AOData Walkthrough

Sara Patterson (last updated 2022-12-18)

The purpose of this tutorial is to demonstrate, step-by-step, how to get started with AOData. The walkthrough 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. You should also get the "aodata-tutorial-package" from Github and dowload the experiment files from the Dropbox link in the repository information.

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".

Opening each of the variables (particularly the core components in the AOData object model within MATLAB's variable viewer and exploring the contents will also help you make the most of the walkthrough. You can click on them in the "Workspace" tab, right click on the variable within the live script and press "Open", or use the `openvar` command from the command line (useful after you're done with the walkthrough and writing your own code). Within this live script, you can also your mouse over any variable to get a little detail on what it is, as long as you've run the code block that first creates the variable.

```
% Create a variable
```

```
a = struct('FieldOne', [1 2 3], 'FieldTwo', 'hello', ...
    'FieldThree', "hi", 'FieldFour', {'hello', 'hi'});
% Open in the variable viewer. The classes created by AOData will be
% similar to structs. You can click on the individual fields to get more
% information about the contents.
openvar('a');
```

The object's created by AOData classes will be similar to the struct above - each field will be a property of the class. You can click the individual fields to get more information about the contents.

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.

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.

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

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
```

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:

```
initializeAOData();
```

A GUI called **PackageManagerApp** 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.

For now, you can close out of it as `initializeAOData()` will have set the necessary initial values. You can return here at any time to add/remove custom packages and repositories by running:

```
AODataManagerApp();
```

Creating a new package

Below is the information for creating a new custom package, which you can return to as you create custom packages (or check the PDF Documentation section on ""). If you want to follow along with the walkthrough, at least make a new folder on your computer to hold the files, even if it isn't tracked with git. At minimium, you should download the "tutorial package" from Github. Inside it, create a folder called "src" and within that a folder called "+tutorial" (explained below).

If you haven't already, make sure to add AOData and the tutorial package to your MATLAB path:

```
% Put the full file path for your package below:
addpath(genpath(''))
```

Create a git repository

Create a new git repository (see the Git Tutorial if you aren't familiar with this step). Name it something that identifies it as your personal AOData package (e.g. "yourname-aodata-package", "yourname-package", etc).

Create a namespace

Read the documentation section on **"Namespaces"** and refer to the linked MATLAB documentation for any additional questions that arise. As with all other concepts introduced here, looking at how AOData is organized can be helpful too.

An example of a class is `aod.core.Device`. The class is named `Device`. `aod` and `core` are the namespaces (`Device` is located in a folder called "+core" which is located in a folder called "+aod"). You will want to make your own namespace in your new repository folder. When in doubt, try the following template: "labname.yourname" (e.g. "williamslab.sara"). You want the namespace to be unique to you.

It can be useful to have packages contained within another folder so I'd recommend creating a folder named "src" within your main folder. If your chosen namespace is "labname.yourname", then inside "src", create a folder named "+labname". Inside that, create "+yourname".

Register with AOData

AOData needs to know your git repositories and packages exist. If you're using git, add the main folder to the "Git Repository" tab. Then make sure the folder containing the package folders is added to the "Search Path" tab (i.e. add "src" because it contains "+tutorial").

```
AODataManagerApp();
```

Example

To familiarize you with process of mapping an experiment to AOData, we'll walk through an example experiment. First, we'll use just the core classes, then we'll develop some custom classes to make the process easier.

A small experiment folder is available at: link. Download this and save it somewhere to your computer. The location where you save it will be your `experimentPath` in the following code.

```
% Save the location of your experiment path
experimentPath = '';
```

The example is based on data from the Rochester one-photon AOSLO. While the processing is specific to these experiments, the resulting understanding of how an experiment maps to AOData should be generic. I've picked a complicated experiment on purpose, to maximally demonstrate as much of AOData's capabilities as possible.

Data on the Rochester 1P-Primate system is acquired from two channels: a reflectance channel focused on the cone mosaic and a fluorescence channel focused on the ganglion cell layer (the RGCs express GCaMP6 (480/520) and a subset are retrogradely-labeled with Rhodamine (555/585). In addition, there's a wavefront sensing channel and a channel for providing visual stimuli to the cones.

The raw, acquired data consists of two simultaneously-acquired videos per epoch: one in the reflectance channel (e.g. Ref/838_20221122_ref_0001.avi) and one from the fluorescence channel (e.g. Vis/838_20221122_vis_0001.avi). The files are named "AnimalID_Date_Channel_EpochID.avi".

I'll explain more about the dataset as it becomes relevant in the walkthrough.

First, we're going to set up some of the Experiment using just the base classes to get a feel for how those work. Next, we're going to start over and develop custom classes, which will streamline the process and customize it to the underlying dataset.

Experiment (aod.core.Experiment)

First let's create an experiment object. Check out the Documentation within the `aod.core.Experiment` file as well as the PDF. As you'll see, there are 3 required inputs (experiment date, the file path to the experiment folder and) and 2 optional parameters (Administrator and Laboratory).

```
EXPERIMENT = aod.core.Experiment('838_ODR_20221122', experimentPath, '20221122',...
    'Administrator', 'Sara Patterson', 'Laboratory', '1P-Primate');
```

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)
        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.

How do you know which optional parameters to define? Those are determined by a protected function called `getExpectedParameters()`

```
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 (none for Entity)
            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 you can get a list of the potential parameters from the `expectedParameters` property of any AOData entity. Each has a name and, optionally, a default value, a validation function and a description.

