AOData Walkthrough

Sara Patterson (last updated 2023-03-12)

The purpose of this tutorial is to demonstrate, step-by-step, how to get started with AOData. This walkthrough will introduce you to AOData and the next (CustomizingAOData.mlx) will take you through the process of setting up a custom AOData package. Before beginning, the walkthrough you should follow the instructions in the documentation for **Installation** and for adding AOData to your search path.

This is a MATLAB LiveScript. Instead of pressing "Run", you should step through it, running the code blocks one-by-one (Ctrl+Enter), exploring the output and reading the associated text. I personally learn by doing and I want AOData to be accessible to people who aren't super comfortable programming, so I've included some little exercises. They would be especially helpful if you aren't super comfortable with MATLAB or object oriented programming in MATLAB. Or if, like me, you start to zone out when confronted with this much text/information to read through. There's a second file "AODataWalkthrough_WithAnswers.mlx" if you get stuck on something.

Finally, the walkthrough is meant to be completed in order, and later sections will assume that you've read the previous sections.

Documentation (where to find help)

This walkthrough is meant to be a companion to the Documentation PDF as well as the in-code documentation (that is, documentation written as comments within the code). The PDF can be reached at the link above and, to access the in-code documentation, you can either use **`doc`** or **`help`** (or just open the code itself with **`edit`**).

```
% This will open up a detailed description of the methods and properties of
% the aod.core.Experiment class. Useful for an overview. You can always
% click on a method to get information on how to use it.
doc aod.core.Experiment
```

```
% For quick details on how to use a method/function, using "help" will
% print usage information to the command line.
help aod.core.Experiment.setHomeDirectory
```

To get a full sense of AOData, check out the accompanying documentation referenced at each step. At this point, pause and read the PDF Documentation's section on "Using the Documentation".

If you aren't comfortable with MATLAB or object oriented programming, opening up the underlying code and checking out how the underlying code is implemented can be helpful too.

```
% Open up the underlying code for aod.core.Experiment in the editor
edit aod.core.Experiment
```

Notes on Object Oriented Programming. AOData relies heavily on classes (files starting with `classdef`). The **properties** are listed at the top, followed by the **methods** which contain the **functions** available to the class. The first method is always the **constructor** (i.e. it "constructs" the object).

Scroll through the rest to see the various functions you can use with the class, particularly the "Public" methods. Functions are organized into `methods` blocks.

- "Private" methods "`methods (Access = private)`" can only be used by code within the class, never from the command line or from subclasses.
- "Protected" methods "`methods (Access = protected)`" can be accessed by code within the class and code within subclasses. They can even be redefined or redefined by subclasses, but not from the command line.
- "Public" methods "`methods (Access = public)`" are like protected methods but you can also use them from scripts and the command line. These are the ones you'll be working with the most and that deserve the most attention.

Keep in mind that classes may be inheriting methods from parent classes (in the case of `aod.core.Experiment` that is `aod.core.Entity`). If you can't find the method you're looking for, check the **superclass** (it's the class name on the first line after "<").

```
classdef Experiment < aod.core.Entity
    % A class called Experiment with superclass aod.core.Entity
end
```

AOData Basics

AOData is a framework for managing experimental data, metadata and code. In other words, AOData is meant to provide a strong, standardized foundation geared towards maximizing reproducibility, accessibility and collaboration. It's a platform for end-users to customize for their individual experiments and workflows. At this point, make sure you've read the documentation sections on **"AOData Object Model"** and how it maps to an HDF5 file. Opening up an example file in **AODataViewer** and clicking around will be helpful too.

You will need to ask a few questions about your data:

1) Conceptually, how does your experiment map onto the AOData object model?

- What are the different types of Calibrations you perform (i.e. power measurements, AO calibrations, PMT optimizations)?
- Is there any information you need to log about the status of your System (i.e. ad hoc additions like NDFs and filters not recorded in the system diagram, serial numbers of devices like PMTs, etc)?
- How many different types of imaging do you do and are they performed in different videos (a.k.a. Epochs)?
- Are there important relationships between entities that cut across the AOData object model and are necessary for understanding the data? For example, are Epochs linked to specific Sources?

2) Logistically, where and how is the information stored?

- Is it hand-written in an imaging log or saved as a file?
- What are the relevant files, where are they located in an experiment folder, what file formats, and how are they tied to the object they describe (e.g. saving the video number in the file name ties it to a specific Epoch)?
- Does any of this information rely on custom code (either to be generated or interpreted)? For example, did you write any code to generate a Stimulus or process a Calibration?

3) Programmatically, what code do you need to write to get that information into AOData's object model?

- What custom classes do you need to create?
- How will you read in your metadata files?
- Of the information related to an entity, what is best suited as metadata (an attribute in HDF5) and what is best-suited for an HDF5 dataset?

This walkthrough is focused on helping you with #3 while providing a familiarity with AOData's functionality that will help with #1. Ultimately, only you know the answers to #2. As you work through the walkthrough, keep an eye towards how the code and concepts introduced could applied/tailored to your own experiments.

Initialization

The first step is initializing AOData. If you haven't already, simply run the line below:

```
AODataManagerApp();
```

A GUI called **AODataManagerApp** should have opened up. There are 3 tabs which reflect the 3 settings AOData adds to your MATLAB user preferences:

1. BasePackage: This is the location on your computer of the main AOData folder. If you move the folder for some reason, you'll want to re-run **`initializeAOData()`**
2. SearchPaths: This a list of the folders containing packages (i.e. the folder containing the first +folder. All subfolders that are packages will be added too). Some parts of AOData require knowledge of all the available classes and custom subclasses. For now, AOData's "src" folder should be already be added as this contains the "+aod" package. The subfolders that are packages do not need to be added here (e.g. "+core", "+persistent").
3. GitRepos: AOData tracks the commit IDs of any folders added here that are git repositories. When you create your own package, if you choose to track it with git (strongly recommended!), then you will add your git repository folder here.

