



MOSAIC Manual

Release 1.0

**Arvind Balijepalli
Jacob Forstater
Joseph Robertson
Canute Vaz**

Questions/Suggestions

Contact Arvind Balijepalli (arvind.balijepalli@nist.gov)
National Institute of Standards and Technology
100 Bureau Drive
Gaithersburg, MD 20899

Report problems with MOSAIC using the issue tracker on GitHub
<https://github.com/usnistgov/mosaic/issues>

Terms of Use

This software was developed at the National Institute of Standards and Technology by employees of the Federal Government in the course of their official duties. Pursuant to title 17 section 105 of the United States Code this software is not subject to copyright protection and is in the public domain. This software is experimental. NIST assumes no responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, or any other characteristic. This software can be redistributed and/or modified freely provided that any derivative works bear some notice that they are derived from it, and any modified versions bear some notice that they have been modified.

Disclaimer

Certain commercial firms and trade names are identified in this document in order to specify the installation and usage procedures adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that related products are necessarily the best available for the purpose.

Contents

1	Introduction	3
2	Getting Started	5
2.1	Binary Installation	5
2.2	Source Installation	5
3	MOSAIC GUI	11
3.1	Interface Overview	11
3.2	Panels A & B: Analysis Setup and Trajectory Viewer	13
3.3	Panels C,D, & E: Blockade Depth Histogram, Statistics, and Event Viewer	14
4	Scripting and Advanced Features	19
4.1	Import Data and Run an Analysis	19
4.2	Advanced Scripting	22
5	Settings File	25
5.1	Settings Layout	25
5.2	Default Settings	26
6	Database Structure and Query Syntax	29
6.1	Metadata Table	29
6.2	Analysis Settings Table	31
6.3	Work with SQLite	31
7	Extend MOSAIC	33
7.1	Read Arbitrary Binary Data Files	33
7.2	Define Top-Level Functionality	34
8	Addons	37
8.1	Mathematica	37
8.2	Matlab	41
8.3	IGOR	42
9	Examples	43
9.1	Single Molecule Mass Spectrometry	43

10 API Documentation	45
10.1 <i>MOSAIC</i> Modules	46
11 Change Log	71
Bibliography	73
Python Module Index	75
Index	77

MOSAIC is a single molecule analysis toolbox that automatically decodes multi-state nanopore data. By modeling the nanopore system with an equivalent circuit, *MOSAIC* leverages the transient response of a molecule entering the channel to quantify pore-molecule interactions. In contrast to existing techniques such as ionic current thresholding [PFR09][RKR10] or Viterbi decoding [Vit67], this technique allows the estimation of short-lived transient events that are otherwise not analyzed.

Nanometer-scale pores have demonstrated potential use in biotechnology applications, including DNA sequencing [KBBD96], single-molecule force spectroscopy [VDKD+09], and single-molecule mass spectrometry [RRS+07]. The data modeling and analysis methods implemented in *MOSAIC* allow for considerable improvements in the quantification of molecular interactions with the channel in each of these applications.

Note: If you use *MOSAIC* in your work, please cite:

A. Balijepalli, J., Ettegui, A. T. Cornio, J. W. F. Robertson K. P. Cheung, J. J. Kasianowicz & C. Vaz, “Quantifying Short-Lived Events in Multistate Ionic Current Measurements.” *ACS Nano* 2014, **8**, 1547–1553.

Introduction

MOSAIC is a modular toolbox for analyzing data from single molecule experiments. Primarily developed to analyze data from nanopore experiments [RBR+12], *MOSAIC* can analyze any data that fit the form [BEC+14]:

$$i(t) = i_0 + \sum_{j=1}^N a_j \left(1 - e^{-(t-\mu_j)/\tau_j} \right) H(t - \mu_j)$$

The above functional form, which represents the response to a step change from one state to another is ubiquitous in many disciplines. By fitting individual state changes to the equation above, *MOSAIC* is able to automatically identify the states corresponding to each change. Moreover this approach allows us to accurately characterize transient events before they asymptotically approach a steady state. In nanopore applications, this has resulted in a 20-fold improvement in the number of states identified per unit time [BEC+14].

MOSAIC offers tremendous flexibility in how it can be used. Nanopore data can be analyzed and visualized using the *MOSAIC GUI* (GUI), which is available as a stand-alone application ([download binaries](#)). This is a convenient way for most users to analyze nanopore data. Advanced users can write their own Python scripts to include *MOSAIC* in their analysis workflow (see [Scripting and Advanced Features](#)). Finally, because *MOSAIC* was designed from the start using object oriented design, developers can easily extend it by combining existing classes to define new functionality or writing their own classes (see [Extend MOSAIC](#)).

Getting Started

2.1 Binary Installation

MOSAIC is available as a pre-compiled binary for Windows and Mac OS X ([download binaries](#)). *MOSAIC* binaries do not need special installation. Under **Mac OS X** open the the downloaded disk image and drag the *MOSAIC* executable to the Applications folder. Under **Windows**, unzip downloaded zip file and move the *MOSAIC* executable to your hard disk.

Note: *MOSAIC* binaries are 64-bit. If you need 32-bit support, please build *MOSAIC* from source as described in the [Source Installation](#) section.

2.2 Source Installation

2.2.1 Install *MOSAIC* on Mac OS X

In the following guide, we provide step-by-step instructions on setting up and running *MOSAIC* on OS X. To simplify the installation, we use [Homebrew](#) to install some required dependencies. [Homebrew](#) requires Apple command line tools, but will directly prompt you to install it on set up.

1. Installing Homebrew

First we will install Homebrew, a useful package manager, to help install some of the dependencies required by *MOSAIC*. You will need administrator access for this step. In the OS X Terminal, run the following command:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Note, if the Apple command line tools are not installed, [Homebrew](#) will prompt you to do so during installation.

Hint: To test if [Homebrew](#) is properly installed, run the following in the terminal: `brew doctor`

To ensure that [Homebrew](#) is set up correctly, add the [Homebrew](#) directory to `~/.bash_profile`. This can be done using the following command:

```
$ echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.bash_profile
```

Hint: If you don't have a `.bash_profile` file in your home directory, you can create one manually using a text editor.

Restart the terminal to update your shell.

2. Installing brewed Python and other necessary packages

MOSAIC is written in [Python 2.7+](#) and utilizes a number of different packages and utilities. In the following we'll install a number of these (specifically, python, gcc, gfortran, qt, and PyQt4). With homebrew this is easy to do in one line! Run the following in the terminal:

```
$ brew install python gcc gfortran qt pyqt
```

At this point, it is a good idea to update the PYTHONPATH environment variable in ~/.bash_profile:

```
$ export PYTHONPATH=$PYTHONPATH:/usr/local/lib/python2.7/site-packages
```

3. (Optional) Install and Setup Virtual Environment

It is generally a good practice to run *MOSAIC* from within a dedicated virtual environment. This minimizes conflicts with other installed programs. While we highly recommend this approach, it is not required to run *MOSAIC*. If you prefer to skip this, move on to the next step now.

To setup a virtual environment, we need two different packages: *virtualenv*, which creates the virtual environments, and *virtualenvwrapper*, a wrapper for *virtualenv* that simplifies set up and use.

To install these and set up the virtual environment wrapper, run the following in a shell:

```
$ pip install virtualenv virtualenvwrapper
```

Hint: Under Ubuntu, you may need install virtualenv and virtualenvwrapper as root. Simply prefix the command above with sudo.

If you would like virtualenvwrapper to be available each time you open a new terminal window, add the line below to ~/.bash_profile on OS X or ~/.bashrc on Linux.

```
source /usr/local/bin/virtualenvwrapper.sh
```

Hint: Depending on the process used to install *virtualenv*, the path to virtualenvwrapper.sh may vary. Find the appropriate path by running `$ find /usr -name virtualenvwrapper.sh`. Adjust the line in your .bash_profile or .bashrc script accordingly.

Open a new shell to make the new virtual environment available. Now we are ready to create a virtual environment. You can choose any name for your virtual environment, here we name it *MOSAIC*:

```
$ mkvirtualenv -p <path to python>/python MOSAIC
```

Hint: We explicitly specify the [Python](#) installation to use. This is not mandatory, but is useful if you have multiple [Python](#) installations on your computer. The `<path to python>` may vary according to the specific version of python you wish to use. In most cases, this will be either `/usr/local/bin/` or `/usr/bin`

4. Installing MOSAIC

First we need to obtain the *MOSAIC* source code. For analyzing publication data, we recommend downloading the latest stable version of the source code ([download source](#)). Alternatively, the latest development version can be downloaded from the [MOSAIC page on Github](#). Here we will show you how to set up *MOSAIC* from the latest stable release:

1. Download the latest release ([download source](#))

2. Create a directory for the project source. In this case we will create a directory called MOSAIC, located in `~/projects/`, where `~` is your home directory.

```
$ mkdir ~/projects/MOSAIC
```

3. Navigate to the directory:

```
$ cd ~/projects/MOSAIC
```

4. Extract the source into this folder.

5. Make sure you are working in the virtual environment we set up in the previous step by typing:

```
$ workon MOSAIC
```

Note: You will notice that (*MOSAIC*) now appears in front of the \$ prompt in your shell. This indicates that the virtual environment is active. We have employed this notation to indicate commands that should be run from inside the virtual environment.

6. *MOSAIC* and its dependencies are built using *setuptools* via a custom command as described below. However, we must first install *cython* manually. Run the following command:

```
(MOSAIC) $ pip install cython
```

7. To install the needed dependencies, navigate to `~/projects/MOSAIC/` and run the following:

```
(MOSAIC) $ python setup.py mosaic_deps
```

8. Finally, add the installation directory (`~/projects/MOSAIC` as set up previously) to your *PYTHONPATH* as shown below. This addition can be made permanent by adding the line below to your *.bash_profile* (OS X) or *.bashrc* (Ubuntu) script.

```
(MOSAIC) $ export PYTHONPATH=$PYTHONPATH:~/projects/MOSAIC
```

2.2.2 Install *MOSAIC* on Ubuntu(14.04)

MOSAIC can be run under Ubuntu using a procedure very similar to *installosx*.

1. Prerequisites

Several prerequisites must be installed prior to building *MOSAIC* dependencies. This is easily accomplished in Ubuntu using the *aptitude* package manager.

Hint: *superuser* privileges are needed when installing *MOSAIC* prerequisites.

```
$ sudo apt-get install python python-dev python-pip python-qt4
freetype* gfortran liblapack-dev libblas-dev
```

Next add the following to `~/.bashrc`

```
export PYTHONPATH=/usr/lib/python2.7/dist-packages
```

2. (Optional) Install and Setup Virtual Environment

It is generally a good practice to run *MOSAIC* from within a dedicated virtual environment. This minimizes conflicts with other installed programs. While we highly recommend this approach, it is not required to run *MOSAIC*. If you prefer to skip this, move on to the next step now.

To setup a virtual environment, we need two different packages: *virtualenv*, which creates the virtual environments, and *virtualenvwrapper*, a wrapper for *virtualenv* that simplifies set up and use.

To install these and set up the virtual environment wrapper, run the following in a shell:

```
$ pip install virtualenv virtualenvwrapper
```

Hint: Under Ubuntu, you may need install *virtualenv* and *virtualenvwrapper* as root. Simply prefix the command above with `sudo`.

If you would like *virtualenvwrapper* to be available each time you open a new terminal window, add the line below to `~/.bash_profile` on OS X or `~/.bashrc` on Linux.

```
source /usr/local/bin/virtualenvwrapper.sh
```

Hint: Depending on the process used to install *virtualenv*, the path to *virtualenvwrapper.sh* may vary. Find the appropriate path by running `$ find /usr -name virtualenvwrapper.sh`. Adjust the line in your `.bash_profile` or `.bashrc` script accordingly.

Open a new shell to make the new virtual environment available. Now we are ready to create a virtual environment. You can choose any name for your virtual environment, here we name it *MOSAIC*:

```
$ mkvirtualenv -p <path to python>/python MOSAIC
```

Hint: We explicitly specify the *Python* installation to use. This is not mandatory, but is useful if you have multiple *Python* installations on your computer. The `<path to python>` may vary according to the specific version of python you wish to use. In most cases, this will be either `/usr/local/bin/` or `/usr/bin`

3. Installing MOSAIC

First we need to obtain the *MOSAIC* source code. For analyzing publication data, we recommend downloading the latest stable version of the source code ([download source](#)). Alternatively, the latest development version can be downloaded from the [MOSAIC page on Github](#). Here we will show you how to set up *MOSAIC* from the latest stable release:

1. Download the latest release ([download source](#))
2. Create a directory for the project source. In this case we will create a directory called *MOSAIC*, located in `~/projects/`, where `~` is your home directory.

```
$ mkdir ~/projects/MOSAIC
```

3. Navigate to the directory:

```
$ cd ~/projects/MOSAIC
```

4. Extract the source into this folder.
5. Make sure you are working in the virtual environment we set up in the previous step by typing:

```
$ workon MOSAIC
```

Note: You will notice that (*MOSAIC*) now appears in front of the `$` prompt in your shell. This indicates that the virtual environment is active. We have employed this notation to indicate commands that should be run from inside the virtual environment.

6. *MOSAIC* and its dependencies are built using *setuptools* via a custom command as described below. However, we must first install *cython* manually. Run the following command:

```
(MOSAIC) $ pip install cython
```

7. To install the needed dependencies, navigate to `~/projects/MOSAIC/` and run the following:

```
(MOSAIC) $ python setup.py mosaic_deps
```

8. Finally, add the installation directory (`~/projects/MOSAIC` as set up previously) to your `PYTHONPATH` as shown below. This addition can be made permanent by adding the line below to your `.bash_profile` (OS X) or `.bashrc` (Ubuntu) script.

```
(MOSAIC) $ export PYTHONPATH=$PYTHONPATH:~/projects/MOSAIC
```

MOSAIC GUI

MOSAIC's GUI interface is designed to allow you to easily setup and run an analysis and to analyze the results of prior trials via a graphical interface; it contains the most commonly used features of *MOSAIC*. The GUI contains modular panels for setting up an analysis, running it, and analyzing the results. Here we give you a brief overview of the graphical interface and its basic use. You can learn more in the [Examples](#) section.

Opening the GUI

If you installed *MOSAIC* from a precompiled binary, you can open the GUI by double clicking the *MOSAIC* icon. Alternatively, if you compiled *MOSAIC* from source code, you can run the GUI from the terminal window – navigate to the installation directory and type:

```
python mosaicgui/mosaicGUI.py
```

Hint: Having trouble getting the GUI to start? Frequently, this arises because your `PYTHONPATH` environment variable is set up incorrectly. To fix this error, first type `echo $PYTHONPATH` in the terminal. If you don't see the path to the *MOSAIC* installation in `PYTHONPATH`, consult the operating-system specific instructions (*OSX* or *Ubuntu*) to help resolve this issue.

3.1 Interface Overview

The main interface consists of five panels which we go over in detail later in this document. Briefly, these are:

1. *Analysis Setup*: This panel is used to set up the analysis parameters.
2. *Trajectory Viewer*: This panel shows a snippet of the ionic current time-series and an all points histogram, used to set the baseline and threshold parameters found in [Panel A: Analysis Setup](#).
3. *Blockade Depth Histogram*: Once the data processing has started, this panel shows a live blockade depth histogram; a query can be defined to restrict the histogram to data which fulfills a user-defined criteria.
4. *Analysis Statistics*: Displays live statistics about the data processed.
5. *Event Viewer*: Displays the partitioned events and their fit. This panel is active only if “Write Events to Disk” is enabled in the [Analysis Setup](#).

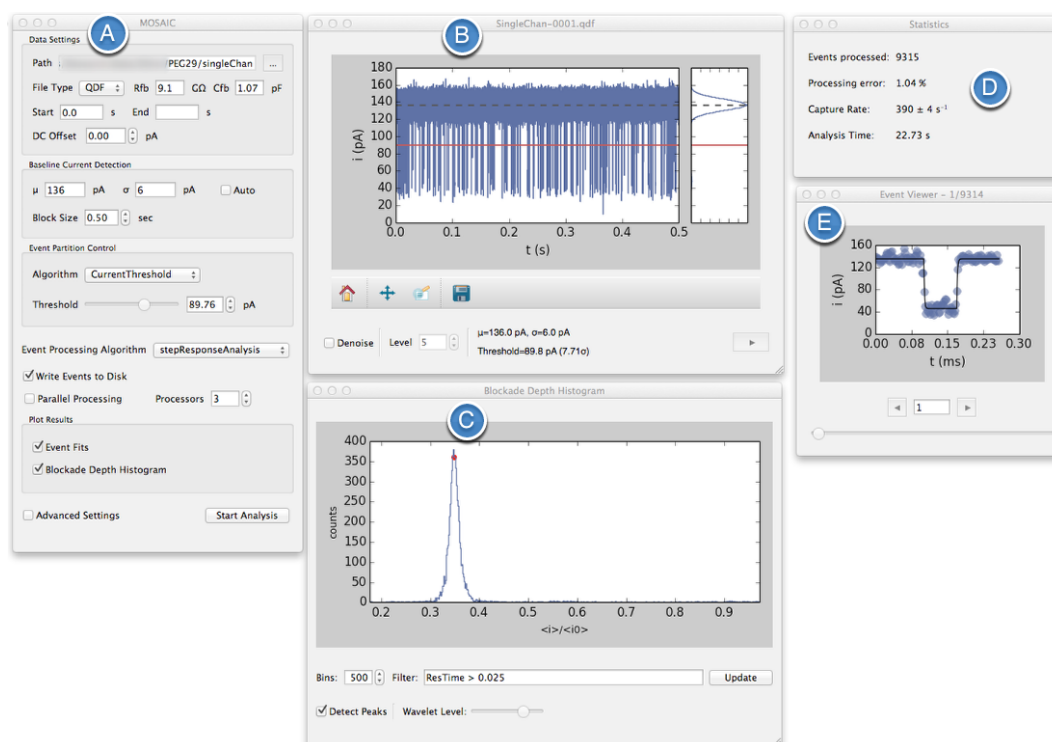


Figure 3.1: Primary panels in *MOSAIC*: (A) Analysis Setup (B) Trajectory Viewer (C) Live Blockade Depth Histogram (D) Live Analysis Statistics (E) Event Viewer.

