handle them in an automatic way when

[20] https://gin.g-node.org/INT/multielectrode_grasp

```
1  import numpy as np
2  import sys, neo
3  import quantities as pq
4  import matplotlib.pyplot as plt
5
6
7  def load_single_channel_data(data_location, selected_channel):
8      """ Loading AnalogSignal and SpikeTrains from a single electrode. """
9      # initialize the io and loade the Neo data structure in lazy mode
10     io = neo.BlackrockIO(data_location)
11     block = io.read_block(lazy=True)
12     # filter to select spiketrains from specific channel
13     spiketrains = block.filter(targdict={'channel_id': selected_channel})
14     # extract corresponding AnalogSignal and trace id
15     for analogsignal in block.segments[0].analogsignals:
16         if selected_channel in analogsignal.array_annotations['channel_ids']:
17             id = np.where(analogsignal.array_annotations['channel_ids'] == selected_channel)[0]
18             break
19     # load analog and spiking data for 10 seconds of recording time
20     analog_data = analogsignal.load(channel_indexes=id, time_slice=(10 * pq.s, 20 * pq.s))
21     spike_data = [st.load(time_slice=(10 * pq.s, 20 * pq.s)) for st in spiketrains]
22     return analog_data, spike_data
23
24
25 def plot_data(analog_data, spike_datas, plot_location):
26     """ Visualize a single AnalogSignal trace with multiple SpikeTrains """
27     # parameters for axis scaling
28     time_scale, voltage_scale = pq.s, pq.microvolt
29     # plot single analogsignal and all spiketrain data
30     plt.plot(analog_data.times.rescale(time_scale), analog_data.rescale(voltage_scale).magnitude,
       ↪  lw=1, label='AnalogSignal')
31     ymax = max(analog_data.rescale(voltage_scale)).magnitude
32     for spike_data in spike_datas:
33         unit_id = spike_data.annotations['unit_id']
34         plt.plot(spike_data.rescale(time_scale), np.ones_like(spike_data) * unit_id * ymax / 6,
           ↪  '|', ms=20, mew=1.5, label='Spikes Unit {}'.format(unit_id))
35     # configure plot labels and add legend
36     plt.xlabel('Time [{}]'.format(time_scale.dimensionality.latex))
37     plt.ylabel('Voltage[{}]'.format(voltage_scale.dimensionality.latex))
38     plt.legend(title='Channel {}'.format(analog_data.array_annotations['channel_ids'][0]),
       ↪  markerscale=0.4, title_fontsize=7, loc=1, prop={'size': 6})
39     # export plot to svg format
40     plt.savefig('{}.svg'.format(plot_location))
41
42
43 # Calling main functions to load data and plot data specified via command line arguments
44 data_location, selected_channel = sys.argv[1:]
45 channel_data = load_single_channel_data(data_location, int(selected_channel))
46 plot_data(*channel_data, data_location.split('/')[-1])
```

Code Listing 4.1: Loading data into the `Neo` framework and visualization using the Matplotlib package. Required packages are imported (line 1-4) and two main functions are defined for loading and visualization of the *Neo* structure: `load_single_channel_data` and `plot_data`. The first one requires the location of the datasets and the selected channel to plot as command line arguments (line 9). Here, the dataset is opened using the `BlackrockIO` in `lazy` mode and the user-specified channel is selected and loaded into memory together with the spiking activity (lines 7-22). The second function visualized a given `AnalogSignal` together with a list of `SpikeTrains` using the *Matplotlib* package (line 25-40). Finally both functions are run subsequently using command line parameters for dataset and channel specifications. For the resulting plot see Fig. 4.6

.

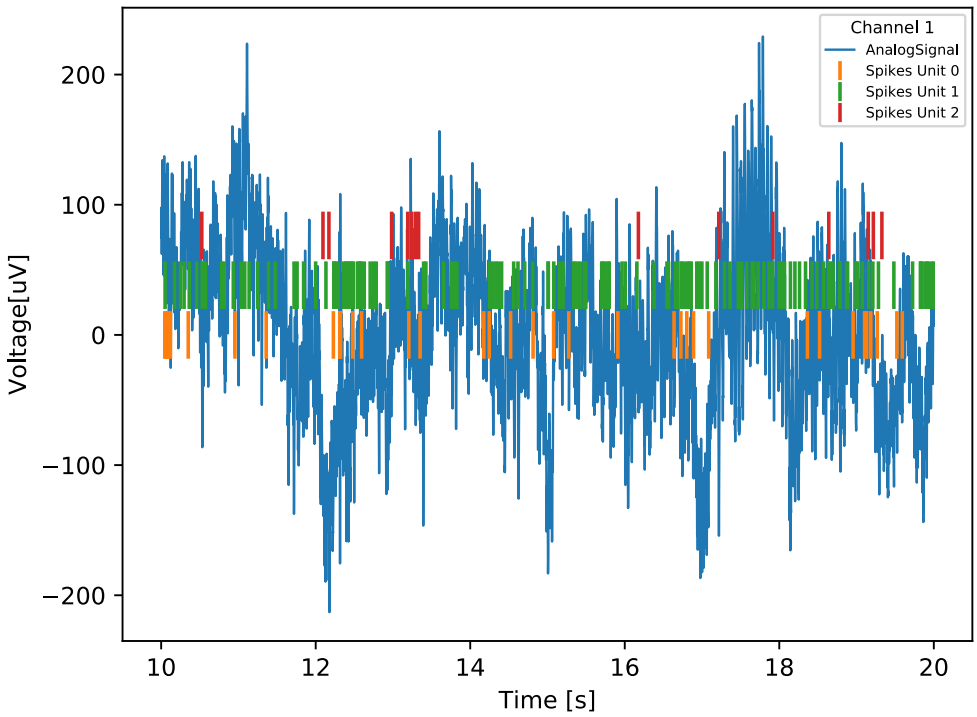


Figure 4.6: Visualizating of the activity recorded on a single electrode. A single trace of an AnalogSignals contains the voltage samples of a recording electrode (blue). The corresponding threshold crossing events are split into three different `SpikeTrain` objects and time stamps are marked as vertical lines with a `Unit` specific offset (orange, green, red).

manipulating the data object. This example demonstrates how to access and modify regular and array annotations using a metadata collection in the odML format.

Code Listing 4.2 demonstrates two aspects of *Neo* annotations: firstly annotation access via the object attributes `annotation` and `array_annotation` and secondly the generation of (array) annotations. The resulting script output is shown in Code Listing 4.3. The dataset is loaded in the same manner as in Code Listing 4.1 and `annotations` and `array_annotations` of selected objects are printed using the `print_annotations_example` function. The resulting printout shows the different levels of annotations, which are automatically generated by *Neo* when loading a dataset from the `Blackrock` format. Annotations on the `Block` level provide general information about the recording session, whereas `SpikeTrains` carry very specific metadata about the identity of the object and its spike sorting classification.

An example for `array_annotations` is provided for an `AnalogSignal`. `AnalogSignal` objects provide the most metadata as they describe all recording traces they contain individually (see Code Listing 4.3 line 15-33). Here the `AnalogSignal` contains 96 recording traces and all array annotations have a matching length of 96. The annotated information covers the identity of the electrodes ('channel_ids' and 'channel_names', line 16f) and identity within the recording system ('connector_ID' and

'connector_pinID', line 19f) as well as signal processing parameters for signal filtering and spike extraction (line 21-30). Here the keys consist of a combination of the file type affected (nsx/nev), the filter border (high/low), the general parameters (freq/energy_threshold/dig(itization)_factor/waveform_size) and filter parameters affected (corner/order/type). In addition, a human readable description as well as information about the originating file is provided ('description', 'nsx' and 'file_origin', line 18, 31f). All `array_annotations` will automatically be adjusted when the underlying data object is modified via *Neo* functions, e.g. via slicing in time.

The second part of the example script (Code Listing 4.2, line 28ff) demonstrates the generation of additional (array) annotations by loading a metadata collection in the odML format, extracting relevant information for the interpretation of an `AnalogSignal` and adding this information to the `AnalogSignal` as array annotation. First, an odML file is loaded using the Python *odML* library (line 31) and all `Sections` describing electrodes are extracted to load a mapping between the Blackrock channel `ID`s provided by the recording system and the spatially ordered `ConnectorAlignedID`s (line 32f), which are defined in order to indicate an electrode's spatial position easily. In the next step, the existing Blackrock `ID` array annotations of an `AnalogSignal` are used to generate the corresponding array of `ConnectorAlignedID`s using the previously extracted mapping b(line 36f). Finally, the new ids are added as a new array annotation to the `AnalogSignal` using the `array_annotate` mechanism (line 38).

The generated array annotations are displayed by printing the complete array annotations of the `AnalogSignal` again, resulting in Code Listing 4.3 line 43ff. Here, the new key 'connector_aligned_ids appears containing the corresponding spatially organized ids of the recording electrodes.

With array annotations and annotations, *Neo* now offers a mechanisms to provide custom information for *Neo* objects as a whole, but also for subsets of data contained by *Neo* objects. With these annotation mechanisms it is now possible to directly add *odML* content to *Neo* objects on multiple levels of the data organization. However, this annotations are not automatically generated yet, since there is no mechanism assigning data in the *Neo* structure and metadata in the *odML* structure. Due to the generality of *Neo* and *odML* structure the is no generic assignment strategy between the two structures possible, but these relations need to be captured explicitly using an extended framework.

4.2.3 Saving data & format conversion

Being able to store intermediate preprocessing steps or analysis results in a persistent manner is important for making workflows reproducible. *Neo* provides the option to store data in plain ASCII, KlustaKwik (Hazan, Zugaro, and Buzsáki, 2006), Nix (Stoewer et al., 2014), binary Matlab, Neuroscience Simulation Data Format (NSDF) (Ray et al., 2016), binary Python pickle and a custom binary format, see also Fig. 4.2. In this example we focus on the Nix format as it provides the most versatile data storage. Nix is based on an hdf5 (The HDF Group, 1997) backend which provides the

```
1  import sys, neo, odml
2
3  def pretty_print_dict(dictionary):
4      """Print individual entries of a dictionary in truncated, individual lines"""
5      for k, v in dictionary.items():
6          res = '  {}: {}'.format(k, str(v)).replace('\n', ',')
7          print(res[:75] + '...' if len(res) > 74 else res)
8
9  def print_annotations(obj, mode='annotations'):
10     """Print annotations / array_annotations of a Neo object"""
11     print(type(obj).__name__)
12     if mode == 'annotations':
13         pretty_print_dict(obj.annotations)
14     elif mode == 'array_annotations':
15         pretty_print_dict(obj.array_annotations)
16     else:
17         raise ValueError('Unknown annotation type {}'.format(mode))
18
19 def print_annotation_examples(block):
20     """ Print some example annotations & array annotations """
21     print('Annotations\n---------------')
22     print_annotations(block)
23     print_annotations(block.segments[0].spiketrains[0])
24     print('Array Annotations\n--------------------')
25     print_annotations(block.segments[0].analogsignals[-1], mode='array_annotations')
26
27 def generate_annotations_from_odml(block, odml_filename):
28     """ Extract mapping information from the odml sheet and  add it as array annotation to the
       ↪  data """
29     # loading odml file and extract electrode id mapping
30     doc = odml.load(odml_filename)
31     electrode_secs = doc.itersections(filter_func=lambda x: x.name.startswith('Electrode_'))
32     mapping = {sec.properties['ID'].values[0]: sec.properties['ConnectorAlignedID'].values[0] for
       ↪  sec in electrode_secs}
33
34     # extract id present in neo block and create new annotation based on mapping
35     original_ids = block.segments[0].analogsignals[-1].array_annotations['channel_ids']
36     connector_ids = [mapping[oid] for oid in original_ids]
37     block.segments[0].analogsignals[-1].array_annotate(connector_aligned_ids=connector_ids)
38
39 # extracting command line parameters, loading data and print default annotations
40 data_location, odml_filename = sys.argv[1:]
41 io = neo.BlackrockIO(data_location)
42 block = io.read_block()
43 print_annotation_examples(block)
44 # extract metadata from odml, add new annotations and print annotations
45 generate_annotations_from_odml(block, odml_filename)
46 print('\n\nNew array annotation "connector_aligned_ids" AFTER annotation generation')
47 print('\tconnector_aligned_ids:
   ↪  {}'.format(block.segments[0].analogsignals[-1].array_annotations['connector_aligned_ids']))
```

Code Listing 4.2: Annotation access and editing with *odML* and *Neo*. Required packages are imported (line 1) and three functions for printing `annotations` and `array_annotations` are defined (line 3-25). `pretty_print_dict` provides functionality for displaying individual dictionary items in separate lines (line 3-6). `print_annotations` uses the previous function to print based on the `mode` keyword the `annotations` or `array_annotations` of a given *Neo* object. `print_annotation_example` prints a selection of `annotations` and `array_annotations` from a *Neo* block structure. The function `generate_annotations_from_odml` extracts metadata information about the mapping of different types of `ids` from an `odML` file and annotates the corresponding `AnalogSignal` (line 27-37). Finally, all functions are demonstrated based on a command line specified data- and metadataset (`data_location` and `odml_filename`, line 40-45). The generated annotations are confirmed via a final call of `print_annotations` (line 46f).

```
Annotations
---------------
Block
  avail_file_set: ['ns2', 'nev']
  avail_nsx: [2]
  avail_nev: True
  rec_pauses: False
SpikeTrain
  id: Unit 1000
  channel_id: 1
  unit_id: 0
  unit_tag: unclassified
Array Annotations
--------------------
AnalogSignal
  channel_names: ['chan1' 'chan2' 'chan3' 'chan4' 'chan5' 'chan6' 'chan7' '...
  channel_ids: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20...
  file_origin: ['datasets/i140703-001.ns2' 'datasets/i140703-001.ns2', 'dat...
  connector_ID: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
  connector_pinID: [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 1...
  nev_dig_factor: [250 250 250 250 250 250 250 250 250 250 250 250 250 250 ...
  nb_sorted_units: [2 1 1 2 1 1 1 2 2 2 2 1 2 2 1 2 1 3 1 1 1 3 2 2 2 3 2 1...
  nev_hi_freq_order: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2...
  nev_hi_freq_type: ['Butterworth' 'Butterworth' 'Butterworth' 'Butterworth...
  nev_lo_freq_order: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2...
  nev_lo_freq_type: ['Butterworth' 'Butterworth' 'Butterworth' 'Butterworth...
  nsx_hi_freq_order: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1...
  nsx_lo_freq_order: [4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4...
  nsx_hi_freq_type: ['Butterworth' 'Butterworth' 'Butterworth' 'Butterworth...
  nsx_lo_freq_type: ['Butterworth' 'Butterworth' 'Butterworth' 'Butterworth...
  description: ['AnalogSignal 0 from channel_id: 1, label: chan1, nsx: 2', ...
  nsx: [2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2...


New array annotation "connector_aligned_ids" AFTER annotation generation
  connector_aligned_ids: [93 92 94 95 75 96 85 97 86 98 87 88 77 99 66 89 76 90 67 79 58 80 78 70
 68 60 69 50 59 40 49 30 83 84 73 74 63 64 53 54 43 55 44 45 33 46 34 65
 24 56 35 47 25 57 26 36 27 37 28 38 29 48 19 39 81 82 71 72 61 62 51 52
 41 42 31 32 21 22 11 12  2 23  3 13  4 14 15  5 16  6 17  7  8 18 20  9]
```

Code Listing 4.3: Output of Code Listing 4.2. Listed are the automatically generated annotations of example *Neo* objects as extracted from `Blackrock` recording files. On the `Block` level these are dealing with general information about available files and interruptions in the recording process. For `SpikeTrains` there is information provided about the identity and classification of the `Unit` the `SpikeTrain` was assigned to. For `AnalogSignals` there are arrays describing the attributes of the individual electrode traces. This includes information about the electrode identities (`channel_ids`), mappings of contacts within the recording system (`connector_IDs`, `connector_pinIDs`) and the applied filter and threshold settings for signal preprocessing and spike extraction.

flexibility of fast and memory efficient access for large datasets. In addition, Nix is designed to capture data as well as metadata, providing the opportunity to unify both in a single file and add links between relevant metadata and the corresponding data. This example demonstrates the conversion of the dataset used already in previous examples from the `Blackrock` format to the *Nix* format. Furthermore, it showcases the addition of metadata from the `odML` format to the same file and the extraction of the data from the *Nix* file. Code Listing 4.4 provides four functions for the back and forth conversion of files in the *Neo* and *odML* format to the *Nix* format (line 4-24). These functions are mostly based on the open-source *nix-odML-converter* library [21]. This library can be used for simple command line conversion between *Nix* and *odML*. Here we demonstrate the usage of the Python interface, permitting a more flexible approach to conversion parameters (e.g. filenames, write modes). We demonstrate the merging of the data together with the metadata into a single *Nix* file using the first two functions (Code Listing 4.4 `save_neo_to_nix` and `save_odml_to_nix`). The last two functions deal with the extraction of the corresponding information of the individual components from the `Nix` file (Code Listing 4.4 `load_odml_from_nix` and `load_neo_block_from_nix`).

With the capability of the *Nix* format to capture data as well as metadata in a combined fashion both modalities can be comprehensively stored in a common framework. Integrating both modalities in the *Nix* format is easy since *Neo* supports the writing of data in the *Nix* format and metadata can be supplemented using the *nix-odML-converter*.

4.3 Comparison of *Neo* and *NWB:N*

The *NWB:N* format (Teeters et al., 2015) is an alternative approach to *Neo* for handling of electrophysiology data. In contrast to *Neo* it specifies a new file format instead of providing interfaces to existing formats. More precisely, *NWB:N* provides a modular framework for the capture of specific electrophysiology modalities. It is accompanied by a number of canonical schema (type) for capturing specific types of data, e.g. behavioural and time series data, imaging recordings, extra and intracellular electrophysiology recordings as well as optogenetic stimulation and optical physiology. Custom schema can be specified to capture data modalities not covered by the set of canonical schema. This makes the *NWB:N* format on the one side more generic than the *Neo* structure, since it permits the implementation of custom schema specifications, but on the other side also more restrictive, because existing schema specifications are very specific and can not be reused in a slightly different context. Also implementation of a custom schema specification extensions in *NWB:N* requires additional background knowledge of the experimenter about the underlying organization of the *NWB* package and the time and effort of implementing such a format. For *Neo* the approach is different: the underlying object structure is generic to automatically cover most of the electrophysiology data and the experiment specificity is introduced via the annotation and array annotation mechanism. This implies that the user only has to understand

[21] https://github.com/G-Node/nix-odML-converter

```
1  import sys, neo, odml, nixio
2  from nixodmlconverter.convert import nixwrite, get_odml_doc, nix_to_odml_recurse
3
4  def save_neo_to_nix(block, nix_filename, **kwargs):
5      """ save Neo structure in Nix format """
6      io = neo.NixIO(nix_filename, **kwargs)
7      io.write_block(block)
8
9  def save_odml_to_nix(odml_filename, nix_filename, **kwargs):
10     """ save odml tree in Nix format """
11     odml_doc = odml.load(odml_filename)
12     nixwrite(odml_doc, nix_filename, **kwargs)
13
14 def load_odml_from_nix(nix_filename):
15     """ load odml document from nix """
16     with nixio.File.open(nix_filename, nixio.FileMode.ReadOnly) as nix_file:
17         odml_doc, nix_sections = get_odml_doc(nix_file)
18         nix_to_odml_recurse(nix_sections, odml_doc)
19     return odml_doc
20
21 def load_neo_block_from_nix(nix_filename):
22     """ loading neo structure from nix using rawio implementation """
23     io = neo.NixIO(nix_filename)
24     return io.read_block()
25
26 # extracting command line parameters and loading original Blackrock data
27 data_location, odml_filename, nix_filename = sys.argv[1:]
28 io = neo.BlackrockIO(data_location)
29 block = io.read_block(lazy=True)
30
31 # save odml and neo block in single nix file
32 save_neo_to_nix(block, nix_filename, mode='ow')
33 save_odml_to_nix(odml_filename, nix_filename, mode='overwrite metadata')
34
35 # extract odml document and neo block from nix file
36 odml_document = load_odml_from_nix(nix_filename)
37 block2 = load_neo_block_from_nix(nix_filename)
```

Code Listing 4.4: Saving data and metadata to *NIX*. Required packages are imported (line 1f) and four functions for the conversion between *Neo* and *odML* on the one side and *Nix* on the other side are defined (line 4-24). The main part of the script (line 26ff) first extract command line specified parameters (location of the dataset and *odML* and *Nix* filenames to be used) and loads the datasets from the `Blackrock` format. Next, the *Neo* block in converted to the *Nix* format (line 32) and the complete metadata collection is added from an *odML* file (line 33). Finally, we demonstrate how to extract the corresponding information again from the *Nix* format using the `load_odml_from_nix` and `load_neo_block_from_nix` functions (line 36f). Please note that the execution time of this code highly depends on the size of the dataset, since for sequentially all parts the complete dataset will is loaded into memory.

the *Neo* objects once and can reuse the knowledge also in different experimental contexts. *Neo* can deal with most of the available electrophysiology file formats and will be extended to cover calcium imaging data in the future. For *NWB:N* there are very limited conversion tools available to convert from specific *Matlab* structures and custom formats to the *NWB:N* format. An interface for the support of additional formats exists, but format converters are in an early development stage. At this point *NWB:N* could benefit from integrating *Neo* to gain basic support for many electrophysiology formats. *NWB:N* has reference implementations in Python as well as Matlab, suiting most of the neuroscientists, whereas *Neo* is only implemented in Python, focusing on non-commercial availability of electrophysiology analysis software.

4.4 Summary

We introduced the *Neo* Python package designed for standardized and efficient representation of electrophysiological data. We presented the development of the package since the original publication and depicted the main features and flexibility of the package in three small usage examples. Finally we provide a concise comparison to the recent *NWB:N* format and highlight the strengths of the software packages. Using the *Neo* package provides a sound foundation for organizing research data according to the FAIR principles as *Neo* i) makes data accessible by supporting the conversion from various file formats to the standardized *Neo* data representation, ii) is an open, freely available and software package and iii) it makes data interoperable by providing a formal, accessible and broadly applicable basis for data representation. In addition *Neo* is supporting the *Nix* file format, which is fulfilling additional FAIR principles. These are i) making the data within the file format findable by using globally unique identifiers for data and metadata objects, ii) permitting the annotation with rich metadata as extensive metadata collections can be stored, iii) connecting data and metadata in a formal way by storing links between data and metadata objects, iv) being an open and free & interoperable framework. Additional aspects of the FAIR principles, like provenance tracking of the data and metadata, require the embedding of *Neo* and *Nix* in a data and metadata pipeline or workflow.

Chapter 5

Workflow management - A new approach for research data and metadata management

The long process that starts from the generation of data, and ends in a scientific publication, can be separated into many individual steps. These subdivision may be coarse, like the separation only of experiment and subsequent analysis. Or they may be very fine, as each individual operation, and substeps within, may constitute and be implemented in independent processes.

Workflow management is the concept to organize these individual steps. The granularity of the steps to manage highly depends on the complexity of the tasks and the diversity of the processing steps. A common and generic example forming such a workflow management system (WMS) is a queuing system used in cluster computing such as *slurm*[1] or *torque*[2] and *maui*[3]. Here users submit a number of, in the simplest case, independent jobs (computing steps) which are then scheduled and distributed to suitable compute resources depending on the requested and available resources. This is a simple example, because the individual processing steps typically do not depend on each other and only the required amount of resources and time needs to be taken into account when organizing the execution. Already with these systems, it is possible to implement more complex scenarios, e.g. by defining an order of execution via dependency statements for individual jobs.

In this chapter we apply the concept of workflow management to solve the data and metadata management issues identified in Section 2.6. A systematic workflow approach for data and metadata management supports the rigorous implementation of the FAIR principles, since the automation of processing steps relies to some extend on FAIR principles, i.e data should have a unique identifier and should be organized in a structured fashion using standardized tools. Additionally, the implementation of a data and metadata workflow increases the reproducibility of the processing steps, as these are

[1] slurm workload manager, https://slurm.schedmd.com/

[2] torque resource manager, http://www.adaptivecomputing.com/products/torque/

[3] maui cluster scheduler, http://www.adaptivecomputing.com/products/maui/

documented and provenance information can be tracked for each step of the workflow. Workflow management is suited to tackle the issues we identified in Section 2.6, since it is designed to coordinate interdependent steps of a process as they occur in the data and metadata pipeline of the Reach-to-Grasp project. In the following, we present available WMSs and identify requirements for the applicability in the context of scientific projects in general as well as for Reach-to-Grasp and similar projects in particular.

For scientific projects like the Reach-to-Grasp experiment described in Chapter 2 there are dependencies between individual steps of the process from data acquisition to publication (see Figs. 2.4 and 2.6). The workflow management concept has been applied in a number of scientific fields like genomics or imaging data. In these fields a systematic approach to data processing and analysis is required and feasible, since they are dealing with large and numerous datasets which exceed manual monitoring or processing power (e.g. Palm et al., 2010). For these and other disciplines, there are a number of platforms and tools available to implement pre & post processing as well as analysis processing steps: *Galaxy*[4], an open, web-based platform providing bioinformatics tools and services for data intensive genomics research; *VisTrails*[5], an open-source scientific workflow and provenance management system that provides support for simulations, data exploration and visualization; *Taverna*[6], a scalable, open source & domain independent tool for designing and executing workflows; *GenePattern*[7], a genomics analysis platform that provides access to hundreds of tools for gene expression analysis, SNP analysis, flow cytometry, RNA-seq analysis, and common data processing tasks; *Renku*[8], an online software platform for reproducible and collaborative data science including workflow management aspects; *Terra*[9], a scalable platform for biomedical research for data analysis and collaboration; *Ugene* (Okonechnikov, Golosova, and Fursov, 2012) a multi platform open-source software for molecular biology; *Luigi*[10], a Python based tool for building complex pipelines of batch jobs; *Airflow* [11], a platform to programmatically author, schedule and monitor workflows; *pinball*[12], a scalable workflow manager with scheduling capability implemented in JavaScript and Python; *Make*, a basic build automation tool available since 1976 with multiple implementations (e.g. gnumake[13]) and *snakemake*[14], a Python based language and execution environment for make-like workflows.

In simple and straightforward projects, one may use plain bash scripts to coordinate the sequential execution of the individual steps of a workflows, mimicking a WMS. However, once the situation grows more complex, the same issues arise as discussed for the Reach-to-Grasp metadata pipeline (Section 2.6): the bash script would form a

[4]Galaxy, https://galaxyproject.org, RRID:SCR_006281
[5]VisTrails, https://www.vistrails.org, RRID:SCR_006261
[6]Taverna, https://taverna.incubator.apache.org, RRID:SCR_004437
[7]GenePattern, http://www.broadinstitute.org/cancer/software/genepattern, RRID:SCR_003201
[8]Renku, https://datascience.ch/renku
[9]Terra, https://terra.bio/
[10]Luigi, https://luigi.readthedocs.io
[11]Airflow, https://airflow.apache.org/index.html
[12]pinball, https://github.com/pinterest/pinball
[13]gnumake, https://www.gnu.org/software/make/
[14]Snakemake, https://snakemake.readthedocs.io/en/stable/, RRID:SCR_003475

monolithic script, trying to cover all possible dependencies between individual scripts resulting in overly complex code. Additionally, the script could only be executed all at once, without taking into account which steps of the workflow are actually required due to updates in the underlying sources files.

Scientific projects, such as the Reach-to-Grasp project presented in Chapter 2, can benefit greatly from a structured workflow approach. However, the development of the workflow should in the best case smoothly integrate with the existing scientific tools and approaches used in the project. In the following, we discuss a number of essential features of a WMS, which are generally required in scientific projects. The WMS should be

slim not introduce unnecessary additional computational overhead

easy not require expert knowledge to implement and configure a workflow

standalone not introduce additional unnecessary dependencies to other projects and programming languages

visual be able to generate a visual overview of the workflow for inspection and debugging purposes

debuggable be easy to debug

active be actively supported

open be open source and freely available

Furthermore, we identify additional, more specific requirements in the context of the Reach-to-Grasp and similar projects. Here, the WMS should support

Python inherently support Python as many of the existing scripts are implemented in Python

integration integrate well with existing scripts as the existing code base should not depend on the WMS.

flexibility provide the flexibility to implement processing steps based on bash (for usage of external tools, e.g. for spike sorting)

HPC support local workflow execution as well as the usage of compute clusters by supporting common queuing systems

The general requirements for a slim and standalone WMS with only minimal overhead already excludes the majority of WMSs listed above as these provide a multitude of features like web applications that are not required in the context of the scientific projects presented here. This rules out *Galaxy*, *Renku*, *Terra* as these are web based WMSs based on a webinterface for user interaction. Ugene and GenePattern focus on very different domains (molecular biology & genomics, respectively) and provide a multitude of tools for these domains, which are not required here. *Pinball* and *Taverna* are Java based which is not used in the context of the Reach-to-Grasp or related projects.

In addition, *Taverna* is a highly developed workflow system with 3500 services available, which surpasses our requirements. *Airflow* and *Luigi* are Python-based WMSs that rely on a custom workflow definition in a Python script, which requires tool-specific knowledge for implementation and maintenance of the workflow. *VisTrails* is Python based as well, but relies on a graphical user interface for the workflow definition, thereby making the implementation of complex workflows cumbersome. *Make* is a well established tool for the organization of build processes and is therefore available on all operating systems. It has been shown that scientific workflows for quality assurance can be implemented using *Make* (Askren et al., 2016). However, the workflow definition via make is laborious as well due to a limited utility functionality. Here, *snakemake* offers a hybrid solution of make and Python. The workflow definition is implemented in a make concept, but Python functionality can be used to facilitate the definition of the workflow within the rigid structure provided by *Make*. Since Python is already the language of choice in the Reach-to-Grasp project, *snakemake* does not introduce any additional language dependencies and is therefore our tool of choice in order to demonstrate the usage of WMSs in the context of data and metadata management in scientific projects.

5.1 Workflow management tools - *Snakemake*

In this section we discuss *snakemake* as a WMS, as it is domain independent, slim and easily integrates with Python based projects, e.g. to the Reach-to-Grasp and related projects (Chapter 2).

Snakemake is a generic workflow management tool derived from the build automation tool make combined with Python features (Köster and Rahmann, 2012). It is available as bioconda[15] and PyPi package[16]. We consider the so far latest version 5.5.4 here.

We demonstrate the basic features of *snakemake* based on two minimal workflow examples. The first one (Code Listing 5.1) demonstrates the basic concept of make and the ability of *snakemake* to define steps in the workflow in which the involved filenames are not known beforehand. The second, more complex workflow (Code Listing 5.2) demonstrates the integration of *snakemake* with Python and showcases its capabilities of flexibly deducing rule dependencies and parameters of individual workflow steps.

The description of individual steps of a workflow within *snakemake* is closely related to *Make*: A processing step is defined via its input and output files (Code Listing 5.1, line 11 and 12). The core of a rule is the instruction how to generate the output files based on the input files. Here *snakemake* offers multiple options based on direct execution of Python scripts or bash scripting. Bash scripts offer the most flexibility and are marked with the `shell` keyword (Code Listing 5.1, line 13). Within executed shell command references to the input and output files can be used via Python based reformatting of the command before execution. E.g. in Code Listing 5.1, line 13 the filename specified by the input of the rule `simple_copy_rule` (line 11) is automatically copied to the filename specified by the output of the rule by using `{input}` and `{output}` in the

[15] https://anaconda.org/bioconda/snakemake
[16] https://pypi.org/project/snakemake

***Snakemake* header**

```
1  # define rule order to first use simple rules if possible
2  rule_order:
3      simple_creation_rule > create_file > simple_copy_rule > copy_file
```

Simple rules

```
5  # simple rules using explicit file names
6  rule simple_creation_rule:
7     output: 'file.md'
8     shell: 'touch {output}'
9
10 rule simple_copy_rule:
11     input: 'file.md'
12     output: 'file.txt'
13     shell: 'cp {input} {output}'
```

Flexible rules

```
15 # flexible rules using wildcards to handle file
   ↪ names
16 rule create_file:
17     output: '{filename}.txt'
18     shell: 'touch {output}'
19
20 rule copy_file:
21     input: '{filename}.md'
22     output: '{filename}.txt'
23     shell: 'cp {input} {output}'
```

Code Listing 5.1: Minimal *snakemake* example workflow. The workflow consists of two rules: i) generation of a markdown file (.md) and ii) conversion to a text file by plain copy of the content into a file with .txt extension. Two versions of each rule are implemented, demonstrating *snakemake* features at different complexities: The simple version of the rule handles filenames explicitly (left), whereas the flexible version of the rule is using wildcards to handle filenames (right). To resolve ambiguities between the two versions of the rules, we define a rule priority order in the first lines of the *snakemake* file.

shell command. The same concept can be used to formulate *snakemake* rules in a more flexible fashion. E.g. in Code Listing 5.1 a set of flexible rules is introduced, which use an additional wildcard `{filename}` to be able to generate and copy not only files with filename `'file.md'`, but any markdown file. Here, the value of the variable `{filename}` is only determined during the execution of the workflow. Therefore, the same rule can be used multiple times within a workflow with different wildcard parameters. Hereby, the value of the wildcard is determined recursively by the required output file.

The dependencies between *snakemake* rules are evaluated based on required files. By default *snakemake* uses the first rule within a `Snakefile` as main rule and tries to execute this rule. Alternatively *snakemake* can be called with a filename as an argument. In this case *snakemake* attempts to build the requested file based on all available rules, thereby matching in and output files of rules and checking the availability of basic input files. For this purpose *snakemake* generates an acyclic directed graph of rule dependencies (e.g. see Fig. 5.1) and infers all wildcard parameters from this. In case multiple rules can be used for generation of the same file a rule priority order can be be defined (Code Listing 5.1, lines 1-3). *Snakemake* only executes rules and creates or overwrites files if the output files of a rule do not exist or the input files have a more recent modification time stamp than the output files. This guarantees that the output files of a *snakemake* workflow are always based on the most recent version of input files and at the same time minimizes the computational overhead, since only required or outdated files are generated.

Snakemake rules can be executed in dedicated, containerized environments. For Python workflows, *snakemake* supports conda environments on a per-rule level. Here,

Snakefile

```
configfile: 'config.yaml'
# extract neo data format to use from configuration
data_format = config['data_format']

# restrict the data extension to use for
wildcard_constraints:
    data_ext=data_format

# plot example data in svg and png format
rule all:
    input: expand('data.{ext}', ext=['png','svg'])

# run python script to generate data
rule create_data:
    output: 'data.{data_ext}'
    conda: 'envs/data_generation_environment.yaml'
    shell: 'python generate_data.py {output}'

# visualize data
rule plot_data:
    input: '{{filename}}.{dext}'.format(dext=data_format)
    output: '{filename}.{ext}'
    conda: 'envs/plotting_environment.yaml'
    shell: 'python plot_data.py {input} {output}'

```

Environments

plotting_environment.yaml

```
name: plotting_environment

dependencies:
    - pip
    - matplotlib
    - pip:
        - nixio==1.5.0b3
        - neo
```

data_generation_environment.yaml

```
name: data_generation_env

dependencies:
    - python=3
    - numpy
    - pip
    - pip:
        - nixio==1.5.0b3
        - neo
```

config.yaml

```
data_format: 'nix'
```

Code Listing 5.2: *Snakemake* example workflow for data generation and plotting. The workflow consists of three rules, for data generation, data visualization and specification of the all output files of the workflow. The first two rules can be executed in dedicated conda environments, specified via the `conda`-directive and are shown on the right. The workflow uses a configuration file (Snakefile, line 1, `config.yaml`), specifying the format for storing *Neo* structures. This specification is also used to provide a constraint for wildcards with the name `data_ext`, which resolves ambiguities between the data generation and visualization rule. The rule `all` is by default executed when *snakemake* is run. It specifies two required output formats of the workflow. For the visualization of the workflow diagram when running the `all` rule, see Fig. 5.1.

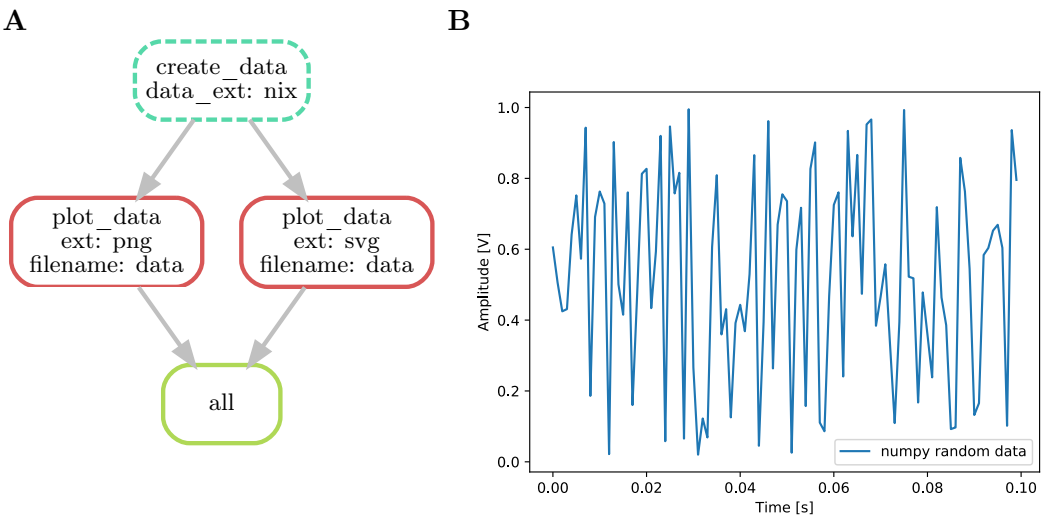


Figure 5.1: *Snakemake* example workflow for data generation and plotting. The workflow diagram (A) and resulting plot (B). The workflow consists of two rules of which the `plot_data` rule is executed twice with different parameters to generate the final plot in two file formats (`ext:svg`, `ex:png`, respectively). Different rules are color coded and the rule name is indicated at the top of each node. The command line parameters of the scripts are indicated below the rule name. The frame style (solid/dashed) indicates if this rule needs to be run to generate a final output file. In this example, the data file was already generated, wherefore *snakemake* would not rerun this rule unnecessarily (dashed box). The arrows indicate the dependencies between the rule executions. Rules at the top need to be executed first, since they generate output files that are required as input for the subsequent rules executions.

the conda environment can be defined via a `yaml` file specifying the conda (and PyPi) dependencies (see Code Listing 5.2 Snakefile, line 16 and 23 and environments). When no cached version of the environment exists or the `yaml` environment definition was updated, *snakemake* builds the environment using conda.

Code Listing 5.2 demonstrates a more complex workflow using two generic Python scripts (Code Listing 5.3). The first script generates data based on the *Neo* package, whereas the second script visualizes any data accessible via *Neo* using the Python *Matplotlib* package. These scripts are implemented to be used as standalone scripts, and require arguments from the command line indicating the filename. Additionally, they do not rely on a fixed data file format, but support any format supported by the *Neo* framework. This, in combination with the explicit definition of the required conda environments in form of `yaml` files makes the scripts highly flexible and generic, such that they can be easily reused in different contexts and projects. Furthermore, the *snakemake* implementation of the workflow keeps the generality of the code by providing flexibility in the used data format, which is defined via an additional configuration `yaml` file, and the usage of wildcards for flexible handling of filenames. The resulting *snakemake* workflow as well as the output visualization of the randomly generated data can be seen in Fig. 5.1.

In addition to the features demonstrated in the example scripts, *snakemake* inte-

Data generation

```
1  import sys, numpy, neo
2  import quantities as pq
3
4  def generate_neo_data():
5      """ Generate Neo block with random data """
6      block = neo.Block(name='generated data
       ↪  block')
7      segment = \
8          neo.Segment(name='generated data
           ↪  segment')
9      analogsignal = \
10         neo.AnalogSignal(
11             numpy.random.random(100)*pq.V,
12             sampling_rate=1*pq.kHz,
13             name='numpy random data')
14     block.segments.append(segment)
15     segment.analogsignals.append(analogsignal)
16
17     return block
18
19 def save_neo_block(block, filename):
20     """ Save Neo block to disc at filename"""
21     with neo.get_io(filename) as io:
22         io.write_block(block)
23
24 if __name__=='__main__':
25     filename = sys.argv[1]
26     block = generate_neo_data()
27     save_neo_block(block, filename)
```

Data visualization

```
1  import sys, neo
2  import matplotlib.pyplot as plt
3
4  def load_neo_block(filename):
5      """ Load data from file into Neo """
6      with neo.get_io(filename) as io:
7          return io.read_block()
8
9  def plot_analogsignal(block, filename):
10     """ Plot first AnalogSignal of Neo """
11     anasig = block.segments[0].analogsignals[0]
12     plt.plot(anasig.times, anasig.magnitude,
       ↪  label=anasig.name)
13     plt.xlabel('Time
       ↪  [{}]'.format(anasig.times.dimensionality.latex))
14     plt.ylabel('Amplitude
       ↪  [{}]'.format(anasig.dimensionality.latex))
15     plt.legend()
16     plt.savefig(filename)
17
18 if __name__=='__main__':
19     neo_filename, plot_filename = sys.argv[1:]
20     block = load_neo_block(neo_filename)
21     plot_analogsignal(block, plot_filename)
```

Code Listing 5.3: Standalone Python scripts used in Code Listing 5.2. The two scripts for data generation and visualization contain generic functions, relying on command line parameters to provide the arguments for the function calls (lines 21-24 and lines 18-21, respectively). The **data generation** is split into two functions, one for generation of the *Neo* structure (`generate_neo_data`) and one for saving the *Neo* structure to disk (`save_neo_block`). The first function generates a *Neo* `Block` containing a single `AnalogSignal` with randomly generated data (lines 4-12). The second function receives a generic *Neo* `Block` and saves it in the format specified by the provided filename (lines 16-19). If the script is executed from the command line, the input parameter `filename` is extracted from the command line arguments and both functions are executed consecutively, passing the *Neo* `Block` from one function the next (lines 21-24). The **data visualization** uses the same concept as the data generation. Here the two internal functions are loading a *Neo* block from the specified data source filename (`load_neo_block`, lines 4-7) and visualize the first `AnalogSignal` of a given plot, saving the result in a requested filename (`plot_analogsignal`, lines 9-16). Both functions are called if the script is called from the command line and the two parameters specifying the data location as well as the output plot filename are extracted from the command line arguments.

grates well distributed storage concepts, such as Google Cloud Storage[17], Dropbox[18], or the secure shell protocol (SSH). Remote file sources are declared in the header of the *snakemake* file and individual files can be referenced from these sources in the same manner as local files. Besides access to remote files, *snakemake* also integrates with high-performance compute clusters by supporting common queuing systems such as the slurm workload manager[19]. Here, a configuration file can be used to specify the cluster job parameters on a rule-level, permitting detailed resource management.

Summary We presented the application of basic *snakemake* features based on two examples demonstrating the modularization of a workflow into individual rules and their file-based dependency handling. We highlighted the flexibility of this approach by introducing wildcard based filename handling and explained the *snakemake* dependency graph. We provided examples of generic standalone Python scripts for seamless integration into *snakemake* rules and demonstrated advanced configuration features of the workflow via additional configuration files and wildcard constraints. We introduced additional features for integration of remote files and cluster usage. The presented features make *snakemake* our tool of choice for the implementation of scientific workflows, as it provides a domain-independent and slim option for workflow definition which integrates well with existing scripted data processing steps.

5.2 Practical application

Snakemake has been applied in a variety of fields and projects. Many of the provided examples and tutorials are set in the field of genomics[20,21]. Here we present a workflow design in the context of the Vision-for-Action project.

5.2.1 The Vision-for-Action project

The Vision-For-Action project builds on top of the Reach-to-Grasp project as it extends the investigation of motor control only to the interaction of motor and visual activity. The involved continuous integration of both visual input and motor control demand a more sophisticated experimental task protocol. The experimental hardware is a real-time visuomotor behavior and electrophysiology recording (RIVER) setup, which utilizes a *Blackrock* system in order to record neuronal activity, as described for the Reach-to-Grasp experiment (Chapter 2). For more details, we refer to Haan et al. (2018) and Haan (2018). The recording system additionally encompasses an eye tracking as well as a hand movement control system and a complex task design, which includes the sequential pointing to up to six targets. To the current date, only a single monkey was recorded in the Vision-for-Action setup.

[17] Google Cloud Storage, https://cloud.google.com/storage/
[18] Dropbox, https://www.dropbox.com
[19] slurm, https://slurm.schedmd.com
[20] https://snakemake.readthedocs.io/en/stable/getting_started/examples.html
[21] https://snakemake.readthedocs.io/en/stable/tutorial/basics.html

The task In the Vision-for-Action experiment a monkey is positioned in front of a horizontal, semi-transparent mirror, onto which a white dot corresponding to its hand position below the mirror, is projected (Fig. 5.2). The monkey is trained to initialize a task by moving the hand cursor into the area of a central, illuminated target. After a waiting period of $200ms$, during which the monkey has to stay in the center, the central target is deactivated and, depending on the task type, one or multiple of 6 peripheral targets are illuminated. To receive a reward, the monkey has to deactivate all illuminated targets by moving the hand cursor into the each of the targets. Depending on the task additional targets appear upon the deactivation of a previous one. Two classic task types are currently implemented: The landing task, in which the monkey is presented a sequence of three peripheral targets, which he has to deactivate by staying in each of the targets for $100ms$ resulting in mostly straight hand movements to the target. In the drawing task, the monkey is presented multiple targets at once and can chose an order and route to deactivate these. In this task type the monkey only needs to touch the target with the hand cursor. It was observed that this typically causes very curved hand movement trajectories.

The setup The RIVER setup consists of three components recording different modalities: i) the neural activity via a *Blackrock* system described in the context of the Reach-to-Grasp experiment ii) the eye movement via a an EyeLink tracking system and iii) the arm movement via a Kinarm motorized exoskeleton (Fig. 5.2).

As in the Reach-to-Grasp experiment, neural activity is recorded using a Utah array recording device. In the Vision-for-Action experiment, however, additional to a single Utah array implanted in motor cortex, four smaller arrays are implanted in visual cortex. 96 active recording electrodes are present in motor cortical area M1 and premotor cortex. Furthermore, 32 active electrodes, arranged in a 6×6 grid, are located in each of the vision-related cortical areas V1, V2, 7a and DP. The neural activity is recorded by two parallel setups (see Fig. 5.2 purple boxes). This includes two separate connectors implanted contralateral to the Utah arrays and ipsilateral to the active hand of the monkey. Each of the two connectors is connected to a separate headstage including a neural signal amplifier, digitizer and converter. The signals of each headstage are then optically transmitted to one of two real-time Neural Signal Processors (NSPs) which perform online signal processing (filtering, spike extraction and sorting). In the next step, the processed neuronal signals of each NSP are transmitted to a corresponding offline Cerebus computer for writing to disk and monitoring by the experimenter.

The active arm of the monkey is attached to a Kinarm system, which can be used to track and interfere with the monkey's movement (Fig. 5.2 green boxes). The Kinarm restricts the hand movement to a horizontal plane below the working space of the monkey and can be used to exert forces onto the monkey's arm. Online feedback of the hand location is provided visually in form of a circular white cursor in the working space.

The gaze position of the monkey is tracked using an EyeLink system (Fig. 5.2, blue boxes). To be able to record the gaze position, the monkey's head is placed in a plastic

head mask, restricting large head movements and providing access to the reward system. The gaze direction is inferred from a video signal that shows the position of the pupil and the corneal reflection of an infrared light source. The raw eye signal is processed online into the final eye position sin gal through the Kinarm real-time system. This conversion depends on the exact location of the monkey's eye with respect to the camera and light source and therefore needs to be calibrated frequently to ensure a stable gaze position recording. For a detailed description of the configuration mechanism and procedure, see Haan et al. (2018).

All online tracked signals, neuronal, Kinarm as well as gaze, are input to at least one of the two NSPs. This results in two sets of *Blackrock* files as original data files generated by the RIVER setup.

The experiment control is implemented as a Simulink model on the Kinarm real-time computer. This model coordinates the experiment using a Stateflow description of the experimental task and by generating corresponding event codes encoding the state of the system. The event codes are unique and globally defined in a generic manner using a 16 bit code. Of the 65536 possible codes, 1246 codes are reserved for generic experimental events like a trial start or the beginning of a trial metadata sequence. This leaves 64290 unused codes, providing sufficient flexibility for future extensions of the scheme. Each event code can contain metadata further specifying properties of a specific event, e.g. the maximum time range the monkey has to reach the next target before the trial is aborted. For robustness, all event codes come in pairs of two, bracketing the metadata information in a start and end bracket. The globally defined experimental codes follow a systematic scheme grouping events based on their first digit in six categories: 'Experiment Metadata', 'Trial Metadata', 'Screen Related', 'Exoskeleton Related', 'Other', 'Behaviour Related' and 'Error'. For more details on the global encoding of metadata see Haan (2018). For a specific task implementation used in the experiment, a mapping of the globally defined codes to the task specific metadata is defined. This approach permits the global use of generic event codes in the recorded data which at the same time is capable of capturing all task specific metadata. The global event codes generated by the Simulink model, together with hand and gaze location information, are forwarded to the Blackrock system and saved together with the neuronal signals.

Synchronization It is essential for the interpretation of the data and their coherent recording from three different systems that these share a common time frame. The RIVER setup produces two sets of *Blackrock* files since all other signals are integrated online during the recording. The two NSPs provide a feature, which permits to synchronize the two systems upon start. To ensure continued synchrony between the two systems, the two NSPs receive common input from the Kinarm system, which can be used offline for validating the common recording time frame.

Data and metadata files The RIVER setup saves multiple signals in parallel, as described in the following. Two sets of *Blackrock* data files are recorded: one containing

file format	content
*.ccf	cerebus configuration
*.nev	digital events • unsorted spike times • spike waveforms • experiment metadata • trial metadata • screen events • exoskeleton events • behavioural events • errors • ...
*.ns2	continuous signals with $1kHz$ sampling rate • eye (gaze) position • hand position • target position • elbow position • joint angles / velocities / accelerations • synchronization pulses
*.n6	continuous signals with $1kHz$ sampling rate • neuronal signals

Table 5.1: Recording file formats and content in the Vision-for-Action project. The cerebus configuration is saved in a custom *Blackrock* configuration format. The nev format contains digital events generated by the NSP based on the neuronal activity (spike detection) and all integrated events received from additional hardware systems, e.g. the Simulink model. Continuous signals are stored in different files depending on the sampling resolution. At a low sampling resolution of $1kHz$ the ns2 signal contains behavioral signals whereas the neuronal high sampling resolution signals are stored in the ns6 file.

neural signals from motor and the other from visual cortical areas in the `ns6` format. In addition, both datasets contain partially identical behavioral signals for the hand, arm and gaze position in the `ns2` format (Table 5.1). Furthermore, the position of the active target is stored with a $1kHz$ sampling rate as well as the synchronization signals which are shared between the two NSP / Cerebus systems.

As for the Reach-to-Grasp experiment, the `nev` file contains online extracted spikes, but in addition it also captures a large amount of structured metadata in form of events, which encode experimental metadata as outlined above. These metadata are complemented by a set of metadata descriptors. These are text files in `csv` format, structured in an *odMLtables* compatible fashion. The files are generated by the experimenter in a manual fashion using *Matlab* routines for generation of repetitive data. The content of these files contains a structure for metadata branches that can be merged and integrated in an hierarchical *odML* metadata collection. A complete set of descriptor files encompasses 9 `csv` files and covers all metadata not captured in the event recording file. These `csv` files store essential metadata about the monkey, the hardware components used, the global and specific codes used in the session, the signal flow for digital and analog sig-

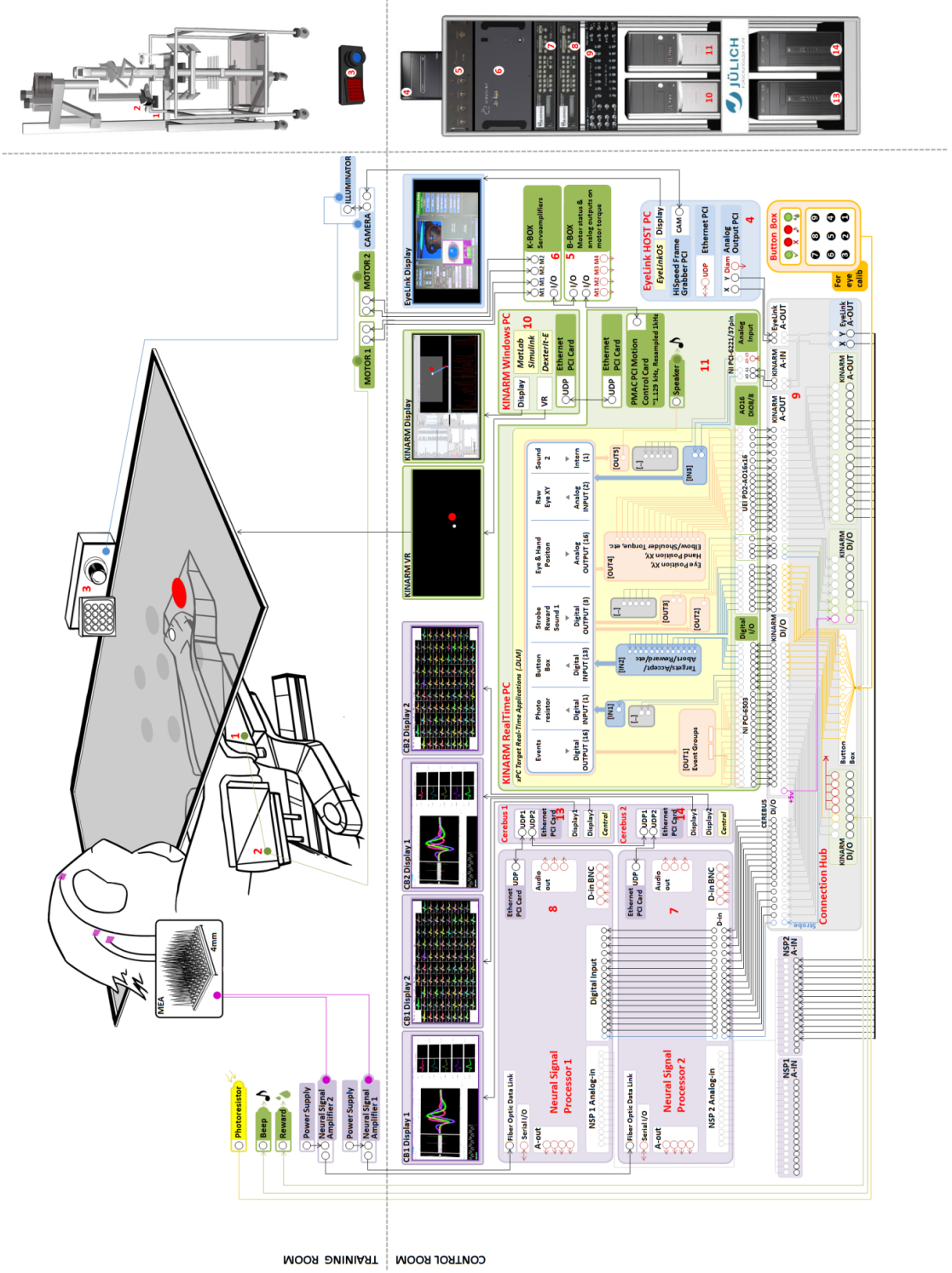


Figure 5.2: The RIVER setup including schematic of hardware components and signal flows. Depicted are the monkey task setup (top right), the recording system and signal flows (bottom left), the monkey chair and Kinarm (top right) and the recording hardware rack (bottom right). Figure from Haan (2018).

Descriptor	Content	Generation
session	• session name • relevant metadata files • task type • ...	once per session semi-automatic visual cross check
subject	• species • active hand • training • ...	onetime, static
Kinarm	• hardware specifications • programming software • ...	onetime, static
eyelink	• hardware specifications • software specifications • ...	onetime, static
blackrock	• Utah arrays • connectors • physical properties • ...	onetime, static
analog_communication	• hardware specifications • pin mapping • ...	onetime, static
digital_communication	• hardware specifications • pin mapping • ...	onetime, static
codes_global	• code mapping & definition	onetime, static
codes_task	• mapping of global codes to task specific metadata	once per task type static

Additional metadata files		
task description (pdf)	extensive human readable task description with sketches	once per task, static
task model (mdl)	model description file as used by Simulink	once per task, static
task parameter file (dtp)	task parameter file as used by Simulink	once per task, static
target picture (png)	image used for visual targets	onetime
calibration parameters (mat)	parameters of the calibration model	onetime
calibration data (mat)	data used for calibration	once per calibration

Table 5.2: Metadata descriptors and supplementary files in the Vision-for-Action project. Nine `csv` descriptor files are required for a complete description of the experiment. Most of these only need to be generated once as the data contained within is constant across consecutive recording sessions. Only the session descriptor needs to be adjusted to each session. There are six additional files which provide supplemental metadata information, e.g. additional configuration and image material used during the recording.

nals in the recording setups and the general description of the recording session as well as meta information about all required descriptor files (Table 5.2 top). Additionally, configuration and supplemental files that can not be captured in a csv file but are used during the recording are referenced in the session descriptor. On the one hand, most of the metadata files are expected to be identical for all sessions. Some on the other hand change with the task type, while only a few need to be generated / tracked explicitly for each session. Nevertheless, since during the life time of an experiment unforeseen changes might occur, such as the replacement of a part of the setup due to malfunction, it is better practice to record all metadata files anew in each session regardless of prior expectations. We also followed this approach in this specific experiment.

5.3 Metadata workflow in the Vision-for-Action project

Based on the metadata source files described in Table 5.2 (descriptors, in addition to a number of supplemental and binary data files) we designed a workflow for metadata collection and enrichment which consists of processing steps that can be classified into five processing categories (Fig. 5.3). In the following, we present concepts and implementations developed for in the context of Vision-for-Action. We design the workflow using *snakemake* in combination with Python scripts, which are implemented in a standalone fashion as described in Section 5.1 (Code Listing 5.3). In the context of this workflow we term these standalone Python scripts 'application' (app). Each app performs only a single, designated processing step based on as few input files as possible, to avoid unnecessary dependencies and provide a processing workflow that is easy to follow.

Grouping of apps We separated apps into groups according to the similarity of their interfaces, i.e. the input parameters the app requires and the type of output it generates. This way a single rule can handle multiple apps in case they have a similar dependency structure and require the same parameters. An example for a group of app with the same interface are metadata apps, coordinated by the `run_metadata_app` rule (Fig. 5.3, green box). This rule covers all apps, which generate metadata based on the original recording data and generate an *odMLtables* compatible `csv` file summarizing the extracted metadata or processing results. These apps require as input parameters the location of the original data files to be loaded as well as the location to which to write the resulting `csv` file. In contrast to this `data_apps` extend the original data (in the *Neo* representation) and generate an *odMLtables* compatible `csv` file. Each generated `csv` file is saved with an app-specific filename and typically contains only a few values of additional metadata, since apps are modularized to cover very specific tasks. In the current workflow, metadata and data apps are implemented in a flexible manner, as these apps are located in a dedicated folder. All apps in these folders are automatically included in the workflow and are executed by the `run_metadata_app` and `run_data_app` rules (Code Listing 5.4).

Two minimal examples depicting two subsets of the workflow are shown in Fig. 5.4. Here we focus on the integration of the original `csv` descriptors into a single *odML* file as well as the execution of preprocessing steps (Fig. 5.4A and B, respectively). The general dependencies between the corresponding rules are visible in Fig. 5.3, whereas the executions of the rule with varying parameters during a run of *snakemake* are depicted in Fig. 5.4. I.e. the two rules handling data and metadata apps (Fig. 5.3, green boxes) each cover a multitude of data and metadata apps. Examples of the apps run by these two rules are depicted in (Fig. 5.4B), e.g. the `run_data_app` rule executes the apps `app_synchrofact_detection`, `app_cross_talk_detection_in_ns6` and `app_saccade_detection` all with the same set of parameters. The large number of apps handled by some of the rules prevents a visualization of the complete workflow in this context, hence only a small selection of apps is depicted Fig. 5.4.

In the first example, covering the merge of descriptors, first all `csv` descriptors need

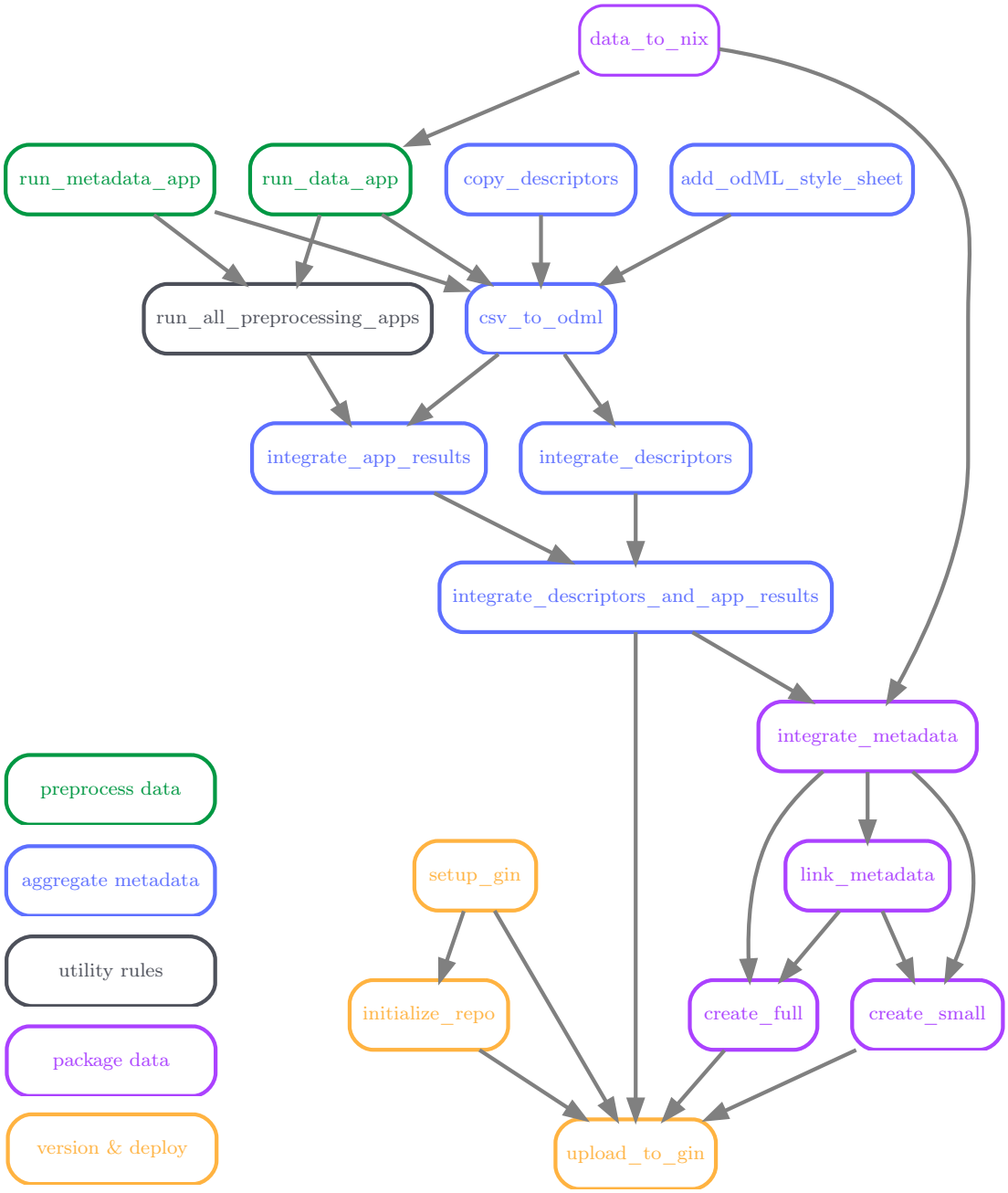


Figure 5.3: Metadata workflow rules for Vision-for-Action experiment. Visualized are only the general dependencies between rules irrespective of the input parameters and multiple executions during the run of the workflow. Data are preprocessed (green boxes) and secondary metadata are extracted. These are together with the primary metadata combined in a single metadata collection (blue boxes). Data are converted to the *Nix* format and packaged together with the metadata in a single file (purple boxes). The resulting files are then put under version control using `gin` and uploaded to a central server (yellow boxes).

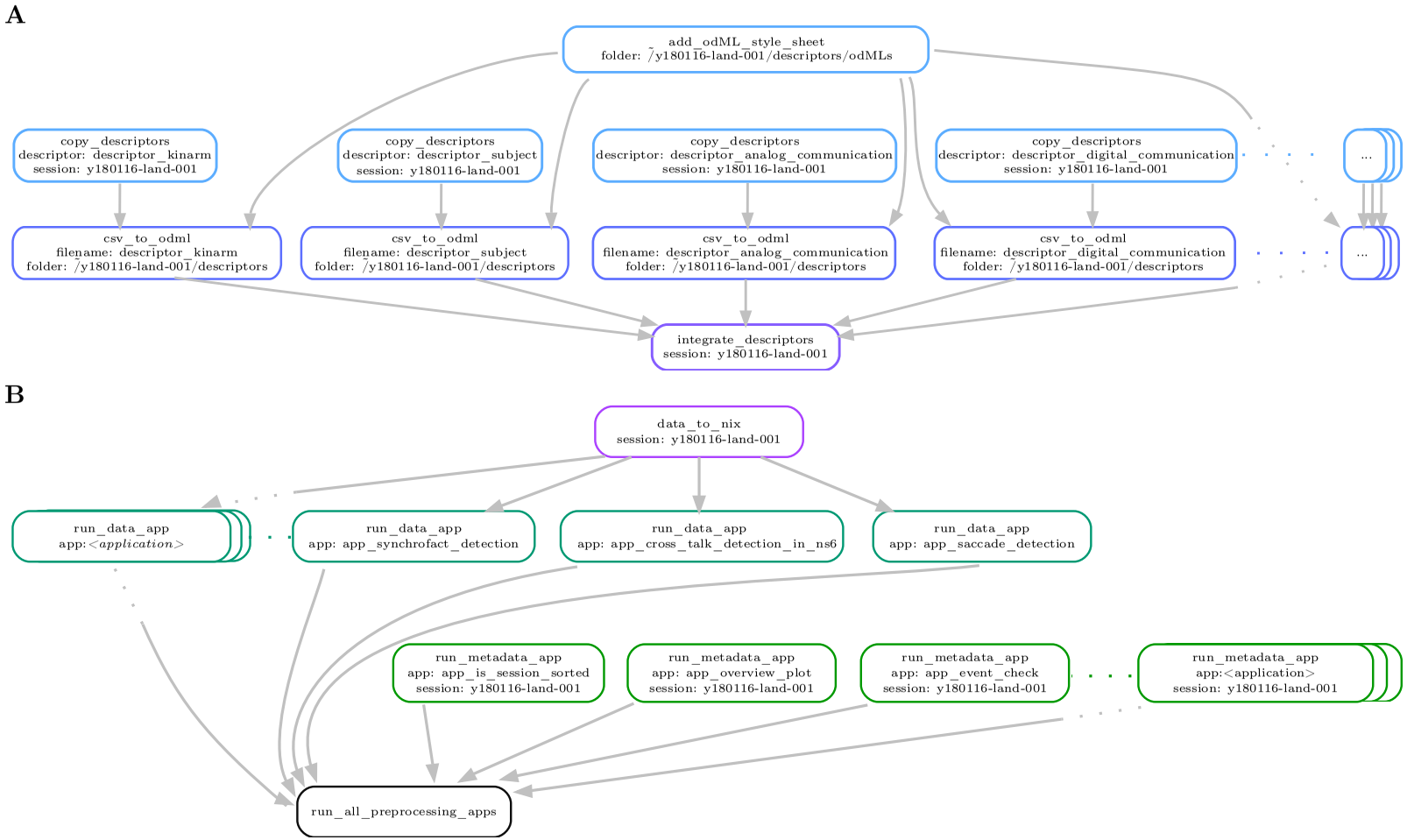


Figure 5.4: Two example metadata workflow steps in the Vision-for-Action project. **A)** Integration of original `csv` descriptors into a single *odML* file by repeated application of the `copy_descriptors` and `csv_to_odml` rule. Each application is specific for a single descriptor and resulting *odML* files are merged via the `integrate_descriptors` rule. A style sheet is automatically downloaded for easy visualization of the generated *odML* files. **B)** Preprocessing and metadata extraction via repeated application of the `run_data_app` and `run_metadata_app` rule. Both rules have similar parameter sets, but `run_data_app` additionally depends on an extendable version of the original recording data structure, which is generated by the `data_to_nix` rule. The rules take as parameter the app to run and the recording session. All rules above are triggered by a utility rule requiring the output files of all preprocessing apps as input.

```
149 rule run_metadata_app:
150     input:
151         script = 'scripts/metadata_apps/{app}.py',
152         original_data = join(DATALOC, '{session}'),
153         utils = UTILDIR,
154     output:
155         join(OUTPUTLOC, '{session}', 'app_stats', '{app}.done'),
156         csv_path = join(OUTPUTLOC, '{session}', 'app_results', 'csv', '{app}.csv')
157     conda:
158         'envs/metadata_env.yaml'
159     shell:
160         '''
161         export PYTHONPATH={input.utils}
162         python {input.script} {input.original_data} {output.csv_path}
163         touch {output}
164         '''
```

Code Listing 5.4: Excerpt of the *snakemake* workflow definition for the Vision-for-Action project. The `run_metadata_app` rule requires all apps located in the `metadata_app` subfolder as well as the locations of the original dataset and the utility functions for this workflow. It generates a `.done` file with an app specific name for housekeeping purposes as well as a `csv` file containing the extracted metadata. The execution environment is defined via a conda environment. The rule executes three lines of bash code for making the utility functions available, running the app with the specific parameters and generating / updating the housekeeping `.done` file. The variables `DATALOC`, `OUTPUTLOC` and `UTILDIR` are fixed path locations within the *snakemake* workflow and either defined via the configuration or are set at the beginning of the workflow description.

to be copied to the working directory of the workflow as the descriptors are stored together with the original data in a read-only folder. This prevents unintentional changes of the original data and makes the workflow less error-prone. Each descriptor file is copied by the `copy_descriptors` rule whereas the descriptor identity is defined as parameter. These copies serve as an input for the `csv_to_odml` rule, which permits the conversion of any *odMLtables* compatible `csv` file to the *odML* format. This step is also run for all descriptors separately. In addition the `csv_to_odml` rule also requires an *odML* style sheet for the user friendly visualization of the *odML* file via `html`. This is a required input file for all realizations of the `copy_to_csv` rule and is downloaded once to the descriptor working directory via the `add_odML_style_sheet` rule. Finally, the `integrate_descriptors` rule uses all previously created *odML files* as input and integrates all *odML* files into a common *odML* file.

In the second example, two types of preprocessing steps are performed: preprocessing and metadata extraction with and without modification or extension of the neuronal data set. Apps that only access the original neuronal data to extract metadata (e.g. data integrity checks) are coordinated by the `run_metadata_app` and access the neuronal data in the original *Blackrock* format. Preprocessing steps that extend the original neuronal dataset (e.g. by performing spike sorting) are handled by the `run_data_app` rule and require a data representation in the generic, open source *Nix* format (see Section 4.2.3), to be able to successively extend the dataset. The initial conversion from the *Blackrock* to the *Nix* format is performed by the `data_to_nix` rule. Here, the order of execution of the data apps is not specified in the *snakemake* workflow as there are no

dependencies between the different runs of the `run_data_app` rule. Hence the order of execution depends on the *snakemake* run and is not predetermined. This means only independent data processing steps can be implemented with this mechanism as no fixed order is guaranteed. Future extensions adding interdependent preprocessing steps can be added as additional rules in the *snakemake* workflow by assigning the new rules with a higher rule priority order than the `run_data_app` rule (see Code Listing 5.1, line 1-3) and introducing additional, explicit dependencies to other rules.

Currently, metadata apps cover aspects of data quality assurance as well as extraction of essential information for easy access. Some examples for data quality assurance apps are listed below:

- check for the existence of all recording files
- check for the integrity of events recorded with both NSP systems. This ensures synchronicity of the datasets between the two independent *Blackrock* recording systems.
- check for integrity of online extracted spikes and continuously sampled raw data. In case of a silent data packet loss during the recording, online extracted spike times and continuous data are not aligned from the time point of data loss (see also Section 2.6). The occurrence of a gap can be automatically detected with a high probability by comparing online generated spiking event times to the continuous recording signal at high sampling rate.

Examples for apps for the automatic extraction of basic metadata are:

- the collection of all channel specific information in a channel-specific *odML* `Section`.
- the evaluation whether a session was offline spike sorted
- the evaluation of the monkey's performance (total number of trials, number of correct trials)
- the creation of overview plots of
  - the raw recorded data for the purpose of visual inspection
  - detected hyper-synchronous events in the spiking data and their complexity distribution (see Appendix D.4)

In contrast to metadata apps, data apps extend the *Neo* data structure. Some examples of implemented and envisioned data apps are:

- the detection of cross talk between individual electrodes and annotation of the corresponding recording traces
- the detection and annotation of hyper-synchronous events in spiking data (see Appendix D.4)
- the extraction of events from continuous recording signals such as
  - the extraction of saccades from the eye (gaze) position
  - the segmentation of the hand movement into sub-trajectories

Code and data folder structure The workflow project is organized in the following structure:

```
workflow folder ..... workflow repository folder
├── Snakefile ..... definition of snakemake workflow
├── config.yaml ..... configuration of snakemake workflow
├── config_template.yaml ..... template for setting up config.yaml
├── envs
│   └── metadata_env.yaml ..... contains conda Python environment definition
└── scripts ..... contains Python scripts coordinated by snakemake
    ├── data_apps ..... contains modifying/extending data apps
    ├── metadata_apps ..... contains aggregating metadata apps
    ├── infrastructure ..... contains general purpose apps
    ├── utilities
    │   └── util.py ..... contains utility functions used by apps
    ├── tests ..... contains test suite for apps
    └── app_example.py ..... template implementation of an app
```

The separation of different types of apps into subfolders permits to automatically collect data and metadata applications during the runtime of the *snakemake* workflow. This avoids hard coding of app names in the workflow definition and provides a more flexible approach that minimizes the need for the user to customize the Snakefile. Within the *snakemake* workflow, different paths need to be configured via the `config.yaml`: i) the location of the original data files, typically only with read access, ii) the output location of the workflow, iii) potentially the server repository the results will be uploaded to. In addition to this, the configuration file can be used to specify a set of recording sessions to run the workflow on. Otherwise all available data folders will be used by default. Within the *snakemake* workflow, multiple folders are used to separate data and metadata at different levels of processing:

```
<session> ..... session specific data and metadata workflow folder
├── <session>_original.nix ..... data as contained in Blackrock files descriptors
├── <session>.nix ..... processed and extended data and metadata
├── <session>_small.nix ..... slim version of the <session>.nix file (no raw data)
├── metadata_complete.odml ..... metadata from descriptors and preprocessing
├── app_results ..... contains metadata output from preprocessing
│   ├── preprocessing_integrated.odml ..... all metadata from preprocessing
│   ├── csvs ..... contains csvs generated by preprocessing steps
│   └── odMLs ..... contains csvs converted to odML format
├── app_stats ..... contains files indicating the execution status of apps
└── descriptors ..... contains descriptor processing steps
    ├── csvs ..... contains data related to csv descriptors
    └── odMLs ..... contains csv descriptors converted to odML format
        └── descriptor_session_integrated.odml ..... all descriptor metadata
```

Metadata aggregation (data & metadata apps) Primary, initial metadata are available as *odMLtables* compatible `csv` descriptors. Secondary, automatically extracted metadata are generated in the same format by a number of preprocessing steps implemented as `metadata` and `data` apps (Fig. 5.3 green boxes). Both sets of `csv` files are converted into the *odML* format using the generic `csv_to_odml` rule which is based on *odMLtables*. In a two level approach, first metadata information originating from descriptors and preprocessing steps are each merged into single *odML* files. In a second step, these are integrated into the complete metadata collection for the given recording session (Fig. 5.3, blue boxes).

Data packaging The original data are stored in a *Blackrock* binary data format, which is optimized for the recording of large data streams. To access, analyze and modify/extend the data we use the open-source *Nix* format which is based on the `hdf5` standard and offers a direct interface to the Python *Neo* package. Hence, all data processing and extension apps are based on a data representation in the *Nix* format, which is generated by the `data_to_nix` rule (Fig. 5.3). This representation is then firstly extended by a number of data preprocessing apps (`run_data_app` rule) and secondly merged with the complete metadata collection (`integrate_metadata` rule). Next, links between the data and metadata within the *Nix* file are established, connecting *Neo* objects to the corresponding sections of the metadata collection (`link_metadata`). To provide appropriately sized packaged data for different analysis purposes, we define two flavours of *Nix* files: a `full` flavour, containing the complete dataset and a `small` flavour containing only memory friendly spiking activity and metadata.

Versioning & deployment We envision the automatic tracking of final metadata and data files generated by the presented workflow using a version control system capable of handling large data files (Fig. 5.3, orange rules). We investigate the integration with *Gin* web service for hosting data, since it is based on the common versioning software stacks *git*[22] and *git-annex*[23]. This requires the configuration of a local and optionally remote repository including access right handling. Automatic versioning and remote hosting of results from the *snakemake* workflow guarantees the consistency of datasets across time. Additionally, the *snakemake* workflow itself can also be tracked in the same repository, assuring a direct link between the workflow result and the contributing source code. Another advantage of hosting the packaged data files remotely is the central storage, providing a single reference location for all scientists working with the data. At the same time the version control system permits easy and clear communication about the data and the up-to-dateness is ensured via the automatic registration of results from the *snakemake* execution. We found that for a one directional interaction in which *snakemake* is only adding results to the repository, the integration of the two systems works smoothly. Potential problems occur in cases when the version control system is used to check out files which act as input files to any rule in the workflow.

[22] git, https://git-scm.com
[23] git-annex, https://git-annex.branchable.com/

Here, the modification time stamp of the file is not conserved between registration in the version control system and the time point of check out. However, since *snakemake* relies on consistent modification timestamps of files for the status of the workflow, this can lead to inconsistencies in the workflow management.

Validation Modularization of the individual processing steps into apps permits the implementation of validation routines to ensure correct functionality of the code. Since in this experiment the apps are Python based, tests can be implemented e.g. via the `unittest`[24] framework. Combining this concept with a versioned data repository, such tests could be integrated with one of the available continuous integration systems (see Section 2.7.1), which automatically trigger validation routines on each code update.

5.3.1 Discussion

Based on the concepts we described above, we implemented a prototype of the data processing workflow. Here, we discuss the individual features exhibited by this approach and contrast it to the workflow implemented for the Reach-to-Grasp project (Chapter 2).

Efficiency & reliability By implementing the workflow in snakemake, inherently only those workflow steps are executed for which updated input files exist. This reduces the amount of overhead and execution time in comparison to a non-modular, scripted, rigid workflow for which all steps can only be executed at once without taking into account intermediate results. At the same time using *snakemake* for determining which steps need to be re-executed is a much more reliable approach in comparison to manual evaluation of the up-to-dateness of intermediate results and initialization subsequent processing steps.

Flexibility & usability Already during the early development and installation of the workflow, output files can be generated (e.g. the complete metadata collection in *odML* format or the recording data in *Nix* format) even if they do not yet contain all information. These preliminary output files will grow proportional to the information contained by the continued extension of the workflow definition. This permits to provide output files to the collaboration community according to the software development philosophy *'Release early, release often'* (Martin, 2008) already at early implementation stages. At the same time the modular structure and simple definition of the workflow in a *snakemake* file permit a flexible extension of the procedure also at later time points during the production, e.g. when a new method for data quality estimation is developed and should be applied to all previously recorded sessions. In simple cases this can be achieved by adding an new app to the `data` or `metadata` apps folder, which will be automatically considered in the next workflow run, thus making maintenance of the workflow easy. For more complex changes, which require additional steps in the

[24] unittest, https://docs.python.org/3.7/library/unittest.html

workflow process, new rules can be added, which will be automatically integrated based on their `input` and `output` file dependencies.

Reusability The presented workflow rules and apps can be grouped into two categories: Those which do require some knowledge about specific aspects of the project and those which only require general information about file formats and generic tools. An example for the first group is the app linking between the data and the metadata part within a *Nix* file (Fig. 5.3, `link_metadata` rule). This rule requires information about the data structure and its interpretation as well as about the metadata originating from the *odML* file to be able to draw semantically useful links between the two. Another example are the various `metadata` apps, which need to be able to identify the relevant information in the source data files to interpret and extract this into a `csv` file. On the other hand, other apps and rules are generic. For example, the conversion from `csv` to the `odML` format does not require information about the actual file content. Other examples for generic rules are the integration of multiple *odML* files into a single file (`integrate_descriptors_and_app_results` rule), the integration of *odML* into *Nix* (`integrate_metadata` rule) or the setup of the *gin* repository. Thus a large amount of apps can be reused across projects, lowering both effort and cost of setting up data management workflows and making processed datasets more similar from different experiments.

Robustness Robustness of the workflow is strengthened in the modularity of the *snakemake* rules: In case one of the rules fails to produce the expected output files, e.g. by encountering invalid or unexpected data, *snakemake* keeps intermediate files. This permits to generate output files under erroneous conditions without explicitly handling all possible exceptions in the individual apps.

Outlook We plan to extend the existing workflow at multiple points. Firstly, on the side of `data` and `metadata` apps, there are a number of steps for preprocessing and information extraction which would simplify data selection for later analysis. This includes for example the definition of trials already in the preprocessing stage to provide a unified trial framework for all collaborators. Similarly, the calculation of common measures of spike train statistics can be performed at that stage and shared between scientists. Additional approaches for data quality assurance are the automatic detection of noise in raw signals and the detection of cross talk between electrodes using a coherency approach. Furthermore, additional metadata not covered by descriptors can be extracted from supplementary files and be added to the metadata collection. A more ambitious, but realistic extension would be to introduce an additional, automatic spike sorting based on the raw recording traces, which can be evaluated against the spike sorting version if manual sorting was performed for the specific session.

Secondly, in case of sequential dependencies between the extensions discussed above, additional, explicit rules for handling these need to be added in the workflow and a rule

order needs to be defined for the disambiguation of the new and existing `data` and `metadata` rules.

Thirdly, the separation of generic from experiment specific apps into a separate Snakefile would highly increase the reusability of the workflow. This utility Snakefile could be integrated in multiple projects as generic rules can be reused in different contexts. Sharing these *snakemake* rules and apps would optimally occur via a separate repository or package collecting general purpose workflow functionality wrapped by *snakemake* rules.

Fourthly, with respect to the different needs for archival of intermediate and final workflow results, the structure of the output can be adjusted to reflect whether the respective content requires archival. An example for the separation on top of the existing folder structure could be as follows: a source folder containing a copy of the original source data and metadata files, a cache folder for all intermediate and volatile files as well as an output folder containing all user relevant results of the workflow (final *odML* and annotated *Nix* files). Thus the cache folder can be removed if required.

Fifthly, to exploit *snakemake* capabilities, the workflow should run in parallel on a compute cluster. With *snakemake* supporting common queuing systems, it facilitates the migration from a local, serial implementation of the workflow towards a parallelized execution on a high performance cluster. As the amount of data increases, such compute power is becoming a necessity.

Future challenges The integration of a file modification timestamp based WMS with a version control system where modified files are based on hashes is not straightforward. Version control systems like git do not track the original modification time stamp of a file, but instead update the timestamp every time they modify the file representation on disc. This can lead to inconsistencies in the workflow management of *snakemake* if a version control system was used to checkout files. In the presented workflow scenario this is not an acute problem, since gin is only used to capture the content of all files of the workflow once at the end of the generation process and not to review older versions of the files. A workaround for avoiding version control and workflow management interference would be to additionally track the original modification time stamp of files and restore this information upon checkout.

The input and output file based workflow description as implemented by *snakemake* leads to frequent reading and writing of data, which could be prevented in a monolithic workflow implementation in a single script, as presented in Section 2.4. Here, the workflow management increases the overhead of data preprocessing. However, there are multiple factors which can counteract or attenuate this effect: i) the usage of efficient reading and writing routines, ii) the reading of only the required part of the data and iii) the workflow management itself, as it only executes required workflow rules. Since *snakemake* is already handling the workflow in an optimized fashion, the most potential for improvement lies in the read and write routines in terms of efficiency and loading of specific data.

The current *snakemake* workflow implementation features a utility script, which

contains centralized functionality needed by multiple apps (e.g. read and write data to *Nix* or `csv`). This script is therefore a required input file for a multitude of rules, and it stated explicitly in all `input` declarations. This duplicated code is not conform with the common software development principle *'Don't repeat yourself'* (Martin, 2008). However, within the framework of *snakemake* up to now, there is no satisfying solution for this except to explicitly list the utility script (see Code Listing 5.4).

Version control was introduced to track changes with each execution of the workflow. However, also hosting the original source files in a version controlled environment has advantages. For example, changes in the source files can automatically trigger the workflow and therefore form a fully automated system to update the packaged data upon updates in the source files. However, the original source files and packaged data should be hosted in two separate repositories as the read and write access to the first one should be much stricter than for the second one. This would require the repository of the original source files to trigger the *snakemake* workflow to build a packaged version of the data and commit it to the second repository. The concept is the same as for continuous integration platforms for software testing, with the only difference of the size of data files handled. Therefore existing systems can potentially be modified to serve this modified purpose. A pilot study investigating the integration of *snakemake* workflows into the *GIN* system started in June 2019[25].

5.3.2 Workflow evaluation

We evaluate the presented *snakemake* workflow with respect to the essential requirements for metadata management workflows in complex, collaborative projects defined in Section 2.7. An overview of the evaluation can be found in Table 5.3.

The presented workflow uses *odML* as a basis for metadata structuring. By using the *odML* framework, common terminologies are automatically defined when setting up the *odML* document (`R1`). This also automatically makes the metadata collection machine and human readable (`R2`), as *odML* is `xml` based and offers user-friendly tools for comprehensive visualization (*odMLtables*, odML-UI, odml_view).

We discussed potential extensions of the workflow including the automatic registration of workflow results at a central server. The combination of the presented workflow with such a remote repository can also be used to host the workflow definition and included scripts (`R4`). This can be achieved by extending the local tracking of versions, e.g. using `git` and `git-annex`, by a remote server, e.g. the *GIN* web service (`R3`). In combination with a fully automatized workflow (`R5`), the packaged data and additional output files can be added automatically to a centralized repository and this way made immediately accessible to all collaboration partners.

The presented workflow does not require manual interaction during the execution and automatically executes only required workflow steps, improving the efficiency and making the workflow less error-prone than a manual execution (`R5`). In case manual preprocessing steps can not be avoided, e.g. for supervised spike sorting, the generated

[25] https://github.com/G-Node/gin-proc

files enter as input files in the workflow. A mechanism for the automatic initiation of a *snakemake* workflow hosted on *GIN* is under development.

The modularization introduced in the workflow by using *snakemake* rules and apps makes the workflow highly flexible for future adaption and extensions (R6). Depending on the required change, additional preprocessing scripts only need to be added while the workflow definition itself remains unchanged. Changes involving the adaptation of the workflow can be implemented by introducing additional input and output files, relating new *snakemake* rules to existing ones. Due to the modularization of the workflow, generic and project specific workflow steps can be easily separated making large parts of the workflow reusable in different contexts (R7). The reusability can be further improved by separation of generic rules in a dedicated, public repository.

The workflow can provide complete provenance tracking and reproducibility when integrating all involved files into the version control system. This includes the workflow definition (Snakefile), all files called by the workflow (apps, utility and other Python scripts, configuration files, conda environment definitions), or corresponding version identifiers of these, as well as the generated output files. In an optimal case this also includes version identifiers of the original, read-only source files. The provenance tracking of Python dependencies can be implemented by specifying exact software versions for the conda environments used for different rules in the workflow or by extracting the current software stack during each run of the workflow (R8).

The workflow generates a compiled version of the data and metadata in a single *Nix* file ensuring the consistency of contained data and metadata (R10). Accessing the data only requires a current installation of the *Neo* package, making the data available also to inexperienced users without installation of additional packages or custom scripts like the `ReachGraspIO`. This lack of dependencies permits the user to benefit from continuously deployed, packaged data as no additional software requirements are introduced when updating the data. This way the user can exploit being continuously updated on the data and metadata side without suffering from software version inconsistencies.

Nix is based on the standardized `hdf5` format. This makes access to the data more memory efficient. It can also be implemented for other programming languages besides Python. Since data and metadata are linked, it makes it easy for users who are not familiar with the metadata structure to access the metadata of a corresponding data object (R9).

By building the workflow based on Python, all utilized software packages are open-source (R11). This includes *odML* and *odMLtables* (Chapter 3) for metadata handling, *Neo* and *Nix* for data representation and storage, and *snakemake* for the implementation of the workflow. Additional canonical Python packages are used in the context of individual apps, like *Matplotlib* for visualization as well as *NumPy* and *SciPy* for efficient array based computation.

Requirement	Brochier et al., 2018	Vision-for-Action
R1: Common terminologies	in project	in project
R2: Structured machine & human readable metadata	yes	yes
R3: Central data and metadata location	no	planned
R4: Version control	no	yes
R5: Mostly automatic metadata compilation	manual initialization	yes
R6: Extendable metadata workflow	minimal	yes
R7: Reusability	partial	yes
R8: Standardized & reproducible preprocessing	no	yes
R9: Easy to access data and metadata for non-experts	partial	yes
R10: Consistent data and metadata	partial	yes
R11: Open source tools	mostly	yes

Table 5.3: Overview of features of the pipeline and workflow applied in the Reach-to-Grasp (Brochier et al., 2018) and Vision-for-Action projects based on requirements for data and metadata workflows as defined in Section 2.6 (extension of Table 2.3). The presented workflow fulfills almost all essential requirements for metadata management workflows in complex, collaborative projects.

5.4 Summary & guidelines

We motivated the need for workflow management tools and introduced appurtenant concepts and implementations. We demonstrated *snakemake* as a flexible and lightweight tool for Pythonic *Make*-style workflow description and explained its main features based on two examples. In the next step we introduced the Vision-for-Action experiment as a source of datasets that exceeds the previous Reach-to-Grasp experiment in size and complexity. We described a workflow to handle data and metadata adhering to the FAIR principles for this complex electrophysiology experiment – from the original recorded data up to user friendly, comprehensively annotated data packages. We also provided detailed examples, discussed features and limitations as well as future plans and challenges. The presented workflow implements the FAIR principles, as it makes data and metadata findable, aggregates a comprehensive metadata collection and combines it with the data in a single, searchable *Nix* file using global identifiers. Since *Nix* is a standardized format, the data and metadata can be easily indexed by databases making the complete dataset also findable by other scientists once the dataset, or even only the metadata, is published. The *Nix* format is an open, free and universally implementable format making the data and metadata easily accessible with standard protocols. The data and metadata representation is interoperable, since the description within the *Nix* file is self contained and references to related metadata can be added using global identifiers. Provenance information can be tracked within the workflow and stored together with the data in a versioned repository making, in combination with the comprehensive metadata collection, the data reusable by other scientists. In addition the workflow implementation provides a foundation for the set up of workflows for other experiments, as generic components were identified and extracted from the workflow. This facilitates the realization of the FAIR principles in future experiments. In addition we abstract general guidelines from the presented workflow to ease the implementation of the FAIR principles for other projects:

G1: Plan ahead Integration and usage of data and metadata is a topic discussed concurrently with the development of an experimental setup. There are typically a number of decisions which can simplify the implementation of the data curation workflow later on, like agreeing on a consistent metadata output format for all hardware components or combining multiple data streams in a single file already in the recording setup instead of realigning and merging these files later during preprocessing steps. Also some time should be spend on the design and structure of the metadata collection to avoid reorganizing metadata in later steps.

G2: Better more than less Tracking more metadata than the essential ones will most likely turn out to be a wise choice later on, when irregularities are discovered or suddenly a seemingly unimportant parameters is needed for the investigation.

G3: Make it explicit Implement the workflow used for data and metadata management explicitly in a formalized workflow language instead of running scripts manually in a certain order. This will make the workflow i) easier understandable, both for others and oneself at a later time ii) provide an overview of the project iii) document your data provenance and preprocessing

G4: Modularize Try to segment the workflow into independent steps, as this will automatically make the workflow easier to understand and reuse, especially when well documented. Try to separate generic processing steps from project specific aspects to improve reusability.

G5: Keep it linear Avoid circular dependencies in the workflow, as these introduce complex dependency structures and indicate a non-optimal segmentation of data / processing steps. When using *snakemake* circular dependencies are innately not permitted.

G6: Think like a user Be user friendly. Try to provide the data and metadata in a format that requires as little prior knowledge as possible. This increases the chances that others can and more importantly, will make use of the data. Additionally, it paves the way for publication of the data without too much additional overhead.

G7: Deploy Continuously This advice from the agile software development approach may as well be applied in a scientific context. Do not attempt to build the perfect workflow, but provide early on intermediate results to collaborators. This enables them to provide valuable feedback and gives them a chance to work with the data while they are still excited about it. For this version management is a prerequisite, since it is essential to be able to communicate about different versions of the workflow output.

Chapter 6

Discussion

The field of experimental neuroscience deals with fundamental questions concerning the functioning of the brain. To answer these questions, experimental data from many different modalities are collected in complex experiments and subsequently combined in analyses. A cornerstone of the scientific method and a prerequisite for derived scientific knowledge, however, is a complete and comprehensive description of the collected data. We present the current state of data and metadata management in the neurosciences based on a complex experiment and identify key issues. We address these by presenting software solutions and tools for simplified and consistent data and metadata handling. This includes the simplified and standardized aggregation of metadata as well as the consistent presentation of data and the systematic structuring of workflows. We extend this by presenting a second, more complex experiment and demonstrating the integration of the tools to form a complete, reproducible workflow.

In Chapter 2 to Chapter 5 we present information processing and related tools and principles in the context of making complex electrophysiological experiments accessible, understandable and usable in a collaborative setting following the concepts described in Zehl et al. (2016) and Zehl (2018). We introduce the FAIR principles for scientific data management and stewardship and evaluate the presented tools and approaches based on these. We describe two experiments and their corresponding data and metadata processing pipelines. In doing so, we focus on the evolution of the approach used in the second experiment based on a rigorous evaluation of the first.

The two presented experiments thus demonstrate the progression of approaches for data management in this context. The first example, described in Chapter 2, is an electrophysiological experiment involving a macaque monkey trained for a reach to grasp task (Reach-to-Grasp experiment). The recording encompasses high-resolution neuronal data as well as online and offline processed data together with task control and behavioural signals. The metadata in this project was collected retrospectively and had to be extracted from a variety of different file formats involving different preprocessing steps in part manually by scientists. We describe the pipeline which was used to create a comprehensive metadata collection for two published datasets and discuss limitations of the chosen approach. Based on the described metadata pipeline we identify essential requirements for the data and metadata handling in complex, collaborative projects.

In Chapter 3 we present *odMLtables*, an open-source tool we developed to aid scientists to collect metadata in a standardized format in their daily routines. *odMLtables* facilitates the interaction with the hierarchical, `xml` based *odML* format by converting between *odML* and a tabular representation of the metadata, making the *odML* format accessible with widely used spreadsheet software. In addition, *odMLtables* implements a number of functionalities that we identified as common steps in creating and using metadata collections. This is an important step for metadata collection, since typically a large fraction of metadata is collected neglectfully and in non-standardized formats.

A comprehensive metadata collection is essential to interpret the data. However, also data require a standardized representation in order to make them easily accessible to scientists. In Chapter 4 we introduce the *Neo* software, which provides a standardized, generic data representation for electrophysiological data. We discuss the evolution of the *Neo* data representation and highlight the most important features, such as the support for numerous proprietary and open electrophysiological file formats as well as the generic structure with a flexible annotation mechanism for custom description of the data. For this purpose, we demonstrate the usage of *Neo* in three scenarios and list open source tools building on the *Neo* structure.

Finally, we introduce the second electrophysiological experiment, conducted after the completion of the Reach-to-Grasp experiment, which is named Vision-for-Action and features a more complex experimental design. This increase in complexity required us to reevaluate our data and metadata workflows in light of the limitation identified in Chapter 2. Here, neuronal signals are recorded from two cortical areas at once using two parallel, synchronized recording setups. In addition, the monkey is trained to track visual targets in a horizontal plane with a manually operated cursor while his eye and hand movements are recorded simultaneously. The organization of data and metadata in this experiment is designed for facilitated access and organization of postprocessing steps. We implement a comprehensive metadata workflow integrating metadata in a modular, automatized fashion using the *snakemake* workflow management system and describe the combination of data and metadata into user-friendly data packages. We evaluate the new workflow based on the requirements identified in Chapter 2 and provide generalized guidelines for the design of future projects.

Combining all of the presented approaches and tools, we suggest a general schema of scientific data and metadata handling from generation to publication (Fig. 6.1) supporting the implementation of the FAIR principles. In this approach, data and metadata are packaged into a common format using a modular workflow, as implemented in the Vision-for-Action project. This workflow can be automatically initialized upon any change in the underlying data or metadata (such as the addition of new data files or changes in the preprocessing steps) and generates a packaged version of the data which is then deployed to a central repository via a continuous integration system. This leads to fast, efficient continuous deployment of data, that is guaranteed to be up-to-date. From there scientists can access the packaged versions and run different analyses or publish the datasets to make them available to the global scientific community, without

the need to supply additional custom codes.

The tools presented in this manuscript can be applied throughout the process from data generation to publication:

odML provides the basic structure for metadata storage during compilation. The final workflow outputs, the packaged data and metadata in the *Nix* format are consistent with this representation. *odML* supports the implementation of the FAIR principles by permitting the definition of common terminologies and providing a standardized, accessible and searchable structure for metadata storage.

odMLtables is used for converting metadata into the standardized *odML* representation and can be used for visual inspection and exploration of the metadata collection in a tabular format at any stage. It supports the implementation of the FAIR principles by facilitating the usage of *odML* in laboratory environments, thereby supporting the spread of standardization of metadata.

Neo is used for data conversion and standardized representation during preprocessing, compilation and analysis steps. It supports the implementation of the FAIR principles by providing a standardized data representation and conversion to the *Nix* format.

Nix is used for combined storage of data and metadata in a standardized, interoperable format adhering to the FAIR principles by implementing persistent identifiers for indexing of data and metadata.

Elephant functionality can be used during preprocessing steps as well as during later analysis steps. *Elephant* contributes to preprocessing steps according to the FAIR principles and builds on data adhering to the FAIR principles and can be used to implement reproducible data analysis using these data.

snakemake is a workflow system which is used for implementing the preprocessing, compilation, integration, packaging and deployment workflow and can as well be used for systematic data analysis at later stages. Utilizing *snakemake* forms a robust foundation for data and metadata management based on the FAIR principles.

Gin is a git-annex based repository that can be used for hosting the original raw data and metadata to provide version control and a basis for a continuous integration system. At the same time, it is our service of choice for version controlling and hosting the packaged metadata and can serve as a platform for publishing the data as it provides a digital object identifier (DOI) service. Also, it allows local deployment on the server at an institution. *GIN* provides a service to make data findable on a global scale according to the FAIR principles by providing storage and digital object identifiers for data publication.

All presented tools fulfill the FAIR principle of being open and free. The majority of the presented tools are not limited in their applicability to neuroscience. Thus, the presented concepts are suitable for any application generating and processing data and are therefore very well positioned for transfer into different domains.

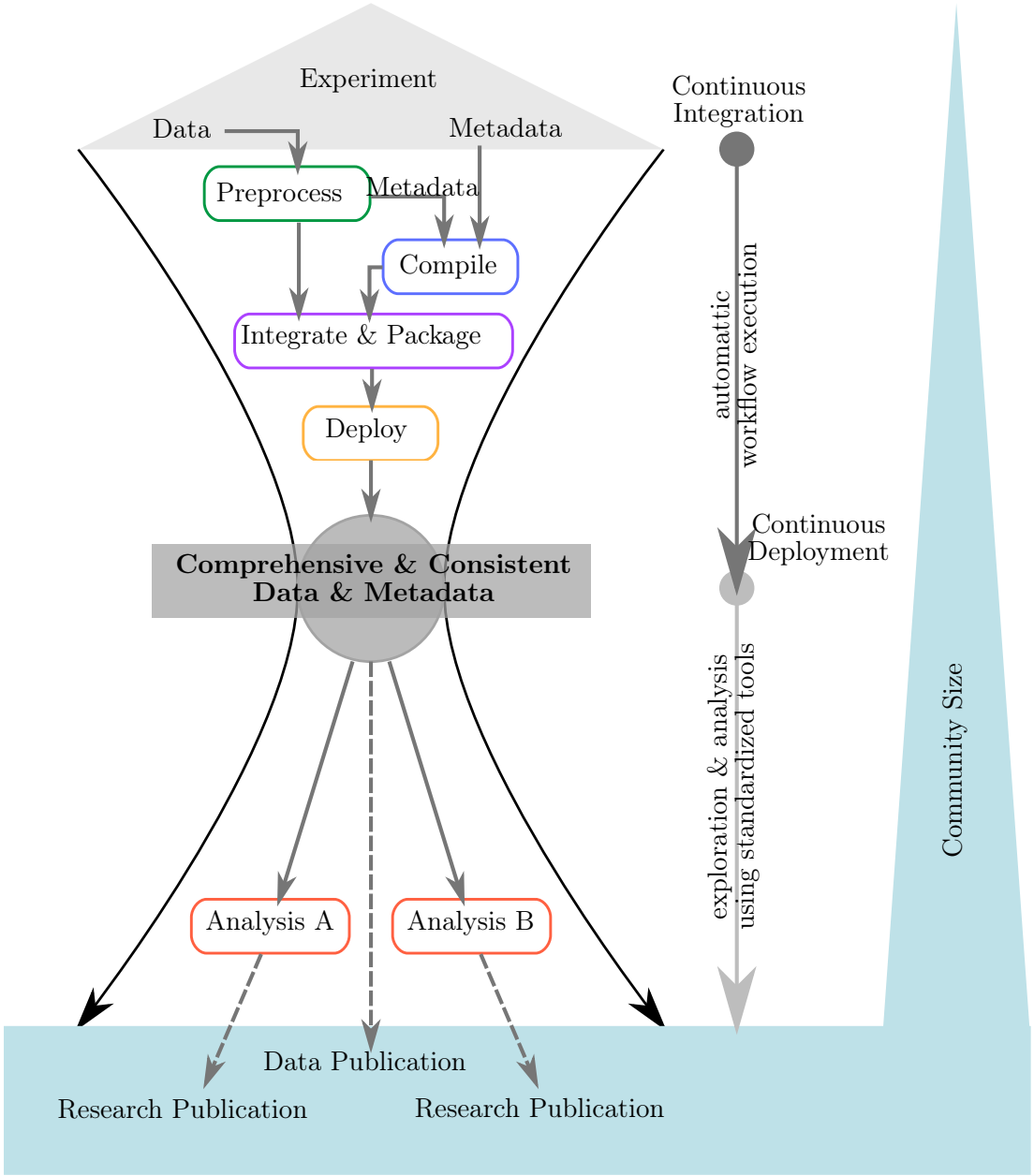


Figure 6.1: General schema of scientific data and metadata handling. Data and metadata generated during an experiment are processed, compiled, integrated and packaged to be deployed to a central data & metadata repository (center). From this reference data and metadata package scientist can run specific analysis and / or publish the packaged data. The process can be initialized by using a continuous integration system, where it is executed automatically and then be deployed in a readily packaged version to a central repository. By following this process, the data can successively be made sense of and used by a larger community up to a global level, when data are published.

6.1 Comparison of Reach-to-Grasp and Vision-for-Action workflows for data and metadata handling

Since the implementation of the original metadata pipeline conceptually following Zehl et al. (2016) and published in the Reach-to-Grasp datasets (Brochier et al., 2018) software tools aiding data and metadata handling in neurosciences evolved (see e.g. Chapters 3 and 4) and shortcomings of the original approach were identified by the use of the data in a large collaborative setting. These changes in tool availability and conceptual differences in the planning and execution of the Vision-for-Action experiment led to different data and metadata approaches in the two projects. We discuss these differences with respect to specific aspects of the experiment in the following.

6.1.1 Experimental design

In the Reach-to-Grasp project systematic metadata collection started during the runtime of the experiment. The lack of a detailed design of a metadata pipeline in preparation for the experiment resulted in metadata being stored in distributed files and various formats, aggravating the systematic collection. To quickly make the most essential metadata accessible with the data, a special loading routine (`ReachGraspIO`) was implemented partially containing hard-coded metadata. This decision complicated the systematic metadata aggregation in the long run by introducing circular dependencies in the metadata pipeline set up around the `IO`, i.e. hard-coded metadata and metadata of the growing metadata collection co-existed, leading to possible contradictions. In the Vision-for-Action project we therefore tried to i) minimize the amount of metadata source files, ii) provide them in a consistent, standardized format that is also human readable and user-friendly and iii) refrained from developing a custom loading code that includes data and metadata processing. The `csv` format together with additional structural restrictions provides suitable tables which can be easily converted to a hierarchical *odML* structure using *odMLtables* (Chapter 3). In addition internal changes in the design of the data recording were implemented: With the RIVER setup, special attention went into the integration of the three different types of recording systems. Instead of running the three systems for tracking neuronal, hand and eye data in parallel, these were integrated with the Kinarm system acting as master system for signal integration and coordination. The two Neural Signal Processors serve as the only output streams of the setup by not only writing neuronal, but also eye and hand signals to disk via the two Cerebus systems. Another improvement implemented in the RIVER setup is the generic encoding of events, which is also used to systematically write parameters and additional metadata into the same files as the neuronal recording data. Storing the complete recording data and metadata in as few files as possible ensures data consistency to a high degree. The introduced generic encoding of events ameliorates the complex event interpretation that was required in the Reach-to-Grasp experiment, where the interpretation of individual events depended on the history of previous events in a complex fashion. With the introduced encoding events have a static interpretation independent

of other events. Additionally, they are more robust against errors in the recording as they always consist of a pair of events, forming an information block. Furthermore, the event encoding can be flexibly used for different modes (e.g. task types) of the recording as each task has a unique mapping from these generic to task specific events.

6.1.2 Concept for metadata aggregation

The concept for compiling a metadata collection differs between the Reach-to-Grasp and the Vision-for-Action experiment (Fig. 6.2). In the former, the metadata structure is defined via a set of templates, which provide an *odML* structure containing default values. These templates are generated dynamically by a script on a session-by-session basis and are merged to build a complete template metadata structure. However, for generating a suitable template structure information from the metadata sources is already required introducing additional interdependencies in the process and complicating the metadata aggregation (Fig. 6.2, red arrow). In the next step, the default values are replaced with the actual metadata entries extracted from the various source files, which again requires knowledge about the odML structure in a semi hard coded fashion, i.e. template generation and population of the template need to be compatible. Additionally, during the aggregation process the metadata pipeline explicitly attempts to resolve a number of interdependencies between the different metadata sources. This example demonstrates that the intended separation between the structure and content of the metadata collection is not feasible due to dependencies between the two, introducing additional overhead and exception handling in the metadata aggregation procedure.

In the Vision-for-Action project, the metadata aggregation is implemented in a different way: The metadata structure is generated by the same functions that extract the metadata from a source file. This way the metadata content and structure for a specific part of information are handled at the same location in the code, rather than being distributed. This simplifies the metadata aggregation process and allows splitting the process into multiple, independent processes (Fig. 6.2).

6.1.3 Changes due to software updates

The Reach-to-Grasp metadata pipeline was implemented using *odML* version 1.3, in which a *odML* `Properties` can only be generated when containing at least one `Value`. This constraint makes the *odML* structure as intermediately generated by the Reach-to-Grasp workflow unnecessarily complicated by enforcing the usage of default values for all potential `Value` entries. With the release of *odML* version 1.4 this constraint was lifted, such that an *odML* structure can be created without any `Value` entries. Updating the Reach-to-Grasp metadata aggregation pipeline to *odML* version 1.4 would therefore simplify the pipeline to a small extend. However, this does not resolve conceptual issues related to the general metadata aggregation concept.

Additionally, when the Reach-to-Grasp metadata pipeline was implemented, *Neo* did not yet support the *Nix* format. For this reason the metadata aggregation pipeline was used to generate a metadata collection in the *odML* format, but did not combine

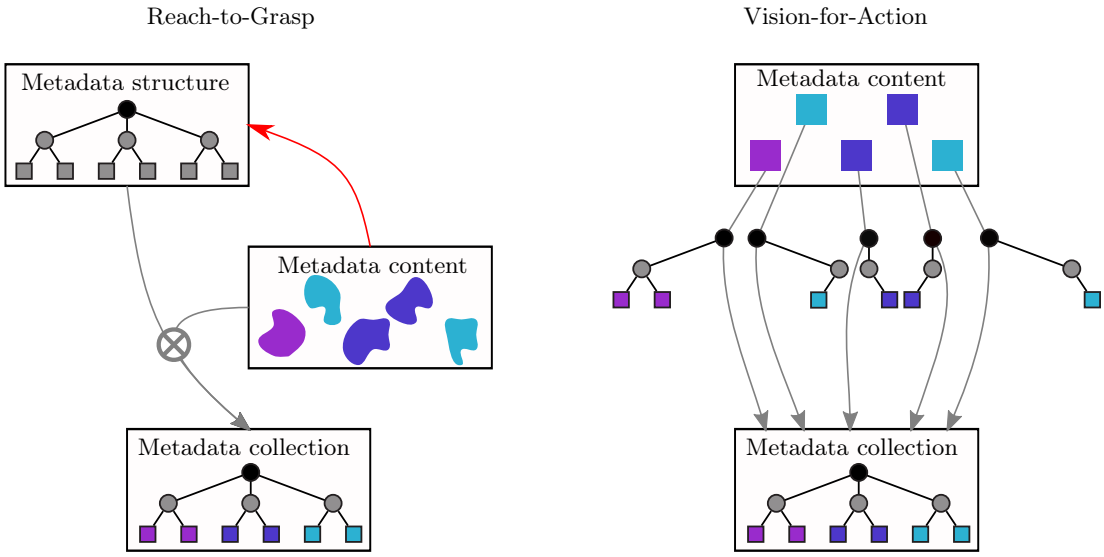


Figure 6.2: Comparison of the metadata aggregation for Reach-to-Grasp and Vision-for-Action experiments. In the Reach-to-Grasp project, the metadata structure generation is strictly separated from the metadata content (left). The integration of data and metadata occurs in a complex operation combining the two aspects. In the Vision-for-Action project the structure is generated piece wise during the metadata is aggregation (right). This approach has the advantage that data and metadata are already combined from the beginning and the integration of the individual components of the metadata collection is straight forward (e.g. by using *odMLtables*).

data and metadata in a single file. For this purpose, the `ReachGraspIO` was implemented to provide a comparable functionality at runtime. However, this introduced additional interdependencies within the project (Section 6.1.2) and attenuated the user-friendliness by requiring the usage of custom code. This situation where code, metadata and data are always required to be fully consistent on the one side, but on the other side the metadata is continuously developed and extended leads to frequent inconsistencies on the user side. This thwarts the implementation of analysis while still using an up-to-date version of data and metadata.

6.1.4 Usability

The Reach-to-Grasp pipeline generates a single *odML* file per recording session. Therefore accessing the data as well as the metadata requires the user to have specific, compatible versions of data and metadata files and software packages set up. This includes the metadata in *odML* format, the data distributed across one `nev`, `ns2` and `ns6` file for each recording session. In addition, the *Neo* and *odML* Python packages, as well as the custom `ReachGraspIO`, is required for accessing all information contained in the data and metadata files. This, however, only provides a basic annotation of the data with some selected content from the metadata collection and does not provide direct linking between data and metadata structures. If the users require additional metadata beyond the annotations, they need to find and extract this information manually from the metadata collection.

Applying the same strategy in the Vision-for-Action project would have resulted in several metadata source files and also two sets of neuronal data files, as the setup contains two NSPs. Therefore, here the data and metadata workflow generates a single *Nix* file combining data and metadata in a singular framework, in which the data and basic metadata are accessible requiring only the Python *Neo* package. For accessing additional metadata any commonly available `hdf5` viewer can be used. A project in development for visualization of *Nix* files taking into account basic metadata is *NixView*[1]. Combining data and metadata in a single file guarantees the correspondence between the two, whereas in the Reach-to-Grasp project this needs to asserted manually.

6.1.5 Pipeline and workflow approach

Within the Reach-to-Grasp project, a single Python script orchestrates the generation of the *odML* structure and enrichment with metadata. This results in convoluted code as one tries to separate the process of building and reading the *odML* structure and metadata sources with moderate success. In the Vision-for-Action project this separation of creating templates and populating them is actively avoided resulting in a much more flexible, reusable and scalable workflow consisting of modular steps interacting only via their in- and output files. In addition, the workflow is easier to understand as the dependencies can be automatically visualized and the workflow can naturally be executed piece wise, which aids troubleshooting and exploration. Furthermore, the workflow steps can be separated into generic and experiment specific components, where the former provide a basis for the exchange and sharing of metadata workflow approaches across projects and laboratories. This automatically makes data and metadata handling more comparable and therefore provides a foundation for exchange and publication of scientific data. For this reason, we abstracted a set of general guidelines for the handling of data and metadata from our experiences, which are described in Section 5.4.

6.2 Outlook

6.2.1 The future of *odMLtables*

odMLtables emerged from a collection of *odML* utility functions that accumulated in the context of the Reach-to-Grasp project. Currently, it is a standalone Python package with key dependencies on *odML* and *PyQt5*. The graphical user interface (gui) provides non-programmatic access to the core features of *odMLtables*. For historic reasons, *odMLtables* internally uses a custom, dictionary-based representation of the *odML* structure. Replacing this with a native *odML* Python object will ensure consistency during the metadata manipulation using *odMLtables*.

In addition to *odMLtables*, a native *odML* editor, *odML-UI*, exists, which only operates on the hierarchical *odML* representation. Currently the four main features of the *odMLtables* gui can be accessed via *odML-UI* if *odMLtables* is installed. However, in

[1] NixView, https://github.com/bendalab/nixview

the long run, it would be most user-friendly to integrate the two tools into a single one thereby reducing the dependencies on the user side and providing a more concise set of tools for metadata handling. With an enhanced *odMLtables* version using a Python *odML* representation internally, integration of the two tools will be straightforward.

The metadata structure used within the *Nix* model is based on *odML* and metadata can easily exported into and read from an *odML* file via the *nix-odML-converter*[2]. This permits the straight forward integration of *Nix* as an additional file format supported by *odMLtables* by utilizing the *nix-odML-converter*.

6.2.2 The future of *Neo*

We described the evolution of the *Neo* package from the original publication by Garcia, Guarino, et al. (2014) to the current version 0.7.2 as well as potential enhancements in future versions in Chapter 4. The basic concepts for capturing data within *Neo* objects are rather stable and also the *Neo* container objects for handling temporal relations between these data objects (`Segments`) did not change in the last releases. In contrast to that, the container objects for capturing channel and general object relations was updated frequently. In the current *Neo* version, this is implemented by `ChannelIndex` objects covering a number of different functionalities for handling object relations. We suggest splitting these different functionalities to create a set of a few simple objects and methods fulfilling the same task. The first step of this was already implemented in the form of array annotations. In the next steps we suggest to introduce `Group` and `View` objects (Section 3.4.4). However, this suggestion is still under debate and will if at all only be adjusted in the future.

With the release of *Neo* version 0.6 a standardized API for readers was introduced. This harmonized the multiple implementation approaches collected in the *Neo* framework and at the same time improved the performance of the readers. However, this only affects the reading aspect of `Neo`. On the writing side, there is no standardization of code, since the writing to multiple formats requires more diverse code organization than funneling different file formats into a single representation. Additionally, the IOs writing capabilities are limited to eight implementations. Thus, the effort of finding a code structure suited for general writing must take that into account. However, two useful extensions on the writing side will be i) a validator, checking the integrity before writing the *Neo* structure to disk and forming a standard of valid and writable *Neo* structures and ii) unittests for ensuring the compatibility between writable and readable *Neo* structures (see Fig. 6.3).

6.2.3 Automated workflow management

With the pilot study investigating the integration of *snakemake* workflows in the *Gin* web service[3] the development takes a direction towards a fully automated data and metadata workflow. Here the creation of new recording data files triggers the workflow

[2] nix-odML-converter, https://pypi.org/project/nixodmlconverter
[3] gin-proc, https://github.com/G-Node/gin-proc

annotating, preprocessing and preparing these data in a version controlled manner for scientific usage. This permits the setup of a workflow based on continuous integration and deployment principles, automatically executing the workflow upon a change in the source files and updating the packaged data stored at a central, versioned location. Here the setup of a system capable of dealing with large datasets as generated by electrophysiological experiments and the sustainable storage of past versions of these large data are issues to be solved before the system can be used in a scientific application.

6.2.4 Data analysis

For analysis of electrophysiological data, there are a number of tools available with different analysis focuses (e.g. see Unakafova and Gail (2019)). Since the data and metadata workflow presented in Chapter 5 creates a comprehensive data representation in the `hdf5` format, in principle any of these toolboxes can be used for analysis as long as `hdf5` reading capability is available in the corresponding programming language. However, analysing the data using a *Neo* based approach provides a direct and simple access to the data. This leaves mainly four tools for data analysis based on *Neo*: *Tridesclous*[4] for spike sorting, *Open Electrophy*[5] (Garcia and Fourcaud-Trocmé, 2009) for viewing and explorative analysis, *SpykeViewer*[6] for navigation, analysis and visualization and *Elephant* for comprehensive data analysis. Of course the data can also be extracted from the *Neo* structure and used in any other Python-based analysis tool, however, this forfeits the inherent data consistency and metadata annotations. Therefore *Elephant* is the tool of choice here. It offers a wide range of basic and a number of advanced methods for the analysis of spiking and continuous neural signal activity. As *Elephant* is a community-driven open-source toolkit also extensions are highly welcome.

Any type of analysis can be supported also from the *Neo* side. Here extending the set of utility functionality, e.g. for data selection and preparation for the analysis would benefit the user independent of the specific analysis toolbox used. Two examples for potential features to be extended and improved on the *Neo* side are i) the filter functionality for efficient selection of data objects based on their attributes and custom annotations ii) the utility functions for user-friendly interaction with memory-optimized *Neo* structures (lazy objects).

6.2.5 Published datasets

The two electrophysiological datasets described in (Brochier et al., 2018) and published in April 2018 have not been reused in an independent study. This might be due to the fact, that even though the dataset is very rich, is also limited to a single cortical area and a monkey performing a very specific task. Therefore the dataset might be of scientific interest for two types of scientists: i) researchers investigating very similar questions to those addressed in the experiment or that require parallel recordings on

[4]Tridesclous, https://tridesclous.readthedocs.io
[5]Open Electrophy, https://pypi.org/project/OpenElectrophy/
[6]SpykeViewer, https://pypi.org/project/spykeviewer/

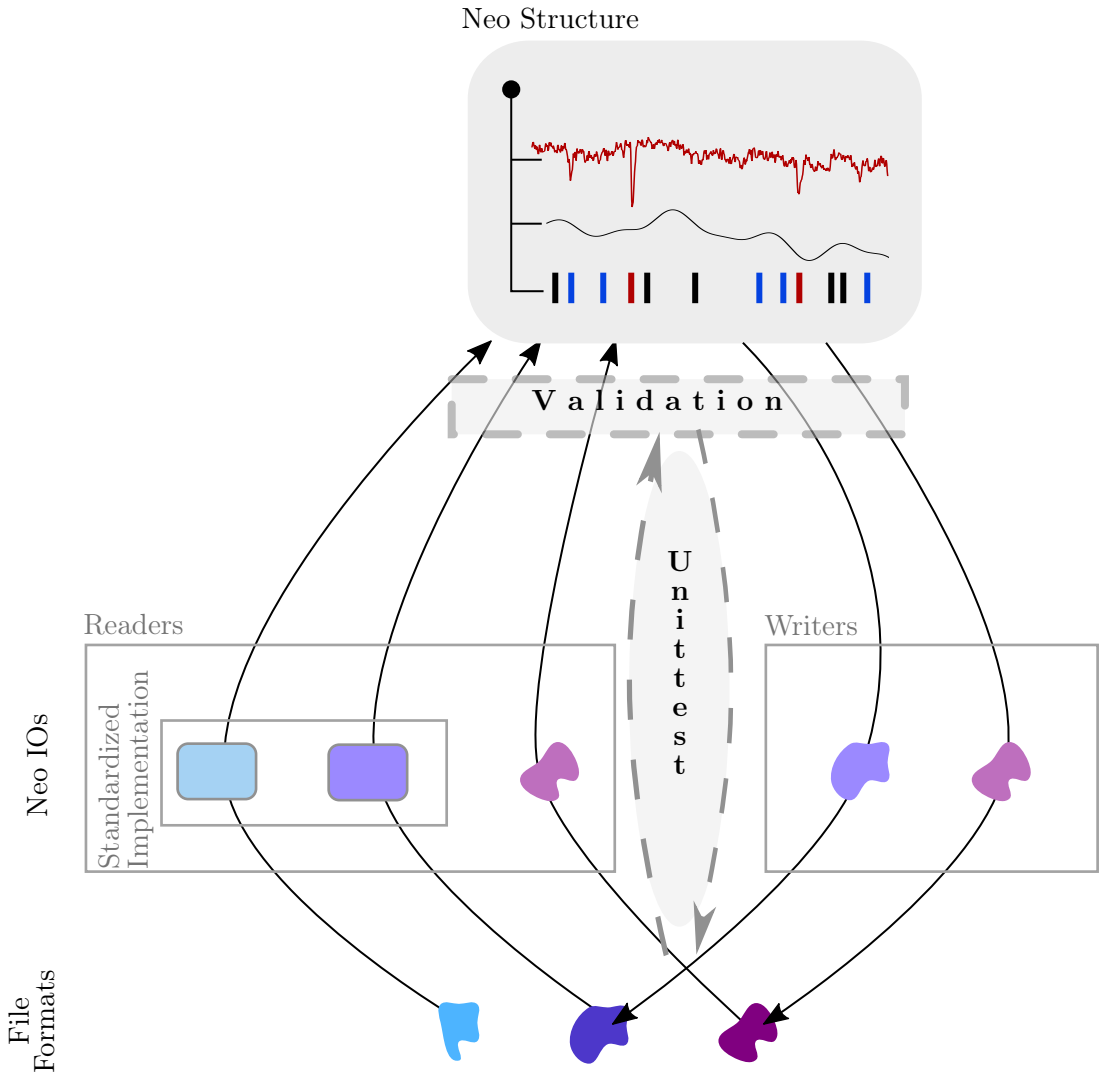


Figure 6.3: *Neo* IOs and future plans. *Neo* supports reading of multiple file formats. A large fraction of these formats is read via standardized implementations of readers whereas other rely on format-specific, custom implementations. In addition, *Neo* can write to a number of formats with custom writers. We suggest to extend the current implementation with a `validation` mechanism as well as a systematic `unittest` approach for formats that are read- and writable which ensures data consistency (dashed boxes).

96 electrodes or ii) researchers looking for a generic example datasets, e.g. to develop new workflows or test new processing and analysis methods. For these, this dataset might not be visible enough or the extensive description dissuasive, as typically much less metadata is required for these purposes. In these cases, the indexing of the dataset by a more general data catalogue will increase the discovery chances of the dataset, e.g. via the Google dataset search[7]. However, the publication of the dataset has paid off as the community now has a publicly available example of the type of data we deal with. The high number of citations[8] further demonstrates the importance and value of doing the effort to publish data. Additionally these datasets have been frequently used for teaching purposes, e.g. in tutorial Jupyter notebooks introducing *snakemake* workflows, *Neo* and or *Elephant*, e.g. at the University of Toronto[10].

6.2.6 Lessons to learn

As already hinted at in the future challenges for the Vision-for-Action workflow (Section 5.3.1) and the guidelines formulated in Section 5.4, we applied a couple of concepts from professional software development in the workflow implementation. Some concepts which can be adapted for example for scientific software development from agile software development are pair programming, continuous integration, short feedback loops and continuous deployment (Shore and Warden, 2007). We already described in detail the integration of continuous integration and deployment in the presented data and metadata workflow. The pair programming technique for joint coding involves two programmers working in a team on a single computer. This technique promotes creative approaches as well as code review during the implementation process. These features are also of advantage for scientific programming and can therefore be easily adopted. The concept of short feedback loops focuses on frequent interaction with the customer / user of the software product to get immediate feedback and quickly adapt to customer needs. In case of the presented workflows, the users would be the scientists using the packaged data and metadata for analysis providing feedback for the implemented features of the data packages. However, recognizing and implementing these concepts requires practice and organization, meaning that the potential to learn from the software development community is great, but it takes time and initiative to be implemented in a scientific environment.

6.2.7 Concept extension

On the software side, the described workflow was developed for the specific research area of neurophysiology by dealing with electrophysiological data from a monkey experiment. However, in principle, the tools presented are capable of dealing with different measurement modalities. For example the description of an electroencephalographic (EEG), a

[7] Google dataset search, https://toolbox.google.com/datasetsearch

[8] number of citations of (Brochier et al., 2018) on 23rd of August 2019 (1 year and 4.5 months after publication) according to PubMed[9]: 6

[10] https://github.com/UofTCoders/Events/issues/239 and https://github.com/UofTCoders/studyGroup/tree/gh-pages/lessons/python/snakemake_elephant_demo

functional magnetic resonance imaging (fMRI) dataset or a spiking network simulation (e.g. using Nest[11]) should be possible using similar means as presented here as the boundaries between the scientific areas are fuzzy. This will permit to easily apply the same analysis methods on datasets e.g. on spiking activity from calcium imaging data and sharp electrodes recordings and will, therefore, form a bridge between the different areas of neuroscience. On a larger scale, the development of concepts and tools for data and metadata management across scientific disciplines is an area of active development (Amari et al., 2002; Cheung et al., 2009; Nichols and Pohl, 2015). Here our efforts to provide a generic tool set located within the field of neuroscience provides a foundation for the integration with other fields of science.

6.2.8 Dissemination of a data and metadata workflow system

The implementation of the data and metadata workflow is a natural precursor to publishing the datasets, such that the research community can benefit from the invested effort. Such a workflow typically covers four aspects of data preparation (see also Fig. 5.3): preprocessing of the data, aggregation of metadata, packaging of the data and metadata and the versioning and deployment of the packages. The former requires data and domain-specific processing steps, as the preparation of the data for analysis is highly specific to the type of data. Here, the implementation of custom code or integration of tools in a scripted fashion can not be avoided. For the second step, the aggregation of metadata, the required effort highly depends on the type of source files generated during the experiment. In the ideal case of a comprehensive set of metadata source files in a standardized format, the aggregation of the metadata collection can be performed completely based on generic workflow steps (e.g. as the ones described in 5). Ideally, these workflow steps are also available publicly in the form of a collection of generic workflow utility components. Packaging the data together with the metadata again highly depends on the standardization of the formats used in previous steps. For standardized formats utility functionality for easy integration of data and metadata should be available. The versioning and deployment steps of the workflow are independent of the scientific discipline and standards used. The automatic deployment of any dataset to a commonly used data repository or hosting platform should be available as a generic component of the workflow management system. In summary, the degree of custom implementation effort for a given project highly depends on the number of standardized tools and formats used within the project. In the ideal case of a very standardized set of tools and formats, the effort for setting up a data and metadata pipeline is minimal and consists mostly of the implementation of suitable preprocessing steps for a particular type of data.

For the successful dissemination of the workflow concept based on *snakemake* as presented here in the field of neuroscience, the establishment of a public collection of snakemake utility functions is essential. These functions will provide the building blocks for setting up data and metadata workflows. By collecting generic rules based

[11] Nest, https://www.nest-simulator.org/, RRID:SCR_002963, doi:10.5281/zenodo.2605421

on tools with standardized interfaces and formats, both, the tools and the community will benefit. The community grows, usability will avail from extended standardization of data and metadata and finally, further facilitation of data publication and sharing will occur.

Extensions The concept of workflow management is not limited to the application in data and metadata management but can be applied e.g. also for data analysis and publication as well as data acquisition. Here potential extensions towards the data acquisition can include the integration of the data acquisition system in the continuous integration system and data workflow. This would permit the prompt execution of preprocessing steps upon data recording, providing immediate feedback to the experimenter about the data quality and potential issues in the recorded data. In the field of electrophysiology a project promoting open hardware and software solutions is Open Ephys[12] (Siegle et al., 2017). This project provides modular open-source hardware for tools for data acquisition and experiment control as well as accompanying software solutions for data acquisition and visualization.

For metadata acquisition besides the presented solution utilizing `csv` files in combination with *odMLtables*, also the integration of electronic labnotebooks (ELNs) is a promising extension to data and metadata workflows in experimental laboratories. This would be one solution to automatize the metadata integration into the workflow as discussed above. The advantage of ELNs is the automatic standardization of metadata within the system of the ELN. Using a workflow management system, this information can be accessed and potentially converted into other standardized metadata formats, e.g. *odML*. One potential tool to perform such a conversion is *odMLtables*, it can act as a bridge between *odML* and ELNs as many of these also support a tabular metadata representation (see Section 3.4).

We discuss the usage of *Gin* as versioning and deployment service in the presented workflow (5.3.1). Here, also other methods for providing the packaged data to collaborators and the scientific community is possible, e.g. the registration of the dataset in a central database or a custom repository. Here, for example, the *Gin* service can be installed on a local server, e.g. in case of infrastructure limitation or data privacy restrictions. This would permit to use the same workflow as presented, only the remote server location and authentication needs to be adapted.

In medical applications typically recorded data are anonymized using a standardized procedure of assigning unique ids to datasets and removing human or machine readable information that allows conclusions about the identity of the patient. This anonymization process can be easily integrated into a workflow by adding a rule for the modification of the data files which can optionally track the patient-identifier mapping in a separate file only accessible to authorized personnel. In case of different levels of authorizations, the data can be anonymized to different degrees and then copied to locations only accessible by personnel with a certain authorization level. This way the

[12]OpenEphys, http://www.open-ephys.org/

anonymization procedure would be robust to human errors and could be even automatized to a degree that no human has access to the patient-identity mapping.

6.2.9 Looking further ahead

A large part of the efforts presented in this manuscript arose from the fact, that data generated by commercial recording systems were not complete and free of unintended signals (artifacts). This lead to two types of development: the aggregation of as much information as possible concerning the recording circumstances and the development of extensive preprocessing steps including artifact detection. The question is: Is this really necessary and will it always be like this?

For the first aspect, this is diligent work requiring first of all commercial recording setup producers to take this aspect into consideration and extend and adjust their systems according to the needs of scientist to comprehensively track the data generation. For custom build setups or integrated systems as in the case of Vision-for-Action it will still be in the scope of duties of the scientist to be aware of this issue and tackle it early in the experiment development process.

The second aspect of artifacts in recording data is an issue typically improving with technical development, e.g. better insulation of cables or less error-prone communication protocols. However, no experimental setup will ever be completely free of artifacts and also a change to the latest technology will never guarantee unintended side effects on the data. Therefore careful quality checks of recording data will always be necessary, only the amount of contaminated data might reduce with technical progress.

For these reasons workflows like the one presented in Chapter 5 deal with a fundamental aspect of contemporary scientific research. Therefore, they should be considered as a key to well-founded scientific findings, and thus may be a key contributor to an expanding field of data science in general.

Appendix A

Supplementary description of the Reach-to-Grasp experiment

A Experimental apparatus

The experimental apparatus was composed by a target object, a table switch, a visual cue, and a reward system. On each recording day, the monkey was seated in a custom-made primate chair and placed in front of that apparatus. The non-working arm of the monkey was loosely restrained in a semi-flexed position. To control the home position of the working hand between the reach-to-grasp movements, the table switch which was installed close to the monkey at waist level, 5cm lateral to the mid-line, needed to be pressed down. The target object was a stainless steel rectangular cuboid (40mm x 16mm x 10mm) rotated 45 degrees around the horizontal axis and pointing towards the monkey (Fig. A.1a). It was located 13cm away from the table switch at 14cm height. The posterior end of the object was attached through a low-friction horizontal shuttle to a counterweight hidden inside the apparatus, which was used to set the object load. The object load was set to one of two possible values to define the force type (LF and HF) needed for pulling the object in each trial by deactivating and activating an electromagnetic weight resting below the counterweight inside the apparatus. When activated, it attached to the counterweight and increased overall weight from usually 100gram to 200gram, which corresponds roughly to a pulling force of 1Newton and 2Newton for LF and HF, respectively.

As already mentioned, the object was equipped with six sensors which monitored the monkey's reach-to-grasp behavior. Four force sensitive resistance sensors (FSR sensors) on the object surface provided continuous measurement of the grip forces applied on the object sides by the index and middle finger, as well as the thumb. The different activation patterns of these four FSR sensors, in particular the different placement of the thumb (see Fig. A.1 a), were used to detect online if the correct grip type was performed. An additional FSR sensor was installed between the object and its counterweight. This FSR sensor was used to measure the horizontally applied force needed to oppose the corresponding object load. Due to the low, but still existing friction of the object moving inside the horizontal shuttle, the measured force signal of this sensor is not

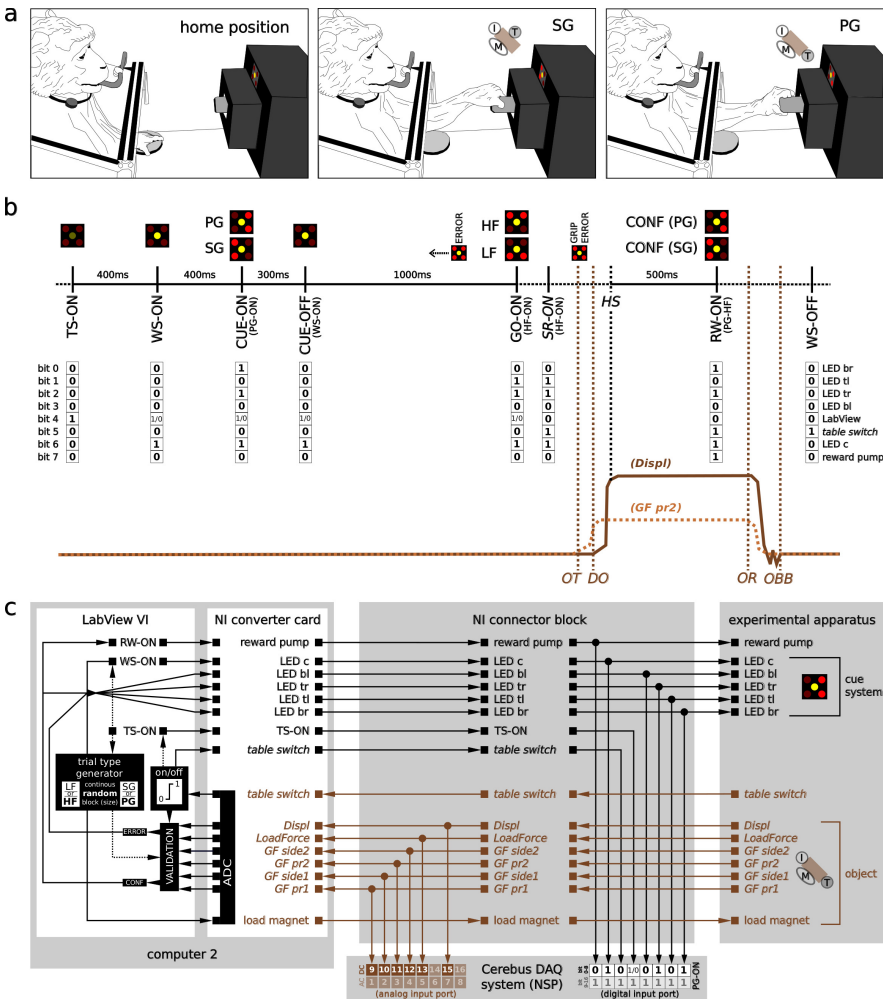


Figure A.1: Overview of the experimental apparatus and behavioral control system. (a) Sketches of the experimental apparatus and the monkey performing the reach-to-grasp task. Left: monkey in its home position with the working hand on the table switch. Middle and right: monkey grasping the object with a side grip (SG) and a precision grip (PG), respectively. Insets of middle and right sketch show the actual position of the index finger (I), the thumb (T), and the middle finger (M) on the object (brown cube) 11 . (b) Trial scheme of an example trial with the respective visual cues (different illumination combinations of the 5 LEDs illustrated on top) shown to the monkey at its respective times. The behavioral events are marked along the time axis (see main text for abbreviations). `Events` with black font mark digitally recorded events, whereas events with brown font indicate events (object touch OT, object release OR, displacement onset DO, and object back to baseline OBB) which were extracted offline from baseline deviations of the analog signals of the object's sensors. Additionally, we indicate by italic fonts events which were generated by the monkey, while all other events are produced by LabView. The 8-bit binary code for the digital event signals sent from LabView VI to the NSP at the respective times is shown below the time axis. Example traces for the analog signals of the HE sensor (Displ; dark solid line) and one of the 4 FSR sensors located at the object's surface (GF pr2, light dotted line) used to monitor the monkeys behavior and extract OT, OR, DO, and OBB are shown at the bottom. (c) Outline of the devices and their wiring controlling the behavior. All analog signal streams are colored in brown, whereas all digital signal streams are colored in black.

sensor	channel ID	label	located at	activated by	used to identify
FSR 1	137	GF pr1	object's top	index finger's touch	PG type
FSR 2	138	GF side1	object's left	middle finger's touch	SG type
FSR 3	139	GF pr2	object's bottom	thumb touch	PG type
FSR 4	140	GF side2	object's right	thumb touch	SG type
FSR 5	141	LoadForce	object's spring	object loading	pulling force
HE	143	Displ	object's shuttle	object displacement	object's position

Table A.1: Overview of the six object sensors used to monitor and control the monkey's behavior. The first four force sensitive resistance (FSR) sensors are used to monitor the applied grip type. They are located on the surface of each object side and are activated by the touch of the corresponding monkey's finger. The fifth FSR is located at the spring counterbalancing the pull resistance of the object and is used to measure the pulling force applied by the monkey. The hall-effect sensor (HE) is located along the low-friction shuttle of the object and used to measure the position of the object. The signals of all sensors are saved in the ns2 with the stated channel ID and label (cf. Fig. A.3).

perfectly proportional to the horizontal force needed to lift the opposed object load, but sufficient to distinguish between LF and HF settings (cf., example in bottom right panel of Fig. A.4 and Fig. A.6). The horizontal displacement of the object over a maximal distance of 15mm was measured by a hall-effect (HE) sensor. All sensors of the object are summarized in Table A.1. The visual cue system, composed of a square of five LEDs (size 10 x 10 mm), was located just above the target object and used to instruct the monkey about the requested behavior. While the central yellow LED was used to warn the monkey that a trial had started, the four red corner LEDs were used to code separately the grip and the force type for the requested trial type of each trial. In this context the illumination of the two left, the two right, the two bottom, or the two top LEDs coded for SG, PG, LF, or HF, respectively (see Fig. A.1 b for illustration). The reward system consisted of a bucket filled with apple sauce and equipped with a feeding tube and a pump allowing to deliver on demand the reward (few drops of the apple sauce) to the monkey (Fig. A.1 a).

A.1 Behavioral control system

The core of the behavioral control system is a custom-made Virtual Instrument (VI) in LabView that controls the digital event sequence and the requested behavior of each trial in a recording. A digital event reflects hereby the activation or deactivation of a physical device of the experimental apparatus. In this context, the LabView VI is responsible to activate and deactivate the LEDs of the visual cue system, the reward pump, and the electromagnet. The latter is not controlled by a digital event, but by an analog square signal that switches the magnet on or off. To control the requested behavior, the LabView VI monitors the monkey's manipulation of the table switch and the target object. The table switch as well as all sensors of the target object produce continuous analog signals that are digitized by the NI converter card and fed into the LabView VI of the setup computer (see Fig. A.2 computer 2). The square signal of the table switch is then online reinterpreted as digital activation or deactivation event.

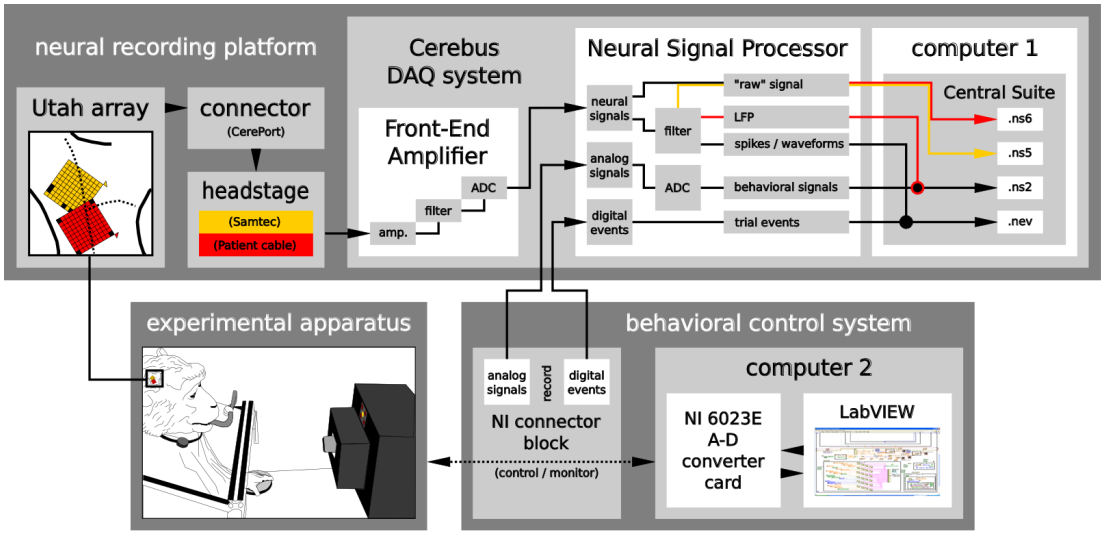


Figure A.2: Overview of the setup. The setup consisted of three main parts: the neural recording platform, the experimental apparatus, and the behavioral control system. The neural recording platform (top) was composed of the implanted Utah array with its corresponding connector (CerePort), a headstage (Samtec or Patient cable), and the Cerebus data acquisition (DAQ) system (i.e. the Front-End Amplifier, Neural Signal Processor (NSP), and the Cerebus control software, Central Suite, installed on the setup computer 1). The experimental apparatus (bottom left) consisted of the physical devices which the monkeys had to interact with (i.e., the visual cue panel (square with 5 LEDs), the target object, the table switch, and the reward system). The behavioral control system (bottom right) was built from hard- and software of National Instruments (NI, National Instruments Corporation, Austin, Texas, USA). It was composed of a NI connector block which was linked via a NI 6023E A-D converter card to setup computer 2 on which the NI system design software, LabView, was running. To record the behavioral data the behavioral control system was interlinked with the neuronal recording platform via the NSP and the NI connector block. Setup differences between the two monkeys are indicated in yellow and red for monkeys L and N, respectively.

Fig. A.1 c displays a schematic diagram on how the physical devices of the experimental apparatus are connected to the setup computer and controlled and monitored by the LabView VI. We will now describe a typical execution of the LabView VI during a recording session in more detail.

The possible trial types were set to SG-LF, SG-HF, PG-LF, and PG-HF, alternating with equal probability randomly in sequence between trials. Once the settings of the overall task were defined, the LabView VI was started to repetitively run and control the event sequence and behavior for each trial during the recording session.

Each single trial was run and controlled as follows:

The LabView VI only started a trial when the monkey deactivated the table switch by pressing and holding it down (home position, Fig. A.1 a, left). This required not much muscle activity, but simply the weight of the monkey's hand on top of the smooth-running switch. If the table switch was deactivated, the LabView VI internally initiated a trial with a short time delay (TS-ON). In parallel, the program picked randomly one of the possible trial types (e.g., SG-HF) and activated or deactivated the electromagnet

accordingly to fit the chosen load force of the object (e.g., activated for HF). To inform (or warn) the monkey that a new trial has started, the central LED was illuminated 400ms after the trial was initiated by the program (WS-ON). Four hundred ms after WS-ON the grip type was revealed to the monkey by illuminating the corresponding corner LEDs of the chosen trial type (CUE-ON, e.g., left LEDs for SG-ON). The LEDs of this first cue were turned off again after 300ms (CUE-OFF). The CUE-OFF was followed by a 1000ms preparatory delay at the end of which the monkey was informed about the upcoming force type by again illuminating the corresponding corner LEDs of the chosen trial type (GO-ON, e.g., top LEDs for HF-ON). This second cue also served as a GO signal for the monkey to initiate the movement which was registered by the activation of the table switch (SR-ON) when the monkey released it after a variable reaction time (RT). The execution of the movement was composed of reaching, grasping, pulling and holding the object in the position window for 500ms. The LabView VI controlled the movement execution online by checking the used grip type, the object displacement and the hold time. For checking the grip type, the grasp of the object was registered by small deflections of the FSR surface sensor signals caused by the monkey's fingers. A FSR sensor was registered as activated if the deflection surpassed a predefined threshold. The pattern of activated FSR sensors was then used by the LabView VI to control if the monkey performed the requested grip type. This meant, in particular, to check for SG and PG, if the FSR sensor on the right (GF side2), or on the bottom (GF pr2) of the object was activated by the monkey's thumb, respectively (see Fig. A.1 a, middle and right). The other 2 sensors that measured force from the index and middle fingers for the 2 grip types (GF side1, and GF pr1) were not controlled online. If the correct grip was detected, the grip cue was illuminated again as a positive feedback. To check the object displacement, the LabView VI measured if the deflection of the HE sensor signal of the object was within the two defined position thresholds (4 and 14mm). The time point at which the displacement signal surpassed the lower threshold was used by the LabView VI to define the estimated start of the holding period (HS) online. If the object remained within the position window for 500ms after the HS was set, LabView activated the reward pump which provided the monkey with a drop of apple sauce as reward for a successful trial. The time until the reward pump was deactivated again by LabView was proportional to the duration of the object hold in the position window, with a maximum duration and with this a maximum amount of reward for a 500ms holding period. With this mechanism, both monkeys rapidly learned to hold the object at least 500ms in nearly all trials. In parallel to the deactivation of the reward pump, LabView turned off all LEDs to indicate that the running trial ended (WS-OFF). The monkey was allowed to release the object at its own pace as soon as it received the reward. A new trial sequence was started by LabView (TS-ON) as soon as the monkey returned to the home position (new deactivation of the table switch).

An abort of the described trial sequence by LabView (error trial) was triggered by the following three scenarios: (i) the monkey released the table switch before the GO cue, (ii) the wrong grip type was registered, and (iii) the object was not pulled and held

long enough in the position window. In case one of these scenarios were registered by LabView the trial was aborted. For monkey L, the LabView VI provided additionally a negative feedback when aborting a trial by flickering all LEDs three times.

As displayed in Fig. A.1 c. the behavioral control system was connected to the NSP of the Cerebus DAQ system to store the trial event sequence and the monkey's behavior of each trial in a recording along with the neural data registered by the neural recording platform. For this, the analog signals of the sensors of the target object were copied from the NI connector block to the analog input port of the Cerebus System NSP via DC coupled BNC cables and connectors. In the NSP they were digitized with a 16-bit resolution at 0.15 mV/bit and a sampling rate of 1kHz and saved in the ns2 file under the channel ids listed in Table A.1. All digital or digitized events that register the activation and deactivation of the table switch, the LEDs of the cue system, and the reward pump, as well as the internally generated digital trial start event (TS-ON) were coded as a 8-bit binary signal (see Table 2.2) and transferred via the NI connector block to a 16-bit DB-37 input port of the NSP where they occupy the first 8 digits (remaining digits are set to 1). In the NSP the now 16-bit binary signal of each event was stored in its decimal representation and with its corresponding time point in the nev file (see Table 2.2 and Fig. A.3).

A.2 Neural recording platform

The recording of the neural signals was performed using a neural recording platform with components produced by Blackrock Microsystems (Salt Lake City, UT, USA, www.blackrockmicro.com). The platform consisted of the multi-electrode Utah array, a headstage, and a Cerebus data acquisition (DAQ) system. The latter is composed of a Front-End Amplifier, a real-time Neural Signal Processor (NSP) and the control software, Central Suite (version 4.15.0 and 6.03.01 for L and N, respectively), running on Windows XP for L, and Windows 7 for N on the setup computer 1 (see Fig. Fig. A.3). The Cerebus DAQ system was also connected to the behavioral control system via the NI connector block to save the analog behavioral data and digital trial event signals that were described in the previous section in parallel with the neural signals. All data were transmitted from the NSP via an ethernet cable to be saved first locally on the setup computer 1. After a recording day, all recordings were transferred to a data server. In the following, we will describe the function of the different components of the neural recording platform in more detail.

The implant location of the Utah array, as well as the electrode configuration of the array of each monkey was described previously (see Fig. 2.2). The electrode identification numbers (IDs) are determined by how the electrodes of the array are wired and connected to the Cerebus Front-End Amplifier. See Appendix A.3 for details.

The analog Blackrock headstage with unity gain (Samtec for monkey L, and Patient Cable for monkey N) was used to reduce the environmental noise. Overall, the reduction of the noise was better with the Patient Cable than with the Samtec headstage.

In the Front-End Amplifier, each of the 96 neural signals was differentially amplified

with respect to the reference input of its corresponding connector bank (gain 5000) and filtered with a 1st-order 0.3Hz high pass filter (full-bandwidth mode) and a 3rd-order 7.5kHz Butterworth low pass filter. After that, the band-pass filtered neuronal signals were digitized with a 16-bit resolution at 0.25V/bit and a sampling rate of 30kHz, in the following called "raw signal". The digitized signals were converted into a single multiplexed optical output and transmitted via a fiber-optic data link to the NSP. In the NSP the raw signals were saved in a ns5-file for monkey L and in a ns6-file for monkey N. The file format depended on the firmware and software version of the Cerebus DAQ system. In addition to the neural signals, the NSP received the analog behavioral signal recorded by the behavioral control system via the analog input port. These behavioral signals were digitized and saved with a sampling rate of 1kHz in a ns2-file. For monkey N, the ns2-file also contained a filtered and downsampled version of the raw signals, in the following called "LFP data". To extract the LFP data, a copy of the raw data was online digitally low-pass filtered at 250Hz (Butterworth, 4th order), and downsampled to 1kHz within the NSP.

The NSP performed also an online spike waveform detection and classification controlled via the Central Suite software. The sorted spikes were used for a first online inspection of the data as well as for selecting and saving the spike waveforms for offline sorting. For this purpose the neuronal raw signals were for monkey L online high-pass filtered at 250 Hz (Butterworth, 4th order) and for monkey N band-pass filtered between 250Hz and 5kHz (Butterworth, 2nd order). Afterwards, the waveforms were detected by threshold crossing (manually set). These waveforms were then sorted by requesting the signal from identified neurons to follow through up to five hoops set by the user (all individually for each channel). To get an overview of the quality of the data during the recordings, the sorted waveforms were displayed in the online classification window provided by Central Suite.

The thresholds (one for each channel) for the spike waveform detection were not modified during a session and were saved in the nev-file for each session along with all other settings (e.g. filter setting etc) and configurations of Central Suite. The data and corresponding settings of Central Suite can also be inspected offline using the Blackrock software CentralPlay even in the absence of the Blackrock hardware system. Each time the high-pass filtered signal passed the threshold, a snippet of 1.6ms (48 samples) for monkey L and 1.3ms (38 samples) for monkey N was cut and saved as potential spike waveform. The snippet was cut with 10 sample points before threshold crossing and 38 or 28 points after for monkey L or N, respectively. Waveforms identified as potential single units (online sorted spikes) were labeled with IDs from 1 to 16. Unsorted waveforms were labeled with ID 0. These potential spike waveforms were saved together with their respective time stamps in the nev-file. Due to the high number of electrodes, online spike-sorting was moderately reliable. We therefore decided to re-sort spiking activity offline on each channel using the Plexon Offline Spike Sorter (Plexon Inc, Dallas, Texas, USA, version 3.3, for details see Appendix B). Results of offline sorting were saved in a copy of the original nev-file with an updated file name.

All data files (nev, ns5/6, ccf) were saved on disk and backed-up on a data server at the end of the recording sessions. The information collected here are partly taken from (Riehle et al., 2013; Zehl et al., 2016).

A.3 Origin of the channel IDs

The neuronal signal inputs to the Front-End Amplifier were grouped into four banks (A-D or 0-3) from which only the first 3 were used. Each bank consists of a male header with 34 pins of which 32 were the neuronal signal input channels. The other two channels served as reference and ground, respectively. In Central Suite, the identification (ID) number of each electrode of the array is defined by the position on the input bank and pin of the Cerebus Front-End Amplifier. For this Central Suite multiplies the bank ID (0, 1, 2, or 3) with the number of pins for neural signal input channels (32) and adds the ID of the pin the electrode is connected to (cf. ID conversion in Fig. A.3). The electrode wiring of the Utah array is, though, not coordinated to the input banks of the Front-End Amplifer which leads to spatially unordered electrode IDs. Nevertheless, Utah arrays are fabricated usually in the same way where the corner electrodes are unconnected leading to a default (unordered) electrode ID configuration (cf. electrode configuration of monkey N in Fig. 2.2). If in the fabrication process one of the corner electrodes was registered to be of significantly higher quality than any other electrodes of the grid, the corner electrode was connected instead and thereby changed the corresponding electrode configuration (cf. electrode configuration of monkey L in Fig. 2.2). This led to the different ID sequences of the arrays for monkey L and N (see Fig. 2.2). To facilitate the comparison of results between arrays with different electrode configurations, we assigned new IDs that reflect the spatial organization of the array. For this we used as reference the lower left corner electrode, when the connected wire bundle is showing to the right. These fabrication-independent, connector-aligned IDs increase linearly from bottom left to top right, line by line. They are also shown in Fig. 2.2 d as gray numbers in the array sketch, which thereby provides the mapping of the Blackrock IDs to the connector-aligned IDs.

A visual summary of the available data is given in Fig. A.4 and Fig. A.5 for monkey L, and Fig. A.6 and Fig. A.7 for monkey N. The first of these figures shows the sequence of trials as well as selected raw recorded time series, spike trains, unit wave forms, and behavioral signals for one particular trial. The second of these figures contrasts parallel neuronal data across channels in a specific trial with neuronal data across trials in a specific channel.

B Data preprocessing

After the recordings, a number of preprocessing steps (pre in the sense of before the actual upcoming data analysis, but being the post-processing after the recording) were performed as described below. This includes (i) the translation of the digital events from their binary codes set by the DAQ system to a human-readable format putting the

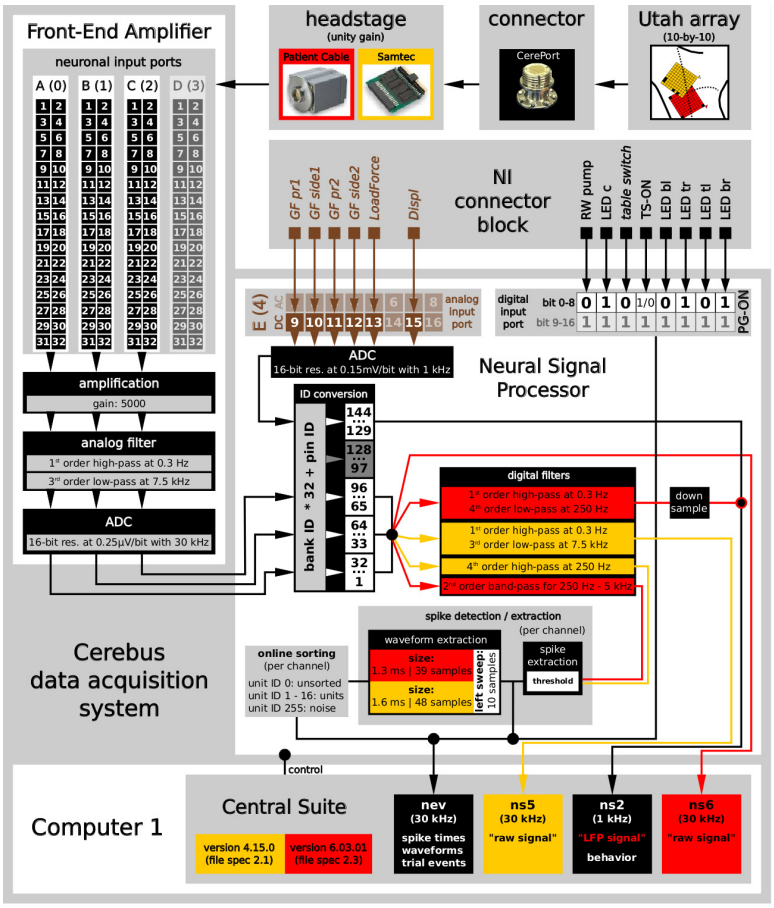


Figure A.3: Sketch of the components related to the recording of the neuronal signals. Data were recorded using a Utah array, which was linked via its connector (CerePort) to a headstage (Samtec or Patient Cable) with a unity gain. From there the neural signals were transferred to the Cerebus Front-End Amplifier, where they were amplified, filtered and digitized. The digitized signals were converted into a single multiplex optical output and sent via a fiber-optic data link to the Neural Signal Processor (NSP), which is controlled by the Cerebus control software (Central Suite). Within the NSP the time points and waveforms of potential spikes were extracted online from a correspondingly processed copy of the neural signals and saved in the nev file. Simultaneously, the continuous raw signals (sampled at 30 kHz) were saved in the ns5 (for monkey L) or ns6 file (for monkey N). In parallel to the neural signals the NSP received also the digital trial events produced by the LabView VI, and the analog signals of the object's sensors via the NI connector block of the behavioral control system. While the digital trial events were saved along with the extracted potential spikes in the nev file, the analog signals of the sensors were digitized and saved in the ns2 file. For monkey N, a filtered and downsampled version of the neural signals (0.3–250 Hz at 1 kHz) was also saved in the ns2 file. Components and settings specific to monkey L and N are indicated by yellow and red, respectively.

events in context of the expected executed trial event sequence, (ii) the offline detection of behavioral trial events and object load force from the analog signals recorded by the sensors of the target object, and (iii) the offline spike sorting.

B.1 Translation of digital events to trial events

Table 2.2 lists the 8-bit combinations that were sent by LabView to the Experimental Apparatus to control the behavior. Following a binary to decimal conversion, they were saved as event codes (Table 2.2) during the experiment along with their time stamps in the .nev file. In the first preprocessing step, these event codes were translated to a human-readable format and put into context of an expected trial event sequence. The validation against the latter was used to identify incomplete, correct and error trials. Error trials were further differentiated into error types (e.g., grip error). This digital event translation and interpretation (cf. Table 2.2) performed automatically within the reach-to-grasp loading routine.

Translation table of the 8 bits to the event codes and their behavioral meaning (labels). The 8 bits (see Table 2.2 for their meaning) were sent from LabView to NSP during the trial sequence (Fig. Fig. A.1). The event codes are the decimal version of the bit sequence assuming another byte with all bits set to 1 in front. The event codes are found in the .nev files with a time stamp and indicate the occurrence of a stimulus / behavioral event as indicated in the center column ('label'). Due to different versions of the LabView control program for monkey L and N (see text for details) the event codes for the same label may be different for the two monkeys. Also some event codes do not have a concrete meaning (miscellaneous) and occur sporadically in the .nev file due to a mistake in the sampling of the digital events - they have to be ignored. In the table the event codes are sorted in sequential order from top to bottom with respect to the task, i.e. their order corresponds to the sequence found in the .nev file in an successful trial.

B.2 Preprocessing of behavioral analog signals

Some behavioral events such as the monkey touching the object or the onset of the object displacement by the monkey were controlled during the experiment, but their online-detected timing was approximate and not saved (see details in section Appendix A.1). However, these events can be relevant for data analysis and they were thus computed offline from the analog signals of the four FSR sensors measuring the monkey's grip and the HE sensor measuring the object displacement. We implemented a custom-made Matlab `Event`-Detection toolbox to detect 8 specific events: the precise timing of object touch (OT) and object release (OR) from the force traces as well as the timing of displacement onset (DO) and object back to baseline (OBB) from the displacement trace, and finally the onset and offset of the plateau phase in the force and displacement traces. The plateau phase of the displacement signal indicates the timing and stability of the holding period, and its onset is used to calculate offline the hold start (HS) signal. The toolbox performed an automatic detection of these events and their timing was first ap-

proximated by threshold crossing and then fine-tuned by back-comparison of the traces with baseline level from the point of threshold crossing. Since the automatic detection was prone to errors, the trials were visually inspected one by one and the timing of the automatically detected events were manually corrected if they did not match the event times as visually identified. In addition, a Matlab script was used to inspect the load force traces in each trial to control if the actual object load corresponded to the programmed object load. This procedure ensured that the electro-magnet controlling the object load was properly activated throughout the recording session.

B.3 Offline spike sorting

The spike waveforms which were extracted and saved (in the nev file) during the recording were offline sorted using the Plexon Offline Sorter (version 3.3.3). To keep the variability in the half-manual spike sorting at a minimum, all sortings were performed by the same person (A. Riehle). The spike sorting started with loading the complete nev file of a session into the Plexon Offline Sorter. The spike sorting was performed on a duplicate of the data file to keep the original data intact. We started by joining all different waveforms extracted online from each channel separately back again into one pool and initially marked as "unsorted waveforms" in the Plexon Offline Sorter. Thereby, we ignored the result of the preliminary online waveform sorting (units 0-16 in the nev file) that was performed during the recording via Central Suite software, which served solely to extract waveforms and gain an overview of the quality of the spiking activity. For the invalidation of cross-channel artifacts (e.g., chewing artifacts) all waveforms that occurred simultaneously on a defined percentage of channels (70%) were marked as "invalidated waveforms" in Plexon Offline Sorter. Such artifacts occurred only in the recording session of monkey L. Furthermore, a waveform rejection was performed. Thereby all waveforms of abnormally large amplitude and/or atypical shape on a channel were manually marked as "invalidated waveforms" in Plexon Offline Sorter.

The actual spike sorting was then performed on the remaining unsorted waveforms (i.e., those not marked as invalidated waveforms) individually for each channel. We used different algorithms to split these waveforms into clusters in a 2- or 3-dimensional principal component (PC) space. The dimensionality of the PC space was chosen according to the best separation. The main algorithms used were K-Means(-Scan) and Valley Seeking (chosen according to the best separation). We used a fixed threshold for outliers (a parameter to be determined in the Plexon Offline Sorter) between 1.8 (K-Means) and 2 (Valley Seeking) to get comparable sorting results. The spikes of the sorted clusters were then controlled using the inter-spike interval (ISI) distributions and the auto- and cross-correlation plots. Units were ordered manually from best to worst (assigning increasing unit IDs 1-16 in the Plexon Offline Sorter) by considering the amplitude of the waveform (the higher the better), the outcomes of the ISI analysis (no or low number of spikes with an ISI smaller than 2 ms), the correlation histograms, and identifiable cluster shapes. Waveforms in the cluster with the highest unit ID (worst) on

monkey	sorting ID	# SUA	# MUA	# electrodes with SUA	# electrodes with SUA or MUA
L	*-02	93	49	65	86
N	*-03	156	19	78	89

Table A.2: Overview of offline sorted single and multi unit activity (SUA and MUA). For the recording of monkey L it was possible to sort out 93 SUAs and 28 MUAs distributed over 65 of the 96 electrodes of the Utah array, with 21 additional electrodes with further MUA recordings. For the recording of monkey N it was possible to sort out 156 SUAs and 8 MUAs distributed over 78 of the 96 electrodes of the Utah array, with 11 additional electrodes with further MUA recordings. For details on the offline spike sorting see Appendix B.3.

a given channel may contain multi-unit activity. Clusters with unacceptable outcomes (completely or partly overlapping waveforms), including those with only a few spikes, left assigned as "unsorted waveforms" in Plexon Offline Sorter. This offline spike sorted nev file was saved under the file name of the original nev file with an added two-digit numeric postfix (e.g. -01). In this file, unit ID 255 contains invalidated waveforms, unit ID 0 contains the unsorted waveforms (that may enter a further cluster analysis for spike sorting), and unit IDs 1-16 contain the time stamps and waveforms of the sorted single- or multi-units (as in the Plexon Offline Sorter). Unit IDs that are considered to represent multi-unit activity are documented in the metadata. The nev file with the sorted units can be loaded again into the Plexon Offline Sorter to visualize all the sorted spikes and rework the spike sorting.

B.4 Code availability

All available code required to access the data as described in Appendix E is stored along with the datasets. The provided code includes, in particular: (i) a snapshot of the Python *Neo* package (see also Chapter 4), (ii) a snapshot of the Python *odML* package (see also Section 1.1.1), (iii) the custom-written ReachGraspIO extending the *Neo* package, (iv) the example script shown and described in Appendix E, (v) the code shown and described in Appendix E demonstrating how to access the data in *Matlab*.

In addition to these frozen versions of the code, we recommend to use updated versions of the code to benefit from future enhancements, bug fixes and increased compatibility with future Python releases or novel applications that rely on recent versions of *Neo* and/or *odML*. Complete link collections to the two libraries can be found online[1],[2]. Importantly, both projects are hosted and version-controlled via Github[3],[4].

[1] *Neo*, http://neuralensemble.org/neo/
[2] *odML*, andhttp://www.g-node.org/projects/odml
[3] *Neo*, https://github.com/NeuralEnsemble/python-neo
[4] *odML*, https://github.com/G-Node/python-odml

C Data records

All data and metadata are publicly available via the data portal of the German Neuroinformatics Node (G-Node) of the International Neuroinformatics Coordination Facility (INCF), called GNData[5]. Table 2.1 provides an overview of the name, size, and content of all files for each published dataset of monkey L and N. The datasets of both monkeys consist of four parts: (i) the primary data are provided as the original data files obtained from the Central Suite software stored in the data format specified by the manufacturer (in particular, nev, ns5 and ns6 format) of the neural recording platform, Blackrock Microsystems; (ii) an offline sorted version of the neural spike data (cf. Appendix B.3) is provided in a second nev file; (iii) metadata are provided as one file per dataset in the *odML* format (Grewe, Wachtler, and Benda, 2011; Zehl et al., 2016); and (iv) a mat file is provided containing the continuous neural raw data together with the offline sorted spike data, both annotated with the corresponding metadata.

Overview of recording days of the published datasets. For both monkeys, we chose to publish the first dataset (rec*-001) of the recording day. For details on the published datasets see Table 2.1.

The dataset l101210-001 from monkey L is the first out of 9 recording sessions conducted on Friday, December 10, 2010, while the dataset i140703-001 from monkey N is the first out of only 3 recording sessions conducted on Thursday, July 3, 2014. Both datasets were recorded in the late morning. The following recording day went on for nearly one hour and a half for monkey L, and one hour for monkey N. Although the recording from monkey N lasted with 16:43 min several minutes longer than the recording from monkey L with only 11:49 min, monkey L executed 204 trials, while monkey N only performed 160 trials in total. However, monkey L performed only 70% of all trials correctly, whereas monkey N successfully completed 90% of all trials during the recording (cf. Table 2.1). Nonetheless, the high percentage of error trials in monkey L are mainly caused by an too early movement onsets reflecting the eagerness, but also the nervousness of the monkey L's character. In contrast to these error types, monkey L used only 12 times the wrong grip compared to monkey N who performed an incorrect grip type 16 times during the recording.

Overview of trials performed during the published datasets. Of the stated number of error trials, the monkey L and N used the wrong grip type in 12 and 16 trials, respectively. In the remaining error trials the monkeys initiated the movement too early. Trial types were altered randomly in the recordings which led to slightly different trial numbers for the different trial types.

For both monkeys the trial types alternated randomly between trials leading to slightly different numbers of trials with the same trial type in the each dataset (cf. Table 2.1).

The quality of the spiking activity in the datasets of both monkeys was high, which allowed us to perform a relatively robust offline spike sorting with high numbers of single unit activity (SUA) distributed over all electrodes of the array (for details see

[5]GNData, http://g-node.github.io/g-node-portal/

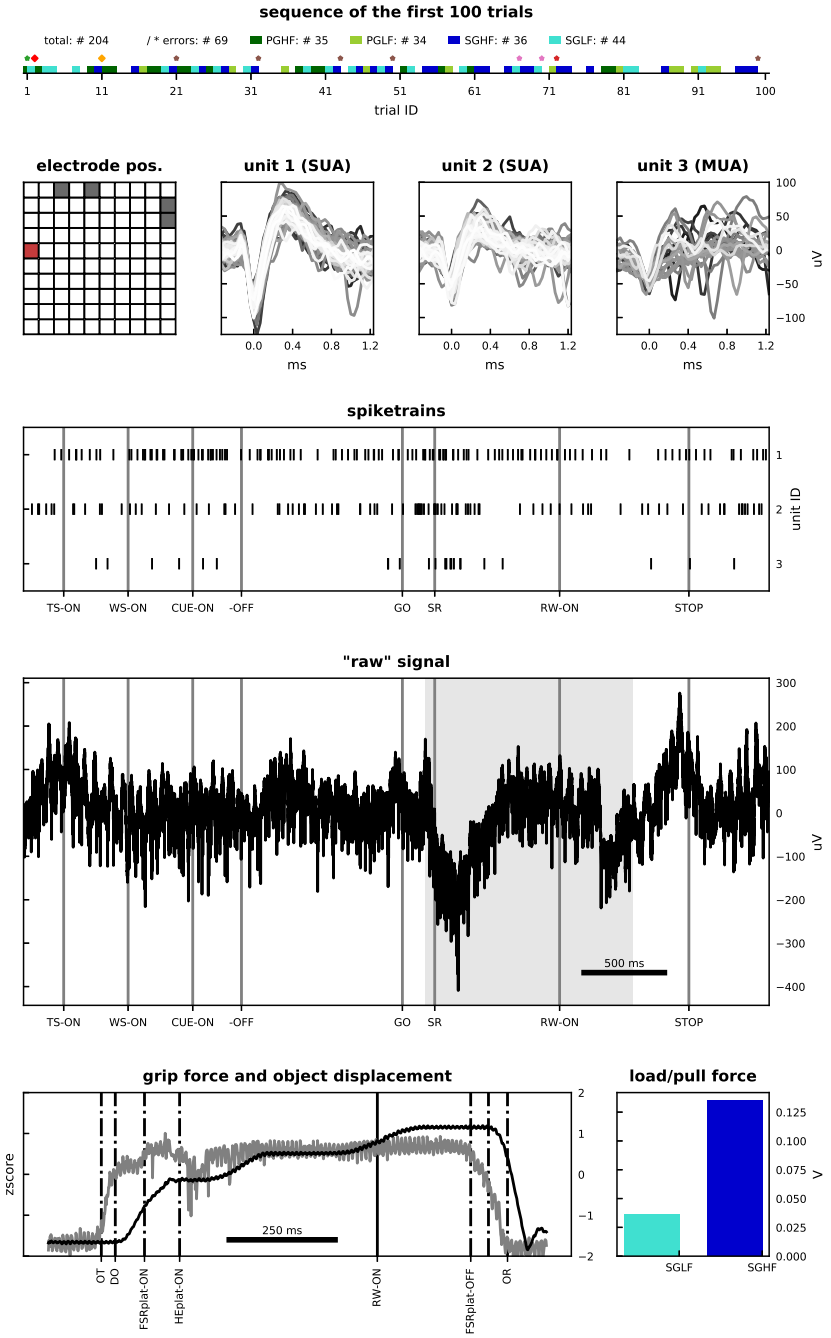


Figure A.4: Overview of data types contained in l101210-001. The figure displays the different data types contained in the selected dataset of monkey L. Top panel: sequence of the first 100 trials (for trial types and errors see color in legend) and the total number of trials (see # for correct, error, trial types in legend); the red diamond marks the selected trial (trial ID: 2) for panels below; the orange diamond marks an additional trial selected to demonstrate load/pull force differences between the averaged load force signals in the bottom right panel. Asterisks indicate error trials (black asterisks: grip errors). Second row, left panel: position of selected electrode (in red) for the data plots (electrode ID: 71). Second row, remaining panels: waveforms of three units from the selected electrode. Third row: spike trains of displayed units for the selected trial. Forth row: raw signal for the selected trial; gray shaded area marks the time window corresponding to the bottom left panel. Bottom left panel: grip force (gray) and object displacement (black) signals for the selected trial. Bottom right panel: averaged load/pull force signals for the duration of the plateau of the grip force signal for the selected LF and HF trial. Important trial events are indicated as vertical lines in the corresponding data plots.

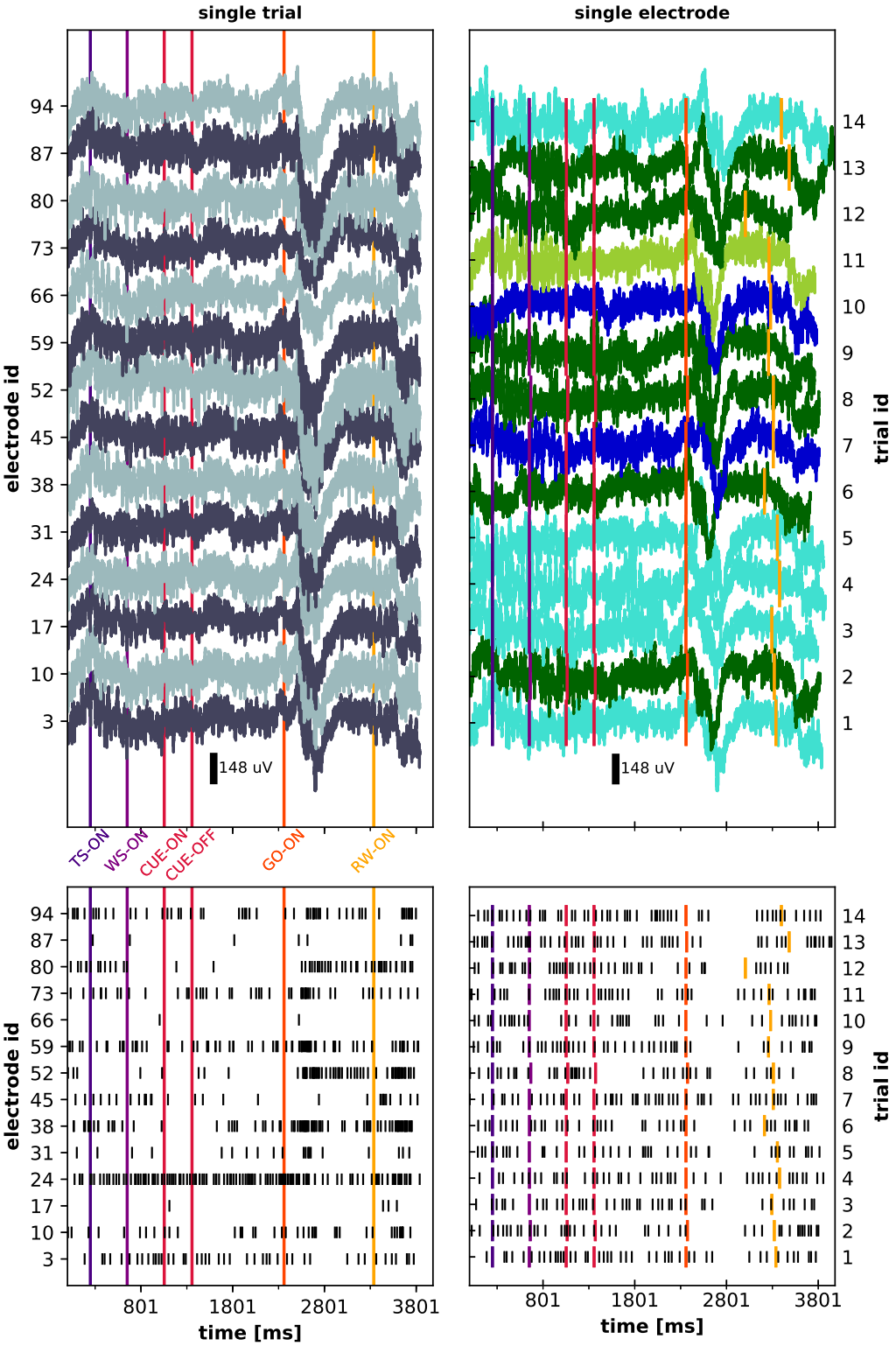


Figure A.5: Overview of raw signal and spike data of monkey L (l101210-001). Left panels: Raw signal (top) and spike data of unit IDs 1 on each given electrode (bottom) for a single trial (trial ID: 2) across a selection of electrodes. Right panels: Raw signal (top) and spike data from single unit ID 1 (bottom) across selected correctly performed trials on one electrode (electrode ID: 3). Trial events (TS-ON, WS-ON, CUE-ON, CUE-OFF, GO-ON, and RW-ON) are indicated as colored vertical lines in each plot. Trial types of selected trials in upper right panels are indicated as color (SGHF: dark blue; SGLF: cyan; PGHF: dark green; PGLF: light green)

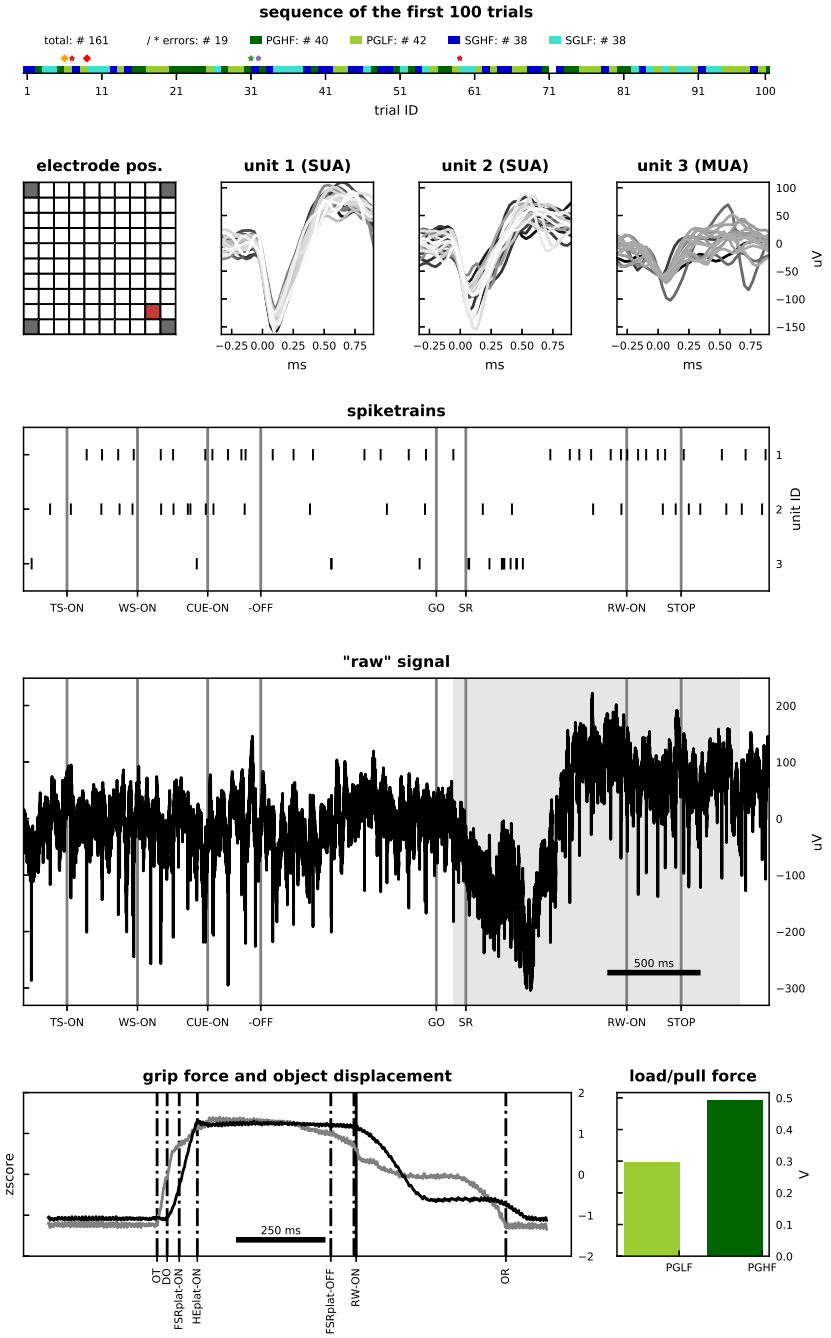


Figure A.6: Overview of data types contained in i140703-001. The figure displays the different data types contained in the selected dataset of monkey N. Top panel: sequence of the first 100 trials (for trial types and errors see color in legend) and the total number of trials (see # for correct, error, trial types in legend); the red diamond marks the selected trial (trial ID: 9) for panels below; the orange diamond marks an additional trial selected to demonstrate load/pull force differences between the averaged load force signals in the bottom right panel. Asterisks indicate error trials (black asterisks: grip errors). Second row, left panel: position of selected electrode (in red) for the data plots (electrode ID: 63). Second row, remaining panels: waveforms of three units from the selected electrode. Third row: spike trains of displayed units for the selected trial. Forth row panel: raw signal for the selected trial; gray shaded area marks the time window corresponding to the bottom left panel. Bottom left panel: grip force (gray) and object displacement (black) signals for the selected trial. Bottom right panel: averaged load/pull force signals for the duration of the plateau of the grip force signal for the selected LF and HF trial. Important trial events are indicated as vertical lines in the corresponding data plots.

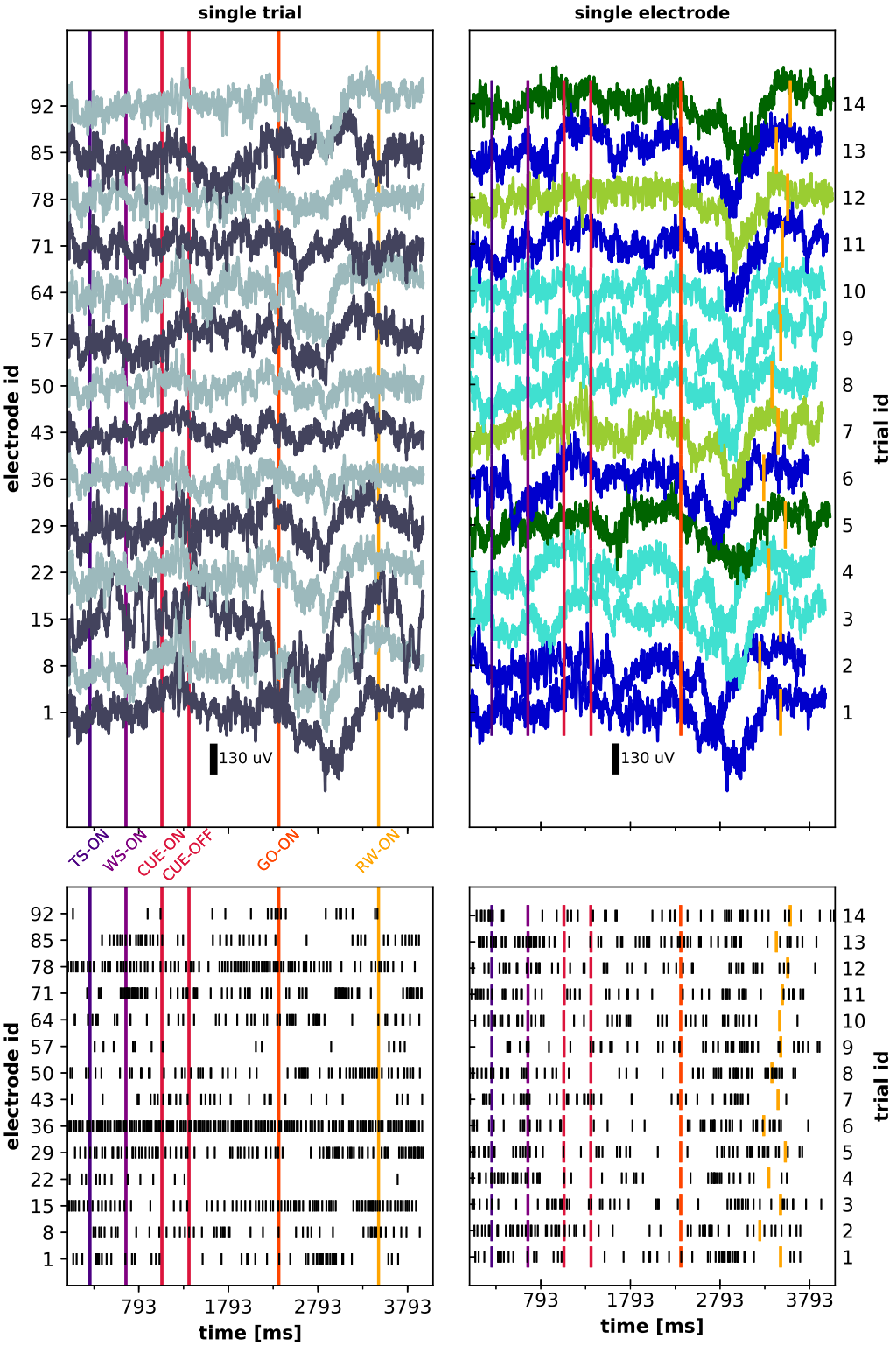


Figure A.7: Overview of LFP and spike data of monkey N (i140703-001). Left panels: LFP data (top) and spike data of unit IDs 1 on each given electrode (bottom) for a single trial (trial ID: 1) across a selection of electrodes. Right panels: LFP data (top) and spike data from single unit ID 1 (bottom) across selected correctly performed trials on one electrode (electrode ID: 1). Trial events (TS-ON, WS-ON,CUE-ON, CUE-OFF, GO-ON, and RW-ON) are indicated as colored vertical lines in each plot. Trial types of selected trials in upper right panels are indicated as color (SGHF: dark blue; SGLF: cyan; PGHF: dark green; PGLF: light green).

Table A.2). For details on how the offline sorting was performed and checked please have a look at Appendix B.3 and Appendix D.4.

D Technical validation

In addition to the above described preprocessing steps that needed to be performed to gain more content of the raw data, some technical validations of the data also had to be conducted. These technical validations include the correction of the irregular alignment data files of the Cerebus DAQ system and a general quality assessment of the data. In order to validate the quality of the recording, a series of algorithms were applied to the data. On the one hand the quality of the LFP signals was assessed per electrode and per trial by evaluating the variance of the corresponding signal in multiple frequency bands. On the other hand the quality of the offline sorted single units (Appendix B.3) was determined by a signal-to-noise measure. In addition, noise artifacts occurring simultaneously in the recorded spiking activity were detected and marked. In the following, we explain these technical validation steps in detail.

D.1 Correction of data alignment

The ns6 file starts always 82 samples later than ns5, ns2 and nev files. This miss-alignment is caused by an error in the Blackrock recording software. However, this shift is correctly recorded in the ns6 file, and therefore will be automatically corrected in the generic *Neo* loading routine (cf., BlackrockIO in Appendix E below). In addition, due to the online filter procedure, the LFP signals in the ns2 file are delayed by approximately 3.6 ms with respect to the time stamps in the nev file and the analog signal of the ns6 file. This offset was heuristically determined, documented in the metadata file, and can be automatically corrected for by the experiment-specific loading routine (cf., ReachGraspIO in Appendix E below). Note that the time stamps of the spike times provided in the nev file correspond to start of the waveform and not to the time point of threshold crossing.

D.2 Quality assessment

The occurrence of noise in electrophysiological recordings is to a certain degree unavoidable and therefore needs to be carefully examined. It depends to a large extent on the quality of the headstage used to record the neurophysiological data. In our data, two different types of headstages were used for the two monkeys - the Samtec-CerePort headstage (monkey L) and the Patient Cable (monkey N). The former is much more sensitive to noise than the latter. The type of noise, its cause and appearance in the data is quite variable. Depending on the direct influence of the different types of noise on subsequent analysis methods, one needs to balance the corresponding data rejection between being very permissive and very conservative. For this reason, it is wise not remove or delete data of bad quality, but instead mark them with the judgment of a corresponding quality assessment procedure. For the here published datasets, we

provide the results of our quality assessment of the electrodes, trials and spiking units along with the analysis parameters of the used procedure in the *odML* metadata files for each recording. The reach-to-grasp IO integrates this information by annotating the corresponding data objects in Neo. This approach not only allows the user to finally decide which data to reject for an analysis, but also provides the opportunity to provide different quality assessments of the same electrode, trial and unit at the same time. This is helpful if one considers that certain types of noise can differently contaminate signals in different frequency bands. For the here published datasets, the quality of the recorded signals was therefore separately tested for the sorted spike data and different frequency bands of the LFP data. The used corresponding procedures are described in detail below.

D.3 LFP data quality

The LFP data were examined for noise in three broad frequency bands excluding the 50Hz European line noise (low: 3Hz - 10Hz, middle: 12Hz - 40Hz, high: 60Hz - 250Hz) in each session individually. The goal of the quality assessment was, first, to detect channels with a noisy signal throughout the session and, second, to detect noisy trials in the remaining "clean" channels. To do so, the analog signals of each electrode were first z-scored and filtered in the three frequency bands (low, middle, and high) using a Butterworth filter (of order 2, 3, and 4, respectively). For each frequency band the quality assessment analysis was carried out separately. The detection of noisy electrodes was performed in three steps:

step 1 The variance of the filtered analog signal of each electrode was calculated over the complete session.

step 2 Out of the 96 resulting variance values, outliers were identified as those values outside a user-defined range. The range was defined as follows: (i) values between a lower (e.g., 25th) and an upper (e.g., 75th) percentile (L and U), (ii) the range of acceptable values was defined by $L - w \cdot (U - L), U + w \cdot (U - L)$,where w is a user-defined whisker coefficient (e.g., w=3).

step 3 The analog signals classified as outliers in step 2 were visually controlled by comparing them to the analog signal of an electrode with a typical variance value. If the results were either too conservative or too permissive, the detection procedure was repeated by manually adapting the chosen parameters (L, U, and w), correspondingly.

The electrode IDs of the final outliers as well as the parameters chosen for their detection were saved in the *odML* metadata file of the corresponding recording and marked as noisy for the tested frequency band.

For the remaining non-noisy electrodes, an analogous procedure was carried out afterwards to detect noisy trials. The procedure differed in one respect: the variance of the filtered analog signal was calculated for each trial on each electrode separately. At the end, the trial IDs of the identified outliers were pooled and marked as noisy for the tested frequency band on all electrodes. The marked trial IDs were saved in the *odML* metadata file of the corresponding recording together with the chosen analysis

parameters for their detection. Note again that with this procedure a trial is marked as noisy on all electrodes as soon as it is classified as noisy on one electrode.

D.4 Spike data quality

To test and judge the quality of the spike data, the results of the offline spike sorting were controlled first, for the signal-to-noise ratio (SNR) from the waveforms of the identified single units and second, for the occurrence of hyper-synchronous event artifacts.

1. To calculate the SNR for each identified unit in the sorting results a method introduced by Hatsopoulos, Joshi, and O'Leary (2004) was used. The SNR was defined as the amplitude (A, trough-to-peak) of the mean waveform ($< w >$) divided by twice the standard deviation of the waveform noise (SD_{noise}) of the defined unit (u): $SNR_u = A_{<w>}/SD_{noise} \cdot 2$,where SD_{noise} was computed by averaging the standard deviations (SDs) obtained from each sample point across the original waveforms (SD of the waveform noise adapted from Nordhausen, Maynard, and Normann (1996) and Suner et al. (2005). For all identified single units in the datasets published here, the determined SNRs ranged between 1.5 and 12. Corresponding to Suner et al. (2005) the quality of the spike sorting of an identified unit is good if the SNR is above 4, is fair if the SNR ranges between 2 and 4, and is poor if the SNR ranges between 1 and 2. Units with an SNR below 1 are not considered as signals. For a conservative analysis of the spike datasets, we recommend to use only single units with a SNR of 2.5 or higher, which was our choice in e.g. Torre, Canova, et al. (2016). The results of the SNR analysis of the performed spike sorting were saved in the *odML* metadata file of the corresponding recording and units were annotated accordingly.

2. Since correlation analysis of spike data is very sensitive to cross-electrode artifacts which would produce unwanted false positive results, we controlled the sorted spike data on their original time resolution ($\delta = 1/30ms$) for potential occurrence of hyper-synchronous event artifacts. For this, we computed the population histogram, i.e. the sum of the spikes across all sorted single units in the dataset in bins of $\delta = 1/30ms$ (sampling resolution of the data), and detected if there were entries ≥ 2. To our surprise these hyper-synchronous spikes, which are likely to be attributed to cross-channel noise, survived the spike sorting including the cross-channel artifact removal by the Plexon Spike sorter. We indeed detected these spike artifacts during a preliminary analysis of a previous study (Torre, Canova, et al., 2016). The number of single units participating in these events ranged from 2 to over 30 and a statistical analysis showed that the frequency of their occurrence largely exceeded the expected value considering the observed population firing rate. Furthermore, a δ-binned time histogram of the population spiking activity triggered around the occurrence times of the hyper-synchronous events revealed also increased spiking activity in the preceding or following bin of the event. For a conservative analysis of the spike datasets, we recommend to treat the spikes participating in a hyper-synchronous event as well as the spikes occurring within a short time interval around this event ($\pm\delta$) as artifacts of unknown origin and to remove them subsequently before performing any analysis of the spike data.

In Torre, Canova, et al. (2016) we combined both quality assessments of the spike data and only considered spikes with a SNR>2.5 and additionally removed all hyper-synchronous events with ≥ 2 spikes.

E Usage notes

In the following, we describe how the provided data files can be practically used in a data analysis scenario. To this end, we first briefly present the open source software libraries we recommend to use in order to access data and metadata using the Python programming language. We also demonstrate how to merge data and metadata in a common representation that facilitates data handling and analysis. Finally, we present an example program that produces a visualization of the most important data items contained in the files, and can be used as a template script for accessing the provided data. All software discussed below is provided in the code subfolder of the provided datasets, and links to the code repositories are listed in Appendix B.4.

As outlined above, the datasets are stored in two types of files. The primary data, and the spike sorted data, are provided in the data format (in particular, the nev, ns5 and ns6 format) specified by Blackrock Microsystems, the manufacturer of the recording hardware. Second, metadata are provided as one file in the *odML* format (Grewe, Wachtler, and Benda, 2011). While data and metadata are provided in documented file formats (see Blackrock[6] and *odML*[7], respectively), the mere knowledge of the highly complex internal structure of the files is insufficient to practically make use of their content. In particular, implementations of corresponding loading routines performed from scratch by individual researchers are likely to be incoherent and error-prone. Thus, in the following we will use two community supported open-source libraries to separately load primary data and metadata into a generic, well-defined data representation.

We chose the data object model provided by the open-source *Neo* library (Chapter 4) (Garcia, Guarino, et al., 2014) as the primary representation of the datasets (Chapter 4). *Neo* provides a hierarchical data structure composed of Python objects that aim to represent electrophysiological data in a generic manner. In addition, *Neo* provides a number I/Os that enable the user to read from (and in part, write to) a large number of open and vendor-specific file formats. In particular, *Neo* provides an I/O module for the file format used by Blackrock Microsystems (class BlackrockIO in file neo.io.blackrockio.py). The output of this I/O is a *Neo* data structure that is a faithful representation of the contents of the primary data files. For detailed information on the structure of the *Neo* data object model, please consult the online documentation[8].

Here, we briefly summarize the output of the reach-to-grasp datasets obtained when calling the I/O. The `read_block` method of an instantiation of the BlackrockIO returns a *Neo* Block object as a top level grouping object representing one recording session. In the hierarchy directly below the Block is one single `Segment` object spanning the complete

[6] Blackrock, http://blackrockmicro.com/
[7] http://www.g-node.org/projects/odml
[8] Neo, http://neo.readthedocs.io/en/latest/index.html

continuous recording, and one `ChannelIndex` object for each of the 96 electrodes of the Utah Array (Fig. 2.2) and each of the 6 sensor signals monitoring the target object manipulation (Appendix A). The data from these 102 recording channels is each saved in one `AnalogSignal` object. All of these are linked to the `Segment` and the respective `ChannelIndex` object. Likewise, the spike times (and optionally, the spike waveforms) of each identified unit are saved to a `SpikeTrain` object. As for the `AnalogSignal` objects, these are linked to the `Segment`, and to the `ChannelIndex` object via a Unit object. Finally, all digital events are saved into a single `Event` object that lists their time of occurrences and the corresponding event IDs. Additional information from the file is provided as annotations on each individual *Neo* object (accessible via the annotation property of the object), in particular as annotations to the top level Block object. Note, that although this generic I/O can be used to access the raw data records, no interpretation of the file contents is given. For example, digital events are not interpreted as behavioral events, but only given as the raw numeric codes shown in Fig. A.2.

In order to access the metadata stored in the *odML* file, we use the corresponding library API python-*odML* described in Grewe, Wachtler, and Benda (2011). In short, *odML* files store metadata in form of hierarchically structured key-value pairs. The *odML* files accompanying the provided datasets contain extensive metadata grouped into different sections describing different aspects of the experiment. A tutorial on how to work with the *odML* library can be found in the online documentation shipped with the library, and a more detailed description of how to manage metadata by example of the *odML* framework can be found in Zehl et al. (2016). In short, the library supports to read the content of an *odML* file, provides an API to navigate through the hierarchical structure, and to extract metadata of interest from the key-value pairs. Thus, the python-*odML* library provides a standardized way to access stored metadata records.

As a next step, we combine the primary data and metadata in a manner that is specific to this experiment and aids the analysis process. To this end, the relevant metadata that were extracted from the *odML* are attached as annotations to data objects in the hierarchical *Neo* structure. For example, metadata information for a particular single unit originating from the spike sorting process may be attached to the *Neo* objects representing the sorted spike data of that unit. The task of combining the primary data and metadata is performed by a custom-written Python class named ReachGraspIO that is derived as child class from Neo's BlackrockIO class. For a full documentation of the input arguments, methods, and outputs of this class, please refer to the class documentation in reachgraspio.py. In short, invoking the `read_block` method of the ReachGraspIO performs the following steps under the hood: (i) read the primary data using the `read_block` method of the parent class (BlackrockIO) as described above, (ii) read the metadata using the python-*odML* library, (iii) interpret event data based on the digital events (e.g., detect trial start or reward), and (iv) add relevant metadata to the *Neo* data object using the annotation mechanism. Thus, the *Neo* Block object returned by the ReachGraspIO contains extensive information attached as annotations of the individual *Neo* objects, in particular, about whether a `SpikeTrain` is classified

as SUA or MUA, about the spatial positioning of electrodes, or about the identities of electrodes that should be rejected. A full list of these metadata annotations can be found in the documentation of the `read_block` method in the file reachgraspio.py.

In summary, for practical purposes, the resulting data structure of the ReachGraspIO hosts a complete representation of the data and a synthesis of the metadata relevant for analysis. This representation may be saved to disk in a standardized container format (e.g., .mat or HDF5), such that the exact same data and metadata context can also be accessed from other programming languages. For illustration, we provide the data object in the Matlab file format (.mat) in the folder datasets_matlab, containing Matlab structs resembling the Python *Neo* objects.

In the following we demonstrate how to use the ReachGraspIO in practice in order to load and visualize the datasets. We follow the file example.py, which is contained as part of the code included with the published datasets. The goal of this program is to create a figure showing the raw signal, LFP, spikes (time stamps and waveforms), and events in a time window (referred to as analysis epoch) around TS-ON of trial 1 for electrode ID 62.

In a first step, we load the data using the ReachGraspIO. Considering that only for monkey N an online filtered version of the LFP data is available in the ns2 file, in the following we calculate offline an LFP signal from all raw signals contained in the ns5 or ns6 files using a non-causal low-pass Butterworth filter implemented in the Electrophysiology Analysis Toolkit (Elephant[9], which provides analysis capabilities for data stored in the *Neo* representation. The parameters of this filter are chosen identical to those of the causal filter for the LFP recorded online in monkey N (Appendix A.2).

In a subsequent step, we extract all TS-ON events in correctly performed trials. To this end, we use the function get_events() contained in the utility module neo_utils.py. The function extracts a list of events contained in one `Event` object of the loaded *Neo* `Block` given the filter criteria specified by the parameter event_prop. In our example, the used filter criteria select all events from the `Event` object "TrialEvents" with a trial_event_labels annotation set to TS-ON, and a performance_in_trial annotation indicating a correct trial.

In a next step, we create `Epoch` objects representing analysis epochs around the extracted TS-ON events. To this end, we use add_epochs() also contained in the utility module neo_utils.py. The function excepts the previously extracted TS-ON events as trigger, and defines epochs of a given duration around this trigger. The resulting `Epoch` object is called "analysis_epochs".

Next, we cut the data according to the analysis epochs and align the cutouts in time. This operation is performed by cut_segment_by_epoch, which returns a list of `Segment` objects, each containing data of one analysis epoch. The `Segments` are annotated by the corresponding annotations of the *Neo* `Epoch`. In addition, the list of `Segment` objects is grouped in a new *Neo* `Block`, named "data_cut_to_analysis_epochs". This representation now enables the analysis of the data across trials in the defined analysis

[9] Elephant, http://neuralensemble.org/elephant/

epochs.

In our example, we show how to create a plot of the data of the analysis epoch in one behavioral trial on the selected electrode. To select the *Neo* `Segment` corresponding to the first correct behavioral trial from the `Block` of the cut data obtained in the previous step, we apply the *Neo* filter() function.

From the selected `Segment`, LFP data and raw signals can be obtained via the `AnalogSignal` objects referenced by the analogsignals attribute, while spike trains and corresponding unit waveforms can be extracted from the `SpikeTrain` objects referenced by the spiketrains attribute. The remainder of example.py uses the matplotlib library to create a figure of the data.

All data and metadata files as well as the code described above can be found in the data repository at GIN[10]. The subdirectory datasets contains all data files and the metadata odML-file for the two provided recording sessions. The subdirectory code contains the files example.py and neo_utils.py. For further reference and inspiration this subdirectory also contains the Python scripts generating the data figures of this manuscript. Furthermore, the subdirectories to code contain frozen versions of the required libraries (*Neo*, *odML*) as well as the custom loading routine combining data and metadata (reachgraspio.py). Finally, the datasets_matlab directory contains the annotated *Neo* data object containing all primary data saved in the mat-file format.

[10]GIN, https://web.gin.g-node.org/INT/multielectrode_grasp

Acknowledgements

My thanks goes to Prof. Dr Sonja Grün for providing an open and creative environment permitting the development of this thesis.

My deepest gratitude goes to Dr. Michael Denker for sharing his excitement and infinite motivation to pursue the rather infrastructural topics covered in this thesis.

Also I would like to thank Dr. Lyuba Zehl for frontiering these topics and laying the foundations for me to build on.

In general I would thank all the lab members of the INM-6 who became friends during my extended time at the INM-6. Thanks for all the fun, boring, tasty, silly, secret, creative and cycling time you shared with me. This specially includes the openness to provide refuge for me being stuck in Jülich or Düren.

I also would like to thank all collaborative partners of the INM-6 I had the chance to work with. This especially includes the ComCo group in Marseille, the G-Node in Munich and the *Neo* collaboration.

My thanks also goes to my current flatmates and friends with whom I also shared very cheerful, tasty, sportive and playful times, although unfortunately I was busy way too often.

I would also like to thank Sam, Kila and their companions, who countless times risked their lifes for my distraction.

Last but not least I would also like to thank my family for their persistent nosiness as well as their support in mental and physical ways. Vielen Dank für den Apfelbrei, Oma!

Financially the work presented in this thesis was partly supported by Helmholtz Portfolio "Supercomputing and Modeling for the Human Brain" (SMHB), EU Grants 604102, 720270 and 785907 (Human Brain Project, HBP), Priority Program SPP 1665 of the DFG (GR1753/4-2 and DE 2175/2-1), Collaborative Research Agreement RIKEN-CNRS, ANR GRASP, CNRS (PEPS, Neuro_IC2010), DAAD, LIA Vision for Action, HDS-LEE: Helmholtz School for Data Science in Life, Earth and Energy (Jülich, Aachen, Köln) and BMBF grants 01GQ1302 and 01GQ1509.

Bibliography

Amari, Shun-Ichi et al. (Dec. 2002). "Neuroinformatics: the integration of shared databases and tools towards integrative neuroscience". eng. In: *J. Integr. Neurosci.* 1.2, pp. 117–128. ISSN: 0219-6352.

Anderson, Christopher J. et al. (Mar. 2016). "Response to Comment on "Estimating the reproducibility of psychological science"". eng. In: *Science* 351.6277, p. 1037. ISSN: 1095-9203. DOI: 10.1126/science.aad9163.

Ascoli, Giorgio A. et al. (2017). "Win-win data sharing in neuroscience". eng. In: *Nat. Methods* 14.2. Citation Key Alias: Ascoli_2017a, pp. 112–116. ISSN: 1548-7105. DOI: 10.1038/nmeth.4152.

Askren, Mary K. et al. (2016). "Using Make for Reproducible and Parallel Neuroimaging Workflow and Quality-Assurance". English. In: *Front. Neuroinform.* 10. ISSN: 1662-5196. DOI: 10.3389/fninf.2016.00002. URL: https://www.frontiersin.org/articles/10.3389/fninf.2016.00002/full#B21 (visited on 08/27/2019).

Assante, Massimiliano et al. (Apr. 2016). "Are Scientific Data Repositories Coping with Research Data Publishing?" eng. In: *Data Science Journal* 15.0, p. 6. ISSN: 1683-1470. DOI: 10.5334/dsj-2016-006. URL: http://datascience.codata.org/articles/10.5334/dsj-2016-006/ (visited on 08/02/2019).

Baker, Monya (May 2016). "1,500 scientists lift the lid on reproducibility". en. In: *Nature News* 533.7604, p. 452. DOI: 10.1038/533452a. URL: http://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970 (visited on 08/02/2019).

Bitzenhofer, Sebastian H. et al. (Feb. 2017). "Layer-specific optogenetic activation of pyramidal neurons causes beta–gamma entrainment of neonatal networks". en. In: *Nature Communications* 8, p. 14563. ISSN: 2041-1723. DOI: 10.1038/ncomms14563. URL: http://www.nature.com/ncomms/2017/170220/ncomms14563/full/ncomms14563.html (visited on 02/24/2017).

Brochier, Thomas et al. (Apr. 2018). "Massively parallel recordings in macaque motor cortex during an instructed delayed reach-to-grasp task". en. In: *Scientific Data* 5, p. 180055. ISSN: 2052-4463. DOI: 10.1038/sdata.2018.55. URL: https://www.nature.com/articles/sdata201855 (visited on 04/16/2018).

Brun, Rene and Fons Rademakers (Sept. 1996). "ROOT - An Object Oriented Data Analysis Framework". In: *Nucl. Inst. & Meth. in Phys. Res. A* 389, pp. 81–86. URL: http://root.cern.ch/.

Burkholder, Tanya et al. (June 2012). "Health Evaluation of Experimental Laboratory Mice". In: *Curr Protoc Mouse Biol* 2, pp. 145–165. ISSN: 2161-2617. DOI: 10.1002/

9780470942390.mo110217. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3399545/ (visited on 02/08/2019).

Candela, Leonardo et al. (2015). "Data journals: A survey". en. In: *Journal of the Association for Information Science and Technology* 66.9, pp. 1747–1762. ISSN: 2330-1643. DOI: 10.1002/asi.23358. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.23358 (visited on 08/02/2019).

Chen, Xiaoli et al. (Feb. 2019). "Open is not enough". En. In: *Nature Physics* 15.2, p. 113. ISSN: 1745-2481. DOI: 10.1038/s41567-018-0342-2. URL: https://www.nature.com/articles/s41567-018-0342-2 (visited on 08/02/2019).

Cheung, Kei-Hoi et al. (July 2009). "Approaches to neuroscience data integration". eng. In: *Brief. Bioinformatics* 10.4, pp. 345–353. ISSN: 1477-4054. DOI: 10.1093/bib/bbp029.

Coles, Simon, Leslie Carr, and Jeremy Frey (Apr. 2008). "Experiences with repositories and blogs in laboratories". en. In: URL: https://eprints.soton.ac.uk/50901/ (visited on 11/14/2018).

Corlan, Alexandru Dan (2004). *Medline trend: automated yearly statistics of PubMed results for any query*. en. URL: http://dan.corlan.net/medline-trend.html (visited on 08/24/2019).

Dale, Darren (2019). *Quantities Documenting*. URL: https://python-quantities.readthedocs.io/en/latest/devel/documenting.html (visited on 07/09/2019).

Deisseroth, Karl and Mark J. Schnitzer (2013). "Engineering Approaches to Illuminating Brain Structure and Dynamics." In: *Neuron* 80.3, pp. 568–577. DOI: 10.1016/j.neuron.2013.10.032.

Denker, Michael and Sonja Grün (2016). "Designing Workflows for the Reproducible Analysis of Electrophysiological Data". en. In: *Brain-Inspired Computing*. Ed. by Katrin Amunts et al. Lecture Notes in Computer Science. Springer International Publishing, pp. 58–72. ISBN: 978-3-319-50862-7.

Denker, Michael, Sébastien Roux, et al. (Dec. 2011). "The Local Field Potential Reflects Surplus Spike Synchrony". en. In: *Cereb. Cortex* 21.12, pp. 2681–2695. ISSN: 1047-3211, 1460-2199. DOI: 10.1093/cercor/bhr040. URL: http://cercor.oxfordjournals.org/content/21/12/2681 (visited on 08/22/2014).

Denker, Michael, Lyuba Zehl, et al. (Mar. 2018). "LFP beta amplitude is linked to mesoscopic spatio-temporal phase patterns". en. In: *Scientific Reports* 8.1, pp. 1–21. ISSN: 2045-2322. DOI: 10.1038/s41598-018-22990-7. URL: https://www.nature.com/articles/s41598-018-22990-7 (visited on 08/05/2019).

Drummond, Chris (2009). "Replicability is not reproducibility: Nor is it good science". In: *In Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*.

Einevoll, Gaute T. et al. (Nov. 2013). "Modelling and analysis of local field potentials for studying the function of cortical circuits". en. In: *Nature Reviews Neuroscience* 14.11, pp. 770–785. ISSN: 1471-0048. DOI: 10.1038/nrn3599. URL: https://www.nature.com/articles/nrn3599 (visited on 08/05/2019).

Eisner, D.A. (Jan. 2018). "Reproducibility of science: Fraud, impact factors and carelessness". In: *J Mol Cell Cardiol* 114, pp. 364–368. ISSN: 0022-2828. DOI: 10.1016/j.yjmcc.2017.10.009. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6565841/ (visited on 08/03/2019).

Ferguson, Adam R. et al. (Oct. 2014). "Big data from small data: data-sharing in the 'long tail' of neuroscience". en. In: *Nature Neuroscience* 17, pp. 1442–1447. ISSN: 1546-1726. DOI: 10.1038/nn.3838. URL: https://www.nature.com/articles/nn.3838 (visited on 08/27/2019).

Fidler, Fiona et al. (Mar. 2017). "Metaresearch for Evaluating Reproducibility in Ecology and Evolution". In: *Bioscience* 67.3, pp. 282–289. ISSN: 0006-3568. DOI: 10.1093/biosci/biw159. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5384162/ (visited on 08/03/2019).

Foltz, Charmaine J and Mollie Ullman-Cullere (1999). "Guidelines for Assessing the Health and Condition of Mice". en. In: *Lab Animal* 28.4, p. 5.

Garcia, Samuel and Nicolas Fourcaud-Trocmé (2009). "OpenElectrophy: An Electrophysiological Data- and Analysis-Sharing Framework". eng. In: *Front Neuroinform* 3, p. 14. ISSN: 1662-5196. DOI: 10.3389/neuro.11.014.2009.

Garcia, Samuel, Domenico Guarino, et al. (2014). "Neo: an object model for handling electrophysiology data in multiple formats". English. In: *Front. Neuroinform.* 8. ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00010. URL: https://www.frontiersin.org/articles/10.3389/fninf.2014.00010/full (visited on 04/24/2019).

Geisler, Wilson S. (2008). "Visual Perception and the Statistical Properties of Natural Scenes". In: *Annu. Rev. Psychol.* 59.1, pp. 167–192. ISSN: 1545-2085. DOI: 10.1146/annurev.psych.58.110405.085632.

Goodman, Steven and Sander Greenland (Apr. 2007). "Why most published research findings are false: problems in the analysis". eng. In: *PLoS Med.* 4.4, e168. ISSN: 1549-1676. DOI: 10.1371/journal.pmed.0040168.

Gorgolewski, Krzysztof J. et al. (June 2016). "The brain imaging data structure, a format for organizing and describing outputs of neuroimaging experiments". eng. In: *Sci Data* 3, p. 160044. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.44.

Grewe, Jan, Thomas Wachtler, and Jan Benda (2011). "A Bottom-up Approach to Data Annotation in Neurophysiology". English. In: *Front. Neuroinform.* 5. ISSN: 1662-5196. DOI: 10.3389/fninf.2011.00016. URL: https://www.frontiersin.org/articles/10.3389/fninf.2011.00016/full (visited on 03/15/2019).

Haan, Marcel Jan de (2018). "Cortical network dynamics during visually-guided motor behavior : Setup development and Preliminary analyses". en. PhD thesis. Dissertation, RWTH Aachen University, 2018. DOI: 10.18154/RWTH-2018-221368. URL: https://publications.rwth-aachen.de/record/717863 (visited on 08/18/2019).

Haan, Marcel Jan de et al. (May 2018). "Real-time visuomotor behavior and electrophysiology recording setup for use with humans and monkeys". In: *Journal of Neurophysiology* 120.2, pp. 539–552. ISSN: 0022-3077. DOI: 10.1152/jn.00262.2017.

URL: https://www.physiology.org/doi/full/10.1152/jn.00262.2017 (visited on 08/18/2019).

Hatsopoulos, Nicholas, Jignesh Joshi, and John G. O'Leary (Aug. 2004). "Decoding Continuous and Discrete Motor Behaviors Using Motor and Premotor Cortical Ensembles". In: *Journal of Neurophysiology* 92.2, pp. 1165–1174. ISSN: 0022-3077. DOI: 10.1152/jn.01245.2003. URL: https://www.physiology.org/doi/full/10.1152/jn.01245.2003 (visited on 08/06/2019).

Hazan, Lynn, Michaël Zugaro, and György Buzsáki (Sept. 2006). "Klusters, NeuroScope, NDManager: A free software suite for neurophysiological data processing and visualization". In: *Journal of Neuroscience Methods* 155.2, pp. 207–216. ISSN: 0165-0270. DOI: 10.1016/j.jneumeth.2006.01.017. URL: http://www.sciencedirect.com/science/article/pii/S0165027006000410 (visited on 07/29/2019).

Hunter, John D. (2007). "Matplotlib: A 2D Graphics Environment". In: *Comput. Sci. Eng.* 9.3, pp. 90–95. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55. URL: http://ieeexplore.ieee.org/document/4160265/ (visited on 07/09/2019).

Ioannidis, John P. A. (Aug. 2005). "Why most published research findings are false". eng. In: *PLoS Med.* 2.8, e124. ISSN: 1549-1676. DOI: 10.1371/journal.pmed.0020124.

— (June 2007). "Why most published research findings are false: author's reply to Goodman and Greenland". eng. In: *PLoS Med.* 4.6, e215. ISSN: 1549-1676. DOI: 10.1371/journal.pmed.0040215.

Jacob, Vincent et al. (May 2010). "The Matrix: A New Tool for Probing the Whisker-to-Barrel System with Natural Stimuli". en. In: *Journal of Neuroscience Methods* 189.1, pp. 65–74. ISSN: 01650270. DOI: 10.1016/j.jneumeth.2010.03.020. URL: https://linkinghub.elsevier.com/retrieve/pii/S0165027010001548 (visited on 02/28/2019).

Jomhari, Nur Zulaiha, Achim Geiser, and Afiq Aizuddin Bin Anuar (2017). *Higgs-to-four-lepton analysis example using 2011-2012 data.* DOI: 10.7483/opendata.cms.jkb8.rr42. URL: http://opendata.cern.ch/record/5500 (visited on 08/03/2019).

Jun, James J. et al. (Nov. 2017). "Fully integrated silicon probes for high-density recording of neural activity". en. In: *Nature* 551.7679, pp. 232–236. ISSN: 1476-4687. DOI: 10.1038/nature24636. URL: https://www.nature.com/articles/nature24636 (visited on 08/05/2019).

Kanza, Samantha et al. (Dec. 2017). "Electronic Lab Notebooks: Can They Replace Paper?" en. In: *Journal of Cheminformatics* 9.1. ISSN: 1758-2946. DOI: 10.1186/s13321-017-0221-3. URL: http://jcheminf.springeropen.com/articles/10.1186/s13321-017-0221-3 (visited on 12/18/2018).

Kelly, Ryan C. et al. (Jan. 2007). "Comparison of recordings from microelectrode arrays and single electrodes in visual cortex". In: *J Neurosci* 27.2, pp. 261–264. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.4906-06.2007. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3039847/ (visited on 07/19/2019).

Köster, Johannes and Sven Rahmann (Oct. 2012). "Snakemake—a scalable bioinformatics workflow engine". en. In: *Bioinformatics* 28.19, pp. 2520–2522. ISSN: 1367-

4803. DOI: 10.1093/bioinformatics/bts480. URL: https://academic.oup.com/bioinformatics/article/28/19/2520/290322 (visited on 08/29/2019).

Kwok, Roberta and S Kanza (2018). "Lab Notebooks Go Digital". en. In: *Nature* 560, p. 269.

Lefebvre, Baptiste, Pierre Yger, and Olivier Marre (2016). "Recent progress in multi-electrode spike sorting methods". eng. In: *J. Physiol. Paris* 110.4 Pt A, pp. 327–335. ISSN: 1769-7115. DOI: 10.1016/j.jphysparis.2017.02.005.

Logothetis, Nikos K. and Brian A. Wandell (2004). "Interpreting the BOLD Signal". In: *Annual Review of Physiology* 66.1, pp. 735–769. DOI: 10.1146/annurev.physiol.66.082602.092845. URL: https://doi.org/10.1146/annurev.physiol.66.082602.092845 (visited on 08/05/2019).

Maldonado, P. et al. (2008). "Synchronization of Neuronal Responses in Primary Visual Cortex of Monkeys Viewing Natural Images". In: *Journal of Neurophysiology* 100.3, pp. 1523–1532. ISSN: 1522-1598. DOI: 10.1152/jn.00076.2008.

Martin, Robert C. (Aug. 2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. English. 1 edition. Upper Saddle River, NJ: Prentice Hall. ISBN: 978-0-13-235088-4.

Mitzdorf, U. (Jan. 1985). "Current source-density method and application in cat cerebral cortex: investigation of evoked potentials and EEG phenomena". In: *Physiological Reviews* 65.1, pp. 37–100. ISSN: 0031-9333. DOI: 10.1152/physrev.1985.65.1.37. URL: https://www.physiology.org/doi/abs/10.1152/physrev.1985.65.1.37 (visited on 08/05/2019).

Miyamoto, Daisuke and Masanori Murayama (2015). "The Fiber-Optic Imaging and Manipulation of Neural Activity during Animal Behavior". In: *Neuroscience Research*. ISSN: 0168-0102. DOI: 10.1016/j.neures.2015.09.004.

Mouček, Roman et al. (2014). "Software and hardware infrastructure for research in electrophysiology". English. In: *Front. Neuroinform.* 8. ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00020. URL: https://www.frontiersin.org/articles/10.3389/fninf.2014.00020/full (visited on 01/23/2019).

Nichols, B. Nolan and Kilian M. Pohl (Sept. 2015). "Neuroinformatics Software Applications Supporting Electronic Data Capture, Management, and Sharing for the Neuroimaging Community". eng. In: *Neuropsychol Rev* 25.3, pp. 356–368. ISSN: 1573-6660. DOI: 10.1007/s11065-015-9293-x.

Nicolelis, Miguel A L. and Sidarta Ribeiro (2002). "Multielectrode Recordings: The next Steps." In: *Curr Opin Neurobiol* 12.5, pp. 602–606. DOI: 10.1016/S0959-4388(02)00374-4.

Nordhausen, Craig T., Edwin M. Maynard, and Richard A. Normann (July 1996). "Single unit recording capabilities of a 100 microelectrode array". In: *Brain Research* 726.1, pp. 129–140. ISSN: 0006-8993. DOI: 10.1016/0006-8993(96)00321-6. URL: http://www.sciencedirect.com/science/article/pii/0006899396003216 (visited on 08/06/2019).

Obien, Marie Engelene J. et al. (2014). "Revealing neuronal function through microelectrode array recordings." In: *Front Neurosci* 8, p. 423. DOI: 10.3389/fnins.2014.00423. URL: http://dx.doi.org/10.3389/fnins.2014.00423.

Ohl, F. W., H. Scheich, and W. J. Freeman (Aug. 2001). "Change in pattern of ongoing cortical activity with auditory category learning". en. In: *Nature* 412.6848, pp. 733–736. ISSN: 1476-4687. DOI: 10.1038/35089076. URL: https://www.nature.com/articles/35089076 (visited on 03/15/2019).

Okonechnikov, Konstantin, Olga Golosova, and Mikhail Fursov (Apr. 2012). "Unipro UGENE: a unified bioinformatics toolkit". en. In: *Bioinformatics* 28.8, pp. 1166–1167. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts091. URL: https://academic.oup.com/bioinformatics/article/28/8/1166/195474 (visited on 08/16/2019).

Open Science Collaboration (Aug. 2015). "PSYCHOLOGY. Estimating the reproducibility of psychological science". eng. In: *Science* 349.6251, aac4716. ISSN: 1095-9203. DOI: 10.1126/science.aac4716.

Palm, Christoph et al. (2010). "Towards ultra-high resolution fibre tract mapping of the human brain - registration of polarised light images and reorientation of fibre vectors". English. In: *Front. Hum. Neurosci.* 4. ISSN: 1662-5161. DOI: 10.3389/neuro.09.009.2010. URL: https://www.frontiersin.org/articles/10.3389/neuro.09.009.2010/full (visited on 08/18/2019).

Parekh, Ruchi, Rubén Armañanzas, and Giorgio A. Ascoli (Apr. 2015). "The importance of metadata to assess information content in digital reconstructions of neuronal morphology". eng. In: *Cell Tissue Res.* 360.1, pp. 121–127. ISSN: 1432-0878. DOI: 10.1007/s00441-014-2103-6.

Pashler, Harold and Eric–Jan Wagenmakers (Nov. 2012). "Editors' Introduction to the Special Section on Replicability in Psychological Science: A Crisis of Confidence?" en. In: *Perspect Psychol Sci* 7.6, pp. 528–530. ISSN: 1745-6916. DOI: 10.1177/1745691612465253. URL: https://doi.org/10.1177/1745691612465253 (visited on 08/03/2019).

PEP 8 – Style Guide for Python Code (2019). en. URL: https://www.python.org/dev/peps/pep-0008/ (visited on 07/04/2019).

Plesser, Hans E. (2018). "Reproducibility vs. Replicability: A Brief History of a Confused Terminology". English. In: *Front. Neuroinform.* 11. ISSN: 1662-5196. DOI: 10.3389/fninf.2017.00076. URL: https://www.frontiersin.org/articles/10.3389/fninf.2017.00076/full (visited on 06/13/2018).

Quaglio, Pietro, Vahid Rostami, et al. (Apr. 2018). "Methods for identification of spike patterns in massively parallel spike trains". en. In: *Biol Cybern* 112.1, pp. 57–80. ISSN: 1432-0770. DOI: 10.1007/s00422-018-0755-0. URL: https://doi.org/10.1007/s00422-018-0755-0 (visited on 08/05/2019).

Quaglio, Pietro, Alper Yegenoglu, et al. (2017). "Detection and Evaluation of Spatio-Temporal Spike Patterns in Massively Parallel Spike Train Data with SPADE". English. In: *Front. Comput. Neurosci.* 11. ISSN: 1662-5188. DOI: 10.3389/fncom.2017.

00041. URL: https://www.frontiersin.org/articles/10.3389/fncom.2017.00041/full (visited on 06/08/2019).

Ray, Subhasis et al. (Apr. 2016). "NSDF: Neuroscience Simulation Data Format". en. In: *Neuroinform* 14.2, pp. 147–167. ISSN: 1559-0089. DOI: 10.1007/s12021-015-9282-5. URL: https://doi.org/10.1007/s12021-015-9282-5 (visited on 07/29/2019).

Rey, Hernan Gonzalo, Carlos Pedreira, and Rodrigo Quian Quiroga (Oct. 2015). "Past, present and future of spike sorting techniques". eng. In: *Brain Res. Bull.* 119.Pt B, pp. 106–117. ISSN: 1873-2747. DOI: 10.1016/j.brainresbull.2015.04.007.

Riehle, Alexa et al. (2013). "Mapping the spatio-temporal structure of motor cortical LFP and spiking activities during reach-to-grasp movements." In: *Front Neural Circuits* 7, p. 48. DOI: 10.3389/fncir.2013.00048. URL: http://dx.doi.org/10.3389/fncir.2013.00048.

Rostami, Vahid et al. (May 2017). "[Re] Spike Synchronization and Rate Modulation Differentially Involved in Motor Cortical Function". Python. In: *ReScience* 3.1, p. 3. DOI: 10.5281/zenodo.583814. URL: https://github.com/ReScience-Archives/Rostami-Ito-Denker-Gruen-2017/blob/master/article/Rostami-Ito-Denker-Gruen-2017.pdf.

Rubacha, Michael, Anil K. Rattan, and Stephen C. Hosselet (Feb. 2011). "A Review of Electronic Laboratory Notebooks Available in the Market Today". en. In: *Journal of Laboratory Automation* 16.1, pp. 90–98. ISSN: 22110682. DOI: 10.1016/j.jala.2009.01.002. URL: http://journals.sagepub.com/doi/10.1016/j.jala.2009.01.002 (visited on 12/18/2018).

Rübel, Oliver et al. (Jan. 2019). "NWB:N 2.0: An Accessible Data Standard for Neurophysiology". en. In: *bioRxiv*, p. 523035. DOI: 10.1101/523035. URL: https://www.biorxiv.org/content/10.1101/523035v1 (visited on 08/28/2019).

Runnarong, Nuttakarn et al. (2019). "Age-related changes in reach-to-grasp movements with partial visual occlusion". eng. In: *PLoS ONE* 14.8, e0221320. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0221320.

Savage, Caroline J. and Andrew J. Vickers (Sept. 2009). "Empirical Study of Data Sharing by Authors Publishing in PLoS Journals". en. In: *PLOS ONE* 4.9, e7078. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0007078. URL: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0007078 (visited on 08/02/2019).

Schwarz, David A. et al. (June 2014). "Chronic, wireless recordings of large-scale brain activity in freely moving rhesus monkeys". en. In: *Nat Meth* 11.6, pp. 670–676. ISSN: 1548-7091. DOI: 10.1038/nmeth.2936. URL: http://www.nature.com/nmeth/journal/v11/n6/full/nmeth.2936.html (visited on 04/19/2017).

Seo, Dongjin et al. (Apr. 2015). "Model validation of untethered, ultrasonic neural dust motes for cortical recording". eng. In: *J. Neurosci. Methods* 244, pp. 114–122. ISSN: 1872-678X. DOI: 10.1016/j.jneumeth.2014.07.025.

Shew, Woodrow L., Timothy Bellay, and Dietmar Plenz (Sept. 2010). "Simultaneous multi-electrode array recording and two-photon calcium imaging of neural activity". In: *J Neurosci Methods* 192.1, pp. 75–82. ISSN: 0165-0270. DOI: 10.1016/j.jneumeth.

2010.07.023. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2934901/ (visited on 07/19/2019).

Shore, James and Shane Warden (Nov. 2007). *The Art of Agile Development: Pragmatic Guide to Agile Software Development*. English. 1 edition. Beijing : Sebastopol, CA: O'Reilly Media. ISBN: 978-0-596-52767-9.

Siegle, Joshua H. et al. (June 2017). "Open Ephys: an open-source, plugin-based platform for multichannel electrophysiology". en. In: *J. Neural Eng.* 14.4, p. 045003. ISSN: 1741-2552. DOI: 10.1088/1741-2552/aa5eea. URL: https://doi.org/10.1088%2F1741-2552%2Faa5eea (visited on 08/30/2019).

Smeets, Jeroen B. J., Katinka van der Kooij, and Eli Brenner (July 2019). "A review of grasping as the movements of digits in space". eng. In: *J. Neurophysiol.* ISSN: 1522-1598. DOI: 10.1152/jn.00123.2019.

Sprenger, Julia (2014). "Spatial Dependence of the Spike-Related Component of the Local Field Potential in Motor Cortex". English. PhD thesis.

Sprenger, Julia et al. (2019). "odMLtables: A User-Friendly Approach for Managing Metadata of Neurophysiological Experiments". In: *Frontiers in Neuroinformatics* 13, p. 62. ISSN: 1662-5196. DOI: 10.3389/fninf.2019.00062. URL: https://www.frontiersin.org/article/10.3389/fninf.2019.00062.

Stoewer, Adrian et al. (2014). "File format and library for neuroscience data and metadata". In: *Frontiers in Neuroinformatics* 27. ISSN: 1662-5196. DOI: 10.3389/conf.fninf.2014.18.00027. URL: http://www.frontiersin.org/neuroinformatics/10.3389/conf.fninf.2014.18.00027/full.

Sukiban, Jeyathevy et al. (July 2019). "Evaluation of Spike Sorting Algorithms: Application to Human Subthalamic Nucleus Recordings and Simulations". eng. In: *Neuroscience* 414, pp. 168–185. ISSN: 1873-7544. DOI: 10.1016/j.neuroscience.2019.07.005.

Suner, S. et al. (Dec. 2005). "Reliability of signals from a chronically implanted, silicon-based electrode array in non-human primate primary motor cortex". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 13.4, pp. 524–541. ISSN: 1534-4320. DOI: 10.1109/TNSRE.2005.857687.

Tebaykin, Dmitry et al. (2017). "Modeling sources of interlaboratory variability in electrophysiological properties of mammalian neurons". In: *Journal of Neurophysiology* 119.4, pp. 1329–1339. DOI: 10.1152/jn.00604.2017. URL: https://doi.org/10.1152/jn.00604.2017.

Teeters, Jeffery L. et al. (2015). "Neurodata Without Borders: Creating a Common Data Format for Neurophysiology". In: *Neuron* 88.4, pp. 629–634. ISSN: 0896-6273. DOI: https://doi.org/10.1016/j.neuron.2015.10.025. URL: http://www.sciencedirect.com/science/article/pii/S0896627315009198.

The HDF Group (1997). *Hierarchical Data Format, version 5*.

Torre, Emiliano, Carlos Canova, et al. (July 2016). "ASSET: Analysis of Sequences of Synchronous Events in Massively Parallel Spike Trains". en. In: *PLOS Computational Biology* 12.7, e1004939. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1004939. URL:

https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1004939 (visited on 08/05/2019).

Torre, Emiliano, Pietro Quaglio, et al. (Aug. 2016). "Synchronous Spike Patterns in Macaque Motor Cortex during an Instructed-Delay Reach-to-Grasp Task". en. In: *J. Neurosci.* 36.32, pp. 8329–8340. ISSN: 0270-6474, 1529-2401. DOI: 10.1523/JNEUROSCI.4375-15.2016. URL: https://www.jneurosci.org/content/36/32/8329 (visited on 08/05/2019).

Unakafova, Valentina A. and Alexander Gail (Apr. 2019). "Comparing open-source toolboxes for processing and analysis of spike and local field potentials data". en. In: *bioRxiv*, p. 600486. DOI: 10.1101/600486. URL: https://www.biorxiv.org/content/10.1101/600486v2 (visited on 08/03/2019).

Vargas-Irwin, C. E. et al. (2010). "Decoding Complete Reach and Grasp Actions from Local Primary Motor Cortex Populations". In: *Journal of Neuroscience* 30.29, pp. 9659–9669. ISSN: 1529-2401. DOI: 10.1523/jneurosci.5443-09.2010.

Verkhratsky, Alexei, O. A. Krishtal, and Ole H. Petersen (2006). "From Galvani to Patch Clamp: The Development of Electrophysiology." In: *Pflugers Arch* 453.3, pp. 233–247. DOI: 10.1007/s00424-006-0169-z.

Vines, Timothy H. et al. (Jan. 2013). "Mandated data archiving greatly improves access to research data". In: *The FASEB Journal* 27.4, pp. 1304–1308. ISSN: 0892-6638. DOI: 10.1096/fj.12-218164. URL: https://www.fasebj.org/doi/full/10.1096/fj.12-218164 (visited on 08/02/2019).

Walt, Stéfan van der, S. Chris Colbert, and Gaël Varoquaux (2011). "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2, pp. 22–30. DOI: http://dx.doi.org/10.1109/MCSE.2011.37. URL: http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2011.37.

Wilkinson, Mark D. et al. (Mar. 2016). "The FAIR Guiding Principles for scientific data management and stewardship". en. In: *Scientific Data* 3, p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18. URL: https://www.nature.com/articles/sdata201618 (visited on 08/02/2019).

Yatsenko, Dimitri et al. (Nov. 2015). "DataJoint: Managing Big Scientific Data Using MATLAB or Python". In: *bioRxiv*. DOI: 10.1101/031658. URL: http://biorxiv.org/lookup/doi/10.1101/031658 (visited on 01/23/2019).

Yeung, Andy W. K. (2017). "Do Neuroscience Journals Accept Replications? A Survey of Literature". English. In: *Front. Hum. Neurosci.* 11. ISSN: 1662-5161. DOI: 10.3389/fnhum.2017.00468. URL: https://www.frontiersin.org/articles/10.3389/fnhum.2017.00468/full (visited on 08/03/2019).

Yu, Byron M. et al. (July 2009). "Gaussian-Process Factor Analysis for Low-Dimensional Single-Trial Analysis of Neural Population Activity". In: *J Neurophysiol* 102.1, pp. 614–635. ISSN: 0022-3077. DOI: 10.1152/jn.90941.2008. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2712272/ (visited on 08/05/2019).

Zehl, Lyuba (2018). *Management of Electrophysiological Data & Metadata - Making complex experiments accessible to yourself and others*. Schriften des Forschungszen-

trums Jülich Reihe Schlüsseltechnologien / Key Technologies 167. Jülich. ISBN: 978-3-95806-311-2.

Zehl, Lyuba et al. (2016). "Handling Metadata in a Neurophysiology Laboratory". English. In: *Front. Neuroinform.* 10. ISSN: 1662-5196. DOI: 10.3389/fninf.2016.00026. URL: https://www.frontiersin.org/articles/10.3389/fninf.2016.00026/full (visited on 01/11/2019).

Zhang, Bo, Ji Dai, and Tao Zhang (Nov. 2017). "NeoAnalysis: a Python-based toolbox for quick electrophysiological data processing and analysis". In: *BioMedical Engineering OnLine* 16.1, p. 129. ISSN: 1475-925X. DOI: 10.1186/s12938-017-0419-7. URL: https://doi.org/10.1186/s12938-017-0419-7 (visited on 07/04/2019).