Extending the Python online analysis toolkit

Benjamin Naecker

July 2, 2018

Introduction This document describes in detail how to extend the Python-based online analysis toolkit, adding new, custom analyses to the package. It assumes knowledge of the recording software system, especially how data is retrieved from the `blds` over the network.

The online analysis toolkit

The recording software system contains a Python-based executable and package which simplify online analysis of retinal data. The executable `oatool` can be run from the command-line and will run one of several existing analyses (mostly receptive fields). One can also create a Python class, subclassing one or more of the classes in the module `oalib`, and use this new analysis from `oatool`. This document will walk through the process of doing this.

Analysis base classes

An online analysis is implemented as a Python class, with specific methods. The interface is defined by the abstract base class `OnlineAnalysis`, which enforces that all online analyses implement the appropriate methods. The `oatool` executable just instantiates the requested subclass, collects data from the `blds`, and calls the appropriate methods in a loop.

The constructor of the base class requires some specific information about the stimulus and the data itself. Each of these attributes is then available to subclasses, if they need to use them during computation. The constructor for the base class has signature:

```
class OnlineAnalysis():
  def __init__(self, stimfile, frame_rate, length, channels):
    pass
```

The `stimfile` argument is the full path of file containing the stimulus (see below for details on the stimulus file), `frame_rate` is the frame rate of the stimulus, in Hz, `length` is the duration of the analysis in seconds, and `channels` is the list of channels on which the analysis is to be performed.

The base class creates a figure in which the analysis will be plotted (if desired). It also defines several useful helper functions, which can be used in any subclass to help compute the desired analysis.

```
class OnlineAnalysis():
  def frame_index_from_time(self, t):
    '''Return the index of the stimulus frame on the screen at time `t`.'''

  def frame_from_time(self, t):
    '''Return the actual stimulus frame on the screen at time `t`.'''

  def frame_times(self, start, stop):
    '''Return an array giving the time of each stimulus frame between the start and stop ti

  def frames_from_time(self, start, stop):
    '''Return the full set of stimulus frames between the start and stop times.'''
```

There are several useful public data attributes of the class. The most important of these is the `current_result` attribute, which is the current, running result of the online analysis. This is set to `None` inside the base class constructor, but it should be used to store the updated result.

Extending the base class

To extend the base class, subclass it and implement the following abstract methods.

```
def update(self, dataframe):
  '''Update the current_result after receiving a new chunk of data.'''

def plot(self):
  '''Plot the current_result in the class's figure, if desired.'''

def finalize(self):
  '''This method is called once, after the analysis has finished. It should
  be used to do things like normalization.
  '''
```

Let's see how this would be done for a new online analsyis, which just plots the activity of each channel over time. "Activity" here will just be defined by the cumulative variance of each channel.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
```

```
from oalib import OnlineAnalysis

class OnlineActivityMap(OnlineAnalysis):
  def __init__(self, stimfile, frame_rate, length, channels):
    super().__init__(stimfile, frame_rate, length, channel)
    self.current_result = np.zeros((len(channels,))) # Start with 0 variance
    self.axes = self.figure.add_subplot(111) # Add axes to the pre-created figure
    self.activity_map = self.axes.bar(self.channels, self.current_result)

  def update(self, dataframe):
    '''Compute the cumulative variance of each channel and normalize, and
    update the title of the axes with the current time in the recording.
    '''
    self.current_result += np.var(dataframe.data, axis=1)[:len(self.channels)]
    self.current_result /= np.max(self.current_result)
    self.axes.set_title('{:d} s'.format(int(dataframe.stop)))

  def plot(self):
    '''Update the height of each rectangle in the bar plot to be the cumulative
    variance of the corresponding channel of data.
    '''
    for ix, rect in enumerate(self.activity_map):
      rect.set_height(self.current_result[ix])
    self.axes.relim()
    self.axes.autoscale()
    plt.pause(0.01) # Forces graphics to update

  def finalize(self):
    pass # Do nothing
```

Running the new analysis with `oatool`

The executable `oatool` can be given a new, custom analysis to run, by specifying the file in which the new analysis class is defined. The executable then looks in this file for a subclass of `OnlineAnalysis`, and instantiates the first such class found as the analysis.

`oatool` then connects to the `blds`, and starts streaming data (optionally triggering the start of the data stream). The core of `oatool` is just a simple loop that basically looks like this:

```
# The analysis class itself is determined dynamically, using either one of the
# included classes or a custom class
```

```
oa = OnlineAnalysisClass(stimfile, frame_rate, length, channels)

# Start at the beginning of the recording
position = 0.0

# Duration is set by the user on the command line, or
# the duration of the recording, whichever is smaller
while position < duration:

  # Get the next chunk of data from the blds. This uses the Python
  # implementation that comes with libblds-client.
  frame = client.get_data(position, position + interval)
  oa.update(frame)
  oa.plot()
  position += interval

# After the loop ends, either because the recording ended, duration was
# exceeded, or the user interrupted the program, finalize the analysis.
oa.finalize()
```

To give `oatool` your custom analysis class, give the full filename to the `--custom-analysis` parameter:

```
$ oatool.py --stimfile stimulus-file --custom-analysis /path/to/custom/analysis.py
```

Stimulus file format

The stimulus file itself is a simple HDF5 file. It should have a dataset called `'/stimulus'`, which should have shape of `(t, x, y)`. t is the number of samples and x and y are spatial dimensions. One or both of the spatial dimensions may be omitted, but the temporal dimension is always required.

The `'/stimulus'` dataset may have an attribute called `'frame-rate'`, which should be a floating point number giving the frame rate in Hz. If this attribute does not exist, the frame rate **must** be specified on the command line with the `--frame-rate` parameter to `oatool`.