You can return here at any time to add/remove custom packages and repositories. We'll discuss this more in the customization tutorial.

Experiment (`aod.core.Experiment`)

To familiarize you with process of mapping an experiment to AOData, we'll walk through an example experiment. First, we'll do a generic experiment without any real data attached. We'll just use the base classes to get a feel for how those work. In the next tutorial, we'll going to start over with a real dataset, using the core classes and developing custom classes (which will streamline the process and customize it to the underlying dataset).

First let's create an Experiment object. There are 3 required inputs: the experiment's name, the file path to the experiment folder and the experiment date). There are also two optional parameters (Administrator and Laboratory) which you are free to edit if you'd like.

```
% Because we won't be using real data, we can use the tutorial folder as
% the experiment folder.
experimentPath = fullfile(getpref('AOData', 'BasePackage'), 'tutorials');

% Create the experiment
EXPERIMENT = aod.core.Experiment('MyDemoExperiment', experimentPath,
getDateYMD(),...
    'Administrator', "Your Name Here", 'Laboratory', "Your Lab Here");
```

How did we know the inputs to provide? You can use `help aod.core.Experiment`, but to get you more comfortable with subclasses, open up `aod.core.Experiment` and find the constructor. You should see the 3 required inputs listed:

```
classdef Experiment < aod.core.Entity

    methods
        % The constructor is a function with the same name as the class
        function obj = Experiment(name, filePath, expDate, varargin)
            % It begins with a call to the superclass, aod.core.Entity
            obj@aod.core.Entity(name, varargin{:});
        end
    end
end
```

You can see the 3 required inputs and a fourth `varargin`. This allows you to provide a variable number of additional arguments (could be 0, could be 4 extra arguments as in the example above). In most cases, the variable inputs should be **key/value pairs**. For example, in the Experiment created above, `'Administrator'` is the **key** and its **value** is `'Sara Patterson'`. These optional parameters are assigned to the `parameters` property and written as attributes of the entity's HDF5 group.

How do you know which optional parameters to define? Those are determined by a protected function defined in `aod.core.Entity` called `getExpectedParameters()`. Subclasses can modify it to add new parameters (or remove inherited ones).

```
classdef Experiment < aod.core.Entity

    methods (Access = protected)
        function out = getExpectedParameters(obj)
```

```
                % This line runs the superclass's version of the method first,
                % so we can get any parameters defined there. aod.core.Entity
                % does not define custom parameters but it does create the
                % expectedParameters object, so always call the superclass.
                out = getExpectedParameters@aod.coreEntity(obj)

                % Subclasses can add their own parameters, as below
                out.addParameter('Administrator', [], @isstring,...
                    'Person(s) who performed the experiment');
                out.addParameter('Laboratory', [], @isstring,...
                    'Lab where the experiment was performed');
            end
        end
    end
```

We'll get into expected parameters definition later on. For now, know that each has a name and, optionally, a default value, a validation function and a description. Also know that you can get a list of the potential parameters from the `expectedParameters` property of any AOData entity

```
disp(EXPERIMENT.expectedParameters)
```

Take a look at what's inside `EXPERIMENT`... If you're new to MATLAB, try both `disp` and `openvar`

```
disp(EXPERIMENT)
```

You will see that the 3 required inputs are properties (`Name`, `homeDirectory`, `experimentDate`). Those will map to HDF5 datasets. The parameters, which map to HDF5 attributes, are within the `parameters` property:

```
disp(EXPERIMENT.parameters)
```

This `parameters` differs from `expectedParameters` above - one tells you which parameters the class *should* have and the other tells you the values of the parameters the object actually has.

Methods shared by all entities

All of the core classes inherit from `aod.core.Entity`. Inheritance means they get all the properties and methods defined by `aod.core.Entity` and can add their own as well. The PDF documentation describes these in depth. The nice thing about inheritance is that it means many aspects of the core classes are identical - if you understand how to use **`aod.core.Experiment`**, then you understand a lot about how to use, for example, **`aod.core.Device`**. Let's go through some of these shared methods:

Accessing parameters. As described above, parameters are specified as **key/value pairs**. That is, each entry has a key (e.g. "Administrator") and a value ("Your Name Here"). For more information on the underlying data structure, check out the PDF's section on "**`aod.util.Parameters`**", then follow the linked resource for learning about MATLAB's documentation for the MATLAB **`containers.Map`** class.

You can get the value of a parameter with **`getParam()`** method, providing the key as the input.

```
% Access the value for the "Laboratory" parameter
disp(EXPERIMENT.getParam('Laboratory'))
```

Editing parameters. You can also add additional parameters to an entity, beyond those specified in `expectedParameters`, using **`setParam()`**. It's good to use the expected parameters where possible to specify the metadata that is important to understand a given entity because they promote consistency. Many data management systems require objects to be fully specified ahead of use - I thought this could be overly restrictive and cumbersome, so the option for *ad hoc* parameters is provided.

```
% Check out the existing parameters
disp(EXPERIMENT.parameters)
% Add a parameter and confirm it is now in "parameters"
EXPERIMENT.setParam('MyNewParam', 1);
disp(EXPERIMENT.parameters)
```

You can re-run **`setParam()`** to change the value of a parameter

```
EXPERIMENT.setParam('MyNewParam', 2);
disp(EXPERIMENT.parameters)
```

You can also specify multiple parameters at once (either as a series of key/value inputs, a `struct` or containers.Map).

```
EXPERIMENT.setParam('ParamTwo', [1 2 3], 'ParamThree', "hello");
disp(EXPERIMENT.parameters)
```

You can remove a parameter with **`removeParam`**`('MyNewParam')`, get the value of a parameter with **`getParam`**`('MyNewParam')` or ask whether a parameter exists with **`hasParam`**`('MyNewParam')`. If you want to change a parameter, just rerun `setParam()` with your different value.

Try changing, removing and re-adding `'MyNewParam'` below. Use `disp` to check the results. If you get stuck, try the `help` or `doc` functions introduced above.

```

```

Warning for ad hoc parameters. The downside to *ad hoc* parameters is that, while they get added to the entity's metadata (bottom right corner in AODataViewer), AOData does not support attaching validation functions or descriptions. Btw, you can find that information in the "expectedParameters" dataset with the Experiment in AODataViewer.

Description. All entities have an property called "`description`". Setting it is optional and can be performed using **`setDescription()`**. Here let's use it to describe the experiment's purpose.

```
EXPERIMENT.setDescription('To demonstrate the use of AOData');
```

If you provide no input to `setDescription()`, the existing description will be cleared.

You should see your description now in the properties when using `disp`.

```
disp(EXPERIMENT)
```

Notes. All entities have a `notes` property for miscellaneous comments. You can add as many notes as you would like using **`setNote()`**. They will be indexed in the order they were added.

```
% Add two notes
EXPERIMENT.setNote('First experiment after latest AO calibration');
EXPERIMENT.setNote('PMT Z position from model eye was off');

% View the experiment's notes
disp(EXPERIMENT.notes)
```

Use **`removeNote()`** to remove one or more notes. You can, for example, remove the first with `removeNote(1)`. To replace a note, for example, the 2nd note use `removeNote("new note", 2)`. To remove all the notes, use **`clearNotes()`**.

If you need practice with MATLAB, try adding removing a note or clearing all notes below. Make sure to add some notes back (you will want them for later steps).

```

```

Files. You can also add file names to an entity using **`setFile()`**. The same function will allow you to change the name of an existing file. Like the parameters, files have a key (descriptive name of the file) and a value (the file path). For example, "Experiment" is a good place to add the file name of the Imaging Log (i.e., the handwritten notes you took while imaging).

```
EXPERIMENT.setFile('ImagingLog', 'ImagingLog.pdf');
% Check to see your new file
disp(EXPERIMENT.files)
```

AOData encourages the use of ***relative file paths***, that is, file paths that are defined relative to the Experiment's `homeDirectory` property.

```
disp(EXPERIMENT.homeDirectory)
```

When you use the **`getExptFile()`** method, the `homeDirectory` will be appended.

```
disp(EXPERIMENT.getExptFile('ImagingLog'))
```

In fact, if you included the whole file path, it would be stripped from the file name before storing it in `files`. See for yourself:

```
EXPERIMENT.setFile('ImagingLog', fullfile(experimentPath, 'ImagingLog.pdf'));
```

```
disp(EXPERIMENT.getExptFile('ImagingLog'))
```

Why relative file paths? AOData is built for collaboration and flexibility. The absolute file paths will vary depending on the computer used. This is also helpful if the absolute file path differs between, say your work computer and your laptop. You can easily change the `homeDirectory` property as demonstrated below in the "Experiment Methods" section, but rewriting a bunch of file paths would be more challenging.

What if you want to log a file that was on the AO system's computer rather than within the experiment folder? The input to `setFile()` will only be altered ***IF*** *the beginning matches the `homeDirectory` property.* If you want to get a file value without appending the `homeDirectory`, simply use **`getFile()`** instead of `getExptFile()`.

```
EXPERIMENT.setFile('OtherImagingLog', 'X:\Users\DiffUser\Documents\ImagingLog.pdf');
disp(EXPERIMENT.getFile('OtherImagingLog'));
```

Try comparing the outputs of `getFile()` and `getExptFile()` for 'ImagingLog':

You can play around some with the files above, adding/removing/clearing. For the purposes of demonstration later, make sure you that, if you clear all the files, you add them back before the next step.

Mapping the Experiment to an HDF5 file

Usually you will want to add more to the experiment before writing it to an HDF5 file. But for the sake of demonstration, let's write it to the HDF5 file to see what it will look like. You'll use the function **`aod.h5.writeExperimentToFile()`**.

```
help aod.h5.writeExperimentToFile
```

Let's name our file "Tutorial_JustTheExperiment.h5".

```
aod.h5.writeExperimentToFile('Tutorial_JustTheExperiment.h5', EXPERIMENT, true);
```

You should now have a file called "Tutorial_JustTheExperiment.h5" in your current directory. If you want it to be saved elsewhere, include the full file path before the file name.

Now let's open the HDF5 file in **`AODataViewer`**. You can specify the file name or leave it blank. If you don't, you'll get a file directory option where you can select the HDF5 file you want to open.

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

By the end of the tutorial, you'll understand all the different aspects visible within AODataViewer. For now, here's a quick tour:

1. Click on the main Experiment folder, and check out the table on the bottom right. These are the HDF5 attributes. Grayed out ones are system attributes (i.e. ones AOData uses behind the scenes). Ones that aren't grayed-out are the entity-specific metadata. You should be able to see all the contents of your Experiment's `parameters` property. Note the UUID parameter - each entity in an experiment will have a unique identifier.
2. By expanding the Experiment node (click on arrow next to it), you can see the contents. You can also see the properties we defined: "`homeDirectory`", "`Name`" and "`experimentDate`" as well as the ones we modified after creating `EXPERIMENT` ("`notes`" and "`files`").
3. If you click on a dataset, you can see the contents in the top right panel (unless it's a very large ND array). You'll notice `files` is handled a bit differently: the contents are visible in the attributes panel instead of the data panel - see the "HDF5" section of the PDF Documentation for more details on why.

As you work through this tutorial, you can write to HDF5 at any point and then open it in a new AODataViewer window to get a sense of how the HDF5 file is built. I'd recommend doing this any time you aren't sure what the last step meant for the Experiment's contents or the final HDF5 file.

Methods and properties specific to Experiment

Subclasses of **`aod.core.Entity`** get all the properties and methods defined there by default. In addition, subclasses can add on new properties and methods. In other words, **`aod.core.Experiment`** is "customizing" **`aod.core.Entity`**. You will do the same with the core classes (like **`aod.core.Experiment`**) when you will do when develop your own packages. **`aod.core.Experiment`** has a few of these methods (in fact, more than any other entity as it is the root for the entire experiment dataset). Here are some relevant ones:

homeDirectory. The path to the experiment folder is stored in Experiment's `homeDirectory` property. As above, it's important that file locations remain flexible so that AOData files will work on different computers and won't break if you move the lcoation of your experiment folder. You can always change the `homeDirectory` property of your experiment with the **`setHomeDirectory()`** function.

```
EXPERIMENT.setHomeDirectory('C:\Users\yourname\newexpfolderlocation\');
```

The Experiment entity is the base for your AOData file and all other entities will be added to it. Any files added to a child entity will use `homeDirectory` in the parent Experiment for relative file paths.

Code. AOData tracks the status of all git repositories logged in **AODataManagerApp** and appends them to the Experiment. This way, you know exactly what code was used to create a given Experiment. This is handled internally.

```
disp(EXPERIMENT.Code)
```

Calling object methods. A quick review: In object oriented programming, you have **classes** that are blueprints for creating **objects**. So `EXPERIMENT` is the object and **`aod.core.Experiment`** is the class.

Classes can define **methods**, such as `setHomeDirectory()`. It's a method that will only work for objects of class **`aod.core.Experiment`**, like `EXPERIMENT`.

```
classdef Experiment < aod.core.Entity
    properties (SetAccess = protected)
        % The experiment's data folder path
        homeDirectory           char
        % ...
    end

    methods
        function setHomeDirectory(obj, filePath)
            % Code here assigns filePath to the homeDirectory property
        end
    end
end
```

The first input to a class method is always `obj` which is an object of class `aod.core.Experiment`. The method knows it only works on **`aod.core.Experiment`** objects, but doesn't know *which* object unless you provide it.

There are two ways to call a class method from an object:

```
% Option One:
EXPERIMENT.setDescription('This is an experiment');
% Option Two:
setDescription(EXPERIMENT, 'This is an experiment');
```

Option 1 is the most common and is the one I'll typically use, unless there is a good reason to use Option Two (occasionally there are, as you'll see later on in the walkthrough).

The Persistent Hierarchy

AOData object model's heirarchy is like a file directory. As mentioned above, Experiment is the "root" or base for the entire AOData file. By root, I mean that every other entity you create will be added to Experiment or to an entity that is added to Experiment.

Experiment
- Source
    - Source
- System
    - Channel
        - Device
- Calibration
- Epoch
    - Registration
    - Response
    - Stimulus
    - EpochDataset
- ExperimentDataset
- Segmentation
- Analysis

If you check out the contents of Experiment again in the Variable UI (or looking at the properties in the code), you will see the hierarchy's implementation. There are 6 properties called "Analyses", "Annotations", "Calibrations", "Epochs", "ExperimentDatasets", "Sources" and "Systems". If you look back at the AOData object model in the PDF documentation, you will see these fall directly under Experiment in the hierarchy.

Source (`aod.core.Source`)

Sources are unique among AOData's entity in that you can have nested Sources - that is, a Source that contains other Sources. This is needed to describe imaging locations at the appropriate level of detail.

Example: Each experiment involves just 1 primate. Within that primate, there are two eyes (OS and OD), both of which may be imaged during an experiment and which have eye-specific metadata like axial length. Finally, within each eye, there are multiple locations imaged. These may not have metadata but still need to be recorded so the data can be sorted by location later.

This situation demonstrates why Source offers a nested hierarchy of Sources. The animal should be a Source within the Experiment. The eye(s) imaged should be Sources within the animal's Source. Finally the locations imaged should be a Source within the correct Eye. The **`aod.core.sources` package** contains classes for this exact situation: **`Subject`**, **`Eye`**, and **`Location`**. All are customized subclasses of **`aod.core.Source`**.

Here's how you can use them to create a Source hierarchy for the situation above. The first input is assigned to the `Name` property. See the underlying the code for `Subject`, `Eye` and `Location` or their help files to learn more about the extra inputs for each.

```
% The subject imaged, in this case a macaque
subject = aod.core.sources.Subject('2018-51',...
    'Sex', "male", 'Age', 6, 'Species', "macaca fascicularis");
% The eyes imaged, in this case both
OD = aod.core.sources.Eye('OD',...
    'AxialLength', 16.88);
OS = aod.core.sources.Eye('OS',...
    'AxialLength', 16.97);
% The specific locations imaged in each eye
rightOD = aod.core.sources.Location('TemporalFovea');
leftOD = aod.core.sources.Location('NasalFovea');
rightOS = aod.core.sources.Location('NasalFovea');
```

Check them out with `disp` to get familiar with the **`aod.core.Source`** entity

```
disp(subject)
```

Now you have created everything, but they are all stand-alone objects. To define their relationships to each other (e.g., make explicit that `OD` contains `rightOD`), you'll need to link them to each other. You also need to link `subject` to the Experiment. This is accomplished with the **`add()`** function.