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

Take a look at what's inside `EXPERIMENT`...

```
openvar('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 ("Sara Patterson"). 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 in `expectedParameters`. It's good to use the expected parameters 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.addParam('MyNewParam', 1);
disp(Experiment.parameters)
```

You can remove a parameter with `removeParam('MyNewParam')`, get the value of a parameter with `get('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.

```

```

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.

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

```
EXPERIMENT.addNote('First experiment after latest AO calibration');
EXPERIMENT.addNote('PMT Z position from model eye was off');
disp(EXPERIMENT.notes)
```

You can, for example, remove the first with `removeNote(1)`. You can clear them all with `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. 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');
```

Why not add the full path? This is because AOData offers the option for *relative file names*, that is, file names that are relative to `experimentPath`, the value of which was stored in the Experiment's `homeDirectory` property. When you use the `getExptFile()` method, the `homeDirectory` property will be appended.

```
disp(EXPERIMENT.homeDirectory)
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 do this? AOData is built for collaboration and 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 always change the `homeDirectory` as demonstrated below in the "Experiment Methods" section.

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\somename\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.

```
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. Check the documentation for `aod.h5.writeExperimentToFile` for the other input information.

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 just 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's parameters and you should be able to see all the parameters within your experiment's `parameters` property.

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`"). If you click on one, you can see the contents in the top right panel. 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 file.

Methods 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 you add new entities to the experiment hierarchy.

The Persistent Hierarchy

As mentioned above, Experiment is the "root" 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. If you check out the contents of Experiment again in the Variable UI (or looking at the properties in the code), you will see 5 properties called "Analyses", "Calibrations", "Segmentations", "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 the most complex parts of the AOData object model, because you can have nested Sources - that is, a Source that contains other Sources. As described below, this is needed to describe imaging locations at the appropriate level of detail.

Each experiment involves just 1 primate. Within that primate, there are two eyes, 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 or eyes imaged should be a Source within the animal's Source. Finally the locations imaged should be a Source within the 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.

```
subject = aod.core.sources.Subject('MW00851',...
    'Sex', 'female', 'Age', 6, 'Species', 'macaca fasciularis');
OD = aod.core.sources.Eye('OD',...
    'AxialLength', 16.56);
rightOD = aod.core.sources.Location('Right');
leftOD = aod.core.sources.Location('Left');
% The right side has GCaMP6 and the left side has both GCaMP6 and rhodamine
% expression. Let's add some custom parameters to for that.
rightOD.setParam('Flurophores', "GCaMP6");
```

```
leftOD.setParam('Fluorophores', ["GCaMP6", "Rhodamine"]);
```

Now you have created everything, but they are all stand-alone objects. To define their relationships to each other (e.g., make explicit the relationship between `OD` and `rightOD`), you'll need to link them to each other, and the experiment. This is accomplished with the **`add()`** function. First, add your `subject` to `EXPERIMENT`. It will go in the `Sources` property. In addtion, `EXPERIMENT` will be set to the `Parent` property of `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.

```
EXPERIMENT.add(subject);
% Now your subject is within the Sources property
disp(EXPERIMENT.Sources)
disp(subject.Parent)
```

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

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

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".

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

```
OD.add([rightOD, leftOD]);
% 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))
```

Searching an Experiment

The syntax for getting `rightOD` and `leftOD` was pretty cumbersome and 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). 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 `search()` is the entity type you want.

I've used MATLAB's new key/value pair syntax for subsequent inputs as I think it makes the code more readable.

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

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 - you should add it and specify the laser line used.

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 reflectance imaging. This includes the wavefront sensing channel and a reflectance imaging channel.

```
system = aod.core.System('ReflectanceImaging');
```

```
EXPERIMENT.add(system);
% Wavefront sensing
channel1 = aod.core.Channel('WavefrontSensing');
% Reflectance imaging
channel2 = aod.core.Channel('Reflectance');
channel2.addDevice(aod.builtin.devices.Pinhole(20));
% Fluorescence imaging
channel3 = aod.core.Channel('Fluorescence Imaging');
channel3.addDevice(aod.builtin.devices.Pinhole(20));
channel3.addDevice(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]);

% Use displayHierarchy to get a full list of all the entities in Experiment
aod.util.displayHierarchy(EXPERIMENT);
```

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.

As promised, 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.

```
% Hint: here's how to get a variable for the pinhole
pinhole = EXPERIMENT.get({'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);
```

Entity 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.

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. 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');
```

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
```

```
% Access the pinhole and check out the name, label and groupName
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()
```

aod.core.Annotation

The Annotation class is designed for you to... Some examples include:

- XY coordinates of cones or any other imaged structure
- Segmentation masks
- Temporal windows for analysis

aod.core.Epoch

The example dataset contains 3 types of imaging that I typically perform within the same experiment. Some metadata is unique to each imaging type and other metadata is shared by all imaging types. There is also both shared and unique post-processing.

- Anatomy: data is acquired from both channels but there is no visual stimulus presented to the cones. The goal is only to visualize the fluorescently-labeled cells.
- Spectral physiology: data is acquired from both channels and a spatially-uniform visual stimulus is presented to the cones through a 3 LED Maxwellian View
- Spatial physiology: data is acquired from both channels. GCaMP6 responses are being measured to a spatial stimulus presented through the scanning system with a 561 nm laser. The stimulus is stabilized on the cone mosaic so remains locked on the same cones in the presence of eye movements.

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

```
disp(experiment.Epochs)
disp(experiment.epochIDs)
```

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.