SpikeSorting.jl Documentation

Release 0.1

Paul Thompson

Dec 03, 2018

Contents

Contents:

CHAPTER 1

Getting Started

SpikeSorting.jl is a Julia implementation of many spike sorting algorithms for neural electrophysiology data. The goal of this project is to design a framework which allows for easy use in 1) real time applications, 2) large scale cluster computing and 3) benchmarking multiple methods against one another. The Julia Language keeps the syntax readible while also allowing for near C performance.

1.1 Overview

The process of "spike sorting" takes an analog, extracellular voltage trace, and determines what components of the signal correpond to electrical activity from nearby neurons. A general workflow for spike sorting would be to 1) detect canidate spikes 2) align candidate spikes 3) extract meaningful features from this signal 4) reduce the dimensionality from a high dimensional feature space to those dimensions which are most meaningful for discrimination and 5) clustering spikes with similar features.

SpikeSorting.jl employs this modular framework, such that one method in a step is compatible with most other existing methods in previous or subsequent steps.

1.1.1 Data Structures

Analog Voltage Input

The methods of SpikeSorting.jl expect to operator on a 2D (M time x N channels) matrix of voltage values.

Sorting

The primary data structure is the Sorting type, which contains the variables necessary for individual methods in the spike sorting workflow outlined above, as well as variables common to all of the methods. An instance of sorting is initialized by providing the desired method for each step in the workflow. For instance:

```
detection=DetectPower() #Power-based spike detection
alignment=AlignMax()     #Align candidate spikes by their maximum voltage
feature=FeatureTime()    #Chose time varying voltage signal as feature
reduction=Reduction()    #Use all time points for clustering steps
cluster=ClusterOSort()   #OSort style clustering (compare clusters with candidate␣
↪spikes by euclidean distance)

mysort=Sorting(detection,cluster,alignment,feature,reduction)
```

In the example above, the detection container of type DetectPower will store all of the necessary variables for power-style detection to take place. Because of the modularity, these data containers can be relatively simple, as is the case with power detection:

```
type DetectPower <: Detect
        a::Int64
        b::Int64
        c::Int64
end
```

Output Buffers

Spikes are characterized by their location in the input voltage array, as well as the cluster they are assigned to. This idea is captured by the Spike type:

```
immutable Spike
        inds::UnitRange{Int64}
        id::Int64
end
```

For online sorting, an output buffer is used to temporarily store newly detected spikes before they are written to disk. Two buffers are used: one to keep record of the spikes detected, and one to keep track of the number of spikes detected on each channel. These can be created with the output_buffer function:

```
channel_num = 64 # number of channels

(buf,nums)=output_buffer(channel_num);
#buf is 100 x 64 array of Spike type, all initialized to (0:0, 0)
#nums is a 64 index array with each element initialized to 0
```

1.1.2 Workflow

Most algorithms require some period of calibration, such as determining the appropriate threshold for detection, or the most discriminatory features to use for clustering. Therefore, some portion of data will need to be used for training. In real time acquisition, this would be the first data collected. For post hoc analysis, this would be some, or all of the data, and then the full dataset can be used after.

1.1.3 Single Channel

To create an instance of spike sorting for a single channel, a complete Sorting type must first be instantiated:

```
detection=DetectPower() #Power-based spike detection
alignment=AlignMax()     #Align candidate spikes by their maximum voltage
feature=FeatureTime()    #Chose time varying voltage signal as feature
reduction=Reduction()    #Use all time points for clustering steps
cluster=ClusterOSort()   #OSort style clustering (compare clusters with candidate␣
↪spikes by euclidean distance)

s = Sorting(detection,cluster,alignment,feature,reduction)
```

To use the your sorting instance, you need a collection of analog voltage signals. This is assumed to be stored in a m x n matrix of Int64s, where m is the length of the sampling period, and n is the number of channels. Most methos for spike sorting require some calibration period, which is called with the cal! method. In addition, the first time you process signals with a new sorting instance, several methods that don't run everytime you calibrate (such as setting a threshold) need to be run; you can invoke these by setting the "firstrun" flag in the cal! method equal to true. Once you have finished calibration, you can call the onlinesort! method.

```
#Single channel sorting workflow. v is assumed to be an m x 1 vector of voltage values

#Create output buffers for single channel
(buf1,nums1)=output_buffer(1);

#First collect voltage trace

#Call calibration with first run flag
cal!(s,v,buf1,nums1,true)

#Define some flag to determine when you want to switch from calibration to online␣
↪sorting
while (calibrate==true)

#collect next voltage traces and overwrite v

        #Call calibration methods
        cal!(s,v,buf1,nums1)

end

#Once calibration is finished, you can perform online sorting instead for incoming␣
↪data
while (sorting==true)
        onlinesort!(s,v,buf1,nums1)
end
```

1.1.4 Multiple Channels

The same methods have also been designed to work with m x n voltage arrays, where n > 1. First, an array of Sorting types needs to be created, which can be invoked with the create_multi method:

```
num_channels=64

s2=create_multi(detection,cluster,alignment,feature,reduction, num_channels);

(buf2,nums2)=output_buffer(num_channels);
```

Now the same processing methods can be called on a 64 column voltage array:

```
cal!(s2,v,buf2,nums2,true); #first run flag set to true
cal!(s2,v,buf2,nums2);
onlinesort!(s2,v,buf2,nums2);
```

1.2 Parallelism

If multiple channels of extracellular recordings are collected simultaneously, and these channels are sufficiently far apart, as is common with multi-electrode arrays, then the spike sorting of each channel can be considered "embarassingly parallel" whereby the sorting of one channel has no impact on another. Right now, SpikeSorting.jl is designed around this principle and can create a Distributed Array of multiple Sorting instances. In this way, each core of a computer or cluster "owns" all of the data in a collection of Sorting instances, and can quickly and independent process channels without message passing back and forth

1.2.1 Implementation

Parallel multi-channel processing works almost identically to single core multi-channel. To create the multi-channel array, specify the parallel flag to be true during initialization:

```
num_channels=64

s3=create_multi(detection,cluster,alignment,feature,reduction, num_channels, true);
(buf3,nums3)=output_buffer(num_channels,true);
```

Now rather than an array of Sorting instance, mysort3 is a Distributed Array of Sorting instances. This can be applied to all of the processing methods as above:

```
cal!(s3,v,buf3,nums3,true); #first run flag set to true
cal!(s3,v,buf3,nums3);
onlinesort!(mysort3,v,buf3,nums3);
```

The code above above may not actually be faster, however, because the matrix v has to be copied to each process during each interation. To get around this, you can store your voltage values in a SharedArray:

```
v2=convert(SharedArray{Int64,2},v);
cal!(mysort3,v2,buf3,nums3,true); #first run flag set to true
cal!(mysort3,v2,buf3,nums3);
onlinesort!(mysort3,v2,buf3,nums3);
```

1.3 Real-Time Application

SpikeSorting.jl is being designed to work on real time incoming electrophysiology using a Julia wrapper for Intan evaluation boards:

https://github.com/paulmthompson/Intan.jl

1.4 Benchmarking

CHAPTER 2

Detection

2.1 Overview

Voltage samples will be compared to some threshold value to determine if spiking has occured. