AOData Documentation

Sara Patterson - November 16, 2022

Contents

1 Introduction

1.1 Motivation

Our experiments generate a massive amount of data, typically spread over many files in multiple folders. Managing that is hard enough, but there's also considerable metadata as well (field of view sizes, imaging light intensities, PMT gains, power measurements, calibrations, pinhole sizes, stimuli, parameters used to generate said stimuli, registrations, parameters used for registration,...). Currently we don't have an easy way of maintaining and accessing everything. Some of this metadata is saved in files generated by the imaging software, others are hand-written in experiment logs (power measurements, notes, etc.). This information is already difficult to access as is and that becomes harder once everything is moved to the NAS.

This already difficult data management task becomes a greater challenge when you move to data analysis. Even the most standard analyses, such as extracting responses from ROIs can have many associated parameters that don't get stored (whether you extracted the raw fluorescence or the dF/F, what background window was used and whether you took the mean or the median). Then there are more specialized analyses which may evolve over time. You can save the output of an analysis, but you might not remember in a few months or years how exactly that output was generated. Of course, you can save the code you used but tracking that down later might be difficult, especially when calling a complicated series of functions/scripts that may change over time and no longer work in the same way.

Most of us have our own solutions to some of these problems, but each of us solving them on our own wastes time and makes it hard for new members to get up and running. These solutions are typically specific to a single programming language, which presents additional barriers to collaboration. Critically, this also leaves us with no standardized approach for data management.

Most often, we end up with highly personalized, inflexible code designed to do something specific. A quote I like is "the greatest limitation in writing software is our ability to understand the systems we are creating".[4] Indeed, code management is a major problem. However, in research, we have an even greater limitation - someone else's ability to understand the system you created. Given the naturally high turnover in academia, someone else unfamiliar with your approach may need to pick up where you left off to complete a project.

These are the issues AOData aims to solve.

1.2 Problems

Normally we write code from an implementation perspective (what code do I need to write to get this analysis done by lab meeting tomorrow). If we write the code to do something specific, of course it's not going to be very adaptable/useful when we need to do something else later on. The design of AOData was done from a problem-oriented perspective. The problems that needed to be addressed were first identified and then the code followed. Here are the problems AOData aims to solve (or at least provide you with the tools for solving):

1. We have no standardized way for storing/accessing the mountains of metadata associated with each experiment, even though much of ARIA works with similar metadata files.
2. There is no one-size-fits-all approach because different types of imaging will generate different combinations of metadata files.
3. While metadata file names for each epoch are predictable, we still need a way of automatically identifying them that is robust to small changes. Robust reading of metadata is important too, we don't want the code to break if a line is added/removed from a .txt file.
4. A strong metadata solution will extend beyond what is generated during an experiment to include stimulus design and analysis.
5. We have individual preferences on how we analyze our post-registration data and need a common foundation that can accommodate and contain those preferences.
6. Our code changes over time and we need to be free to make necessary changes without fear of breaking past code.
7. Metadata and data must be saved in a format appropriate for long-term archival. This format should be platform/language-independent, fast and capable of handling large heterogeneous datasets.

1.3 Solutions

- The code base is tracked with git and the latest commit ID is written with the HDF5 file. If the code changes in a year, you can still run code developed with a previous version by rolling back the repository to an earlier commit.

- The end product, an HDF5 file, is portable, platform-independent and can be taken to languages other than MATLAB. The HDF5 format is self-describing, in that metadata can be attached to the data it describes.
- The framework is flexible and open-ended. Enough organization is established through the core classes to ensure a baseline level of standardization in how the components interact, without placing unnecessary restrictions on implementation. The goal is that you can fill in the code you've already written with minimal changes.
- Object oriented design - work with components that make intuitive sense, like "Device" and "Registration". Reusability. Modularity minimizes side effects.
- Cooperates with existing lab experiment file structures, no need to alter how data is stored.

1.4 Design Considerations

A key gap in our current framework is a method for condensing our massive experiment files into a smaller representation containing only the most important elements. Most of us do this to some degree when analyzing our data, but the results are typically saved in some language-specific format (e.g. MATLAB .mat files). This is problematic for storing complex data as it is not a self-contained, platform-independent solution. Ultimately, I chose to use Hierarchical Data Format (HDF) (specifically HDF5, the newest version) files[2] in a way that cooperates with our existing experiment file structure (experiment videos are not written...). HDF5 is used widely in both academia and industry and is a "defacto" standard for large, heterogeneous datasets. I did explore a few file storage alternatives to HDF5, in particular, the Advanced Scientific Data Format (ASDF) developed for the James Webb Space Telescope to solve some complaints with FITS.[3] The readable YAML component was a major draw over HDF5 but ASDF also doesn't have MATLAB support. Writing that was too time-consuming and choosing Python would have turned off most ARIA members who prefer MATLAB. The lack of MATLAB support ruled out several other file formats as well. Finally, I considered writing to JSON or YAML files without using a more specialized format like HDF5,[1] but reading/writing was significantly slower and this approach limited the size of data that could be written.

I also considered other frameworks for organizing/managing experimental data before creating my own, particularly Symphony-DAS and NeurodataWithoutBorders (NWB).[5,6] Symphony-DAS has an excellent framework but was intended to operate with data acquisition and would need some work to apply to imaging data. We want a system that operates offline, independent of data acquisition. NWB is designed to provide a single format that applies to all neuroscience data. It's highly specified which provides a lot of overhead, especially considering much AO data isn't necessarily "neuroscience data". I considered modifying both to fit our experiments, but it was ultimately simpler to create something new. However, many of their ideas are adopted here.

2 Getting Started

2.1 Installation

2.1.1 Downloading AOData

You can get the latest version of the AOData on Github: https://github.com/sarastokes/ao-data-tools. Download git if you do not have it already, then go to the folder you want to clone AOData into. Right click and select "Git Bash". Then type in:

```
git clone https://github.com/sarastokes/AOData --recurse-submodules
```

I would strongly recommend using git. In case you are not familiar, there are many reasons for using git but here are a few relevant ones:

- You can easily keep up-to-date with changes to AOData. You will undoubtedly have feedback and suggestions that I will want to implement. If I update the repository accordingly, you can update your local copy with a single command.
- If you consistently track your own code with git, you don't have to worry about backwards compatibility issues. If you, for example, change the inputs to a function, your earlier analysis code calling that function will no longer work. Avoiding changing your functions when those changes are warranted isn't great either. If you want to repeat that analysis later and have been tracking your code with git, you can just roll back your repository and run it as it existed at the time you did your analysis.

Long story short: it won't take you long to get going with git and it'll make your life infinitely easier in the future. If you're still not convinced, you can click on the green button labeled Code at the URL above and download the code as a zip file.

2.1.2 Starting AOData in MATLAB

Add the folder and all the subfolders to your MATLAB path with:

```
addpath(genpath('yourfilepath/ao-data-tools'));
```

You will need to add AOData to your MATLAB path every time you open MATLAB. If you intend to use AOData frequently and want to avoid this, add the above line to your startup file If you do not have one, create a file called `startup.m` in the "MATLAB" folder that is created within Documents when you download MATLAB. This file will run every time you start up MATLAB.

```
initializeAOData();
```

You only need to initialize AOData once. Rerun this if you move the location of the AOData folder on your computer. This function stores information about AOData's location on your computer in MATLAB's user preferences. After completion it opens `AODataManagerApp`. If you are using any user-defined AOData packages, add them. This is an important step because AOData If you create a new package after initialization, you can return

2.2 Dependencies

You will need MATLAB to create the initial data files for your experiment. The code is tested on MATLAB 2022n with both PC and Mac; beyond that all bets are off. The base AOData code does not require any additional MATLAB toolboxes. Once the HDF5 file is created, you can use it in any programming language that can read an HDF5 file (virtually all major programming languages support HDF5).

Code I didn't write myself is included in the `lib` folder. There are some miscellaneous functions (usually the authors have their information in the documentation), as well as three toolboxes: JSONlab 2.0, ReadImageJROI and appbox.

3 Framework

An overview of the AOData object model is shown below. Each item in the list corresponds to a core class in AOData. These classes aren't meant to be used as is and they probably can't as they don't contain any implementation that could limit their use. The idea is that it's always easier to add code than to change core code (which undoubtedly causes side-effects and backwards-compatibility issues).

Instead, they establish the components of an experiment, the organization of those components within an experiment and the relationships between them. They are meant to be as generic and flexible as possible rather than specialized for a specific purpose (like physiology experiments in primates on a specific system).

Experiment

- Source
  - Source (nestable)
- System
  - Channel
    - Device
- Calibration
- Annotation
- Epoch
  - Registration
  - Response
  - Stimulus
  - Dataset
- Analysis

A note on terminology: Throughout the documentation, the components of the AOData object model are referred as entities. For example, Epoch and Calibration are both *entities* and "Epoch" and "Calibration" are their *entity types*. This is done for clarity; object-oriented programming is used extensively in AOData and it would be confusing not

to distinguish the core AOData objects from other types of objects. Technically speaking though, in the MATLAB implementation, Calibration, Epoch, etc. truly are entities because they inherit from a superclass called Entity, which ensures they all behave in the same way.

You will need to subclass these to customize them for your experiments. By using the AOData object model, your code will inheriting the organization established by these components. It also means that anyone familiar with the basic framework can look at your code and understand what is going on, without reading the code itself.

4 Using AOData's Documentation

AOData contains extensive documentation within the class and function files and that is the authoritative source for understanding how to use AOData's code. For any function or class, you obtain help by typing in one of the two commands:

```
% For help on a function or class
doc functionName
doc className
% For help on a class method (a function within a class)
doc className/functionName
```

If you don't want the full documentation and instead just want to quickly know how to use something, you can exchange `doc` with `help` to print the core documentation to the command line.

This documentation is meant to provide the other information about AOData: a high-level description of the software and details on implementation that would be difficult to discern without reading the full codebase.

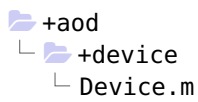
A few conventions are used throughout the documentation. Classes are marked as `MyClass`, properties of classes are marked as `MyProperty`, functions are marked as `MyFunction()` and parameters are marked as `MyParameter`. Miscellaneous code outside a code listing environment is marked as `MyCode`.

5 Background

A basic working knowledge of MATLAB is assumed. Some quick background on HDF5 files and MATLAB concepts that you might not find in an intro class or encounter in your own programming are explained below.

5.1 Packages

Folders beginning with a plus sign are interpreted as **packages**. So to call Device in the file structure below, you need to use `aod.core.Device`, not `Device`.



Why not just use regular file folders? Using packages reduces conflicts and increases code clarity. An example: Say I make an object for the Toptica in the 1P system that inherits from `aod.core.Device`. I could name it `Toptica` or `sara.devices.Toptica`. The first option might work in the short-term. However, Jennifer Hunter's lab has a Toptica too and they might have their own `Toptica` class. Most of us only use one system so that could be fine. Except that we borrowed Jennifer's Toptica so now I need a second Toptica class (it's worth keeping them separate as Jennifer's has one less laser line and higher max powers for the other lines). If they have their own package, I just need that and then I don't have to write any code at all! If they don't, I could make the class and call it either `TopticaTwo` or `hunterlab.devices.Device`. The second option is clearer and specifies exactly which Toptica is being used used (I *might* remember which Toptica is `TopticaTwo`, but no one else will know). It may even help us remember to return Jennifer's Toptica when we're done.

There is an `import` function to simplify calling members of a class within a package. Try to limit its use to within functions. If you use it in the base workspace, you will lose the code clarity advantages of using packages.

```
% To use just aod.core.Device without the package names
import aod.core.Device
device = Device([]);
% To use all core classes without 'aod.core'
import aod.core.*
stimulus = Stimulus([]);
```

```
% To clear all imports
clear import
```

5.2 Object Oriented Programming

Object oriented programming (OOP) is too large a topic to quickly explain here, but I'll touch on a few basics to give you a quick background. Each component in the framework is a **class** and when you instantiate it, you get an **object**. You use OOP all the time in your code (`double`, `char` and `struct` are all classes and when you run a function like `str2double()`, you're converting the object from one class to another).

Classes have **properties**, **methods** and **events**. An imperfect analogy: a function gives you an unlabeled can of soup. A class gives you a can of soup with a label and nutrition facts (properties) along with a can opener, a microwave and a spoon (methods). When the soup expires you get a notification that it expired (event).

If you aren't familiar with OOP, this will be the biggest learning curve for using AOData. The tutorials have information on how to use object-oriented programming within the context of AOData and there are also resources listed in 13. Once you know what the code is supposed to do, classes can be way easier to work with than functions and scripts.

Object oriented programming is an approach that focuses on organizing code into self-contained units called objects. These objects contain both the data and the functions that operate on the data, enabling modular and organized code. There are a number of benefits:

- **Improved readability and understandability.** OOP allows for creation of clear, intuitively named objects that represent real-world concepts, making it easier to understand the purpose and function of code.
- **Reusability**. OOP allows for the creation of generic objects that can be easily reused in different contexts, reducing the need to write new code for scratch. This saves time and reduces the potential for errors.
- **Easier debugging and maintenance.** OOP allows for the isolation of individual objects, making it easier to identify and fix any errors that may arise. In addition, OOP allows for the modification of individual objects without affecting the rest of the code, making it easier to maintain and update the code over time.

To get started, you will want to be familiar with constructors, property and method specification, how access is specified and how inheritance works. You will also want to be comfortable with Static methods and Dependent properties. You can wait on other topics and learn about them as they become relevant. MATLAB's documentation and particularly their examples that demonstrate building classes are worthwhile. A few more resources: an intuitive explanation from the Sampath lab that's particularly strong with inheritance (here).

5.3 HDF5 Files

A processed experiment from AOData is written as Hierarchical Data Format (HDF) file (specifically HDF5, the newest version). The file format was initially developed by the US National Center for Supercomputing Applications and is now supported by the HDF Group, a non-profit committed to ensuring continued development and accessibility of HDF. An intuitive high-level explanation can be found here.

5.3.1 Benefits of HDF5

First, why use HDF5 files? A few advantages:

1. HDF5 is an **open-source**, **platform-independent** and **portable**. The HDF Group is committed to providing long-term development and accessibility. For archival purposes, HDF5 is vastly preferable to the approaches we typically use (.mat or .npy files). For now, you have to create your HDF5 file in MATLAB, but then you can use it in virtually any programming language you want.
2. HDF5 files allow you to **organize** your data within groups resembling a traditional file directory.
3. HDF5 files can be **self describing**. This means the file itself as well as the groups and datasets within the file can have associated metadata. Life is way easier when the metadata is stored with the object it describes. A major component of AOData is providing an interface to allow you to take full advantage of this feature.
4. HDF5 files support **heterogeneous** data, so you can have any combination of standard data types within a single file. There are also no size limits.
5. HDF5 is both time and memory **efficient**. The file format supports lossless compression, reducing the overall size of your data. The entire dataset also doesn't have to be read into memory; you only need to read in the datasets you actually want. Finally, reading/writing data from an HDF5 file is *fast*, much much faster than any other file format I explored.

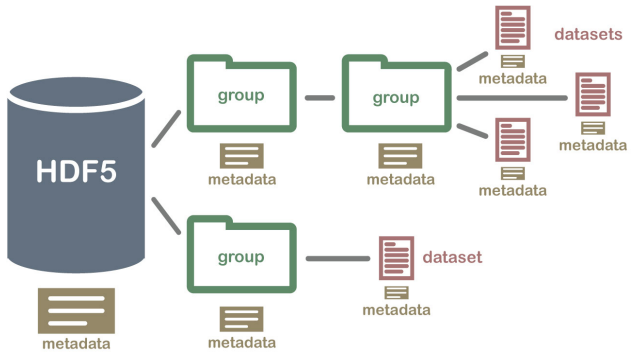


Diagram of the HDF5 file format, from the National Ecological Observatory Network. The metadata attached to groups and datasets are called "attributes".

5.3.2 Components of an HDF5 File

The two main aspects of an HDF5 file are **groups** and **datasets**. In addition there are **attributes** that can be attached to groups and datasets.

- **Groups** define organization within the file and can optionally have associated metadata (attributes). Every HDF5 file contains a root group that can contain datasets or other groups. Working with groups is similar to working with directories and files in UNIX. For example: `/` is the root group, `/foo` is a member of the root group called `foo`, `.` signifies the specified file, group or object.
  Each instance of the core classes in the AOData framework (see 3 on "Framework") above are written as a group in the HDF5 file.
- **Datasets** organize and contain "raw" data values. A dataset consists of the data itself and metadata (attributes) that describes the data. Most MATLAB data types have a comparable HDF5 datatype, with one exception - multi-level structures (a `struct` where one or more fields is another `struct` or another compound datatype like `table` or `containers.Map`). The details of conversion to HDF5 datatype are discussed in 10.
- **Attributes** can optionally be associated with HDF5 objects. They have two parts: a *name* and a *value*. Typically an attribute is small and contains metadata about the object it is attached to, either a group or a dataset.
- **References** are low-level pointers to other objects. They can be stored and retrieved as data, created as an attribute or an entire dataset of reference type. There are two types of references, the first is used often in AOData for links between entities.
  1. *Object references* point to a particular object in a file, either a dataset or a named datatype.
  2. *Region references* always point to a dataset and additionally contain info about a certain selection (*dataset region*) on that dataset.

5.3.3 Viewing HDF5 files

A disadvantage of using HDF5 files is that they are binary and not human-readable (i.e. you can't pull the file up in a text editor and read it, like for a JSON file). Instead, you will need some sort of software to visualize the contents of an HDF file. The most commonly used is the HDF Group's HDFView. This is definitely worth installing.
HDFView is very generic because HDF5 files can be very diverse. AOData HDF5 files are built around the AOData object model, which provides a predictable structure. I wrote `AODataViewer` to be designed around the AOData framework.

```
AODataViewer('myfile.h5');
```

There is also a standalone compiled version that does not require a MATLAB license: link.

There are a number of nice options if you're comfortable with Python, such as h5pyViewer and ViTables. Regardless of the language you intend to use, I recommend first looking through the files with `AODataViewer` because it's designed to facilitate understanding the AOData framework.

Figure here. Understanding how AOData writes experiment data to an HDF5 file is described in 10 on "HDF5 File Details". Groups are shown with folders and datasets are shown with files. Groups that are containers for

multiple entities are italicized, the other groups correspond to entities. Checking out the example file will make this clearer.

5.4 Advanced MATLAB Concepts

This section has information on a few MATLAB-specific topics (other languages probably have similar things using different names) that aren't routinely covered in the introductory documentation but are critical to how AOData operates. *You don't need to read and understand this prior to using AOData!* You may never need to deal with these directly; after all, the whole point of abstraction in 5.2 is that complicated behind-the-scenes code is handled and kept hidden from end-users. At some point though as you work with AOData, you might have questions about these topics and can circle back to the information below:

5.4.1 Handle vs. Value Classes

When you work with `char` and `double`, you are working with value classes.

```
% Create a double variable
a = 3.14
% Copy it to another variable
b = a;
% Change the value of b
b = 3.14159;
% You have changed b, but not a
disp([a, b])
```

By contrast, most classes in AOData are handle classes. Handle classes provide a "handle" to the object, which is stored only one in memory:

```
% Create a standalone Device class. It's from ThorLabs but you don't remember the Model
device1 = aod.core.Device([], 'Manufacturer', 'ThorLabs', 'Model', 'idk');
% Copy set a second variable equal to device1
device2 = device1;
% Now you remember the Model number
device2.setParam('Model', 'P20K');
% The Model parameter is now changed for device2...
disp(device2.parameters)
% ...and for device1, because both variables were just handles to data that exists only once in memory
disp(device1.parameters)
```

Why is this relevant to AOData? See below:

```
% Create a channel
channel = aod.core.Channel('MyChannel');
% Create a device and add it to the channel
device = aod.core.Device('Manufacturer', 'ThorLabs', 'Model', 'idk');
channel.add(device);
% Change a parameter on the device variable
device.setParam('Model', 'P20K');
% And it is also changed within the Channel!
disp(channel.Devices.parameters)
```

5.4.2 matlab.mixin.Heterogeneous

When you create custom classes in MATLAB, you can only concatenate members of the same class. This gets annoying when you have multiple subclasses of one of your custom classes and want to make them into an array (which we do for all containers). By subclassing `matlab.mixin.Heterogenous` in each core class (other than Entity), your subclasses of, for example, `aod.core.Device` can be concatenated into an array. A quirk of using `matlab.mixin.Heterogeneous` is that all methods in the class subclassing it must be either `Sealed` or `Abstract`. You don't have to continue this practice when subclassing the core classes and do not need to declare inheritance again from `matlab.mixin.Heterogeneous`. Also, your classes cannot inherit two classes that use `matlab.mixin.Heterogeneous` (e.g. both Device and Channel), though I can't imagine why you would need to do that anyway.

5.4.3 Enumerations

Enumerations are useful when you have a list of things and you want to standardize their behavior. MATLAB's documentation (see 13) will do a better job describing this than I will, but two things are worth mentioning here. First, entity behavior is codified in an enumeration (`aod.core.EntityTypes`). Second, you may want to include a

static `init` method if you want to read back your own enumerations classes from HDF to MATLAB. You don't need to deal with enumerations to use AOData, but they can be helpful (for more examples see `sara.EpochTypes` and `sara.SpectralTypes`).

5.4.4 Dynamic Properties

The MATLAB `dynamicprops` class enables ad-hoc addition of properties to specific instances of a class (often called "instance properties" in other programming languages). The advantage of dynamic properties is that they allow the persistent interface to adapt to customized subclasses of the core classes without any prior knowledge of said subclass (other than which entity type it is).

6 Workflow

To use AOData for an experiment, you will follow the workflow below.

1. Define how your experiment will map to the AOData object model in MATLAB using the Core Interface (see 7). Creating the subclasses will require an initial investment, after which users will only need to work in the core interface when changes are made to an experiment.
2. Experiment specification file (example).
3. Write your experiment to the HDF5 file with `aod.h5.writeExperimentToFile`.
4. Once you have the HDF5 file, you can continue your analysis in any language you would like. If you choose to use MATLAB for analysis, you will access your HDF5 file through the Persistent Interface (see 9). You can still make changes to the HDF5 file at this point and the techniques for doing so are defined below.

You will need to define subclasses of the core classes to define how exactly you want your experiment to be set up. This will require an initial investment of time to set the classes up. Once established, it is unlikely you will need to modify these unless you change up how you're performing your experiments (which can often mean different data types, metadata and/or parameters).

7 Core Interface

This information is compiled here to serve as a single resource for core classes. It may not be the best gateway for learning about the core classes for the first time. If the documentation below seems like too much information, try to read through 7.1 on `aod.core.Entity` and then read 11.1 on "Class Design Principles", before returning. However, each class and function is thoroughly documented in the source code. For more detail on any aspect below, try using the `help` command in MATLAB:

```
% Get documentation for a class or function
help aod.core.Experiment
% Get documentation for a function within a class
help aod.core.Experiment\add
```

The documentation for each class includes (if relevant), the following information:

className (parentClassName(s))

The constructor will be shown. There may be two version, one with the minimal number of required inputs and a second showing the optional inputs.

```
obj = MyClass(inputs)
```

Properties: - Class properties that are written as datasets, with two exceptions: `aod.core.Entity`'s properties are written as parameters and anything designated as a container is a group that contains other entities.

- `MyProp`

Parameters: Optional inputs sent to `aod.util.Parameters` and written as attributes

- `MyParam`

Methods:

- `outputs = MyFcn(obj, inputs)`

Examples: Subclasses within AOData that demonstrate implementation

- `SubclassName`

Methods may be listed with some combination of Abstract, Sealed, Static, Protected or Private. Properties may be listed as Abstract, Dependent, Protected or Private. If access isn't specified, assume it is Public. If these terms aren't familiar, check out the resource recommendations in 5.2 on "Object Oriented Programming".

7.1 ao.core.Entity (handle)

Abstract parent class to all core classes that are persisted into the HDF5 file. Because all core classes inherit `aod.core.Entity`, they all operate in a similar way and if you learn the basics of this parent class, you'll already need to know most of how the subclasses work.

Before getting into the code documentation, it is worth explaining it in the context of AOData's goals. The main question in designing `aod.core.Entity` was: What should all components have in common? and How can they be implemented in a flexible/generic way to avoid unnecessarily restricting subclasses? In an ideal world, `aod.core.Entity` will be set in stone as any change here will change the rest of the framework. So it needs to provide enough options (optional options, not required options - you should be able to use what you want and ignore the rest) to be robust and flexible. To ensure end-users aren't depending on critical functionality, the things that `aod.core.Entity` absolutely *must* do to make the code work should be hidden from the average user.

```
% Note that aod.core.Entity is abstract and never directly instantiated.
% This is the constructor that subclasses will call in their constructors
obj = aod.core.Entity(name)
```

Properties:

- `Name` - user-defined name for the entity
- `description` - a space for describing the Entity is (in case the class, properties and parameters are not sufficient)
- `parameters` - `aod.util.Parameters` containing entity metadata
- `files` - `aod.util.Parameters` containing file names associated with the entity
- `notes` - a space for miscellaneous comments

Private Properties: These are handled behind the scenes and average users shouldn't need to worry about them

- `Parent` optional, another entity that the current entity falls under in the hierarchy (e.g. the Parent for an Epoch is the Experiment). This is set when you add the entity to another entity using `add()`.
- `UUID` - a randomly-generated unique identifier from `aod.util.generateUUID()`. Behind the scenes, these are used to search through entities and their HDF5 counterparts, providing a solution for indexing which is one of HDF5's larger shortcomings. See 10.4.1 on `aod.h5.EntityManager` for the solution.

Dependent Properties:

- `label` - automatically generated label based on data and metadata. Set to the output of `getLabel()`

Public methods:

- `setDescription(obj, txt)`
- `addNote(obj, txt)`
- `removeNote(obj, ID), removeNote(obj, 'all')`

- `add(obj, entity)` - Add an entity

- `tf = hasParam(obj, 'paramName')`
- `addParam(obj, 'paramName', paramValue)` - add parameters as key/value pair(s) or a struct
- `removeParam(obj, paramName)`
- `paramValue = getParam(obj, 'paramName', mustReturnParam)`

- `tf = hasFile(obj, fileKey)`
- `setFile(obj, fileKey, filePath)`
- `fileValue = getFile(obj, fileKey)`
- `fileValue = getExpFile(obj, fileKey)` - appends the current `homeDirectory` to the returned file path
- `removeFile(obj, fileKey)`

- `fPath = getHomeDirectory(obj)` - once an entity has been added to the Experiment or to another entity that is added, this function will return the `homeDirectory` of `aod.core.Experiment`
- `h = ancestor(obj, entityType)` - recursively searches `Parent` to find an entity of the specified type.
- `assignUUID(obj, UUID)` - assign a specific UUID to an entity. UUIDs are generated automatically for all entities and must be unique within in an experiment. However, some entities may not be unique across experiments. Use this method if you would like to have, for example, a Source or Device that is identifiable by the same UUID in each experiment.

Protected methods:

- `value = getLabel(obj)` - defines `label`. Default value is the class name without packages but subclasses should tailor it, if needed.
- `setParent(obj, entity)`
- `sync(obj)` - For updating entity after added to the main experiment hierarchy. Use this is the entity needs information from other entities (e.g. `Parent`)

7.2 aod.core.Experiment (aod.core.Entity)

The parent class for all other entities, containing everything associated with a specific experiment. Experiment is written first to the HDF5 file and the other entities are written in the order shown below.

```
obj = aod.core.Experiment('name', experimentDate, 'homeDirectory')
obj = aod.core.Experiment('name', experimentDate, 'homeDirectory',...
    'Administrator', "YourName", 'System', "Primate1P")
```

Properties:

- `homeDirectory` - file path to experiment folder
- `experimentDate` - date that the experiment occurred (yyyyMMdd)
- `Analyses` - container for all Analyses within the Experiment
- `Calibrations` - container for all Calibrations within the Experiment
- `Epochs` - container for all Epochs within the Experiment
- `Sources` - container for all Sources within the Experiment
- `Annotations` - container for all Annotations within the Experiment
- `Systems` - container for all Systems within the Experiment

Dependent Properties:

- `epochIDs` - the IDs of all epochs in Experiment
- `numEpochs` - the number of epochs in the Experiment

Parameters:

- `Administrator` - who performed the experiment
- `System` - the name of they system used

Methods: This may seem like a lot, but many of the methods are conceptually identical (e.g. if you know how to use `add`, then you know how to use all the add methods)

- `setHomeDirectory(obj, filePath)` - change file path to experiment data folder
- `add(obj, entity)` - add Analysis, Annotation, Calibration, Epoch, Annotation, Source or System
- `remove(obj, entity, ID)` - remove an Analysis, Annotation, Calibration, Epoch, Source or System. ID should be the index of the entity/entities (or epochID for Epochs) to remove or "all" to clear all
- `epoch = id2epoch(obj)` - input ID and returns Epoch, if exists
- `idx = id2index(obj, ID)` - input ID and return the index of the Epoch in Epochs

7.3 aod.core.Source (aod.core.Entity, matlab.mixin.Heterogeneous)

The source of the data acquired during the experiment.

```
obj = aod.core.Source(parent, 'name')
```

Sealed Methods:

- `add(obj, source)` - add source
- `remove(obj, sourceID)` - remove a child source
- `clearChildSources(obj)` - clear all child sources
- `sources = getParents(obj)`
- `ID = getParentID(obj)`

Sources are nestable. For an example hierarchy, see the following Source subclasses where you might have 2 locations in 1 eye of 1 subject. When specifying the source for `aod.core.Epoch`, use the most specific Source in the hierarchy.

7.3.1 aod.core.sources.Subject (aod.core.Source)

Top-level source (person, monkey, mouse, model eye).

```
obj = aod.core.Subject(name)
obj = aod.core.Subject(name, 'Species', value, 'Sex', value, 'Demographics', value)
```

Parameters:

- `Species`
- `Sex` - male, female or unknown
- `Age`
- `Demographics` - fluorophore, transgenic line, disease

7.3.2 aod.core.sources.Eye (aod.core.Source)

An Eye within the Subject. `Name` is restricted to being either OS or OD.

```
obj = aod.core.sources.Eye(name)
```

Properties:

- `name` - inherited from Entity, but with added restriction of being either 'OS' or 'OD'

7.3.3 aod.core.sources.Location (aod.core.Source)

A location within an Eye.

```
obj = aod.core.sources.Location(name)
```

7.4 aod.core.Calibration (aod.core.Entity, matlab.mixin.Heterogeneous)

Represents measurements of the system (e.g. spectra/power of light sources, optimized positions of PMT).

```
obj = aod.core.Calibration(name, calibrationDate)
```

Properties:

- `calibrationDate` - date calibration was performed

Methods:

- `setCalibrationDate(obj, calDate)` - calDate should be `datetime` or a `char` with the format yyyyMMdd

Examples:

- `aod.builtin.calibrations.PowerMeasurement`
- `aod.builtin.calibrations.RoomMeasurement`
- `aod.builtin.calibrations.ChannelOptimization`

7.5 aod.core.System (aod.core.Entity, matlab.mixin.Heterogeneous)

Represents the system configuration you used to acquire the data. While there is only one system, you may use it in different ways within or between experiments, depending on the type of imaging your're doing. The reasoning behind this organization is that you may used one system in two different ways during an experiment (e.g. calcium imaging requires a different set of channels than rhodamine imaging)

```
obj = aod.core.System(name)
```

Because it is hard to automate system names, `name` is a required input

Properties:

- `Channels` - container for all `aod.core.Channel`s within the System

Sealed Methods:

- `add(obj, channel)` - Add a Channel to the system
- `remove(obj, ID)` - ID is the index of the channel to remove. The standard remove syntax is also accepted: `remove(obj, entityType, ID)`, but entityType must be Channel
- `getChannelDevices(obj)`

7.6 aod.core.Channel (aod.core.Entity, matlab.mixin.Heterogeneous)

A single light path within the System. For example, the calcium imaging channel contains a Mustang laser, a bandpass filter, a PMT and a pinhole. The specifics of these devices may change between experiments (for example, you may test different pinhole sizes for fluorescence imaging) or even within experiments (trying out different dichroic filters).

```
obj = aod.core.Channel(name)
```

Properties:

- `Devices` - container for all `aod.core.Device`s within the Channel

Sealed Methods:

- `add(obj, device)` - Add a Device to the channel
- `removeDevice(obj, ID)`
- `clearDevices(obj)`

7.7 aod.core.Device (aod.core.Entity, matlab.mixin.Heterogeneous)

Any sort of Device within a Channel worth logging. You don't necessarily need to add every single mirror in the light path, just the components that are critical and may have parameters that change between experiments. See **11.1** on "Class Design" for a thorough explanation of how this class was designed that introduces the principles behind all other core classes.

```
obj = aod.core.Device(name)
obj = aod.core.Device(name, 'Model', value, 'Manufacturer', value);
```

Parameters:

- `Manufacturer`
- `Model`

Examples:

- `aod.builtin.devices.BandpassFilter`
- `aod.builtin.devices.DichroicFilter`
- `aod.builtin.devices.LightSource`
- `aod.builtin.devices.Pinhole`
- `aod.builtin.devices.PMT`

7.8 aod.core.Annotation (aod.core.Entity, matlab.mixin.Heterogeneous)

Spatial regions within your acquired data. These could be ROIs in a physiology experiment, coordinates of labeled cells in anatomical imaging or a subregion of the field of view that omits the zeroed pixels after registration.

```
obj = aod.core.Annotation()
obj = aod.core.Annotation(data)
```

Properties:

- `Data`
- `Source`

Protected Properties:

- `Reader` - `aod.util.FileReader`

Sealed Protected Methods:

- `setData(obj, data)`
- `setSource(obj, source)`

Examples:

- `aod.builtin.annotations.Rois`

7.9 aod.core.Epoch (aod.core.Entity, matlab.mixin.Heterogeneous)

A single period of data acquisition within the Experiment. Epochs differ from the other entities in one key way: because they already have an intuitive numeric identifier (`ID`, the video number in the experiment), this is used rather than their index in the `Epochs` container for methods like `remove()`

```
obj = aod.core.Epoch(ID)
```

Properties:

- `ID` - (required) number of epoch within the experiment. ID must be an integer, though not necessarily consecutive, to facilitate indexing and sorting of Epochs. If you have some other non-integer specification for individual trials, your subclasses can use the `setName()` function to define that convention and assign it to `Name`.
- `startTime` - time the epoch started (`datetime`)
- `Timing` - timestamps for data acquisition during the epoch
- `Registrations` - container for `aod.core.Registration`
- `Responses` - container for `aod.core.Response`
- `Stimuli` - container for `aod.core.Stimulus`
- `Datasets` - container for `aod.core.Dataset`
- `Source` - link to `aod.core.Source` for the Epoch
- `System` - link to `aod.core.System` for the Epoch

Sealed Methods:

- `setSource(obj, system)`
- `setSystem(obj, system)`
- `add(obj, entity)` - Add a Dataset, Registration, Response or Stimulus to the Epoch
- `remove(obj, entityType, ID)` - Remove...
- `get(obj, entityType, varargin)`

7.10 aod.core.Registration (aod.core.Entity, matlab.mixin.Heterogeneous)

Any registration applied to the raw data acquired during an Epoch. For example, if you use Qiang's registration software, you can use `aod.builtin.registrations.StripRegistration` to store the metadata and specify whether you used frame or strip.

```
obj = aod.core.Registration(name, registrationDate)
```

Properties:

- `registrationDate` - date registration was performed

Methods:

- `setRegistrationDate(obj, regDate)`

Examples:

- `aod.builtin.registrations.RigidRegistration`
- `aod.builtin.registrations.StripRegistration`

7.11 aod.core.Stimulus (aod.core.Entity, matlab.mixin.Heterogeneous)

Stimuli presented during the Epoch. Optionally, the contents of `aod.core.Stimulus` can be generated automatically by passing a `aod.util.Protocol` used to create the stimulus presented during the trial.

```
obj = aod.core.Stimulus(name)
obj = aod.core.Stimulus(name, protocol)
```

Properties:

- `Calibration` - `aod.core.Calibration` used to design the stimulus, if applicable
- `protocolClass`
- `protocolName`

Methods:

- `setCalibration(obj, calibration)` - Set Calibration used, if wasn't already extracted from Protocol
- `setProtocol(obj, protocol)` - Set protocol, if wasn't already passed as an input to the constructor
- `protocol = getProtocol(obj)`

Examples:

- `aod.builtin.stimuli.ImagingLight` - Example of stimulus defined without a Protocol

7.12 aod.core.Dataset (aod.core.Entity, matlab.mixin.Heterogeneous)

Space for storing datasets acquired during the Epoch. Could be the acquired videos if you would like to store them in the HDF5 file, or some other "meta-dataset" related to the Epoch, like information about the wavefront sensing. If there is just one thing being saved, consider setting it to `Data` for consistency. If there are multiple things being saved, your subclasses can add additional properties, or ignore `Data` all together if it doesn't make sense for your purpose (empty properties will not be saved to the HDF5 file and are not required for reading the entities back in).

```
obj = aod.core.Dataset(name)
obj = aod.core.Dataset(name, data)
```

Properties:

- `Data`

Sealed Protected Methods:

- `setData(obj, data)`

7.13 aod.core.Response (aod.core.Entity, matlab.mixin.Heterogeneous)

A time-varying signal extracted from the data acquired during an Epoch. You can set `Timing` if it differs from the timing set at the `Epoch` level, otherwise it will be written as a link to Epoch's `Timing`

```
obj = aod.core.Response(name)
```

Properties:

- `Data`
- `Timing` - inherited from `aod.core.Epoch`'s `Timing` if unset

Sealed Methods:

- `setData(obj, data)`
- `addTiming(obj, timing)`

Examples:

- `aod.builtin.responses.RegionResponse`
- `sara.responses.Dff`
- `sara.responses.Fluorescence`

Keep in mind that you can set the `Parent` prior to adding the response to an Epoch. This is often needed for initializing a Response, as information from the parent Epoch might be required. All of the examples above take this approach and can be used as reference.

7.13.1 aod.core.responses.RegionResponse (aod.core.Response)

A Response associated with an instance of `aod.core.Annotation`.

```
obj = aod.core.responses.RegionResponse(parent, varargin)
```

7.14 aod.core.Analysis (aod.core.Entity, matlab.mixin.Heterogeneous)

Represents any analysis performed on entities within the Experiment. This class standardizes recording of the entities analyzed, otherwise it's a blank slate to fill in with the implementation. Placing your code within an Analysis class allows you to save both the results of the analysis and how the results were generated.

```
obj = aod.core.Analysis('name')
obj = aod.core.Analysis('name', analysisDate)
```

Properties:

- `analysisDate` - date analysis was performed

Methods:

- `setAnalysisDate(obj, analysisDate)`

7.15 Quick Reference

A quick overview of classes and constructors:

```
experment = aod.core.Experiment(name, homeFolder, experimentDate);
experment = aod.core.Experiment(name, homeFolder, experimentDate, 'Administrator', "adminName", 'System', "systemName
    ");

obj = aod.core.Source(name)

system = aod.core.System('systemName');

channel = aod.core.Channel('channelName');

device = aod.core.Device('name');
device = aod.core.Device('name', 'Manufacturer', "manufacturerName", 'Model', "modelName");

calibration = aod.core.Calibration('name', calibrationDate);
calibration = aod.core.Calibration('name', calibrationDate, "Administrator", "adminName");

epoch = aod.core.Epoch(ID);
epoch = aod.core.Epoch(ID, 'Source', value, 'System', value);

registration = aod.core.Registration('name', regDate)
registration = aod.core.Registration('name', regDate, "Administrator", "adminName")

stimulus = aod.core.Stimulus(name);
stimulus = aod.core.Stimulus(name, protocol);

dataset = aod.core.Dataset(name);
dataset = aod.core.Dataset(name, data);

response = aod.core.Response(name);
```

8 Support Classes

8.1 aod.core.EntityTypes (enumeration)

Lays out the business logic of the different entity types. Unlikely that you will need to make any changes here but it's worth checking out if you're interested in how the entities are organized.

8.2 aod.util.Factory (handle)

Factory classes simplify the creation of standardized entities. For example, if you have some standard Devices you frequently use, you could subclass Device and create a class for each or you could create a pipeline with Factory to keep OOP from getting out of control. See `sara.factories.ChannelFactory` for an example.

Factory is an abstract class with two abstract methods, a public `get()` method and a static `create()` method. The `get()` method is where your implementation goes: define the parameters you want to provide and how you will use them to return a specific entity.

```
% Instantiate the object
obj = MyFactory();
% Get the entity based on some number of inputs
entity = get(obj, varargin)
```

Sometimes you might want to have some properties associated with your Factory class, which means you need to instantiate the object before calling `get()`. If you're lazy like me and don't want to always be instantiating first, especially for factories that have no associated properties, there's the static `create()` method, which wraps the two lines above.

```
% Get the entity using create
entity = MyFactory.create(varargin);
```

Here's a blueprint for a Factory class showing `get()` and `create()`.

```
classdef MyFactory < aod.util.Factory
    methods
        function obj = get(varargin)
            % Implementation: use parameters to return a specific entity
        end
    end

    methods (Static)
        function entity = create(varargin)
            obj = MyFactory();
            entity = obj.get(varargin);
        end
    end
end
```

8.3 aod.util.Parameters (containers.Map)

This is a wrapper for MATLAB's builtin `containers.Map` class, so you if know how to use that then you know how to use `aod.util.Parameters`. The only differences are better command line display (each key and the associated values are displayed) and a standardized approach for writing to HDF5 files.

This class is central to AOData because it provides a means of defining properties for entities in a flexible way that avoids hard-coding and class incompatibility as you change the underlying code. Each core class has a `aod.util.Parameters` class. The contents of an `aod.util.Parameters` property will be written as **attributes** of its parent entity in the HDF5 file. The contents of an attribute should be reasonably small (some text, a number, maybe a short row of numbers. If needed, more info can be found in 10.3 on "Reading and Writing Entities to HDF5 files"). **In general, entity parameters are metadata and entity properties are data**. There are undoubtedly exceptions, but it's a good rule of thumb.

More info here. See `aod.core.Devices` for an example on specification of default class parameters.

There is one other use for `aod.util.Parameters` and that is providing a `containers.Map`-like data type to store files associated with an entity (e.g. `files` for `aod.core.Epoch`). Any property named `files` will be written as a group with the contents as attributes. This is a bit odd but it was the best option available as I didn't want to clutter up the entity attributes or datasets with a bunch of file names.

8.4 aod.util.template.ExpectedParameter (handle)

Expected parameters are provided to the `aod.util.ParameterManager` present in `expectedParameters` property of every subclass of `aod.core.Entity`.

AOData users do not need to directly create expected parameters and can instead provide the inputs to the `add` method of `aod.util.ParameterManager`.

8.5 aod.util.ParameterManager (handle)

The ParameterManager class provides users with an opportunity to specify which parameters the entity should have. Each AOData class inheriting from `aod.core.Entity` has a dependent property called `expectedParameters`, which is set by the output of the function `getExpectedParameters`.

No parameters are set in `aod.core.Entity`; however, several core classes define them. For example, `aod.core.Device` has two expected parameters (Manufacturer and Model), which are defined as follows:

```
function value = getExpectedParameters(obj)
    % Subclasses should always call the inherited method first, putting their
    % superclass after the @.
    value = getExpectedParameters@aod.core.Entity(obj)
    % Add two new parameters
    value.add('Manufacturer', [], @ischar, 'Manufacturer of the Device');
```

```
    value.add('Model', [], @ischar, 'Model of the Device');
end
```

When you create a device, you will see both listed in the `expectedParameters` property

```
device = aod.core.Device("TestDevice");
disp(device.expectedParameters)

ParameterManager with 2 parameters:
    Model: "Default = [], Validation = @ischar, Description = []"
    Manufacturer: "Default = [], Validation = @ischar, Description = []"
```

8.6 aod.util.Protocol (handle)

Protocols define how stimuli are constructed from a basic set of parameters and how they are transformed into the format the stimulator requires. See 11.3 on "Creating A Protocol" for a detailed explanation.

```
obj = aod.util.Protocol(calibration, varargin)
```

Abstract Methods:

- `sampleRate` - the rate data is acquired (Hz)
- `stimRate` - the rate stimuli are presented (Hz)

Properties:

- `Calibration`
- `dateCreated`

Abstract Methods:

- `stim = generate(obj)`
- `writeStim(obj, fName)`
- `value = samples2sec(obj)`

Methods:

- `value = sec2pts(obj)`
- `value = pts2sec(obj)`
- `value = sec2samples(obj)`
- `value = samples2sec(obj)`

How is `aod.util.Protocol` different from `aod.core.Stimulus`? A Protocol defines how the stimulus was created and provides a record of the parameters used. A Stimulus is what actually happened during the experiment (e.g. for the LEDs we have two outputs: a JSON file with the LED voltages at each stimulation interval which is 2 ms and a CSV file with the voltages at each frame along with the timing of each frame).

8.7 aod.util.FileReader (handle)

FileReader classes provide a designated, contained location for reading files. Create one per metadata file you want to read in, then you can easily add/remove/alter how the files are read without disrupting any other code. The files created during an experiment change with the type of trial you run and with subclasses of `aod.core.Epoch`, you can define exactly which are needed.

Without knowing the contents of your files and the information you want to extract, how can `aod.util.FileReader` help you read files in a robust way, beyond just providing a contained location for reading each file?

8.8 aod.util.FileManager (handle)

The `aod.util.FileManager` class houses pipelines for automatically identifying specific experiment files. Not every file need to be assigned through `aod.util.FileManager`, but if you have a large number of files generated per Epoch, for example, it can be useful to segregate file name buil

Examples:

- `sara.util.EpochFileManager`

8.9 aod.util.InputParser (inputParser)

A wrapper for the MATLAB's builtin `inputParser` class that simplifies setting three defaults that would otherwise need to be retyped with every class that has optional parameter inputs. If you know how to use `inputParser`, then you know how to use `aod.util.InputParser` (see 13 for relevant documentation - mostly you need `addParameter`). Why are these defaults important? You don't need to know this to use AOData, but in case you're wondering:

1. `KeepUnmatched=true` - This is the most important. When you have a subclass that takes a variable number of inputs (`varargin`) as an input `inputParser` and also want to pass those same variable inputs to the parent class's `inputParser`, you need `KeepUnmatched=true` to avoid errors at each stage for unrecognized parameters. The downside is incorrect parameters won't error, instead they will just be ignored.
2. `PartialMatching=false` - Disabling partial matching was a hard choice but it can cause fatal errors for classes where a text input is the required positional parameter before `varargin` is a char/string input. If it partially matches one of the parameters, input parsing won't work. Given that most core classes do have a text input (`Name`) prior to `varargin`, some of those errors were going to be inevitable.
3. `CaseSensitive=false` - Having two parameters to the same class that are identical other than their case (e.g. `'MyParam'` and `'myparam'`) is bad practice and should never be done, so we might as well make the parsing case-insensitive.

9 Persistent Interface

The largest investment for AOData tools is creating customized subclasses for writing the data into an HDF5 file. The interface for reading files back in requires no additional customization. Load your file in with:

```
experiment = loadExperiment(hdfName);
```

The organization mirrors that of the core classes used to build an HDF5 file (Experiment contains Analyses, Calibrations, Epochs, Segmenations, Sources and Systems... Systems contain Channels and Channels contain Devices, etc). Each entity has a `parameters` property containing all the parameters specified when building the experiment. Entities will have the same properties defined in the core classes and custom subclasses. All you need to worry about is navigating through the experiment to find the information you need.

```
% Get all the calibrations
calibrations = experiment.Calibrations

% Get the first epoch
epoch1 = experiment.Epochs(1);

% Check out the parameters for the second epoch:
experiment.Epochs(2).parameters

% Get the responses for the first 5 epochs
experiment.Epochs(1:5).Responses
```

How does this work? The long story short is that there are dedicated "persistent" versions of each core class specifically for reading information back from an HDF5 file (e.g. for `aod.core.Epoch` there is a corresponding `aod.persistent.Epoch`). These classes ensure the core components of AOData maintain their structure and relationships. Beyond that, any additional datasets found in an entity's group within the HDF5 file are added as dynamic properties (also known instance properties) using the `dynamicprops` superclass.

Reading the HDF5 files is fast to begin with and made faster by "lazy loading" which basically means that entities are not read from the HDF5 file until requested. Once requested, they are added to a cache so they only need to be read in once. This all happens behind the scenes.

9.1 Modifying Persisted Experiments

You can continue to edit your HDF5 file from the persistent interface. The same functions used to build the experiment in the core interface are still there, like `setParam(), removeParam()`. Changes made will immediately be reflected in the underlying HDF5 file.

- `setParam(obj, paramName, paramValue)`
  Change the value of an existing parameter or create a new one.
- `rmParam(obj, paramName)`
  Remove an existing parameter

- `deleteEntity(obj)`
  Delete an existing entity

9.2 Customizing the Persistent Interface

Data read in from an HDF5 file is meant to serve as the foundation for user-defined workflows. These can be custom scripts, functions or classes. Any function that takes an entity from the core framework can also accommodate the comparable entity from the persistent interface.

One optional tactic is to customize the persistent interface. For example,

10 HDF5 File Details

High-level summary: When you write an entity to an HDF5 file, the entity will be written as a **group**. Public properties will be written as **datasets** within the group. Anything in a `aod.util.Parameters` object will be written as an **attribute** of the group. Properties referring to other entities (e.g. Parent) are written as **references**. Several entities also contain default groups which are *containers* for other groups (e.g. Experiments have Calibrations, a container group for all the individual Calibration groups).

10.1 Entity locations within the HDF5 file

Each entity type lives in a specific part of the HDF5 folder. The `aod.core.Experiment` group is written in the root group. Virtually all other entities are found within container groups which are created by default for the entities below. For example, when you add the Experiment to an HDF5 file, 5 default container groups are created within the Experiment group. When you add calibrations to the Experiment, each is a new group within `/Experiment/Calibrations`.

```
Experiment
├─ Sources - container for all Sources within the Experiment
├─ Systems - container for all Systems within the Experiment
├─ Calibrations - container for all Calibrations within the Experiment
├─ Epochs - container for all Epochs within the Experiment
├─ Annotations - container for all Annotations within the Experiment
└─ Analyses - container for all Analyses within the Experiment
```

When you add an Epoch to the Experiment, four default container groups are created

```
Epochs
├─ Datasets - container for all Datasets within the Epoch
├─ Registrations - container for all Registrations within the Epoch
├─ Responses - container for all Responses within the Epoch
└─ Stimuli - container for all Stimuli within the Epoch
```

For each System added to the Experiment, one default container group is created for Channels. For each Channel added to a System, one default container is created for Devices.

```
Experiment
└─ Systems
   └─ System1
      └─ Channels
         └─ Channel1
            └─ Devices
```

Finally, sources are *nestable* and can contain other sources. Accordingly, each Source will have a Sources container. This allows you to specify Sources at whatever level of detail you want. For an example of how this could work, see

the Source hierarchy detailed in 7.3

```
Experiment
└ Sources - container for all top-level Sources within the Experiment
  └ Source1
    └ Sources - container for all Sources within the Source
```

10.2 Naming Entities for HDF5 Files

The first and most important point to address is what each entity group will be named. Much like a normal file directory, HDF files can't have two groups with the same name at the same level. To ensure you don't overwrite other entities, there needs to be a standardized way of naming each group. This is why all entities have the `Name` and `label`. Sometimes you can come up with a unique name based on the entity parameters. For example, you can automate naming of entities in the Device class `aod.builtin.devices.BandpassFilter` from the `Wavelength` and `Bandwidth` parameters. If you specify a `Name`, that is always used as the group name. If `Name` is empty, `label` is used.

Setting `Name` is required in the constructors of the core classes. If you *really* don't want to name one of these, just pass an empty for that argument (`[]`).

10.3 Reading and Writing Entities to HDF5 Files

1. `writeExperimentToFile(hdfName, experiment)`
2. `writeEntity(hdfName, entity)`
3. `write(hdfName, hdfPath, dsetName, data)`,
   `read(hdfName, hdfPath, dsetName, data)`,
   `writeAttributesByType()`,
   `readAttributesByType()`
4. `aod.h5.HDF5`

When writing an entity, properties are handled as follows:

- Empty properties are not written
- Transient properties and properties without public GetAccess are not written. Dependent properties are written unless they are also marked Hidden.
- Properties referring to other entities (that are not containers, e.g. `Epochs` in `aod.core.Experiment`) are written as link within the entity group. Data is written in an order that ensures linked entities are written before the entity containing their link. When adding links to custom subclasses, keep in mind the order of entities in the AOData object model, as this is the order they are written to the HDF5 files.
- All other properties are written as datasets within the entity group. Virtually all MATLAB data types are supported, except multilevel `struct`. Most other data types translate well, with a few quirks:
  - HDF5 doesn't have a `datetime` class, so these properties are written as a text dataset, with attributes specifying the format to aid conversion when reading the HDF5 file back into MATLAB.
  - For an `enumeration` property, the enumerated type is written as a string dataset. The name of the enumeration class is written as an attribute of the dataset. This works if the enumeration class has a static method called `init()` that takes the enumerated type name as a char and returns the enumerated type (see `aod.core.EntityTypes` for an example). HDF5 does have enumeration support, but this works just as well.
  - Several entities have a `files` property of class `aod.util.Parameters`. The contents are written as attributes of a text dataset with some placeholder text.
  - If `Timing` is present but empty and the parent class has `Timing`, a link will be written to that dataset. This allows, for example `aod.core.Response` to inherit timing from `aod.core.Epoch`
  - Data of class `table` are converted to as `struct`, then written as compound data types

  Each dataset is written with an attribute describing the original MATLAB type to ensure the data is read back in appropriately.
- The contents of `aod.util.Parameters` classes are written as attributes of the entity group.

- Attributes should be small, consistent with them representing metadata rather than the data itself - some text, a number, maybe a row of numbers. I'm not sure what the hard size limit is (if there is one) but if you're writing some ND array it's worth thinking about whether it truly qualifies as metadata.
- Arrays of more than one `string` and `struct` cannot be written as attributes.
- You can write `datetime` but it will be converted to `char`, without the guidelines described above for `datetime` datasets enabling accurate conversion back to `datetime` when read into MATLAB.

10.4 HDF5 Classes

10.4.1 aod.h5.EntityManager (handle)

A well-known shortcoming of the file directory organization within HDF5 files is the difficulty indexing and searching contents. The solution for AOData is `aod.h5.EntityManager`. Each entity has a unique identifier (UUID). As a result groups with a UUID attribute must reflect entities within the persistent hierarchy. When you need to search through your HDF5 file for some attribute/dataset/group, use the entity groups as the base for your search, then search the paths returned by the EntityManager.

```
obj = aod.h5.EntityManager('myfile.h5');
% Get a table listing each UUID, the corresponding class and the path within the HDF5 file
T = table(obj);
```

10.4.2 aod.h5.HDF5

The goal of AOData is to abstract the messy details of HDF5 writing, so you never have to worry about it in writing your own code. If you do need delve into HDF5 writing for some reason, this class provides the interface for mapping data to and from HDF5 files. MATLAB has high-level functions (`h5write`, `h5read`, `h5info`, `h5readatt`, `h5writeatt`); however, they still require a decent understanding of how HDF5 I/O works. Other aspects aren't supported by the high-level functions and require directly working with the C API. A consistent interface to all these functions is provided by `aod.h5.HDF5`. This is the back-end for the even higher-level HDF5 functions provided by AOData (`aod.h5.writeEntity()`, `aod.h5.write()`, `aod.h5.read()`).

Every method of `aod.h5.HDF5` is Static. There's no constructor or properties, it's just a collection of related functions in a single location.

11 Using AOData

11.1 Class Design Principles

Before delving into the specifics of getting set up, first let's just take a general principles look at the design of just one of the classes, `aod.core.Device`, and how it should be used. First, how do we use devices in an experiment? It turns out we can use them in many ways, often for slightly different purposes sometimes even within a single experiment. For example, we can use the Toptica to image fluorescence, in which case it's part of a channel that involves a PMT and a pinhole. We can also use it to provide visual stimuli to the cones and in this case there is no PMT and pinhole. So the use of a device depends on the Channel or light path it's being used in. This is why Devices fall underneath Channels in the persistent hierarchy. An important point is that Devices depend on Channels but not vice versa. The `aod.core.Channel` class is carefully designed to avoid any code that requires knowing something specific about the contents/identity of a Device. This enforces a *unidirectional dependency* which is good practice in software design because it limits unintended side-effects, thus keeping you from accidentally breaking your code.

Second, we can use the same system in different ways. In the example above, we're using one system for anatomy and for physiology. The channels involved and the devices within them are different, even though it's technically the same "System". The different "Channels" and their devices make sense under a larger group that defines how the system is being used. This is why channels fall under systems in the persistent hierarchy and why there is the capacity for you to specify more that one System (they're more like "System Configurations" but that was a long class name).

How is this implemented? In `aod.core.EntityTypes`, there is a function called `validParentTypes()`. There is also a method called `setParent()` which takes an input, confirms it is a subclass of the valid parent types defined in `aod.core.EntityTypes`.

So that was the logic behind the location of `aod.core.Device`. Now what do we put in the class? When designing the classes, the overarching goal was to standardize behavior while being as generic and unrestrictive as possible. That being said, the purpose here is archiving the many aspects of our experiments that are relevant to interpreting

the results, like the relationship between Devices, Channels and Systems described above. I took the perspective of asking What does someone else need to know to about this device to understand it 10 years from now? Then I asked whether that information could apply to every single device and if the answer was no, it's not included in `aod.core.Device`. I settled on two parameters: `Manufacturer` and `Model`. I considered serial number as well, but that was only useful for some devices (e.g. a PMT) and completely unnecessary for others (e.g. a pinhole).

The next question is just how important `Manufacturer` and `Model` is for a device. It's good information to have and we should have it for all devices, but should I force people to provide as a required input? I decided that was unnecessarily restrictive - a 20 micron pinhole is a 20 micron pinhole even if I don't know for sure that you bought it from ThorLabs. Instead both parameters are optional inputs handled with `aod.util.InputParser` (see below).

```
classdef Device < aod.core.Entity

    function obj = Device(name, varargin)
        obj = obj@aod.core.Entity(name);

        % Define the two parameters decided above, making them optional
        ip = aod.util.InputParser();
        addParameter(ip, 'Model', [], @ischar);
        addParameter(ip, 'Manufacturer', [], @ischar);
        parse(ip, varargin{:});

        % Add to the entity's parameter property
        obj.setParam(ip.Results);
    end
end
```

So that's how `aod.core.Device` is created (more or less; behind the scenes there is a lot going on, but you don't have to deal with it). By itself, `aod.core.Device` isn't particularly useful, it's more of a solid foundation for subclasses.

What subclasses should you make? The goal here is to add information important for interpreting the experimental data. It's up to you to decide what that means. Personally, I'm not logging every single mirror in the light path - that's what the system diagram is for - instead, I'm putting the light sources, PMTs and parts of the system that we change frequently (for FAOSLO, this is filters, pinholes and NDFs). My reasoning is that when I put an NDF in front of a light source or test out different band-pass filters in front of the PMT, I'm making changes that generally are not included in the system diagram and only I know about.

Let's say we want to make a subclass for dichroic filters, such as `aod.builtin.devices.DichroicFilter`. What are the defining features of a dichroic filter? Basically, all of them have a cutoff wavelength and are either high- or low-pass. These are good parameters to specify: `Wavelength` and `Bandwidth`. Unlike `Manufacturer` and `Model`, these are essential to understanding the bandpass filter. There's no point in going to the trouble of specifying a bandpass filter in your HDF5 file if you're not going to include this information. Accordingly, the should be required parameters, specified with `addRequired` to `aod.util.InputParser`.

There could be some data associated with the bandpass filter, such as the transmission spectra. This would be a good property: `transmission`. However, we can understand the filter without it or look it up online using `Model` and `Manufacturer`, so `transmission` shouldn't be required. If you want to add it, you can do it after constructing the object using `setTransmission()`.

```
classdef BandPassFilter < aod.core.Entity

    properties (SetAccess = protected)
        transmission
    end

    methods
        function obj = BandPassFilter(varargin)
            obj = obj@aod.core.Device([], varargin{:});

            ip = aod.util.InputParser();
            addParameter(ip, 'Wavelength', [], @isnumeric);
            addParameter(ip, 'Bandwidth', [], @isnumeric);
            parse(ip, varargin{:});

            obj.setParam(ip.Results);
        end

        function setTransmission(obj, transmission)
```

```
            assert(isnumeric(transmission), 'Transmission must be numeric!');
            obj.transmission = transmission;
        end
    end

    methods (Access = protected)
        function value = getLabel(obj)
            value = sprintf('%ux%uBandpassFilter',...
                obj.getParam('Wavelength'), obj.getParam('Bandwidth'));
        end
    end
end
```

11.2 Robust file reading

The option to store not only the output, but also the file name and how the file was read.

11.3 Creating a protocol

Each subclass of `aod.util.Protocol` must define two properties: `sampleRate`, the rate data is sampled in Hz, and `stimRate`, the rate stimuli are presented in Hz.

The stimulus is created and written with the following methods:

1. `calculateTotalTime(obj)`: determine how the total stimulus time is calculated
2. `stim = generate(obj)`: calculates normalized stimulus values (between 0 and 1). This method separates the space/time specification of a stimulus from details about how it gets mapped to the stimulation software. If Qiang changes how stimuli are provided to the stimulation software in the future, you won't need to change this part of your code.
3. `stim = mapToStimulator(obj)`: Apply any alterations necessary to convert the output of generate into whatever your stimulator requires. For example, apply nonlinearities, conversions to specific data types or turn your normalized stimulus into separate signals for 3 LEDs.
4. `fName = getFileName(obj)`: Defines how your stimulus file name will be constructed from the parameters. Optional
5. `writeStim(obj, fileName)`: Outputs the calculated stimulus as needed for imaging software. If you do not provide the fileName input, the output of `getFileName()` will be used.

Below is a template for defining a new protocol:

```
classdef MyProtocol < aod.util.Protocol

    % Define properties specific to this protocol
    properties
        myProp1
        myProp2
    end

    % Set the following two properties required of all protocols
    properties (Access = protected)
        sampleRate = 25            % Frequency that data is sampled (Hz)
        stimRate = 500             % Frequency that stimuli are presented (Hz)
    end

    methods
        function obj = MyProtocol(varargin)
            obj = obj@aod.util.Protocol(varargin{:});

            % Parse properties specific to this protocol
            ip = aod.util.InputParser();
            addParameter(ip, 'MyProp1', 1, @isnumeric);
            addParameter(ip, 'MyProp2', 50, @isnumeric);
            parse(ip, varargin{:});

            obj.myProp1 = ip.Results.MyProp1;
            obj.myProp2 = ip.Results.MyProp2;
        end

        function stim = generate(obj)
```

```
            % Your code here
        end

        function stim = mapToStimulator(obj)
            stim = obj.generate();
            % Your code defining how stim is mapped to the stimulator, if necessary
        end

        function writeStim(obj, fName)
            % Define how the stimulus is written to a file
            if nargin < 2
                fName = obj.getFileName();
            end
            stim = obj.mapToStimulator();
        end
    end
end
```

For example, here's a protocol called `Steps` for the LEDs on the 1P primate system. Data is sampled at 25 Hz (`sampleRate`) and the update rate for the LEDs is 500 Hz (`stimRate`). This protocol begins at a baseline value (`baseIntensity`) for a set amount of time (`preTime`), increases (`contrast`) for a set amount of time (`stimTime`), then returns to the baseline value (`baseIntensity`) for a set amount of time (`tailTime`), This creates an achromatic step in contrast.

```
classdef Step < aod.util.Protocol

    % Protocol-specific, all public access properties are saved
    properties
        ledMaxPowers        % Max powers for each LED, [R G B]
        preTime             % time before step is presented (sec)
        stimTime            % time step is presented (sec)
        tailTime            % time after step is presented (sec)
        contrast            % change in value during step (-1 to 1)
        baseIntensity       % baseline value for the LEDs (0 to 1)
    end

    % These properties must be set by subclasses
    properties (SetAccess = protected)
        sampleRate = 25
        stimRate = 500
    end

    % These properties are derived and don't need to be saved
    properties (Access = private)
        amplitude
    end

    methods
        function obj = ContrastStep(ledMaxPowers, varargin)
            obj = obj@aod.util.Protocol(varargin{:});

            obj.ledMaxValues = ledMaxValues;

            % Parse optional key/value inputs
            ip = inputParser();
            addParameter(ip, 'PreTime', 1, @isnumeric);
            addParameter(ip, 'StimTime', 5, @isnumeric);
            addParameter(ip, 'TailTime', 1, @isnumeric);
            addParameter(ip, 'BaseIntensity', 0.5, @isnumeric);
            addParameter(ip, 'Contrast', 1, @isnumeric);
            parse(ip, varargin{:});

            obj.preTime = ip.Results.PreTime;
            obj.stimTime = ip.Results.StimTime;
            obj.tailTime = ip.Results.TailTime;
            obj.baseIntensity = ip.Results.BaseIntensity;
            obj.contrast = ip.Results.Contrast;

            % Derived properties
            if obj.baseIntensity == 0
```

```
                obj.amplitude = obj.contrast;
            else
                obj.amplitude = (obj.baseIntensity*obj.contrast) + obj.baseIntensity;
            end
        end

        function stim = generate(obj)
            % Define how the stimulus is created
            totalPts = obj.sec2pts(obj.totalTime);
            stim = obj.baseIntensity + zeros(1, totalPts);

            prePts = obj.sec2pts(obj.preTime);
            stimPts = obj.sec2pts(obj.stimPts);

            stim(prePts+1:prePts+stimPts) = obj.amplitude;
        end

        function stim = mapToStimulator(obj)
            % Define mapping to stimulator, if necessary
            stim = obj.generate();
            ledValues = data .* obj.ledMaxPowers';
        end

        function fName = getFileName(obj)
            if obj.baseIntensity == 0
                fName = 'intensity_increment_';
            elseif obj.contrast > 0
                fName = 'contrast_decrement_';
            elseif obj.contrast < 0
                fName = 'contrast_increment';
            end
            fName = [fName, sprintf('_%up_%uc_%us_%ut',...
            100*obj.baseIntensity, 100*obj.contrast,...
            obj.stimTime, obj.totalTime)];
        end

        function writeStim(obj, fName)
            % Define how the stimulus is written to a file
            if nargin < 2
                fName = obj.getFileName();
            end
            ledValues = obj.mapToStimulator();
            makeLEDStimulusFile(fName, ledValues);
        end
    end

    methods (Access = protected)
        function value = calculateTotalTime(obj)
            % Define how the total stimulus time is calculated
            value = obj.preTime + obj.stimTime + obj.tailTime;
        end
    end
end
```

You might notice a this protocol used few functions and properties that weren't defined. These are inherited from `aod.util.Protocol`. For example, several methods are defined for converting between seconds and samples (`obj.sec2samples(sec)`, `obj.samples2sec(samples)`) which use the value you set for `sampleRate`. To convert between seconds and the stimulator's timing, there's `obj.sec2pts(sec)` and `obj.pts2sec(pts)` which use your value for `stimRate`. There's also a property `totalTime` which calls the function `calculateTotalTime()`. See the `aod.util.Protocol` code for more.

12 Miscellaneous FAQs

What to do when a constructor needs information from the parent entity

Supply the parent entity as an input to the constructor and assign it within the constructor with `setParent()`. See `aod.builtin.registrations.StripRegistration` or `aod.builtin.responses.RegionResponse` for an example.

How to keep UUID consistent for an entity that is shared between experiments
By default, UUIDs are randomly generated in the `aod.core.Entity` constructor and will change each time you instantiate an object (even if every other aspect is the same). To set a consistent UUID, use the `setUUID()` function, which is provided for Source, System, Channel and Device. See `sara.factories.SubjectFactory` for an example.

13 Code Documentation

Here I've picked out a few links to MATLAB's documentation that are particularly relevant for subjects in AOData. This is typically just a subset of the documentation MATLAB provides for each concept, so if you have a question that these links don't answer, scroll to the bottom of the documentation page. There MATLAB provides links to other related documentation topics that will likely address your question.

- Object Oriented Programming (specific to MATLAB)
  - User-Defined Classes - start here
  - Class Constructor Methods
- Properties
  - Property Definition
  - Property Attributes
- Methods
  - Method Attributes
- Inheritance
  - Hierarchies of Classes - Concepts
  - Subclass Syntax
  - Design Subclass Constructors
- Packages
  - Packages Create Namespaces
  - Import Classes
- Input Parsing
  - Parse Function Inputs
  - inputParser
  - varargin
- Advanced
  - Comparison of Handle and Value Classes
  - dynamicprops, Adding Dynamic Properties to an Instance
  - matlab.mixin.Heterogeneous, Handle-Compatible Classes and Heterogeneous Arrays
  - Enumerations, Named Values

References

[1] Dragly, S.A. et al (2018) Experimental directory structure (Exdir): An alternative to HDF5 without introducing a new file format. *Frontiers in Neuroinformatics*, 12, 16

[2] Fold, M. et al (2011) An overview of the HDF5 technology suite and its application. *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 36-47

[3] Greenfield, P., Droettboom, M., & Bray, E. (2015). ASDF: A new data format for astronomy. *Astronomy and Computing*, 12: 240-251

[4] Osterhout, J. (2018) A Philosophy of Software Design. 2nd edition

[5] Rubel, O. et al (2019) NWB:N 2.0: An accessible data standard for neurophysiology. *bioRxiv*

[6] Teeters, J.L. et al. (2015) Neurodata Without Borders: Creating a common data format for neurophysiology. *Neuron*, 88, 629-634