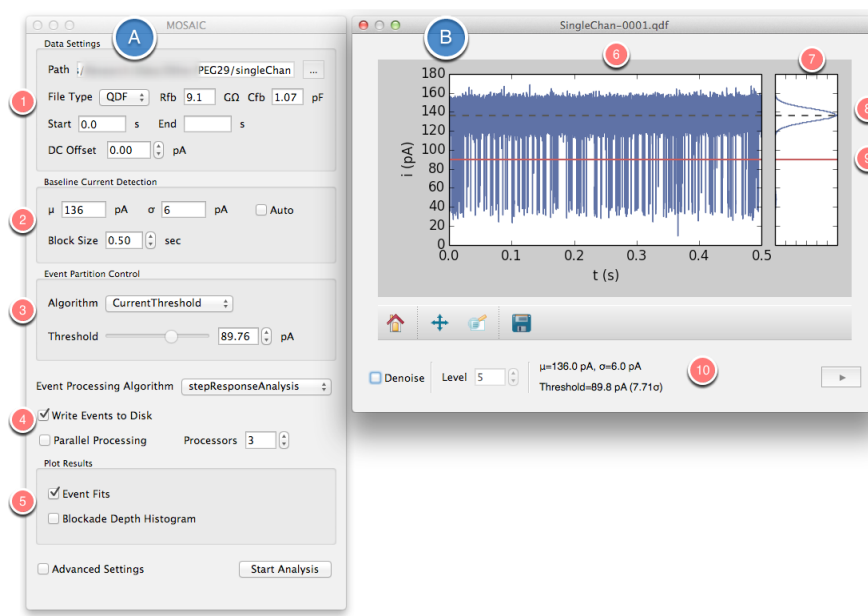


Figure 3.2: Overview of Panels A & B: (A) Analysis setup panel (B) Trajectory viewer panel

3.2 Panels A & B: Analysis Setup and Trajectory Viewer

3.2.1 Panel A: Analysis Setup

1. Data Settings

- **Path:** Allows user to set the directory containing files to analyze. Click the "..." icon to navigate to the directory.
- **File Type:** The GUI is natively compatible with either ABF or QDF Files, this field is automatically populated based on the files in the directory you've chosen. The **Rfb** and **Cfb** parameters are needed to correctly analyze QDF files (see [qdfTrajIO](#) for more information)
- **Rfb & Cfb:** MOSAIC supports the QUB QDF file format used by the [Electronic Biosciences](#) Nanopatch system. Two additional parameters, the feedback resistance (Rfb) in Ohms and capacitance (Cfb) in Farads are required to appropriately convert the measurements to ionic current.
- **Start and End:** These parameters allow you to analyze a range of your data. Choose the starting and ending times if you'd like to analyze a small time segment of your data. If this is left blank, all data will be analyzed.
- **DC Offset:** If your measurement contains a systematic bias, it can be manually corrected by entering the DC offset here.

2. Baseline Current Detection

- μ : Mean baseline current, in picoamperes (pA). This is shown schematically in the trajectory viewer (see Label #8). When *Auto* is selected, this will be greyed out and labeled `<auto>`
- σ : Noise level (in pA). This is expected noise level of your baseline. Typically one would set this to the measured RMS noise of the open channel state at the cutoff frequency. When *Auto* is selected, this will be greyed out and labeled `<auto>`.
- **Auto:** Checking this box enables automatic detection of the mean baseline current (μ) and noise level (σ). When auto is enabled, the values chosen by the software will be displayed in the trajectory viewer panel (see Label #10)
- **Block Size:** Controls the amount of data examined to determine the baseline. This also controls the amount of data shown in the trajectory viewer.

3. Event Partition Control

This panel is used to set the current threshold used for event detection

- **Algorithm:** Currently, the only event partitioning algorithm enabled is *CurrentThreshold*.
- **Threshold:** This is used to set the minimum current threshold used to partition events with the *CurrentThreshold* algorithm.

4. Event Processing Setup

Event Processing Algorithm: The GUI supports two event processing algorithms, i) *StepResponseAnalysis* and ii) *MultiStateAnalysis*. *StepResponseAnalysis* is the default analysis, and should be used with data sets with unimodal events. For events with multiple states or steps the *MultiStateAnalysis* algorithm, which is capable of automatically analyzing events with N states, should be used. Note that *StepResponseAnalysis* is a restricted case of *MultiStateAnalysis* and is more computationally efficient to run if you have unimodal (or single states) data.

- **Write Events to Disk:** When this box is checked, the data points for each partition events are written to the [SQLite](#) database. When this is checked it is possible to view the individual fits of each in the *Event Fits* panel.

Hint: When *Write Events to Disk* is checked, your database can become extremely large! This is because MOSAIC is effectively writing most of your time-series to the database. Note that the fit parameters are *always* written to the database.

- **Parallel Processing and Processors:** Parallel processing can be enabled by checking this box. This box will be greyed out if the python module *ZeroMQ* is not installed. The *Processors* box allows you to select the number of processors used in the analysis. It is important to note that the GUI will occupy one processor, so choosing 3 processors will actually use a total of 4 processors.

5. Plot Results and Advanced Settings

- **Event Fits:** Checking this box will show the events viewer (Panel E). This can also be accessed from the file menu *View>Plots>Event Fits*. If *Write Events to Disk* is not enabled this checkbox will be greyed out.
- **Blockade Depth Histogram:** Checking this box will show the blockade depth histogram (Panel C). This can also be accessed through the file menu *View>Plots>Blockade Depth Histogram*.
- **Advanced Settings:** This opens a dialog window to manually edit settings not otherwise accessible in the GUI. See the *Settings File* section for further details.

3.2.2 Panel B: Trajectory Viewer

This panel shows a segment of the data time series. The file currently being displayed is shown at the top of the window. If data from multiple files are loaded, the last filename is displayed. The length of time displayed in the window is controlled by *BlockSize* in Panel A (see #2).

6. Time Series (Trajectory)

- This plot shows the ionic current time series, of length *BlockSize*. Other features in the panel (such as histogram, denoising, etc.) only utilize the data in the window for their calculations.

7. All Points Histogram

- This shows a histogram of the time series data shown in #6.

8. Dashed line indicates mean baseline current

9. Detection threshold level indicated by solid red line

10. Navigation, Denoising, and Statistics

- **Navigation Tools:** Tools to navigate the plot window are shown below the time-series plot. These can be applied to either the trajectory or all points histogram plots. The arrow bar on the bottom right of the trajectory viewer can be used to advance to the next data block.
- **Denoising** Wavelet denoising can be activated by clicking , the denoising level is enabled here, the level of denoising can be varied between 1 and 5.

Warning: Wavelet-based denoising is currently an experimental feature and should be used with caution.

- **Baseline Statistics:** The mean baseline current, standard deviation, and the threshold used for event detection (specified as a multiple of the standard deviation in parenthesis) correspond to the settings in the main window. If the baseline current detection is set to *auto* these values will update as each data segment is examined. The size of this segment is determined by the *Block Size* setting. In the figure above, the *Block Size* is set to 0.5 s.

3.3 Panels C,D, & E: Blockade Depth Histogram, Statistics, and Event Viewer

3.3.1 Panel C: Blockade Depth Histogram

This window shows the blockade depth histogram calculated from the meta-data output by *MOSAIC*.

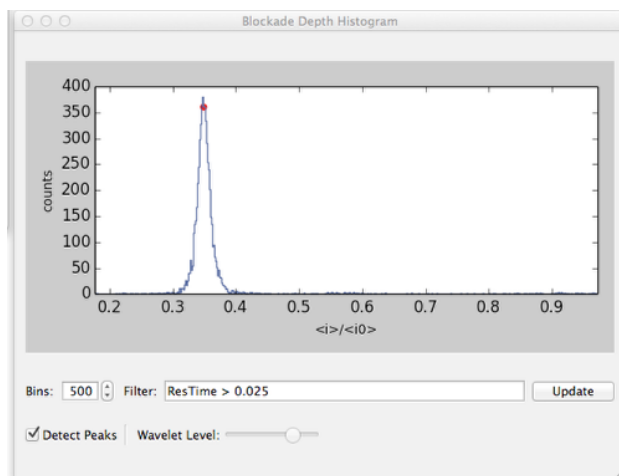


Figure 3.3: Blockade depth histogram

- **Filter:** The data displayed in the histogram can be restricted to events that fulfill specific user-defined criteria. For instance, the default filter `ResTime > 0.025` only includes events longer than 0.025 ms (or 25 μ s). The GUI uses a [SQL select](#) statement to restrict the events included in the histogram. The text in the *Filter* field represents the part of the query after the *where* clause, and allows the user to use standard [SQL](#) syntax to narrow the results in the plot. See the *working-with-sqlite-sec* section for details on [SQL](#) syntax.
- **Bins:** The number of bins in the histogram are defined here. By default, 500 bins are used, but the user can change this necessary.
- **Detect Peaks:** Checking *Detect Peaks* enables a wavelet-based peak detection algorithm. The wavelet level slider controls the sensitivity of the peak detection. Sliding it to the right will decrease the number of peaks picked up. The peaks detected are represented with red dots. Mousing over the detected peaks cause the coordinates of the peak to be displayed in the lower right hand corner of the window. The detected peaks can also be exported to a CSV file from the file menu `File>Save Histogram`.

3.3.2 Panel D: Statistics

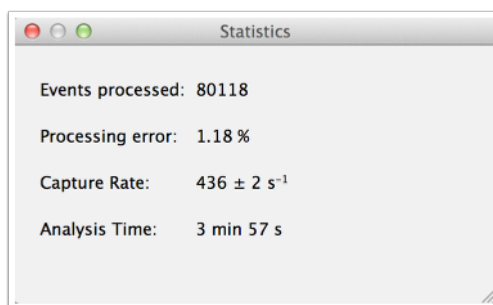


Figure 3.4: Live statistics window

The *Statistics Window* is displayed when a new analysis is started and displays:

- **Events Processed:** The number of events processed.
- **Processing Error:** The processing error rate (i.e. the percentage of events for which fit has failed).
- **Capture Rate:** An estimate of the mean capture rate.

- **Analysis Time:** The amount of data processed (in seconds).

3.3.3 Panel E: Event Viewer

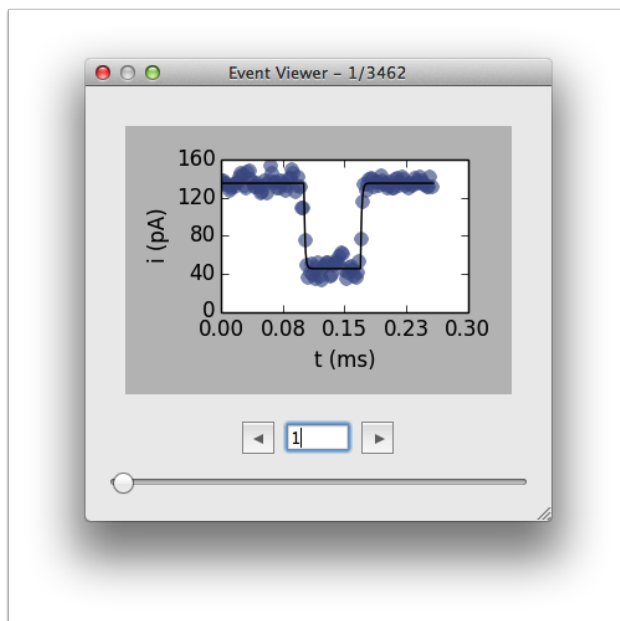


Figure 3.5: Event viewer window

If *Write to Disk* is enabled, this panel allows you to view the first 10,000 events processed. This is useful to ensure the quality of the analysis and to debug potential problems with the settings.

3.3.4 Console Log

When processing is complete, this panel displays a log of the analysis. This log contains useful information such as the analysis settings, the number of events fit, baseline drift, open channel conductance, etc. This file is written to the database and can be accessed later.

3.3.5 Advanced Settings

This dialog allows you to manually edit advanced settings for uncommon use cases not natively accessible from within the GUI. Further information can be found in the [Settings File](#).

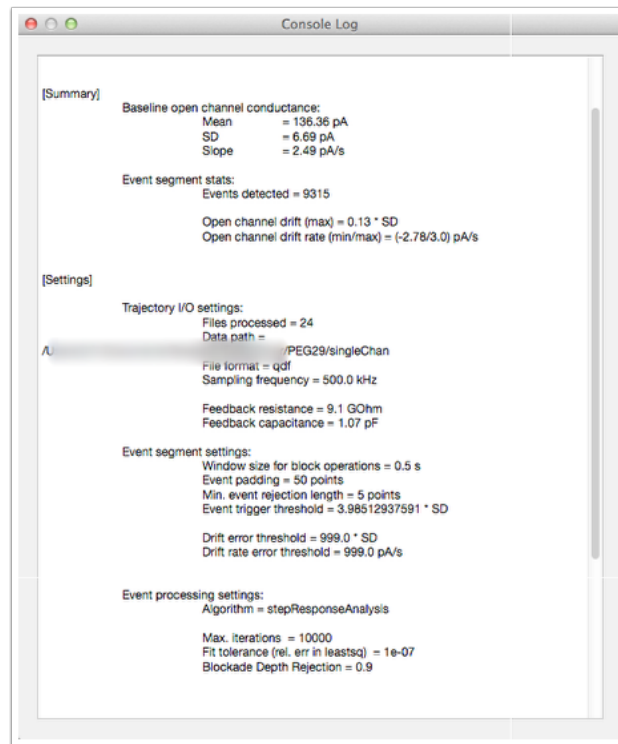


Figure 3.6: Console log window

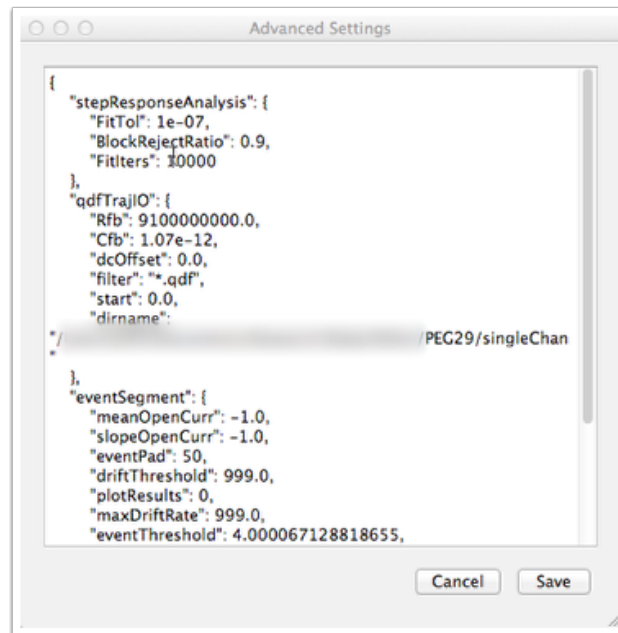


Figure 3.7: Advanced settings window

Scripting and Advanced Features

The analysis can be run from the command line by setting up a [Python](#) script. Scripting allows one to build additional analysis tools on top of *MOSAIC*. The first step is to import *MOSAIC* as shown below.

```
import mosaic
```

Alternatively, one can import sub-modules of *MOSAIC* directly into a script to access other parts of the system as shown below.

```
import mosaic.qdfTrajIO as qdf
import mosaic.abfTrajIO as abf

import mosaic.SingleChannelAnalysis
import mosaic.eventSegment as es
import mosaic.stepResponseAnalysis as sra
import mosaic.besselLowpassFilter as bessel
```

4.1 Import Data and Run an Analysis

Once the required modules are imported, a basic analysis can be run with the code snippet below. The top-level object that is used to configure and run a new analysis is `SingleChannelAnalysis`, which takes five arguments: i) the path to the data directory, ii) a handle to a *TrajIO* object that reads in data (e.g. `abfTrajIO`), iii) a handle to a data filtering algorithm (e.g. `besselLowpassFilter` or `None` for no filtering), iv) a handle to a partitioning algorithm (e.g. `eventSegment`) that partitions the data and v) a handle to a processing algorithm (e.g. `stepResponseAnalysis`) that processes individual blockade events.

```
# Process all ABF files in a directory
analysisObj=mosaic.SingleChannelAnalysis.SingleChannelAnalysis(
    '~/ReferenceData/abfSet1',
    abf.abfTrajIO,
    None,
    es.eventSegment,
    sra.stepResponseAnalysis
)
```

The analysis is started by calling the `Run()` function.

```
analysisObj.Run()
```

The code listing above analyzes all ABF files in the specified directory. Handles to trajectory I/O, data filtering, event partitioning and event processing are controlled with their corresponding sections in the *Settings File*. Default settings used to read ABF files are shown below.

```
"abfTrajIO" : {
    "filter"      : "*.abf",
    "start"       : 0.0,
    "dcOffset"    : 0.0
}
```

MOSAIC also supports the QUB QDF file format used by the [Electronic Biosciences](#) Nanopatch system. This is accomplished by replacing `abfTrajIO` in the previous example with `qdfTrajIO`. Settings for QDF files require two additional parameters to be specified in the settings file, the feedback resistance (Rfb) in Ohms and capacitance (Cfb) in Farads as described in the *API Documentation*. A sample section of the settings file to read QDF files, followed by Python code required to run an analysis, is shown below.

```
"qdfTrajIO": {
    "Rfb"          : 9.1e+12,
    "Cfb"          : 1.07e-12,
    "dcOffset"     : 0.0,
    "filter"       : "*.qdf",
    "start"        : 0.0
}

# Process all QDF files in a directory
mosaic.SingleChannelAnalysis.SingleChannelAnalysis(
    '~/ReferenceData/qdfSet1',
    qdf.qdfTrajIO,
    None,
    es.eventSegment,
    sra.stepResponseAnalysis
).Run()
```

Upon completion the analysis writes a log file to the directory containing the data. The log file summarizes the conditions under which the analysis were run, the settings used and timing information.

Start time: 2014-10-05 11:53 AM

[Status]

```
Segment trajectory: ***USER STOP***
Process events: ***NORMAL***
```

[Summary]

Baseline open channel conductance:

```
Mean   = 136.0 pA
SD     = 5.5 pA
Slope  = 0.0 pA/s
```

Event segment stats:

Events detected = 11306

Open channel drift (max) = 0.0 * SD

Open channel drift rate (min/max) = (-2.77/3.0) pA/s

[Settings]

```
Trajectory I/O settings:
Files processed = 27
```

```

Data path = ~/ReferenceData/qdfSet1
File format = qdf
Sampling frequency = 500.0 kHz

Feedback resistance = 9.1 GOhm
Feedback capacitance = 1.07 pF

Event segment settings:
  Window size for block operations = 0.5 s
  Event padding = 50 points
  Min. event rejection length = 5 points
  Event trigger threshold = 2.36363636364 * SD

  Drift error threshold = 999.0 * SD
  Drift rate error threshold = 999.0 pA/s

Event processing settings:
  Algorithm = stepResponseAnalysis

  Max. iterations = 50000
  Fit tolerance (rel. err in leastsq) = 1e-07
  Blockade Depth Rejection = 0.9

```

[Output]

```

Output path = ~/ReferenceData/qdfSet1
Event characterization data = ~/ReferenceData/qdfSet1/eventMD-20141005-115324.sqlite
Event time-series = ***enabled***
Log file = eventProcessing.log

```

[Timing]

```

Segment trajectory = 98.03 s
Process events = 0.0 s

Total = 98.03 s
Time per event = 8.67 ms

```

4.1.1 Filter Data

```

# Filter data with a Bessel filter before processing
mosaic.SingleChannelAnalysis.SingleChannelAnalysis(
    '~/ReferenceData/abfSet1',
    abf.abfTrajIO,
    bessell.bessellLowpassFilter,
    es.eventSegment,
    sra.stepResponseAnalysis
).Run()

```

MOSAIC supports filtering data prior to analysis. This is achieved by passing the *dataFilterHnd* argument to the `SingleChannelAnalysis` object. In the code above, the ABF data is filtered using a `bessellLowpassFilter`. Parameters for the filter are defined within the settings file as described in the [Settings File](#) section.

```
"besselLowpassFilter" : {  
    "filterOrder"      : "6",  
    "filterCutoff"     : "10000",  
    "decimate"         : "1"  
}
```

A similar approach can be used to filter data using a `waveletDenoiseFilter` or a tap delay line (`convolutionFilter`). Additional filters can be easily added to *MOSAIC* as described in *Extend MOSAIC*.

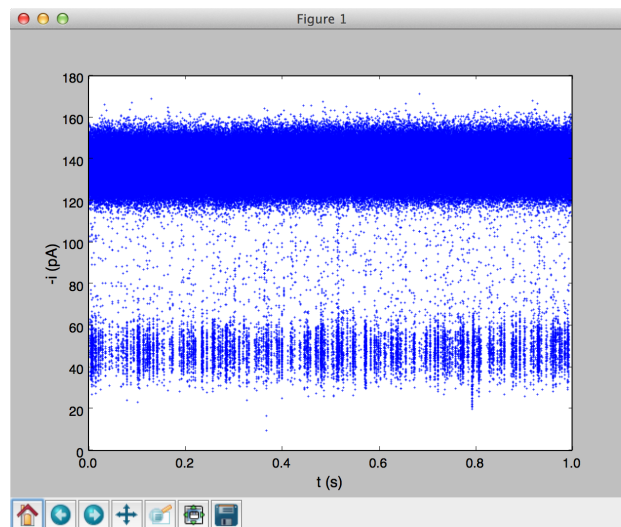
4.2 Advanced Scripting

Scripting with `Python` allows transforming the output of the *MOSAIC* further to generate plots, perform additional analysis or extend functionality. Moreover, individual components of the *MOSAIC* module, which forms the back end code executed in the data processing pipeline, can be used for specific tasks. In this section, we highlight a few typical use cases.

Plot the Ionic Current Time-Series

```
import mosaic.abfTrajIO as abf  
import matplotlib.pyplot as plt  
import numpy as np  
  
abfDat=abf.abfTrajIO(dirname='~/abfSet1/', filter='*.abf')  
plt.plot( np.arange(0,1,1/500000.), b.popdata(500000), 'b.', markersize=2 )  
plt.xlabel("t (s)", fontsize=14)  
plt.ylabel("-i (pA)", fontsize=14)  
plt.show()
```

It is useful to visualize time-series data to highlight unique characteristics of a sample. For example the sample code above was used to load 1 second of monodisperse PEG28 data, sampled at 500 kHz. The data was read using a `abfTrajIO` object similar to the examples above. The `popdata()` command was used to take 500k data points (or 1 second) and then plot a time-series using `matplotlib` (see figure below). Calling `popdata()` again will return the next n points.



Estimate the Channel Gating Duration

Scripting can be used to obtain statistics from the raw time-series. In the code snippet below, we estimate the amount of time a channel spends in a gated state by combining modules defined within *MOSAIC*. The analysis is performed in blocks for efficiency. We first define a Python function that takes multiple arguments including *TrajIO* object, the threshold at which we want to define the gated state in pA (*gatingcurrentpa*), the block size in seconds (*blocksz*), the total time of the time-series being processed in seconds (*totaltime*) and the sampling rate of the data in Hz (*fshz*). The function then calculates the number of blocks in which the channel was in a gated state and returns the time spent in that state in seconds.

```
import mosaic.abfTrajIO as abf
import numpy as np

def estimateGatingDuration( trajioobj, gatingcurrentpa, blocksz, totaltime, fshz ):
    npts = int((fshz)*blocksz)
    nblk = int(totaltime/blocksz)-1

    # Iterate over the blocks of data and check if the channel was in a gated state.
    # The code below returns the mean ionic current of blocks that are below the gating
    # threshold (gatingcurrentpa)
    gEvents = filter( lambda x:x<float(gatingcurrentpa),
                      [ np.mean(trajioobj.popdata(npts)) for i in range(nblk) ])

    return len(gEvents)*blocksz

abfObj=abf.abfTrajIO(dirname='~/abfSet1',filter='*.abf')
print estimateGatingDuration( abfObj, 20., 0.25, 100, abfObj.FsHz )
```

Plot the Output of an Analysis

This final example shows how one can use *MOSAIC* to process an ionic current time-series and then build a custom script that further analyses and plots the results. This example uses single-molecule mass spectrometry (SMMS) data [RRS+07], described in more detail in the *Single Molecule Mass Spectrometry* section .

In the code below, we first process all the ABF files in a specified directory similar to the examples in previous sections. Upon completion of the analysis, the results are stored in a *SQLite* database, which can be then queried using the structured query language (SQL).

```
import mosaic.qdfTrajIO as qdf
import mosaic.abfTrajIO as abf

import mosaic.SingleChannelAnalysis
import mosaic.eventSegment as es
import mosaic.stepResponseAnalysis as sra

import glob
import pylab as pl
import numpy as np
import mosaic.sqlite3MDIO as sql

# Process all ABF files in a directory
mosaic.SingleChannelAnalysis.SingleChannelAnalysis(
    '~/ReferenceData/abfSet1',
    abf.abfTrajIO,
    None,
    es.eventSegment,
    sra.stepResponseAnalysis
).Run()

# Load the results of the analysis
```

```

s=sql.sqlite3MDIO()
s.openDB(glob.glob("~/ReferenceData/abfSet1/*.sqlite")[-1])

# We first set up a string that holds the query to retrieve the analysis results. Note that {col}
# will be replaced with the name of the database column when we run the query below.
q = "select {col} from metadata where ProcessingStatus='normal' and ResTime > 0.2 \
    and BlockDepth between 0.15 and 0.55"

# Now we run two separate queries - the first returns the blockade depth
# and the second returns the residence time. Note that we simply take the query
# string 'q' above and replace {col} with the column name.
x=np.hstack( s.queryDB( q.format(col='BlockDepth') ) )
y=np.hstack( s.queryDB( q.format(col='ResTime') ) )

# Use matplotlib to plot the results with 2 views:
# i) a 1D histogram of blockade depths and
# ii) a 2D histogram of the residence times vs. blockade depth
fig = pl.gcf()
fig.canvas.set_window_title('Residence Time vs. Blockade Depth')

pl.subplot(2, 1, 1)
pl.hist(x, bins=500, histtype='step', rwidth=0.1)
pl.xticks(())
pl.ylabel("Counts", fontsize=14)

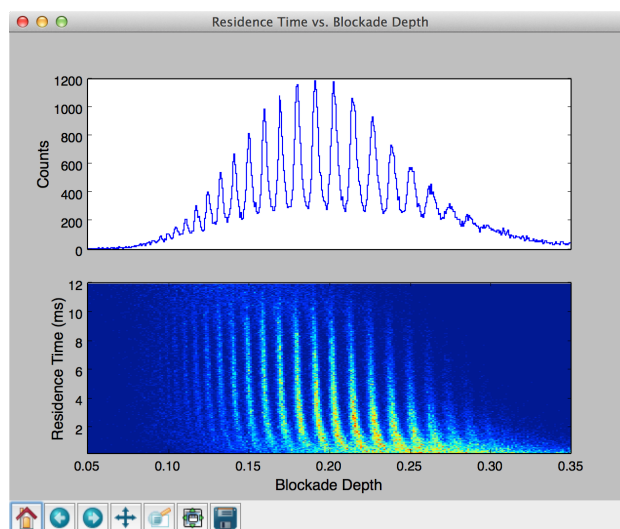
pl.subplot(2, 1, 2)
pl.hist2d(x,y, bins=500)

pl.xlabel("Blockade Depth", fontsize=14)
pl.ylabel("Residence Time (ms)", fontsize=14)
pl.ylim([0.2, 20])

pl.show()

```

Running the code above generates a two pane plot using `matplotlib`. The top pane contains a histogram of the blockade depth, while the bottom pane plots a 2D histogram of residence time vs. blockade depth.



Settings File

MOSAIC stores its settings in the **JSON** format. When using the graphical interface, a settings file is generated automatically upon starting an analysis, or by clicking *Save Settings* in the *File menu* (see *MOSAIC GUI*).

5.1 Settings Layout

JSON is a human readable file format that consists of key-value pairs separated by sections. Each section in a JSON object consists of a section name and a list of string key-value pairs.

```
{
    "<section name>" : {
        "key1" : "value1",
        "key2" : "value2",
        ...
    }
}
```

MOSAIC settings define a new section for each class, with key-value pairs corresponding to class attributes that are set upon initialization. This is illustrated below for the `stepResponseAnalysis` class. The `stepResponseAnalysis` section in the settings file holds parameters corresponding to the `stepResponseAnalysis` class. Note that the section name in the settings file is identical to the corresponding class name. Three parameters are then defined within the section that control the behavior of the class.

```
{
    "stepResponseAnalysis" : {
        "FitTol" : "1.e-7",
        "FitIters" : "50000",
        "BlockRejectRatio" : "0.9"
    }
}
```

Finally, `stepResponseAnalysis` is initialized by defining class attributes corresponding to the key-value pairs in the settings file.

```
try:
    self.FitTol=float(self.settingsDict.pop("FitTol", 1.e-7))
    self.FitIters=int(self.settingsDict.pop("FitIters", 5000))
```

```
        self.BlockRejectRatio=float(self.settingsDict.pop("BlockRejectRatio", 0.8))

except ValueError as err:
    raise commonExceptions.SettingsTypeError( err )
```

5.2 Default Settings

```
{
    "eventSegment" : {
        "blockSizeSec"           : "0.5",
        "eventPad"               : "50",
        "minEventLength"         : "5",
        "eventThreshold"         : "6.0",
        "driftThreshold"         : "999.0",
        "maxDriftRate"           : "999.0",
        "meanOpenCurr"           : "-1",
        "sdOpenCurr"             : "-1",
        "slopeOpenCurr"          : "-1",
        "writeEventTS"           : "1",
        "parallelProc"           : "0",
        "reserveNCPUs"           : "2"
    },
    "singleStepEvent" : {
        "binSize"                : "1.0",
        "histPad"                : "10",
        "maxFitIters"             : "5000",
        "a12Ratio"               : "1.e4",
        "minEvtTime"              : "10.e-6",
        "minDataPad"             : "75"
    },
    "stepResponseAnalysis" : {
        "FitTol"                 : "1.e-7",
        "FitIters"               : "50000",
        "BlockRejectRatio"       : "0.9"
    },
    "multiStateAnalysis" : {
        "FitTol"                 : "1.e-7",
        "FitIters"               : "50000",
        "InitThreshold"          : "5.0"
    },
    "besselLowpassFilter" : {
        "filterOrder"            : "6",
        "filterCutoff"           : "10000",
        "decimate"               : "1"
    },
    "waveletDenoiseFilter" : {
        "wavelet"                : "sym5",
        "level"                  : "5",
        "thresholdType"          : "soft",
        "thresholdSubType"       : "sqtwolog"
    },
    ,
    "abfTrajIO" : {
        "filter"                 : "/*.abf",
        "start"                  : 0.0,
        "dcOffset"               : 0.0
    }
}
```

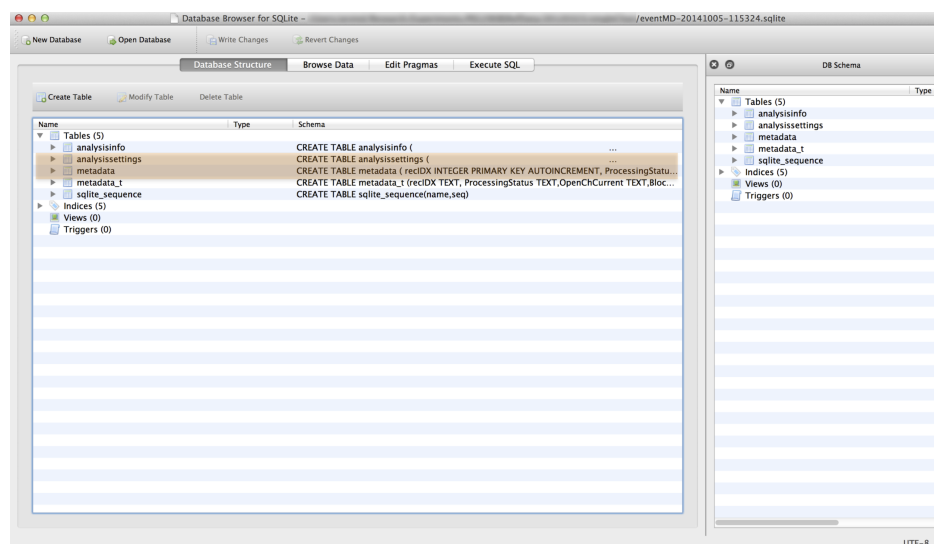


```
    },  
    "qdfTrajIO": {  
        "Rfb"           : 9.1e+12,  
        "Cfb"           : 1.07e-12,  
        "dcOffset"      : 0.0,  
        "filter"        : "*.qdf",  
        "start"         : 0.0  
    }  
}
```


Database Structure and Query Syntax

MOSAIC stores the output of an analysis in a *SQLite* database. Database files are stored in the same directory as the data being processed. Each analysis creates a new database file named *eventMD-<date>-<time>.sqlite*, where *<date>* is the date the analysis was performed (e.g. 20140929 for Sep 29, 2014) and *<time>* is the analysis start time (e.g. 112937 for 11:29:37 AM).

SQLite databases store data in tables similar to spreadsheets, where each table is analogous to a sheet in an Excel spreadsheet. Databases generated by *MOSAIC* can be inspected using a database viewer, for example the open source *DB browser for SQLite*. *MOSAIC* outputs databases with multiple tables as seen from the figure below. Databases output by *MOSAIC* contain four tables: i) *analysisinfo* contains general information about the analysis such as the data path, analysis algorithm etc., ii) *analysissettings* contains a *JSON* formatted string with the analysis settings, iii) *metadata* holds the output of the analysis, and iv) *metadata_1* lists the data types for each column in *metadata*. Two tables most relevant to the analysis (*metadata* and *analysissettings*) are discussed in detail below.



6.1 Metadata Table

The *metadata* table contains the primary output of the analysis. *MOSAIC* processes individual blockade events from a time-series of ionic current. The parameters describing each event (or metadata) are stored in individual rows of

the *metadata* table in the database file. The column names describe the metadata and are unique to the processing algorithm used. For example, the column names for the *stepresponse-page* algorithm are shown below. The column names for *multistate-page* differ from this list.

```
{
    ProcessingStatus,
    OpenChCurrent,
    BlockedCurrent,
    EventStart,
    EventEnd,
    BlockDepth,
    ResTime,
    RiseTime,
    AbsEventStart,
    RedChiSq,
    TimeSeries
}
```

Note that the column names can be used in constructing queries passed to SQLite, and is described in more detail in the *working-with-sqlite-sec* section and the *Scripting and Advanced Features* section. The first example SQL query below returns the *BlockDepth* column (ratio of *BlockedCurrent* to *OpenChCurrent*). One can imagine assembling more complex queries for example restricting the results to events whose residence time is greater than 0.2 ms as seen from the second example query below.

```
select BlockDepth from metadata where ProcessingStatus='normal'
```

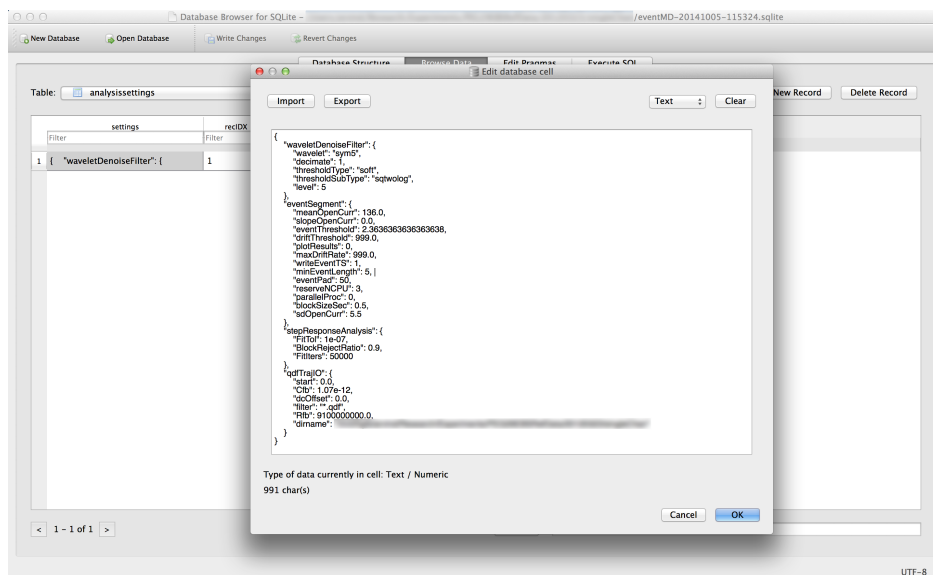
```
select BlockDepth from metadata where ProcessingStatus='normal' and ResTime > 0.2
```

A typical *metadata* table for the *stepresponse-page* algorithm is shown below. The *ProcessingStatus* column is a text field that should read *normal* if the fit for a particular event was successful. If a failure occurred the corresponding error code (e.g. *eInvalidFitParameters*) is stored and all other columns (except *TimeSeries*) are set to -1. If event time-series storage was requested, then the *TimeSeries* column will store the ionic current data for that entry in binary format.

recIDX	ProcessingStatus	OpenChCurrent	BlockedCurrent	EventStart	EventEnd	BlockDepth	ResTime	RConstant	AbsEventStart	duceChSquan	TimeSeries
184	normal	136.048481...	47.8884398...	0.102442...	0.2063717...	0.3519954...	0.1039288...	0.0017467...	458.10244...	1.0428571...	kpkLTtWwY...
185	normal	136.388727...	49.4119037...	0.101160...	0.2531239...	0.3622872...	0.1519637...	0.0020683...	468.72916...	1.0333333...	Snwks2FRY...
186	normal	136.295727...	48.7546930...	0.099610...	0.1772262...	0.3577125...	0.0776155...	0.0010883...	469.76761...	1.0451127...	vCqthIEVU...
187	normal	136.728134...	48.4160741...	0.099204...	0.1834360...	0.3541046...	0.0842319...	0.0030160...	473.43920...	1.0483870...	+HTs4mq...
188	normal	136.163621...	75.2090860...	0.075814...	0.0837541...	0.5523434...	0.0079394...	0.0001326...	473.60581...	1.0722891...	xR+Ary2Y...
189	normal	136.479801...	98.4793567...	0.099866...	0.1059172...	0.7215672...	0.0060503...	0.0001528...	474.81986...	1.0645161...	ymZDSQY...
190	normal	135.580797...	49.4561541...	0.099369...	0.2429424...	0.3647725...	0.1435727...	0.0014158...	478.93336...	1.0348837...	0lhycrIEA...
191	eInvalidFitParams	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	-1.0	o2rFWsqY...
192	normal	136.060431...	48.0983706...	0.011056...	0.1306499...	0.3535074...	0.1195933...	0.0031418...	479.48305...	1.0540540...	REVstQgY...
193	normal	135.394986...	48.6723594...	0.099333...	0.2170996...	0.3594842...	0.1177660...	0.0023736...	480.64333...	1.0425531...	x8pPu49QY...
194	normal	136.532025...	49.9108603...	0.079606...	0.1311501...	0.3655615...	0.0515433...	0.0020511...	480.84560...	1.0512820...	AqNGvByAY...
195	normal	136.016300...	49.8596692...	0.098892...	0.1492097...	0.3665712...	0.0503174...	0.0015597...	481.54889...	1.0512820...	eDSY4ajfYE...
196	normal	136.775120...	47.7841358...	0.101028...	0.2825822...	0.3493627...	0.1815534...	0.0016714...	482.09102...	1.0315789...	r+DU/bKky...
197	normal	137.516752...	56.9740218...	0.100124...	0.1413854...	0.4143060...	0.0412613...	0.0012905...	483.30812...	1.0545454...	b7wAu0EY...
198	normal	135.728152...	58.8360340...	0.101791...	0.1117746...	0.4334843...	0.0099829...	0.0001607...	485.43179...	1.0612244...	ImOSccrY...
199	normal	137.152200...	46.6580419...	0.099647...	0.2092531...	0.3401917...	0.1096055...	0.0025605...	486.42764...	1.0410958...	o/axZY9yY...
200	normal	137.110974...	48.7001968...	0.099491...	0.1532354...	0.3551881...	0.0537435...	0.0021878...	487.13949...	1.0483870...	s7gWEGpY...

6.2 Analysis Settings Table

The *analysissettings* table contains a single text entry that stores the settings file for the analysis. This allows any database opened with the *MOSAIC* GUI to retrieve settings that correspond to the analysis results in the file. As seen from the figure below, the settings file is in the **JSON** format as described in the *Settings File* documentation.



6.3 Work with SQLite

MOSAIC stores the output of an analysis in a SQLite database as described in the *Database Structure and Query Syntax* section. Interacting with the data through the **Structured Query Language (SQL)** is a flexible approach to further analyze or plot the output. Here we provide a few detailed examples of the common ways in which the output of *MOSAIC* can be queried for further processing. While this section is not a comprehensive SQL tutorial, it provides common use cases to allow you to get started.

One way to retrieve data from a **SQLite** database is to use the *select* command. In its simplest form, a *select* query can return the entire contents of a table using the syntax below. The statement below selects all columns (*select **) from the table specified by *<tablename>*.

```
select * from <tablename>
```

The power of SQL lies in its ability to restrict results to match specific criteria. This is accomplished with the *where* clause described next. SQL queries can be very fast even for large databases. It is often desirable to only include events that were successfully fit in a plot or other analysis. All *eventprocess-page* algorithms implemented in *MOSAIC* store a *ProcessingStatus* column in the output database. This enables one to easily query events that were successfully processed. This is easily accomplished with the query below, which returns all columns for events that were successfully processed (*ProcessingStatus=normal*).

```
select * from metadata where ProcessingStatus='normal'
```

It is not always necessary to retrieve every column for events that fit a certain criteria. For example, *gui-blockdepth-sec* in the GUI displays a histogram of the blockade depths that match a user specified criteria. This is accomplished within the GUI by a query similar to the one shown below. There are two important differences between the query below and previous examples: i) by replacing *** with *BlockDepth*, we only retrieve the *BlockDepth* column for events

that meet the criteria specified after the *where* clause, and ii) selection criteria specified after where can be compound statements or even nested as seen in the examples below.

```
select BlockDepth from metadata where ProcessingStatus='normal' and ResTime > 0.2
```

```
select BlockDepth from metadata where ProcessingStatus='normal' and ResTime > 0.2  
and BlockDepth between 0.1 and 0.5
```

Multiple columns can be retrieved from a table by providing a comma separated list of column names after the *select* clause. As in previous cases, only events that meet a specified criteria are returned. The results can be ordered using *order*. In this example we sort the results in ascending order by the *AbsEventStart* column.

```
select BlockDepth, ResTime, AbsEventStart from metadata where ProcessingStatus='normal'  
order by AbsEventStart ASC
```

Finally, SQL allows the number of results returned to be limited using the *limit* clause. In this example, we limit the query results to the first 500 rows that meet our criteria.

```
select AbsEventStart from metadata where ProcessingStatus='normal'  
order by AbsEventStart ASC limit 500
```

Extend *MOSAIC*

MOSAIC was designed from the start using object oriented tools, which makes it easy to extend. *Meta-Classes* define interfaces to five key parts of *MOSAIC*: time-series IO (`metaTrajIO`), time-series filtering (`metaIOFilter`), analysis output (`metaMDIO`), event partition and segmenting (`metaEventPartition`), and event processing (`metaEventProcessor`). Sub-classing any of these meta classes and implementing their interface functions allows one to extend *MOSAIC* while maintaining compatibility with other parts of the program. We highlight these capabilities via two examples. In the first example, we show how one can extend `metaTrajIO` to read arbitrary binary files. In the second example, we implement a new top-level class that converts files to the comma separated value (CSV) format.

7.1 Read Arbitrary Binary Data Files

In this first example, we implement a class that can read an arbitrary binary data file and make its data available via the interface functions in `metaTrajIO`. This allows the newly implemented binary data to be used across *MOSAIC*. A complete listing of the code used in this example (`binTrajIO`) is available in the API documentation.

The new binary IO class is implemented by sub-classing `metaTrajIO` as shown in the listing below.

```
class binTrajIO(metaTrajIO.metaTrajIO):
```

Next, we must fully implement the `metaTrajIO` interface functions (`__init__()`, `readdata()` and `__formatsettings__()`). Note that the arguments of each function must match their corresponding base-class versions. For example the `__init__()` function only accepts keyword arguments and is defined as shown below.

```
def __init__(self, **kwargs):
```

The `__init__()` function checks the arguments passed to `kwargs` and raises an exception if they are not defined.

```
    if not hasattr(self, 'SamplingFrequency'):
        raise metaTrajIO.InsufficientArgumentsError("{0} requires the sampling rate in Hz to be defined")
    if not hasattr(self, 'PythonStructCode'):
        raise metaTrajIO.InsufficientArgumentsError("{0} requires the Python struct code to be defined")
```

Next we define the `readdata()` function that reads in the data and stores the results in a numpy array. This array is then passed back to the calling function.

```
def readdata(self, fname):
    tempdata=np.array([])
```

```
# Read binary data and add it to the data pipe
for f in fname:
    tempdata=np.hstack(( tempdata, self.readBinaryFile(f) ))

return tempdata
```

Finally, we implement the `_formatsettings()` that returns a formatted string of the settings used to read in binary data.

```
def _formatsettings(self):
    """
        Return a formatted string of settings for display
    """
    fmtstr=""

    fmtstr+='\n\t\tAmplifier scale = {0} pA\n'.format(self.AmplifierScale)
    fmtstr+='\n\t\tAmplifier offset = {0} pA\n'.format(self.AmplifierOffset)
    fmtstr+='\n\t\tHeader offset = {0} bytes\n'.format(self.HeaderOffset)
    fmtstr+='\n\t\tData type code = \' {0} \'\n'.format(self.PythonStructCode)

    return fmtstr
```

The newly defined `binTrajIO` class can then be used as shown below and in *Scripting and Advanced Features*.

```
# Process all binary files in a directory
mosaic.SingleChannelAnalysis.SingleChannelAnalysis(
    "~/RefData/binSet1/",
    bin.binTrajIO,
    None,
    es.eventSegment,
    sra.stepResponseAnalysis
).Run()
```

Similar to other *TrajIO* objects, parameters for `binTrajIO` are obtained from the settings file when used with `SingleChannelAnalysis`. Example settings for `binTrajIO` that read 16-bit integers from a binary data file, assuming 50 kHz sampling, are shown below.

```
"binTrajIO" : {
    "filter"                : "*bin",
    "AmplifierScale"         : "1.0",
    "AmplifierOffset"       : "0.0",
    "SamplingFrequency"     : "50000",
    "HeaderOffset"          : "0",
    "PythonStructCode"      : "'h'"
}
```

7.2 Define Top-Level Functionality

New functionality can be added to *MOSAIC* by combining other parts of the code. One way of accomplishing this is by defining new top-level functionality as shown in the following example. We define a new class that converts data from one of the supported data formats to comma separated text files (CSV). A complete listing of the `ConvertToCSV` class in this example is available in the API documentation.

The `__init__` function of `ConvertToCSV` class accepts two arguments: a `trajIO` object and the location to save the converted files. If the output directory is not specified, the data is saved in the same folder as the input data. The data conversion is performed by the `Convert()` function, which saves the data in blocks controlled by the `blockSize`

parameter. `Convert()` saves each block to a new CSV file, named with the filename of the input data followed by an integer number (see the API documentation for `__filename()` for additional details).

```
class ConvertToCSV(object):
    def __init__(self, trajDataObj, outdir=None):
        self.trajDataObj=trajDataObj
        self.datPath=trajDataObj.datPath

        # If outdir is None, save the CSV files to the same directory as the data.
        if outdir==None:
            self.outDir=self.datPath
        else:
            self.outDir=outdir

        self.filePrefix=None
        self._creategenerator()

    def Convert(self, blockSize):
        data=numpy.array([], dtype=numpy.float64)

        try:
            while(True):
                (self.trajDataObj.popdata(blockSize)).tofile(
                    self.__filename(),
                    sep=', '
                )
        except EmptyDataPipeError:
            pass
```

The `ConvertToCSV` class can now be used with any `trajIO` object as seen below.

```
ConvertToCSV( abfTrajIO(dirname="~/RefData/abfSet1/", filter="*abf") ).Convert(
    blockSize=50000)

ConvertToCSV( qdfTrajIO(dirname="~/RefData/qdfSet1/", filter="*qdf", Rfb="2.1E+9",
    Cfb="1.16E-12") ).Convert(blockSize=50000)

ConvertToCSV( binTrajIO(dirname="~/RefData/binSet1/", filter="*bin", AmplifierScale=1.0,
    AmplifierOffset=0.0, SamplingFrequency=50000, HeaderOffset=0,
    PythonStructCode='h') ).Convert(blockSize=50000)
```

Finally, since `ConvertToCSV` accepts a `trajIO` object, we can apply a lowpass filter to the data before converting it to the CSV format. This is accomplished by passing the `datafilter` option to the `trajIO` object as described in the [Filter Data](#) section. In the example below, we convert ABF files to the CSV format after applying a lowpass Bessel filter to the data.

```
ConvertToCSV( abfTrajIO(dirname="~/RefData/abfSet1/", filter="*abf",
    datafilter=mosaic.besselFilter
) ).Convert(blockSize=50000)
```


The output of *MOSAIC* is often processed further to generate plots or perform more sophisticated analysis. We facilitate this process by providing addon packages that make it easy to import the *SQLite* database generated by a *MOSAIC* analysis into *mathematica-addons-sec*, *matlab-addons-sec* or *igor-addons-sec*. The interfaces for these programs are described in more detail in this section.

8.1 Mathematica

8.1.1 Installation

The analysis output generated by *MOSAIC* can be imported into *Mathematica* for further processing. This accomplished with two packages: the low level *mathematicaMosaicutilsSec* and *mathematica-mosaicanalysis-sec*, which contains additional analysis routines. The addon package must first be installed to one of the locations in the *Mathematica* path. Alternatively, the required package files can be installed to the *Applications* folder using *setuptools* on Mac OS X and Linux by issuing the command below in the root folder of the *MOSAIC* code. Instructions for installing the package files for Windows are available [here](#).

```
python setup.py mosaic_addons --mathematica
```

8.1.2 MosaicUtils

MosaicUtils provides low level functions to interact with a database output by *MOSAIC*.

PrintMDKeys[*dbfile*]

Returns a list of column headings from the *metadata* table.

Args

- *dbfile* : full path to the database file

Returns A list of column names in the table *metadata*.

PrintMDTypes[*dbfile*]

Returns a list of column types from the *metadata* table.

Args

- *dbfile* : full path to the database file

Returns A list of column types in the table *metadata*.

QueryDB[*dbfile*, *query*]

Queries the *metadata* table using the supplied SQL query.

Args

- *dbfile* : full path to the database file
- *query* : a SQL query

Returns A nested list of query results.

PlotEvents[*dbfile*, *FsKHz*, *options*]

Plot the event-time series if stored in the database (see the [Settings File](#) section for details on saving time-series to the analysis output).

Args

- *dbfile* : full path to the database file
- *FsKHz* : sampling frequency in kHz.
- *options*: (optional) *AnalysisAlgorithm*-> "MultiStateAnalysis" for *multiStateAnalysis, not supplied for the *stepResponseAnalysis* algorithm.

Returns A dynamic object that allows the user to browse event time-series and fits.

MosaicUtils Examples

Once installed as described above, *MosaicUtils* must be imported as shown below.

```
In[1]= <<MosaicUtils`
```

SQL queries require the exact column names when querying data from a table (see [Database Structure and Query Syntax](#)). Column names in the *metadata* table, which stores the main results from the analysis can be retrieved using the *PrintMDKeys* function as shown below. In this example, the column names returned correspond to an analysis performed using the *stepResponseAnalysis* algorithm.

```
In[2]= PrintMDKeys["<mosaicroot>/data/eventMD-PEG29-Reference.sqlite"]
```

```
Out[2]= {"recIDX", "ProcessingStatus", "OpenChCurrent",
         "BlockedCurrent", "EventStart", "EventEnd",
         "BlockDepth", "ResTime", "RCConstant", "AbsEventStart",
         "ReducedChiSquared", "TimeSeries"
        }
```

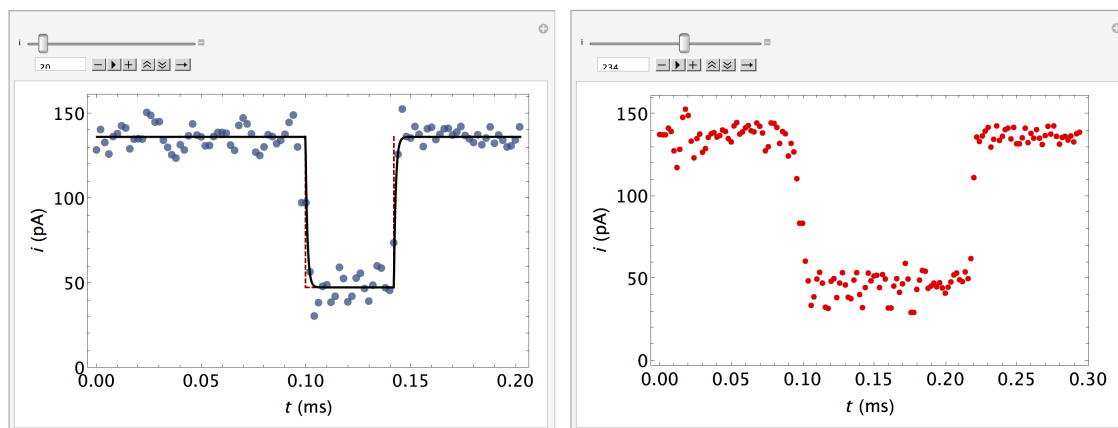
The *MosaicUtils* package allows the output of *MOSAIC* to be queried just like from Python. This accomplished using the *QueryDB* function. In the example below, we retrieve a column that returns the start time of the first 10 entries in the *metadata* table that have their *ProcessingStatus* set to *normal*. The results are then returned in a standard list. Note that *QueryDB* accepts a standard SQL query as described in more detail in the [Database Structure and Query Syntax](#) section.

```
In[3]= QueryDB[
    "<mosaicroot>/data/eventMD-PEG29-Reference.sqlite",
    "select AbsEventStart from metadata where ProcessingStatus='normal' limit 10"
]

Out[3]= {
    {1.84376}, {4.54439}, {5.26933}, {6.01253}, {6.80369},
    {8.48988}, {10.841}, {11.2246}, {13.2892}, {16.3983}
}
```

Finally, the addon package allows us to plot individual events if time-series data was stored in the database. This is accomplished using the `PlotEvents` function, and provides a convenient tool to visually inspect the output of a *MOSAIC* analysis. In the example below, we inspect the events stored in the reference PEG28 data set included with *MOSAIC*. `PlotEvents` returns a dynamic object that allows the user to inspect all the events in a database. An event that was properly characterized by the code is plotted with *blue* markers (*left*). The plot is overlaid with the optimized fit function (*black*) and an idealized pulse (*red dashed*). Events that were not properly fit are plotted with *red* markers (*right*).

```
In[4]= PlotEvents["<mosaicroot>/data/eventMD-PEG29-Reference.sqlite", 500]
```



8.1.3 MosaicAnalysis

MosaicAnalysis builds on the *MosaicUtils* package and provides basic analysis functions such as estimating the capture rate of molecules partitioning into a channel, or the mean residence time. Additionally, new functionality can be created by combining the functions defined below.

ScaledSingleExponentialFit`[hist, lambda, lambda0]`

Scale the histogram with the number of counts in the first bin. Fit a single exponential of the form $a \exp(-t/\tau)$ to the scaled histogram.

Args

- *hist* : a histogram with format `{{bin1, counts1}, {bin2, counts2}, ..., {binN, countsN}}`
- *lambda* : parameter of the distribution. This symbol must be passed from the calling function.
- *lambda0* : initial guess for *lambda*.

PlotScaledSingleExponentialFit`[hist, ftfunc, plotopts]`

Args

- *hist* : a histogram with format `{{bin1, counts1}, {bin2, counts2}, ..., {binN, countsN}}`
- *ftfunc* : an optimized fit, defined as a virtual function.
- *plotopts*: a list of options to control the plot output.

CaptureRate`[arrtimes, stime, etime, nbins, plotopts]`

Estimate the capture rate of molecules by a channel by analyzing the arrival times of individual molecules. The arrival times of a stochastic process follow a single exponential distribution. This function first calculate a histogram of arrival times and then fits a single exponential function to the data.

Args

- *arrtimes* : a list of absolute start times (*AbsEventStart*) queried from a database.
- *stime* : lower limit of the arrival times distribution
- *etime* : upper limit of the arrival times distribution
- *nbins* : number of bins
- *plotopts* : a list of options to control the plot output.

Returns The mean capture rate, a plot of the underlying distribution of arrival times, the arrival times distribution and the optimized fit function.

ArrivalTimes[*abseventstart*]

Calculate the arrival times from a list of the absolute start time of each event in a data set.

Args

- *abseventstart* : a list of absolute start times (*AbsEventStart*) queried from a database.

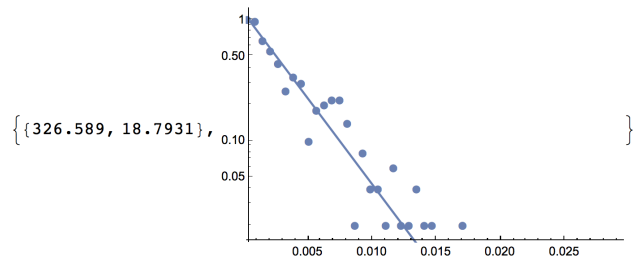
Returns A list of arrival times.

MosaicAnalysis Examples

```
In[1]= <<MosaicUtils`
In[2]= <<MosaicAnalysis`
```

In the following example, we estimate the capture rate of PEG28 from the reference data set included with the *MOSAIC* source. The first argument fo *CaptureRate* is a list of the absolute start time of each event in the database. This data can be obtained using the query shown below. The remaining arguments to *CaptureRate* define the parameters of the arrival times distribution, the lower and upper limit of the arrival times and the number of bins. The function returns the mean capture rate and standard error, as well as a plot that shows the underlying arrival times distribution.

```
In[3]= CaptureRate[
  QueryDB[
    "<mosaicroot>/data/eventMD-PEG29-Reference.sqlite",
    "select AbsEventStart from metadata where
      ProcessingStatus='normal' and ResTime > 0.01"
  ], 0.0, 0.05, 50
]
```



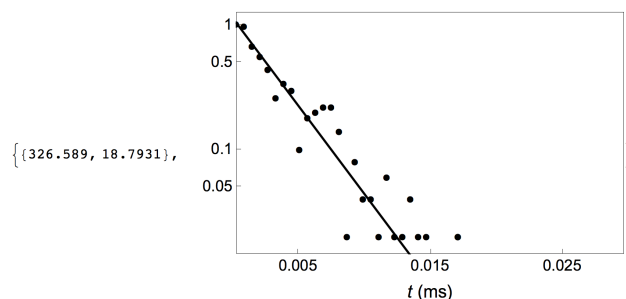
The capture rate plot above can be formatted by supplying the optional *plotopts* argument, which uses standard *Mathematica* plot options, as seen in the example below. This is particularly helpful to customize the output of the plot, for example for publication ready graphics.

```
In[4]= CaptureRate[
  QueryDB[
    "<mosaicroot>/data/eventMD-PEG29-Reference.sqlite",
    "select AbsEventStart from metadata where
      ProcessingStatus='normal' and ResTime > 0.01"
```

```

], 0.0, 0.05, 50,
    {Frame -> True, FrameLabel -> {Style["t (ms)", 16], ""},
FrameTicks -> {{{0.05, 0.1, 0.5, 1}, None}, {{0.005, 0.015, 0.025},
    None}}, FrameTicksStyle -> 14, PlotStyle -> {Black, Thick},
ImageSize -> 400}
]

```



8.2 Matlab

The SQLite database output by MOSAIC can be further processed using [MATLAB](#). The data can then be stored in an array in the [MATLAB](#) Workspace, and then manipulated as desired.

The features, of opening, querying, and storing as an array, are made available in the [MATLAB](#) script `openandquery.m`. The script does not use the [MATLAB](#) Database Manager GUI, a part of the Database Toolbox, which requires a paid license. Instead, an open-source alternative, [mksqlite](#), an interface between [MATLAB](#) and [SQLite](#) is used.

This section of the manual provides information on how to set up the `mksqlite`- package for use with [MATLAB](#), and how to use the `openandquery.m` script.

All code has been successfully tested with MATLAB 2013a, MATLAB 2014a, G++ 4.7 in Ubuntu 14.04 LTS, and Windows Visual C++ 2010. Also, [SQLite](#) must be installed prior to performing the following steps.

8.2.1 mksqlite Documentation

Information about [mksqlite](#), such as function calls and examples, is available in the [MKSQLITE: A MATLAB Interface to SQLite](#) documentation.

8.2.2 Installing mksqlite in Ubuntu 14.04 LTS

Download the latest [mksqlite](#) source files from [SourceForge](#) Unzip the files to a folder, and note the path to that folder (e.g., `/home/mksqlitefolder`) Open [MATLAB](#), and change the current path to that of the `mksqlite` folder In the Command Window, type `buildit`, and press Enter to build `mksqlite` (this will run the `buildit.m` script). If the MEX files do not build, one of the following two problems may be why: i) a compiler may not be installed – see the [MathWorks page on Supported and Compatible Compilers](#) to select and install a compiler, or ii) errors are generated during compilation of `mksqlite.cpp`. In the latter case, see the “How to build `mksqlite` MEX file `mksqlite.mexa64` in Linux?” thread in the [MathWorks MATLAB Answers forum](#). If the build proceeds without errors, you will first see the notification “compiling release version of `mksqlite`...” in the Command Window, followed by “completed.”

Note: GCC/G++ Version (in Linux)

You may have to install a version of GCC/G++ that is compatible with with your specific MATLAB release. If so, check out the linked discussion thread on MATLAB Central on how to [set up a MEX Compiler](#).

8.2.3 Installing mksqlite in Windows 7

The installation steps are essentially the same as for Ubuntu, except a different compiler (e.g., contained in Windows SDK 7) may instead have to be installed. If the SDK installer says it cannot proceed, quit the installation, uninstall previous instances of Microsoft Visual C++ 2010, and then install Windows SDK 7 again.

8.2.4 Opening, Querying, and Closing the MOSAIC Output Database

The MATLAB script `openandquery.m` contains all of the commands to: Open a MOSAIC database (e.g., `eventMD-PEG29-Reference.sqlite`) Query the database Save queried data elements into a structure Close the database Convert the structure into a multi-dimensional array, that can be easily manipulated in [MATLAB](#)

Two changes must be made to the `openandquery` m-file by the end-user: The path to the database file must be changed for each database you wish to access. An example path in Linux would be `/home/Data/eventMD-PEG29-Reference.sqlite`, and in Windows `C:\Data\eventMD-PEG29-Reference.sqlite`. The query string can be changed as needed. More information about queries is available in the [Database Structure and Query Syntax](#) section.

8.2.5 Example

The reference database file provided with MOSAIC is `eventMD-PEG29-Reference.sqlite`, located in the data folder of the source code root directory. This database contains the results of an analysis performed using the `stepResponseAnalysis` and consists of the data fields:

```
{recIDX, ProcessingStatus, OpenChCurrent, BlockedCurrent, EventStart, EventEnd, BlockDepth, ResTime,
```

In the `openandquery` script modify line 20 by typing in, within the quotes, the correct path to the database file.

```
dbname = '/home/Data/eventMD-PEG29-Reference.sqlite';
```

The query in line 23 is to read the names of all fields in the database. The names, along with column ID, and data type, are stored in the structure `fieldnames`. You may double-click on the variable `fieldnames` in the Workspace, which will open the structure for you to read the field names in which you are interested.

```
fieldnames = mksqlite('PRAGMA table_info(metadata)');
```

Next, modify line 24 to include the query. In this example we want to select (and later manipulate) the data stored in the fields `AbsEventStart` and `BlockDepth`. This is where `mksqlite` comes in: the query arguments to the `mksqlite()` function. For more information about using the `mksqlite.m` function check out the `mksqlite` documentation.

```
querytemp = mksqlite('select AbsEventStart, BlockDepth from metadata');
```

No other changes are required. Run the script. The queried data are stored in the variable `data`, seen in the MATLAB Workspace (with value 418x2 double). This variable is a 2-column matrix. The first column contains all 418 data elements of the field `AbsEventStart`, and the second column contains all elements of the field `BlockDepth`. Note that the query above can be replaced with any standard SQL query as outlined in the [working-with-sqlite-sec](#) section.

8.3 IGOR

IGOR

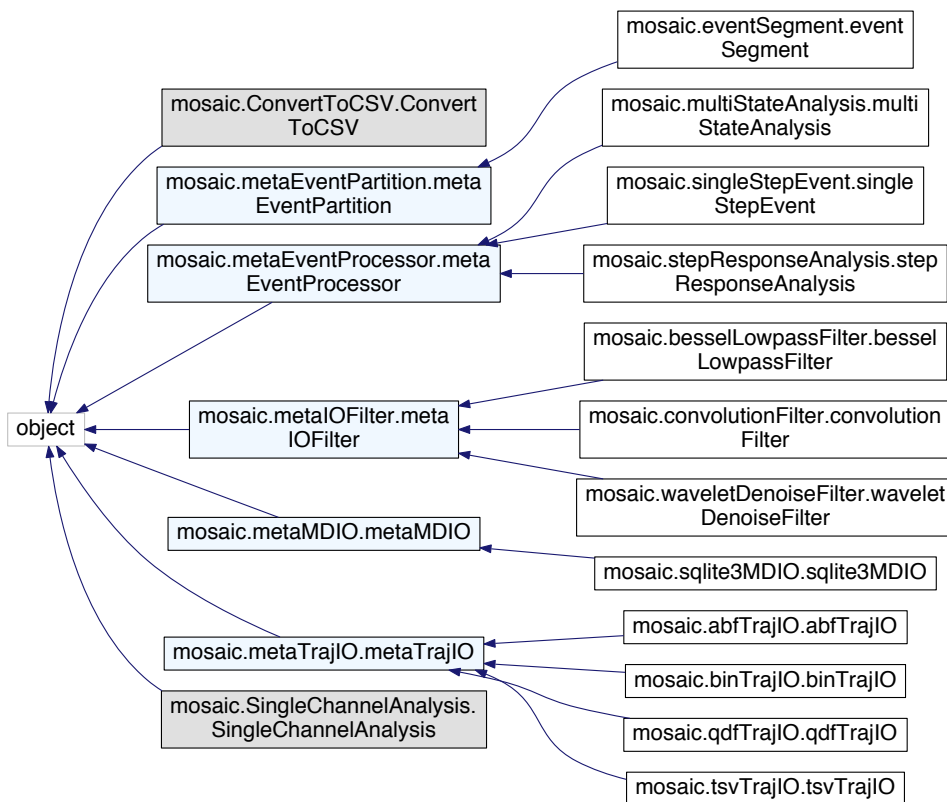
CHAPTER 9

Examples

9.1 Single Molecule Mass Spectrometry

API Documentation

MOSAIC is designed using object oriented tools, which makes it easy to extend. The API documentation provides class level descriptions of the different modules that can be used in customized code. Meta-Classes (in *blue* below) define interfaces to five key parts of *MOSAIC*: time-series IO (*metaTrajIO*), time-series filtering (*metaIOFilter*), analysis output (*metaMDIO*), event partition and segmenting (*metaEventPartition*), and event processing (*metaEventProcessor*). Sub-classing any of these meta classes and implementing their interface functions allows one to extend *MOSAIC* while maintaining compatibility with other parts of the program. The diagram below shows the class inheritance in *MOSAIC*, with top-level classes in *gray*.



10.1 MOSAIC Modules

10.1.1 Top-Level Interfaces

mosaic.SingleChannelAnalysis module

Top level module to run a single channel analysis.

Created 05/15/2014

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

5/15/14 AB Initial version

```
class mosaic.SingleChannelAnalysis.SingleChannelAnalysis (dataPath,      trajDataHnd,
                                                           dataFilterHnd,  eventParti-
                                                           tionHnd, eventProcHnd)
```

Bases: `object`

Run a single channel analysis. This is the entry point class for the analysis.

Parameters

- *dataPath* : full path to the data directory
- *trajDataHnd* : a handle to an implementation of `metaTrajIO`
- *dataFilterHnd* : a handle to an implementation of `metaIOFilter`
- *eventPartitionHnd* : a handle to a sub-class of `metaEventPartition`
- *eventProcHnd* : a handle to a sub-class of `metaEventProcessor`

Run (*forkProcess=False*)

Start an analysis.

Parameters

- *forkProcess* : start the analysis in a separate process if *True*. This option is useful when the main thread is used for other processing (e.g. GUI implementations).

Stop ()

Stop a running analysis.

mosaic.ConvertToCSV module

Top level module to convert any data file readable by `TrajIO` objects into a comma separated value text file.

Created 10/13/2014

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

`ConvertToCSV._filename()`

Return a output filename that contains the data file prefix and the block index.

`ConvertToCSV._creategenerator()`

Create a new filename generator if the file prefix has changed. The generator returns a filename incremented by a counter each time its `next()` function is called.

class `mosaic.ConvertToCSV.ConvertToCSV(trajDataObj, outdir=None)`

Bases: `object`

Convert data read from a sub-class of `metaTrajIO` to a comma separated text file

Parameters

- *trajDataObj* : a `trajIO` data object
- *outdir* : the output directory. Default is *None*, which causes the output to be saved in the same directory as the input data.

Convert (*blockSize*)

Start converting data

Parameters

- *blockSize* : number of data points to convert.

10.1.2 Meta-Classes

`mosaic.metaEventPartition` module

class `mosaic.metaEventPartition.metaEventPartition(trajDataObj, eventProcHnd, eventPartitionSettings, eventProcSettings, settingsString)`

Bases: `object`

Warning: This metaclass must be sub-classed. All abstract methods within this metaclass must be implemented.

A class to abstract partitioning individual events. Once a single molecule event is identified, it is handed off to an event processor. If parallel processing is requested, detailed event processing will commence immediately. If not, detailed event processing is performed after the event partition has completed.

Parameters

- *trajDataObj* [properly initialized object instantiated from a sub-class] of `metaTrajIO`.
- *eventProcHnd* [handle to a sub-class of `metaEventProcessor`. Objects of] this class are initialized as necessary
- *eventPartitionSettings* : settings dictionary for the partition algorithm.
- *eventProcSettings* : settings dictionary for the event processing algorithm.
- *settingsString* : settings dictionary in JSON format

Common algorithm parameters from settings file (`.settings` in the data path or current working directory)

- *writeEventTS* : Write event current data to file. (default: 1, write data to file)
- *parallelProc* : Process events in parallel using the `pproc` module. (default: 1, Yes)

•*reserveNCPU* : Reserve the specified number of CPUs and exclude them from the parallel pool

_init (*trajDataObj*, *eventProcHnd*, *eventPartitionSettings*, *eventProcSettings*)

Important: Abstract method: This method must be implemented by a sub-class.

This function is called at the end of the class constructor to perform additional initialization specific to the algorithm being implemented. The arguments to this function are identical to those passed to the class constructor.

_stop ()

Important: Abstract method: This method must be implemented by a sub-class.

Stop partitioning events from a time-series

_eventsegment ()

Important: Abstract method: This method must be implemented by a sub-class.

An implementation of this function should separate individual events of interest from a time-series of ionic current recordings. The data pertaining to each event is then passed to an instance of *metaEventProcessor* for detailed analysis. The function will collect the results of this analysis.

PartitionEvents ()

Partition events within a time-series.

Stop ()

Stop processing data.

formatoutputfiles ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a formatted string of output files.

formatsettings ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a formatted string of settings for display

formatstats ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a formatted string of statistics for display

mosaic.metaEventProcessor module

class mosaic.metaEventProcessor.**metaEventProcessor** (*icurr*, *Fs*, ****kwargs**)

Bases: `object`

Warning: This metaclass must be sub-classed. All abstract methods within this metaclass must be implemented.

Defines the interface for specific event processing algorithms. Each event processing algorithm must sub-class metaEventProcessor and implement the following abstract functions:

- **processEvent** [process raw event data and populate event meta-data. Store each] piece of processed event data in a class attribute starting with 'md'. For example, the blockade depth meta-data can be defined as 'mdBlockadeDepth'
- **printMetadata** : print meta-data set by event processing in a human readable format.

Parameters

- *icurr* : ionic current in pA
- *Fs* : sampling frequency in Hz

Keyword Args

- *eventstart* : the event start point
- *eventend* : the event end point
- *baselinestats* : baseline conductance statistics: a list of [mean, sd, slope] for the baseline current
- *algosettingsdict* : settings for event processing algorithm as a dictionary
- *absdatidx* : index of data start. This arg can allow arrival time estimation.
- *datafilehnd* : reference to an metaMDIO object for meta-data IO

__init (****kwargs**)

Important: Abstract method: This method must be implemented by a sub-class.

__metaEventProcessor__mdformat (*dat*)

Round a float to 3 decimal places. Leave ints and strings unchanged

__processEvent ()

Important: Abstract method: This method must be implemented by a sub-class.

mdAveragePropertiesList ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a list of meta-data properties that will be averaged and displayed at the end of a run. This function must be overridden by sub-classes of metaEventProcessor. As a failsafe, an empty list is returned.

mdHeadingDataType ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a list of meta-data tags data types.

mdHeadings ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a list of meta-data tags for display purposes.

mdList ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a list of meta-data set by event processing.

processEvent ()

This is the equivalent of a pure virtual function in C++.

rejectEvent (status)

Set an event as rejected if it doesn't pass tests in processing. The status is assigned to mdProcessingStatus.

writeEvent ()

Write event meta data to a metaMDIO object.

mosaic.metaIOFilter module

class mosaic.metaIOFilter.**metaIOFilter** (**kwargs)

Bases: object

Warning: This metaclass must be sub-classed. All abstract methods within this metaclass must be implemented.

Defines the interface for specific filter implementations. Each filtering algorithm must sub-class metaIOFilter and implement the following abstract function:

• *filterData* : apply a filter to self.eventData

Parameters

- *decimate* : sets the downsampling ratio of the filtered data (default:1, no decimation).

Properties

- *filteredData* : list of filtered and decimated data
- *filterFs* : sampling frequency after filtering and decimation

__init (**kwargs)

Important: Abstract method: This method must be implemented by a sub-class.

filterData (*icurr*, *Fs*)

Important: Abstract method: This method must be implemented by a sub-class.

This is the equivalent of a pure virtual function in C++.

Implementations of this method MUST store (1) a ref to the raw event data in `self.eventData` AND (2) the sampling frequency in `self.Fs`.

Parameters

- *icurr* : ionic current in pA
- *Fs* : original sampling frequency in Hz

filterFs

Return the sampling frequency of filtered data.

filteredData

Return filtered data

formatsettings ()

Important: Abstract method: This method must be implemented by a sub-class.

Return a formatted string of filter settings

mosaic.metaMDIO module

class `mosaic.metaMDIO.metaMDIO`

Bases: `object`

<p>Warning: This metaclass must be sub-classed. All abstract methods within this metaclass must be implemented.</p>
--

This class provides the skeleton for storing metadata generated by algorithms. It also provides an interface to query metadata, for example in a SQL database.

Properties

- *dbColumnNames* : a list of database column names

_opendb (*dbname*, ***kwargs*)

Important: Abstract method: This method must be implemented by a sub-class.

_initdb (***kwargs*)

Important: Abstract method: This method must be implemented by a sub-class.

_colnames (*table=None*)

Important: Abstract method: This method must be implemented by a sub-class.

closeDB ()

Important: Abstract method: This method must be implemented by a sub-class.

initDB (***kwargs*)

Initialize a new database file.

Parameters

The arguments passed to init change based on the method of file IO selected, in addition to the common args below:

- *dbPath* : directory to store the MD database ('<full path to data directory>')
- *colNames* : list of text names for the columns in the tables
- *colNames_t* : list of data types for each column.

openDB (*dbname*, ***kwargs*)

Open an existing database file.

Parameters

- *dbname* : directory to store the MD database ('<full path to data directory>')

See also:

The arguments passed to init change based on the method of file IO selected, in addition to the common args.

queryDB (*query*)

Important: Abstract method: This method must be implemented by a sub-class.

Query a database. :Parameters:

- *query* : query string

See also:

See specific implementations of metaMDIO for query syntax.

readAnalysisInfo ()

Important: Abstract method: This method must be implemented by a sub-class.

Read analysis information from the database.

readAnalysisLog ()

Important: Abstract method: This method must be implemented by a sub-class.

Read the analysis log from the database.

readSettings ()

Important: Abstract method: This method must be implemented by a sub-class.

Read JSON settings from the database.

writeAnalysisInfo (*infolist*)

Important: Abstract method: This method must be implemented by a sub-class.

Write analysis information to the database. Note that subsequent calls to this method will overwrite the analysis information entry in the table.

Args

- *infolist* [A list of strings in the following order [*datPath*, *dataType*, *partitionAlgorithm*, *processingAlgorithm*, *filteringAlgorithm*.] *datPath* : full path to the data directory
dataType : type of data processed (e.g. ABF, QDF, etc.)
partitionAlgorithm : name of partition algorithm (e.g. eventSegment)
processingAlgorithm : name of event processing algorithm (e.g. multStateAnalysis)
filteringAlgorithm : name of filtering algorithm (e.g. waveletDenoiseFilter) or None if no filtering was performed.

writeAnalysisLog (*analysislog*)

Important: Abstract method: This method must be implemented by a sub-class.

Write the analysis log string to the database. Note that subsequent calls to this method will overwrite the analysis log entry.

Args

- *analysislog* : analysis log string to save

writeRecord (*data*, *table=None*)

Important: Abstract method: This method must be implemented by a sub-class.

Write data to a specified table. By default table is None. In this case sub-classes should fall back to writing data to a default table.

writeSettings (*settingsstring*)

Important: Abstract method: This method must be implemented by a sub-class.

Write the settings JSON object to the database.

Args

- *settingsstring* : a **JSON_** formatted settings string.

mosaic.metaTrajIO module

```
class mosaic.metaTrajIO.metaTrajIO (**kwargs)
```

Bases: `object`

Warning: This metaclass must be sub-classed. All abstract methods within this metaclass must be implemented.

Initialize a TrajIO object. The object can load all the data in a directory, N files from a directory or from an explicit list of filenames. In addition to the arguments defined below, implementations of this meta class may require the definition of additional arguments. See the documentation of those classes for what those may be. For example, the qdfTrajIO implementation of metaTrajIO also requires the feedback resistance (Rfb) and feedback capacitance (Cfb) to be passed at initialization.

Keyword Args

- *dirname* : all files from a directory ('<full path to data directory>')
- *nfiles* : if requesting N files (in addition to dirname) from a specified directory
- *fnames* [explicit list of filenames ([file1, file2,...]). This argument] cannot be used in conjunction with dirname/nfiles. The filter argument is ignored when used in combination with fnames.
- *filter* : '<wildcard filter>' (optional, filter is '*' if not specified)
- *start* : Data start point in seconds.
- *end* : Data end point in seconds.
- *datafilter* [Handle to the algorithm to use to filter the data. If no algorithm is specified, datafilter] is None and no filtering is performed.
- *dcOffset* : Subtract a DC offset from the ionic current data.

Properties

- *FsHz* [sampling frequency in Hz. If the data was decimated, this] property will hold the sampling frequency after decimation.
- *LastFileProcessed* : return the data file that was last processed.
- *ElapsedTimeSeconds* : return the analysis time in sec.

Errors

- *IncompatibleArgumentsError* : when conflicting arguments are used.
- *EmptyDataPipeError* : when out of data.
- *FileNotFoundError* : when data files do not exist in the specified path.
- *InsufficientArgumentsError* : when incompatible arguments are passed

```
__init__ (**kwargs)
```

Important: Abstract method: This method must be implemented by a sub-class.

This function is called at the end of the class constructor to perform additional initialization specific to the algorithm being implemented. The arguments to this function are identical to those passed to the class constructor.

`_formatsettings()`

Important: Abstract method: This method must be implemented by a sub-class.

Return a formatted string of settings for display

ElapsedTimeSeconds

Important: Property

Return the elapsed time in the time-series in seconds.

FsHz

Important: Property

Return the sampling frequency in Hz.

LastFileProcessed

Important: Property

Return the last data file that was processed

formatsettings()

Return a formatted string of settings for display

popdata(*n*)

Pop data points from self.currDataPipe. This function uses recursion to automatically read data files when the queue length is shorter than the requested data points. When all data files are read, an EmptyDataPipeError is thrown.

Parameters

- *n* : number of requested data points

Returns numpy array with requested data

Errors

- *EmptyDataPipeError* : if the queue has fewer data points than requested.

popfnames(*n*)

Pop *n* filenames from the start of self.dataFiles. If filenames run out, simply return the available names.

Parameters

- *n* : number of requested filenames

Returns List of filenames if successful, empty list if not files remain

previewdata(*n*)

Preview data points in self.currDataPipe. This function is identical in behavior to popdata, except it does not remove data point from the queue. Like popdata, it uses recursion to automatically read data files when the queue length is shorter than the requested data points. When all data files are read, an EmptyDataPipeError is thrown.

Parameters *n* : number of requested data points

Returns numpy array with requested data

Errors

- *EmptyDataPipeError* : if the queue has fewer data points than requested.

readdata (*fname*)

Important: Abstract method: This method must be implemented by a sub-class.

Read the specified data file(s) and return the data as an array. Set a class property *Fs* with the sampling frequency in Hz.

Parameters

- *fname* : list of filenames

10.1.3 Time-Series IO

mosaic.abfTrajIO module



A TrajIO class that supports ABF1 and ABF2 file formats via *abf/abf.py*. Currently, only gap-free mode and single channel recordings are supported.

Created 5/23/2013

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

5/23/13 AB Initial version

```
class mosaic.abfTrajIO.abfTrajIO(**kwargs)
```

```
    Bases: mosaic.metaTrajIO.metaTrajIO
```

Parameters

In addition to *metaTrajIO.__init__* args, None

readdata (*fname*)

Read one or more files and append their data to the data pipeline. Set a class attribute *Fs* with the sampling frequency in Hz.

Parameters

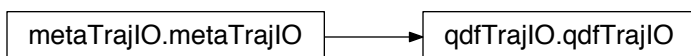
- *fname* : list of data files to read

Returns None

Errors

- *SamplingRateChangedError* : if the sampling rate for any data file differs from previous

mosaic.qdfTrajIO module



QDF implementation of meta-

TrajIO. Uses the readqdf module from EBS to read individual qdf files.

Created 7/18/2012

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

7/18/12 AB Initial version

2/11/14 AB Support qdf files that save the current in pA. This needs
format='pA' argument.

```
class mosaic.qdfTrajIO.qdfTrajIO(**kwargs)
    Bases: mosaic.metaTrajIO.metaTrajIO
```

In addition to the base class init, check if the feedback resistance (Rfb) and feedback capacitance (Cfb) are defined to convert qdf binary data into pA

Parameters

In addition to metaTrajIO.__init__ args,

- *Rfb* : feedback resistance of amplifier
- *Cfb* : feedback capacitance of amplifier
- *format* : 'V' for voltage or 'pA' for current. Default is 'V'

Returns None

Errors

- *InsufficientArgumentsError* : if the mandatory arguments Rfb and Cfb are not set

readdata (*fname*)

Read one or more files and append their data to the data pipeline. Set a class attribute Fs with the sampling frequency in Hz.

Parameters

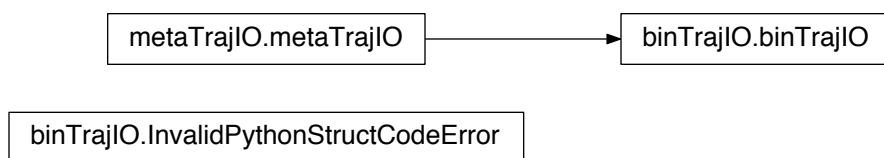
- *fname* : list of data files to read

Returns None

Errors

- *SamplingRateChangedError* : if the sampling rate for any data file differs from previous

mosaic.binTrajIO module



Binary file

implementation of `metaTrajIO`. Read raw binary files with specified record sizes

Created 4/22/2013

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

7/27/14 AB Update interface to specify python `PythonStructCode` instead of `RecordSize`. This will allow any binary file to be decoded
The `AmplifierScale` and `AmplifierOffset` are set to 1 and 0 respectively if `PythonStructCode` is an integer or short.
4/22/13 AB Initial version

class `mosaic.binTrajIO.binTrajIO` (**kwargs)
Bases: `mosaic.metaTrajIO.metaTrajIO`

Read a binary file that contains single channel ionic current data and calculate the current in pA after scaling by the amplifier scale factor and removing any offsets.

Parameters

In addition to `metaTrajIO.__init__` args,

- *AmplifierScale* : full scale of amplifier (in pA) that varies with the gain
- *AmplifierOffset* : current offset in the recorded data
- *SamplingFrequency* : sampling rate of data in the file in Hz
- *HeaderOffset* : ignore first 'n' bytes of the file for header (default: 0 bytes).
- *PythonStructCode* : Single character code for a python struct (see [Python struct docs](#)).

Returns: None

Errors:

- *InsufficientArgumentsError* : if the mandatory arguments Rfb and Cfb are not set

readdata (*fname*)

Read one or more files and append their data to the data pipeline. Set a class attribute Fs with the sampling frequency in Hz.

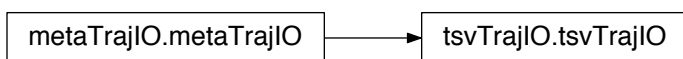
Parameters

- *fname* : list of data files to read

Returns None

Errors None

mosaic.tsvTrajIO module



jIO that reads tab separated valued (TSV) files

An implementation of metaTra-

Created 7/31/2012

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

7/31/12 AB Initial version

6/30/13 AB Added the 'separator' kwarg to the class initializer to allow any delimited files to be read. e.g. ' ' (default), ',', etc.

class mosaic.tsvTrajIO.**tsvTrajIO** (**kwargs)

Bases: mosaic.metaTrajIO.metaTrajIO

Perform additional initialization checks. Check if kwarg 'timeCol' is set to a number.

In addition to metaTrajIO.__init__ args, :Optional Parameters:

- *headers* : If True, the first row is ignored (default: True)
- *separator* : set the data separator (default: ' ')

Either:

- *Fs* [Sampling frequency in Hz. If set, all other options are ignored] and the first column in the file is assumed to be the current in pA.

Or:

- ***nCols*** [number of columns in TSV file (default:2, first column is time) in ms and second is current in pA)
- ***timeCol*** : explicitly set the time column (default: 0, first col)
- ***currCol*** : explicitly set the position of the current column (default: 1)

If neither 'Fs' nor { 'nCols', 'timeCol','currCol' } are set then the latter is assumed with the listed default values.

readdata (*fname*)

Read a single TSV file and append its data to the data pipeline. Set/update a class attribute Fs with the sampling frequency in Hz.

Parameters

- *fname* : list of data files to read

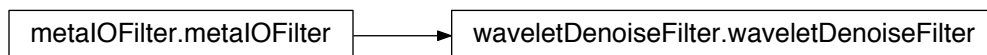
Returns None

Errors

- *SamplingRateChangedError* : if the sampling rate for any data file differs from previous

10.1.4 Time-Series Filters

mosaic.waveletDenoiseFilter module



Implementation of a wavelet based denoising filter

Created 8/31/2014

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

Author Arvind Balijepalli

ChangeLog

8/31/14 AB Initial version

```
class mosaic.waveletDenoiseFilter.waveletDenoiseFilter (**kwargs)
    Bases: mosaic.metalOFilter.metalOFilter
```

Keyword Args

In addition to metalOFilter args,

- *wavelet* : the type of wavelet
- *level* : wavelet level
- *threshold* : threshold type

filterData (*icurr*, *Fs*)

Denoise an ionic current time-series and store it in self.eventData

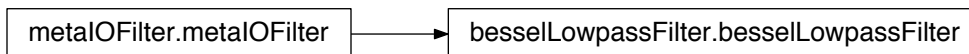
Parameters

- *icurr* : ionic current in pA
- *Fs* : original sampling frequency in Hz

formatsettings ()

Return a formatted string of filter settings

mosaic.besselLowpassFilter module



Im-

plementation of an 'N' order Bessel filter

Created 7/1/2013

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

7/1/13 AB Initial version

class mosaic.besselLowpassFilter.**besselLowpassFilter** (**kwargs)

Bases: mosaic.metaIOFilter.metaIOFilter

Keyword Args

In addition to metaIOFilter.__init__ args,

- *filterOrder* : the filter order
- *filterCutoff* : filter cutoff frequency in Hz

filterData (*icurr*, *Fs*)

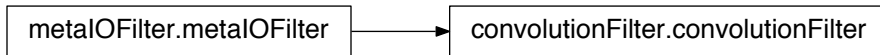
Denoise an ionic current time-series and store it in self.eventData

Parameters

- *icurr* : ionic current in pA
- *Fs* : original sampling frequency in Hz

formatsettings ()

Return a formatted string of filter settings

mosaic.convolutionFilter module

tion of a weighted moving average (tap delay line) filter

Implementa-

Created 8/16/2013**Author** Arvind Balijepalli <arvind.balijepalli@nist.gov>**License** See LICENSE.TXT**ChangeLog**

8/16/13 AB Initial version

class mosaic.convolutionFilter.**convolutionFilter** (**kwargs)

Bases: mosaic.metaIOFilter.metaIOFilter

Keyword Args**In addition to metaIOFilter.__init__ args,**

- *filterCoeff* : filter coefficients (default is a 10 point uniform moving average)

filterData (*icurr*, *Fs*)

Denoise an ionic current time-series and store it in self.eventData

Parameters

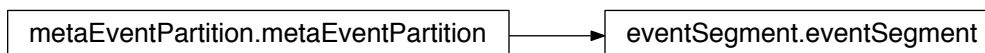
- *icurr* : ionic current in pA
- *Fs* : original sampling frequency in Hz

formatsettings ()

Return a formatted string of filter settings

10.1.5 Event Partition and Segment

mosaic.eventSegment module



Partition a trajectory into individual events and pass each event to an implementation of eventProcessor

Created 7/17/2012

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

5/17/14 AB Delete plotting support

5/17/14 AB Add metaMDIO support for meta-data and time-series storage

2/14/14 AB Pass absdatidx argument to event processing to track absolute time of event start for capture rate estimation.

6/22/13 AB Use plotting hooks in metaEventPartition to plot blockade depth histogram in real-time using matplotlib.

4/22/13 AB Rewrote this class as an implementation of the base class metaEventPartition. Included event processing parallelization using ZMQ.

9/26/12 AB Allowed automatic open channel state calculation to be overridden. To do this the settings “meanOpenCurr”, “sdOpenCurr” and “slopeOpenCurr” must be set manually. If all three settings are absent or set to 01, they are automatically estimated.

Added “writeEventTS” boolean setting to control whether raw events are written to file. Default is ON (1)

8/24/12 AB Settings are now read from a settings file that is located either with the data or in the working directory that the program is run from. Each class that relies on the settings file will fallback to default values if the file is not found.

7/17/12 AB Initial version

```
class mosaic.eventSegment.eventSegment (trajDataObj, eventProcHnd, eventPartitionSettings,
                                         eventProcSettings, settingsString)
```

Bases: `mosaic.metaEventPartition.metaEventPartition`

Implement an event partitioning algorithm by sub-classing the metaEventPartition class

Settings In addition to the parameters described in metaEventPartition, the following parameters from are read from the settings file (.settings in the data path or current working directory):

- **blockSizeSec** [Functions that perform block processing use this value to set the size of] their windows in seconds. For example, open channel conductance is processed for windows with a size specified by this parameter. (default: 1 second)
- **eventPad** : Number of points to include before and after a detected event. (default: 500)
- **minEventLength** : Minimum number points in the blocked state to qualify as an event (default: 5)
- **eventThreshold** [Threshold, number of SD away from the open channel mean. If the $\text{abs}(\text{curr})$ is less] than $\text{abs}(\text{mean}) - (\text{eventThreshold} * \text{SD})$ a new event is registered (default: 6)
- **driftThreshold** [Trigger a drift warning when the mean open channel current deviates by 'driftThreshold'*] SD from the baseline open channel current (default: 2)
- **maxDriftRate** [Trigger a warning when the open channel conductance changes at a rate faster] than that specified. (default: 2 pA/s)
- **meanOpenCurr** [Explicitly set mean open channel current. (pA) (default: -1, to] calculate automatically)
- **sdOpenCurr** [Explicitly set open channel current SD. (pA) (default: -1, to] calculate automatically)
- **slopeOpenCurr** [Explicitly set open channel current slope. (default: -1, to] calculate automatically)

formatsettings ()

Return a formatted string of settings for display in the output log.

formatstats ()

Return a formatted string of statistics for display in the output log.

10.1.6 Event Processing

mosaic.stepResponseAnalysis module



A class that extends metaEventProcessing to implement the step response algorithm from [Balijepalli:2014]

Created 4/18/2013

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

11/7/14 AB Error codes describing event rejection are now more specific.

11/5/14 AB Fixed a bug in the event fitting logic that prevented

long events from being correctly analyzed.

5/17/14 AB Modified md interface functions for metaMDIO support

2/16/14 AB Added new metadata field, 'AbsEventStart' to track

global time of event start to allow capture rate estimation.

6/20/13 AB Added an additional check to reject events

with blockade depths > BlockRejectRatio (default: 0.8)

4/18/13 AB Initial version

class `mosaic.stepResponseAnalysis.datblock` (*dat*)

Smart data block that holds a time-series of data and keeps track of its mean and SD.

class `mosaic.stepResponseAnalysis.stepResponseAnalysis` (*icurr, Fs, **kwargs*)

Bases: `mosaic.metaEventProcessor.metaEventProcessor`

Analyze an event that is characteristic of PEG blockades. This method includes system information in the analysis, specifically the filtering effects (through the RC constant) of either amplifiers or the membrane/nanopore complex. The analysis generates several parameters that are stored as metadata including:

1. Blockade depth: the ratio of the open channel current to the blocked current
2. Residence time: the time the molecule spends inside the pore
3. **Rise time: the 1/RC of the response to a step input (e.g. the entry or exit of the molecule into or out of the nanopore).**

When an event cannot be analyzed, the blockade depth, residence time and rise time are set to -1.

formatsettings ()

Return a formatted string of settings for display

mdAveragePropertiesList ()

Return a list of meta-data properties that will be averaged and displayed at the end of a run.

mdHeadingDataType ()

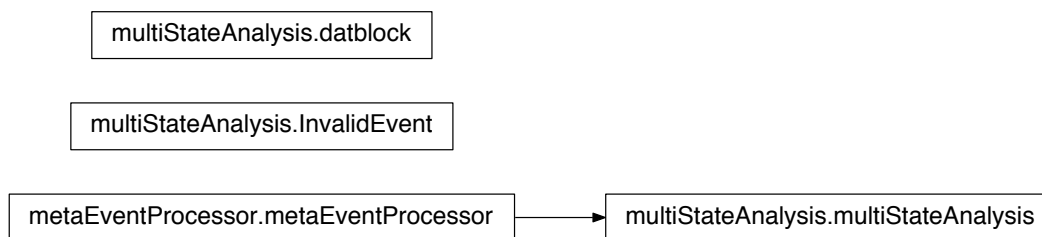
Return a list of meta-data tags data types.

mdHeadings ()

Explicitly set the metadata to print out.

mdList ()

Return a list of meta-data from the analysis of single step events. We explicitly control the order of the data to keep formatting consistent.

mosaic.multiStateAnalysis module

Analyze a multi-step event

Created 4/18/2013

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

11/7/14 AB Error codes describing event rejection are now more specific.

11/6/14 AB Fixed a bug in the event fitting logic that prevents the analysis of long states.

8/21/14 AB Added AbsEventStart and BlockDepth (constructed from mdCurrentStep and mdOpenChCurrent) metadata.

5/17/14 AB Modified md interface functions for metaMDIO support

9/26/13 AB Initial version

class `mosaic.multiStateAnalysis.datblock` (*dat*)

Smart data block that holds a time-series of data and keeps track of its mean and SD.

class `mosaic.multiStateAnalysis.multiStateAnalysis` (*icurr*, *Fs*, ***kwargs*)

Bases: `mosaic.metaEventProcessor.metaEventProcessor`

Analyze a multi-step event that contains two or more states. This method includes system information in the analysis, specifically the filtering effects (through the RC constant) of either amplifiers or the membrane/nanopore complex. The analysis generates several parameters that are stored as metadata including:

1. Blockade depth: the ratio of the open channel current to the blocked current
2. Residence time: the time the molecule spends inside the pore
3. **Tau: the 1/RC of the response to a step input (e.g. the entry or exit of the molecule into or out of the nanopore).**

When an event cannot be analyzed, the blockade depth, residence time and rise time are set to -1.

formatsettings ()

Return a formatted string of settings for display

mdAveragePropertiesList ()

Return a list of meta-data properties that will be averaged and displayed at the end of a run.

mdHeadingDataType ()

Return a list of meta-data tags data types.

mdHeadings ()

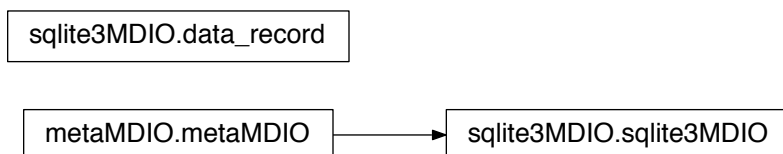
Explicitly set the metadata to print out.

mdList ()

Return a list of meta-data from the analysis of single step events. We explicitly control the order of the data to keep formatting consistent.

10.1.7 Data Output

mosaic.sqlite3MDIO module



metaMDIO to implement SQLite support for metadata storage.

A class that extends

Created 9/28/2014

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

11/9/14 AB Implemented the analysis log I/O interface for sqlite3 databases.

9/28/14 AB Initial version

```
class mosaic.sqlite3MDIO.data_record(data_label, data, data_t)
    Bases: dict
```

Smart data record structure that automatically encodes/decodes data for storage in a sqlite3 DB.

```
class mosaic.sqlite3MDIO.sqlite3MDIO
    Bases: mosaic.metaMDIO.metaMDIO
```

10.1.8 Miscellaneous

mosaic.settings module

Load analysis settings from a JSON file.

Created 8/24/2012

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

8/20/14 AB Changed precedence of settings file search to datapath/.settings,
datapath/settings, coderooot/.settings and coderooot/settings

8/6/14 AB Add a function to parse a settings string.

9/5/13 AB Check for either .settings or settings in data directory
and code root. Warn when using default settings

8/24/12 AB Initial version

class mosaic.settings.settings (datapath, defaultwarn=True)
Initialize a settings object.

Args

- *datapath* : Specify the location of the settings file. If a settings file is not found, return default settings.
- *defaultwarn* : If *True* warn the user if a settings file was not found in the path specified by *datapath*.

getSettings (section)

Return settings for a specified section as a Python dict.

Args

- *section* : specifies the section for which settings are requested. Returns an empty dictionary if the settings file doesn't exist the section is not found.

mosaic.utilities.ionic_current_stats module

Created 10/30/2014

Author Arvind Balijepalli <arvind.balijepalli@nist.gov>

License See LICENSE.TXT

ChangeLog

10/30/14 AB Initial version

mosaic.utilities.ionic_current_stats.OpenCurrentDist (dat, limit)
Calculate the mean and standard deviation of a time-series.

Args

- *dat* : time-series data
- *limit* : limit the calculation to the top 50% (+0.5) of the range, bottom 50% (-0.5) or the entire range (0). Any other value of *limit* will cause it to be reset to 0 (i.e. full range).

mosaic.utilities.util module

A collection of utility functions

`mosaic.utilities.util.avg(dat)`

Calculate the average of a list of reals

`mosaic.utilities.util.commonest(dat)`

Return the most common element in a list.

`mosaic.utilities.util.decimate(dat, size)`

Decimate dat for a specified window size.

`mosaic.utilities.util.filter(dat, windowSz)`

Filter the data using a convolution. Returns an array of size $\text{len}(\text{dat}) - \text{windowSz} + 1$ if dat is longer than windowSz.

If $\text{len}(\text{dat}) < \text{windowSz}$, raise `WindowSizeError`

`mosaic.utilities.util.flat2(dat)`

Flatten a 2D array to a list

`mosaic.utilities.util.partition(dat, size)`

Partition a list into sub-lists, each of length size. If the number of elements in dat does not partition evenly, the last sub-list will have fewer elements.

`mosaic.utilities.util.sd(dat)`

Wrapper for numpy std

`mosaic.utilities.util.selectS(dat, nSigma, mu, sd)`

Select and return data from a list that lie within $n\text{Sigma} * \text{SD}$ of the mean.

Change Log

v1.0b3.2

- [GUI] Misc bug fixes
- [Addons] Added code to import MOSAIC output into Matlab (pull requests #18 and #20)
- [Addons] Updated [Mathematica](#) addons to automatically decode multi-state data.
- Resolves issues #16 and #22

v1.0b3.1

- [GUI] Added multiState support to mosaicgui.
- Analysis information such as algorithms used, data type, etc. are now stored within a MDIO database.
- [GUI] Autocomplete in mosaicgui only suggests database columns that are valid when used in a query.
- Reorganized [Mathematica](#) addon code.

v1.0b3

- Fixed a bug that prevented events longer than ~700 data points from being correctly analyzed.
- Fixed a problem that prevented event data from being correctly padded before analysis.
- Resolves #2. TrajIO settings are now read in from the settings file.
- [GUI] Resolves #3. Threshold entry box in GUI becomes nonresponsive when meanOpenCurr is negative.
- [GUI] Resolves #4. Analysis fails when using wavletDenoiseFilter from GUI.
- [GUI] Histogram in BlockDepthViewer window can be saved to a CSV file from the File Menu.
- Analysis log is saved to the MDIO database.
- [GUI] ConsoleLogViewer displays the analysis log saved in the MDIO database.
- [GUI] Added a new dialog that displays an experimental feature warning wavelet-based denoising is selected.
- Updated error codes reported in database to be more descriptive of the failure.
- Improved and expanded unit testing framework.
- Moved installation and testing to setup tools.

v1.0b2

- [GUI] Fixed threshold update error from 1.0b1.

- Considerably improved automatic open channel state detection.
- The default settings string is now included within the source code.
- Implemented new top-level class ConvertToCSV that allows conversion of data read by any TrajIO object to comma separated files.
- Updated build system and unit testing framework.
- [GUI] Misc UI updates.

v1.0b1

- [GUI] Added a menu option to save a settings file prior to starting the analysis.
- [GUI] Current threshold is now defined by an ionic current. The trajectory viewer displays the deviation of the threshold from the mean current.
- Analysis settings are saved within the analysissettings table of the sqlite database. When an analysis database is loaded into the GUI, settings are parsed from within the database.
- When an analysis file is loaded, widgets in the main window remain enabled. This allows starting a new analysis run with the current settings.
- [GUI] Implemented an analysis log viewer that displays the event processing log.
- [GUI] Initial commit of wavelets based peak detection in blockdepthview.
- [GUI] Added all points histogram to trajectory viewer.
- *Known Issues:* Selecting automatic baseline detection can sometimes cause the threshold in the trajectory viewer to change. Moving the slider will cause the settings and trajectory windows to synchronize.

Bibliography

- [BEC+14] Arvind Balijepalli, Jessica Ettegui, Andrew T Cornio, Joseph W F Robertson, Kin P Cheung, John J Kasianowicz, and Canute Vaz. Quantifying short-lived events in multistate ionic current measurements.. *ACS Nano*, 8(2):1547–1553, February 2014.
- [KBBD96] JJ Kasianowicz, E Brandin, D Branton, and DW Deamer. Characterization of individual polynucleotide molecules using a membrane channel. *P Natl Acad Sci USA*, 93(24):13770–13773, 1996.
- [PFR09] Daniel Pedone, Matthias Firnkes, and Ulrich Rant. Data Analysis of Translocation Events in Nanopore Experiments. *Anal. Chem.*, 81(23):9689–9694, December 2009.
- [RBR+12] Joseph E Reiner, Arvind Balijepalli, Joseph W F Robertson, Jason Campbell, John Suehle, and John J Kasianowicz. Disease Detection and Management via Single Nanopore-Based Sensors.. *Chem. Rev.*, 112:6432–6451, November 2012.
- [RKR10] Joseph W F Robertson, John J Kasianowicz, and Joseph E Reiner. Changes in ion channel geometry resolved to sub-ångström precision \emph{via} single molecule mass spectrometry.. *J Phys-Condens Mat*, 22(45):454108, November 2010.
- [RRS+07] Joseph W F Robertson, Claudio G Rodrigues, Vincent M Stanford, Kenneth A. Robinson, Oleg V Krasilnikov, and John J Kasianowicz. Single-molecule mass spectrometry in solution using a solitary nanopore. *P Natl Acad Sci USA*, 104(20):8207–8211, 2007.
- [VDKD+09] S. Van Dorp, U.F. Keyser, N.H. Dekker, C Dekker, and S.G. Lemay. Origin of the electrophoretic force on DNA in solid-state nanopores. *Nat Phys*, 5(5):347–351, 2009.
- [Vit67] A J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inf. Theory*, 13(2):260–269, 1967.

Python Module Index

m

`mosaic.abfTrajIO`, [56](#)
`mosaic.besselLowpassFilter`, [61](#)
`mosaic.binTrajIO`, [58](#)
`mosaic.ConvertToCSV`, [46](#)
`mosaic.convolutionFilter`, [62](#)
`mosaic.eventSegment`, [63](#)
`mosaic.multiStateAnalysis`, [66](#)
`mosaic.qdfTrajIO`, [57](#)
`mosaic.settings`, [67](#)
`mosaic.SingleChannelAnalysis`, [46](#)
`mosaic.sqlite3MDIO`, [67](#)
`mosaic.stepResponseAnalysis`, [64](#)
`mosaic.tsvTrajIO`, [59](#)
`mosaic.utilities.ionic_current_stats`,
 [68](#)
`mosaic.utilities.util`, [69](#)
`mosaic.waveletDenoiseFilter`, [60](#)

Symbols

[_colnames\(\)](#) (mosaic.metaMDIO.metaMDIO method), [51](#)
[_creategenerator\(\)](#) (mosaic.ConvertToCSV.ConvertToCSV method), [47](#)
[_eventsegment\(\)](#) (mosaic.metaEventPartition.metaEventPartition method), [48](#)
[_filename\(\)](#) (mosaic.ConvertToCSV.ConvertToCSV method), [47](#)
[_formatsettings\(\)](#) (mosaic.metaTrajIO.metaTrajIO method), [54](#)
[_init\(\)](#) (mosaic.metaEventPartition.metaEventPartition method), [48](#)
[_init\(\)](#) (mosaic.metaEventProcessor.metaEventProcessor method), [49](#)
[_init\(\)](#) (mosaic.metaIOFilter.metaIOFilter method), [50](#)
[_init\(\)](#) (mosaic.metaTrajIO.metaTrajIO method), [54](#)
[_initdb\(\)](#) (mosaic.metaMDIO.metaMDIO method), [51](#)
[_metaEventProcessor__mdformat\(\)](#) (mosaic.metaEventProcessor.metaEventProcessor method), [49](#)
[_opendb\(\)](#) (mosaic.metaMDIO.metaMDIO method), [51](#)
[_processEvent\(\)](#) (mosaic.metaEventProcessor.metaEventProcessor method), [49](#)
[_stop\(\)](#) (mosaic.metaEventPartition.metaEventPartition method), [48](#)

A

[abfTrajIO](#) (class in mosaic.abfTrajIO), [56](#)
[avg\(\)](#) (in module mosaic.utilities.util), [69](#)

B

[besselLowpassFilter](#) (class in mosaic.besselLowpassFilter), [61](#)
[binTrajIO](#) (class in mosaic.binTrajIO), [58](#)

C

[closeDB\(\)](#) (mosaic.metaMDIO.metaMDIO method), [52](#)
[commonest\(\)](#) (in module mosaic.utilities.util), [69](#)
[Convert\(\)](#) (mosaic.ConvertToCSV.ConvertToCSV method), [47](#)
[ConvertToCSV](#) (class in mosaic.ConvertToCSV), [47](#)
[convolutionFilter](#) (class in mosaic.convolutionFilter), [62](#)

D

[data_record](#) (class in mosaic.sqlite3MDIO), [67](#)
[datblock](#) (class in mosaic.multiStateAnalysis), [66](#)
[datblock](#) (class in mosaic.stepResponseAnalysis), [65](#)
[decimate\(\)](#) (in module mosaic.utilities.util), [69](#)

E

[ElapsedTimeSeconds](#) (mosaic.metaTrajIO.metaTrajIO attribute), [55](#)
[eventSegment](#) (class in mosaic.eventSegment), [63](#)

F

[filter\(\)](#) (in module mosaic.utilities.util), [69](#)
[filterData\(\)](#) (mosaic.besselLowpassFilter.besselLowpassFilter method), [61](#)
[filterData\(\)](#) (mosaic.convolutionFilter.convolutionFilter method), [62](#)
[filterData\(\)](#) (mosaic.metaIOFilter.metaIOFilter method), [50](#)
[filterData\(\)](#) (mosaic.waveletDenoiseFilter.waveletDenoiseFilter method), [61](#)
[filteredData](#) (mosaic.metaIOFilter.metaIOFilter attribute), [51](#)
[filterFs](#) (mosaic.metaIOFilter.metaIOFilter attribute), [51](#)
[flat2\(\)](#) (in module mosaic.utilities.util), [69](#)
[formatoutputfiles\(\)](#) (mosaic.metaEventPartition.metaEventPartition method), [48](#)
[formatsettings\(\)](#) (mosaic.besselLowpassFilter.besselLowpassFilter method), [61](#)

formatsettings() (mosaic.convolutionFilter.convolutionFilter method), 62
formatsettings() (mosaic.eventSegment.eventSegment method), 64
formatsettings() (mosaic.metaEventPartition.metaEventPartition method), 48
formatsettings() (mosaic.metaIOFilter.metaIOFilter method), 51
formatsettings() (mosaic.metaTrajIO.metaTrajIO method), 55
formatsettings() (mosaic.multiStateAnalysis.multiStateAnalysis method), 66
formatsettings() (mosaic.stepResponseAnalysis.stepResponseAnalysis method), 65
formatsettings() (mosaic.waveletDenoiseFilter.waveletDenoiseFilter method), 61
formatstats() (mosaic.eventSegment.eventSegment method), 64
formatstats() (mosaic.metaEventPartition.metaEventPartition method), 48
FsHz (mosaic.metaTrajIO.metaTrajIO attribute), 55

G

getSettings() (mosaic.settings.settings method), 68

I

initDB() (mosaic.metaMDIO.metaMDIO method), 52

L

LastFileProcessed (mosaic.metaTrajIO.metaTrajIO attribute), 55

M

mdAveragePropertiesList() (mosaic.metaEventProcessor.metaEventProcessor method), 49
mdAveragePropertiesList() (mosaic.multiStateAnalysis.multiStateAnalysis method), 66
mdAveragePropertiesList() (mosaic.stepResponseAnalysis.stepResponseAnalysis method), 65
mdHeadingDataType() (mosaic.metaEventProcessor.metaEventProcessor method), 49
mdHeadingDataType() (mosaic.multiStateAnalysis.multiStateAnalysis method), 67
mdHeadingDataType() (mosaic.stepResponseAnalysis.stepResponseAnalysis method), 65
mdHeadings() (mosaic.metaEventProcessor.metaEventProcessor method), 50

mdHeadings() (mosaic.multiStateAnalysis.multiStateAnalysis method), 67
mdHeadings() (mosaic.stepResponseAnalysis.stepResponseAnalysis method), 65
mdList() (mosaic.metaEventProcessor.metaEventProcessor method), 50
mdList() (mosaic.multiStateAnalysis.multiStateAnalysis method), 67
mdList() (mosaic.stepResponseAnalysis.stepResponseAnalysis method), 65
metaEventPartition (class in mosaic.metaEventPartition), 47
metaEventProcessor (class in mosaic.metaEventProcessor), 49
metaIOFilter (class in mosaic.metaIOFilter), 50
metaMDIO (class in mosaic.metaMDIO), 51
metaTrajIO (class in mosaic.metaTrajIO), 54
mosaic.abfTrajIO (module), 56
mosaic.besselLowpassFilter (module), 61
mosaic.binTrajIO (module), 58
mosaic.ConvertToCSV (module), 46
mosaic.convolutionFilter (module), 62
mosaic.eventSegment (module), 63
mosaic.multiStateAnalysis (module), 66
mosaic.qdfTrajIO (module), 57
mosaic.settings (module), 67
mosaic.SingleChannelAnalysis (module), 46
mosaic.sqlite3MDIO (module), 67
mosaic.stepResponseAnalysis (module), 64
mosaic.tsvTrajIO (module), 59
mosaic.utilities.ionic_current_stats (module), 68
mosaic.utilities.util (module), 69
mosaic.waveletDenoiseFilter (module), 60
multiStateAnalysis (class in mosaic.multiStateAnalysis), 66

O

OpenCurrentDist() (in module mosaic.utilities.ionic_current_stats), 68
openDB() (mosaic.metaMDIO.metaMDIO method), 52

P

partition() (in module mosaic.utilities.util), 69
PartitionEvents() (mosaic.metaEventPartition.metaEventPartition method), 48
popdata() (mosaic.metaTrajIO.metaTrajIO method), 55
popfnames() (mosaic.metaTrajIO.metaTrajIO method), 55
previewdata() (mosaic.metaTrajIO.metaTrajIO method), 55
processEvent() (mosaic.metaEventProcessor.metaEventProcessor method), 50

Q

qdfTrajIO (class in mosaic.qdfTrajIO), [57](#)
 queryDB() (mosaic.metaMDIO.metaMDIO method), [52](#)

R

readAnalysisInfo() (mosaic.metaMDIO.metaMDIO method), [52](#)
 readAnalysisLog() (mosaic.metaMDIO.metaMDIO method), [52](#)
 readdata() (mosaic.abfTrajIO.abfTrajIO method), [56](#)
 readdata() (mosaic.binTrajIO.binTrajIO method), [59](#)
 readdata() (mosaic.metaTrajIO.metaTrajIO method), [56](#)
 readdata() (mosaic.qdfTrajIO.qdfTrajIO method), [57](#)
 readdata() (mosaic.tsvTrajIO.tsvTrajIO method), [60](#)
 readSettings() (mosaic.metaMDIO.metaMDIO method), [52](#)
 rejectEvent() (mosaic.metaEventProcessor.metaEventProcessor method), [50](#)
 Run() (mosaic.SingleChannelAnalysis.SingleChannelAnalysis method), [46](#)

S

sd() (in module mosaic.utilities.util), [69](#)
 selectS() (in module mosaic.utilities.util), [69](#)
 settings (class in mosaic.settings), [68](#)
 SingleChannelAnalysis (class in mosaic.SingleChannelAnalysis), [46](#)
 sqlite3MDIO (class in mosaic.sqlite3MDIO), [67](#)
 stepResponseAnalysis (class in mosaic.stepResponseAnalysis), [65](#)
 Stop() (mosaic.metaEventPartition.metaEventPartition method), [48](#)
 Stop() (mosaic.SingleChannelAnalysis.SingleChannelAnalysis method), [46](#)

T

tsvTrajIO (class in mosaic.tsvTrajIO), [59](#)

W

waveletDenoiseFilter (class in mosaic.waveletDenoiseFilter), [60](#)
 writeAnalysisInfo() (mosaic.metaMDIO.metaMDIO method), [53](#)
 writeAnalysisLog() (mosaic.metaMDIO.metaMDIO method), [53](#)
 writeEvent() (mosaic.metaEventProcessor.metaEventProcessor method), [50](#)
 writeRecord() (mosaic.metaMDIO.metaMDIO method), [53](#)
 writeSettings() (mosaic.metaMDIO.metaMDIO method), [53](#)