First, add your `subject` to `EXPERIMENT`. Two things happen:

1. You will now find `subject` within the Experiment's `Sources` property.

2. You will now have the Experiment in `subject`'s `Parent` property .

```
EXPERIMENT.add(subject);
```

As expected from the AOData Object Model, all entities will have a `Parent`, except for Experiment as it's the root/top-level entity.

```
% Now your subject is within the Sources property:
disp(EXPERIMENT)
% And the Parent property in subject is the experiment:
disp(subject)
```

Add eyes to the subject. Now, we'll add the eyes to `subject`. They will now be visible in the `Sources` property of `subject`.

```
subject.add([OD, OS]);
disp(subject)
```

To understand why we can add these to `subject` and see the changes reflected in `EXPERIMENT.Sources`, see the section of the PDF Documentation on "Handle vs. Value Classes".

All entities must be part of the Experiment to be written to the final AOData file. This is accomplished by adding each entity to Experiment itself, or another entity that is already added to Experiment. Basically the Experiment object needs to "know" about each entity.

Because `subject` has been added to `EXPERIMENT`, adding `OD` to `subject` means it's now part of the Experiment as well. You can confirm this by demonstrating that `OD` is now accessible from `EXPERIMENT`.

```
% You can access OD and OS through Experiment:
disp(EXPERIMENT.Sources(1).Sources);
```

Add locations to their respective eye. Next, add the the locations (`rightOD`, `leftOD`) to the appropriate eye (`OD`).

```
OD.add([rightOD, leftOD]);
OS.add(rightOS);
% There should now be two sources within OD
disp(EXPERIMENT.Sources(1).Sources(1))
```

When you ask for the `Sources` within `OD`, you get an array of two Sources - `rightOD` will be first because it was the first one added to the Experiment, `leftOD` will be the second.

```
disp(EXPERIMENT.Sources(1).Sources(1).Sources(1))
disp(EXPERIMENT.Sources(1).Sources(1).Sources(2))
```

As promised above, all the details you learned about the `notes`, `description`, `files`, etc apply to the Source entity as well so you can use those to fill out information on the Source entities as needed.

```
% Describe the fluorophores expressed in different imaging regions
rightOD.setDescription('GCaMP6s expression');
leftOD.setDescription('GCaMP6s and rhodamine expression');
```

Demonstrate this for yourself by adding some of this metadata to another source

```

```

A quick warning regarding human imaging: if you intend to share your dataset, beware of identifying details in `parameters`. Consider putting any identifying demographics or protected health information in `files` associated with the subject's Source entity. Or make sure that you have a version with identifying `parameters` removed for sharing.

Searching an Experiment

The syntax for accessing `leftOD` was pretty cumbersome.

```
% Access leftOD
Experiment.Sources(1).Sources(1).Sources(2)
```

It also requires that you remember which Source within `OD` was 1 and which was 2 (or at least using `disp` to check before using one in subsequent code). My goal was to make AOData easy to use, and accessing `leftOD` is not.

This is where the **`get()`** function comes in, which allows you to request entities of a specific type that meet some criteria - this could be `name`, class, subclass or even a specific parameter. I'll demonstrate some here and you can check out the PDF Documentation section "Searching within the Core Interface".

The first input to **`get()`** is the entity type you want (e.g., 'Source', 'Epoch', etc.). This is optional, though it does make the search a little faster (in the core interface there's no speed benefit, but once you start querying multiple HDF5 files, it will help). Additional arguments are queries within cells. Read the AOQuery documentation for details on the queries are setup.

```
out = EXPERIMENT.get('Source', {'Name', "NasalFovea"});
disp(out);
```

You use **`get()`** from any entity, but you will only be able to search that entity's children (it's like the "Search with folder and subfolders" functionality in a file directory). So you could have run the same query above with `OD`.

```
out = OD.get('Source', {'Name', "NasalFovea"});
disp(out);
```

Removing entities. Ideally once you are all set up with AOData, you'll have automated pipelines to create your experiments. While you set those up, you'll probably make mistakes and need to remove entities from time to time. You can do so with `remove()` in two ways:

1. By index. For example, to remove leftOD, run:
2. By query. This works identically to `get()`. You specify the entity type and some queries.
3. All. To remove all child entities, specify `'all'`

```
% Remove leftOD by index
EXPERIMENT.Sources(1).Sources(1).remove(2);
% Alternative syntax with object inside "remove":
remove(EXPERIMENT.Sources(1).Sources(1), 2);

% Remove leftOD by query
EXPERIMENT.remove('Source', {'Name', "Right"});
% Remove all locations from OD
EXPERIMENT.Sources(1).Sources(1).remove('all');

% Combine get and remove. This gets OD then removes all child sources
remove(EXPERIMENT.get('Source', {'Name', "OD"}, 'all')
```

If the syntax where `remove()` is listed first is confusing, go back to the Calling Object Methods section. We used it here because we want to use a method of the entity which is the output of a `get()` query

Handling of empty datasets. An entity's properties (defined in the `properties` block of each class and their superclasses) will be written as HDF5 datasets within the entity. AOData provides lots of useful default datasets (e.g., notes, files, descriptions), but you may not take advantage of them for every entity. **If the value of a dataset is empty, it will not be written to the HDF5 file.** You can confirm this by writing your experiment to an HDF5 file again and searching for "notes" within your new Sources. A goal of AOData was to make the HDF5 files interpretable and a bunch of empty datasets you didn't use would have cluttered things up unnecessarily.

To learn more about how any entity will be written to the HDF5 file, use `aod.h5.getPersistedProperties()`.

It will print four groups:

1. "Persisted Properties" - these will be written as datasets
2. "Attribute Properties" - these will be written as attributes (these are aod.core.Entity properties)
3. "Abandoned Properties" - these will not be written because they are either Transient or GetAccess=private
4. "Empty Properties" - these will not be written because they are empty

System Hierarchy (`aod.core.System`, `aod.core.Channel`, `aod.core.Device`)

You're typically using just one physical system during an experiment, but you may be using that system in different ways, either within an experiment or between experiments. The System hierarchy allows you to document these **system configurations**. Within a system, you likely have multiple **channels** (e.g., a wavefront sensing channel, a reflectance imaging channel, a channel that provides stimulation but no data is collected, etc.) and acquired data may use some or all of those channels. And within those channels, you have **devices**.

Which devices should you record? Of course, if you want you can provide an exhaustive list of all the mirrors and lenses, but those could also be obtained from a system diagram. The key details to include are the ones you might not know about from the system diagram. There are two classes of details to include (at least in my experience, perhaps you can think of more):

- Devices that are absent from or not fully specified by the system diagram. These are devices outside the system diagram (e.g., maybe you added an NDF in front of the light source) or ones that aren't fully described and may be frequently changed for different experiment goals (e.g., maybe you have "Dichroic Filter" and "Pinhole" in your diagram but the one you use may vary with the experiment.
- Devices that are not created equal. You likely have a PMT in your system diagram, but not the specific PMT's serial number. PMTs aren't created equal and swapping them could lead to changes in other metadata that would be unexplained without knowledge of the PMT swap (e.g., PMT gain used during each trial). It is valuable to record the specific PMT and serial number in your system configuration, and, if you're feeling thorough, the manufacturer stats for your specific PMT. Same goes for light sources - you never know when you're going to need to change your wavefront sensing beacon and documenting the swap will provide important context for interpreting the power levels used in your experiments.
- Devices with multiple functions. Maybe you have a light source with several laser lines and different experiments use different wavelengths.

It's worth looking at your system diagram and making a comprehensive list of which should be included (even if they aren't changing in your current experiments).

The example below sets up a simple system hierarchy for using an AOSLO for simultaneous imaging of cones and a fluorophore, GCaMP6s. For brevity, a lot of the relevant devices are omitted...

```
% Create the system and add it to the experiment
system = aod.core.System('ReflectanceImaging');
EXPERIMENT.add(system);
```

Make a few channels and add some devices (for demonstration purposes, not meant to be comprehensive).

```
% Create the channels and add a few devices
channel1 = aod.core.Channel('WavefrontSensing');
channel2 = aod.core.Channel('Reflectance');
channel2.add(aod.builtin.devices.Pinhole(20));
channel3 = aod.core.Channel('Fluorescence Imaging');
channel3.add(aod.builtin.devices.Pinhole(20,...
    'Manufacturer', "ThorLabs", "Model", "P20A"));
channel3.add(aod.builtin.devices.BandpassFilter(520, 15,...
    'Manufacturer', "Semrock", 'Model', "TODO"));

% Don't forget to add the channels to the System
system.add([channel1, channel2, channel3]);
```

If this seems like a lot of information to be typing in for every experiment, don't worry - the customization tutorial will cover methods for automating standardized components of your system.

When you want to get an overview of the entities within an experiment, try **`aod.util.displayHierarchy()`**. You should see your newly added channels, devices and systems.

```
disp(aod.util.displayHierarchy(EXPERIMENT));
```

The information introduced for Experiment about files, parameters, descriptions and notes applies to these entities as well. Try adding a parameter to `system`, and a file to `channel3` and a description to the pinhole within `channel3`. You'll need to use the `get()` function to access the pinhole as we didn't create an variable for it.

```
pinhole = EXPERIMENT.get('Device', {'Class', 'aod.builtin.devices.Pinhole'});
```

```

```

Now would be a good time to overwrite your existing AOData HDF5 file and check out the new entities you have added.

```
aod.h5.writeExperimentToFile('Tutorial_WithSystem.h5', EXPERIMENT, true);
AODataViewer('Tutorial_WithSystem.h5');
```

Entity Names.

Before moving on, let's address the naming of entities...

User-defined names. By default, the first argument to any AOData entity is its name (set to the `Name` property). Every entity needs to have a name that will be used for its group in the final HDF5 file. An important restriction to keep in mind about HDF5 is that all groups at a specific level must have unique names. It's like a file system in that regard - you can't have two files called 'MyImage.png' saved in the same folder, but you can have a second 'MyImage.png' saved in a subfolder.

If you add an entity that has the same name as another entity within it's HDF5 parent group, you'll get a warning:

```
% Try to add a second System with the same name
EXPERIMENT.add(aod.core.System('ReflectanceImaging'));
```

The second System is still there, but you'll want to change its name using **`setName()`**. If you don't heed the warning and change the name, the 2nd system merge with the first when you write it to the HDF5 file.

```
EXPERIMENT.Systems(2).setName('SecondSystem');
```

Automated naming. However, you might notice that we didn't have to name the pinhole or bandpass filters in the example above. Just because the groups need names doesn't mean you should always have to type in a name - in fact, this could lead to unnecessary inconsistencies between experiments (i.e., maybe you named your pinhole "20microns" one week and "Pinhole20microns" another week).

Often entities will have predictable names. Custom subclasses can take advantage of this by defining how to name an entity based on their properties/parameters using the **`getLabel()`** function. The output of this function is set to the `label` property. Unlike `Name`, you cannot directly set the `label` property, it can only be the output of an entity's `getLabel()` function.

```
classdef Pinhole < aod.core.Device

    methods
        function obj = Pinhole(diameter, varargin)
            % The first input (name) is left empty when calling the superclass
            obj@aod.core.Device([], varargin{:});
            % ...
        end
    end

    methods (Access = protected)
        function value = setLabel(obj)
            % Make a label that includes the pinhole diameter
            value = ['Pinhole_', num2str(obj.getParam('Diameter')), 'microns'];
        end
    end
end
```

Note that **`getLabel()`** is a "protected" method (as introduced above, subclasses can use it internally but you can't call it from the command line). Why? You don't need to because the output is assigned to the property `label`.

```
% Access the pinhole
pinhole = EXPERIMENT.get('Device', {'Class', 'aod.builtin.devices.Pinhole'});
pinhole = pinhole(1);
% and check out the name, label and groupNam
disp(pinhole.Name)          % This is the user-defined name (empty)
disp(pinhole.label)         % This is the automated label ('Pinhole_20microns')
disp(pinhole.groupName)     % This is what the HDF group will be called
```

You can change a name at any point using **`setName()`** as below.

```
pinhole.setName('Bob');
disp(pinhole.Name)          % This is the user-defined name ('Bob')
disp(pinhole.label)         % This is the automated label ('Pinhole_20microns')
disp(pinhole.groupName)     % This is what the HDF group will be called
```

If you no longer want your pinhole's HDF5 group to be named Bob and wish to have its group name be the automated label, you will have to reset the `name` property. This is because **`Name` always takes precedence over `label`** when determining the HDF5 group name.

```
pinhole.setName()  % Clear pinhole's name
disp(pinhole.groupName)
```

Epoch (`aod.core.Epoch`)

An Epoch is a period of continuous data acquisition within an Experiment. The Epoch class, **`aod.core.Epoch`** has only one required input, ID.

First, let's just add the epoch IDs (these could be the IDs associated with each video.

```
epochIDs = [1, 2, 4:6];
for i = 1:numel(epochIDs)
    EXPERIMENT.add(aod.core.Epoch(epochIDs(i)));
end
disp(EXPERIMENT)
```

You should see the 5 epochs in the "Epochs" property and the IDs of each Epoch in the "epochIDs" property.

Nonconsecutive epoch IDs. Note that the epoch IDs above aren't consecutive. There are many reasons why you might omit a trial and have no need of storing it with the final data file (e.g., maybe you ended the recording early because you realized you selected the wrong protocol or forgot to open the shutter). However, if your data files are identified by trial number, you'll need to maintain the appropriate ID numbers.

AOData supports non-consecutive IDs by providing a few convenience functions: **`id2epoch()`** and **`id2index()`**. You may find these handy once you're analyzing specific epochs.

```
% If you want the epoch with the ID #4, you can't get it by the index
% within the Epochs container:
disp(EXPERIMENT.Epochs(4))          % Returns Epoch #5
% Here's how to get Epoch #4 without remembering which were omitted
epoch4 = EXPERIMENT.id2epoch(4)     % Returns Epoch #4
% Here's how to get the epoch's index within the Epoch's container
idx = EXPERIMENT.id2index(4);         % Returns 3
disp(EXPERIMENT.Epochs(idx))        % Returns Epoch #4
```

Epoch has a number of unique properties (which will be written as datasets within each Epoch's HDF5 group), more so than any other entity besides Experiment. Let's look at how these are defined:

```
classdef Epoch < aod.core.Entity

    properties (SetAccess = private)
        % Epoch ID number in Experiment
        ID (1,1)          double      {mustBeInteger}
    end

    properties (SetAccess = {?aod.core.Epoch, ?aod.core.Experiment})
        % Time the Epoch (i.e. data acquisition) began
        startTime         datetime = datetime.empty()
        % Timing of samples during the Epoch
        Timing (:,1)      {mustBeA(Timing, ["double", "duration"])} = []
        % ...
    end

    properties (SetAccess = protected)
        % Source where data was acquired during the Epoch
        Source          {mustBeEntityType(Source, 'Source')} = aod.core.Source.empty()
```

```
            % System used for data acquisition during the Epoch
            System          {mustBeEntityType(System, 'System')} = aod.core.System.empty()
        end
    end
```

For a full description on property validation, check out MATLAB's documentation (link!).

Note that most properties have private or protected SetAccess. This means you can't set them directly. How do you set a property like `Timing` then? When in doubt, check the entity's methods:

```
doc aod.core.Epoch
```

You should see a method called **`setTiming()`** listed. Now ask for the help:

```
help aod.core.Epoch.setTiming
```

Hopefully you gathered that your input must be one of two classes: duration or double. If you aren't familiar with MATLAB's **`duration`** class, try `doc duration` to learn more. It's useful when working with time and offers some nice advantages over using `double`. AOData will make sure all **`duration`** data is written in a consistent time unit (seconds) to reduce heterogeneity in timing specification - but your data needs to be `duration` for AOData to recognize that it represents time.

```
% Let's say you have 25 frames per second and you're not using a scanning
% system where there is some variability in sample times.
epochTiming = 1/25:1/25:5;              % You imaged for five seconds
epochTiming = seconds(epochTiming);     % Convert to duration

% Set the timing to all epochs at once:
EXPERIMENT.Epochs.setTiming(epochTiming);
```

Links. AOData's object model defines a fixed set of relationships between entities (e.g., Experiments contain Systems, Systems contain Channels, etc.). But sometimes, you'll need to specify relationships that cut across this hierarchy. For example, the basic Epoch class **`aod.core.Epoch`** has two properties linking to other entities: `Source` and `System`. Any property containing another AOData entity will be written to the HDF5 file as a **soft-link**. These are basically like "shortcuts" on a Windows file system. The shortcut gives you immediate access to a file or folder in another location, without duplicating the underlying file/folder.

As you might have seen in the methods list above, there are two associated methods: **`setSource()`** and **`setSystem()`**.

```
% Let's assign the regions imaged
EXPERIMENT.Epochs(1:2).setSource(leftOD);
setSource(EXPERIMENT.id2epoch(1:2), leftOD);
setSource(EXPERIMENT.id2epoch(4), rightOD);
setSource(EXPERIMENT.id2epoch(5:6), rightOS);

% And they all used the same system configuration
```

```
EXPERIMENT.Epochs.setSystem(EXPERIMENT.Systems(1));
```

```
aod.h5.writeExperimentToFile('Tutorial_withEpochs.h5', EXPERIMENT, true);
AODataViewer('Tutorial_withEpochs.h5');
```

Explain soft-link following.

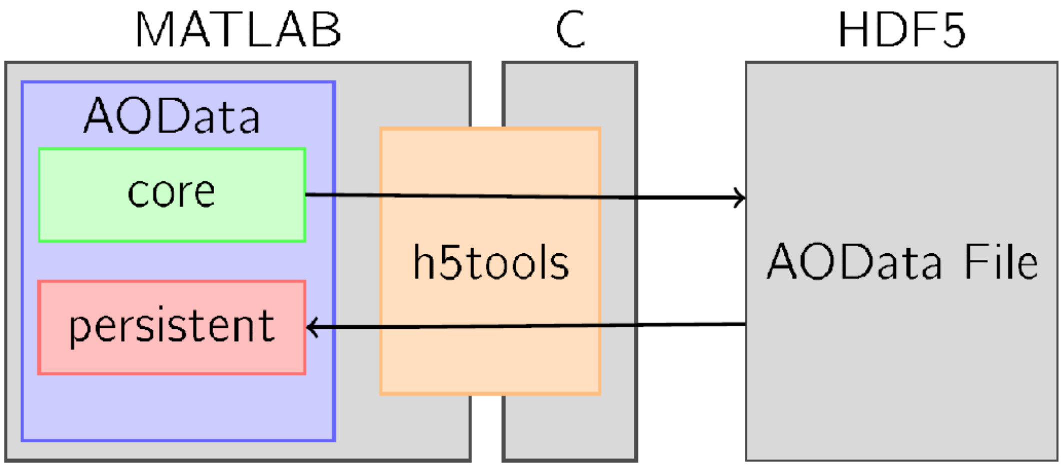
Persistent Interface

Now let's read in the AOData HDF5 file we just created.

```
pEXPT = loadExperiment('Tutorial_withEpochs.h5');
disp(pEXPT)
```

The persistent interface is largely interchangeable with the core interface, with one big difference:

- **Core interface:** used to define the mapping of information to an AOData file using customized subclasses of the core AOData objects
- **Persistent interface:** reflects the contents of the HDF5 file and is agnostic to the MATLAB classes used to create the file.



You will find the methods of Experiment are pretty much identical to those in the core interface and they work the same way. You can continue to modify persisted entities and add new ones with identical functions as in the core interface. There are two things to know first.

Read only mode. Persisted AOData files have a property called `readOnly` which is true by default. You will need to set it to false before modifying the AOData file's entities. Also, you should close out of AODataViewer before making any changes.

```
pEXPT.setReadOnlyMode(false);
```

Setting parameters.

```
pEXPT.setParam('PostPersistParam', "hi");

% Now let's look at the AOData file and see if it's there
AODataViewer(pEXPT);
```

Remember to close out before making more changes! It's possible to keep using an existing AODataViewer window after making changes to the underlying files, but your changes will not be there and I can't promise you won't run into errors.

Adding new entities. Recall when we add a new entity to a parent entity (e.g., adding Analysis to an Experiment), two things happen:

1. **The entity is added to the appropriate container property in the parent entity** (e.g., a new Epoch goes in Experiment's Epochs property). Placing an entity into the parent's container (Epochs) lets the Experiment hierarchy know about the entity. Once a new entity is added to the container of a persisted experiment, it is written to the HDF5 file.
2. **The entity's Parent property is set.** Setting the Parent property let's the child entity access information from the full Experiment hierarchy.

We can take advantage of #2 to create a new core entity that "behaves" as if it is part of the full experiment, but isn't persisted. This way, you can create, for example, a new Analysis and debug/develop it independent of the HDF5 file, then only add it once it's truly ready to become part of the persisted record.

```
analysis = aod.core.Analysis('TestAnalysis', 'Parent', EXPERIMENT);
disp(pEXPT)
disp(analysis)
```

```
% Once you're ready to add the analysis to the HDF5 file
pEXPT.add(analysis);
disp(pEXPT)
```

```
AODataViewer('Tutorial_withEpochs.h5')
```

Response (`aod.core.Response`)

A response is extracted from a specific spatiotemporal region of an Epoch's acquired data. Some examples include an image representing the average for each pixel or a timecourse average over a specific region of pixels (typically specified as an Annotation). The Response must specify the data from which the response is extracted - two approaches are enabled in in `aod.core.Response`, but subclasses could define alternatives, if needed.

1. By specifying the name of a file within the parent Epoch's `files` property. If the file requires specialized import, a `aod.util.FileReader` subclass can be provided as well. If not, one of the built-in readers will be assigned based on the file's extension (see `findFileReader` for supported extensions).
2. By specifying a Dataset associated with the parent Epoch.
3. By providing it directly to the **`Data`** property. This is not recommended as you won't know where the data came from - if you choose this method make it explicit where you got the data (e.g., one of the files in the Response or Epoch's `files` property).

For demonstration purposes, we will use #3 though because it's simple. This response will be the average of all pixels in each frame.

```
% One data point per time point
data1 = randn([1, numel(epochTiming)]);

resp1 = aod.core.Response('AverageTimecourse', 'Data', data1);
experiment.Epochs(1).add(resp1);
```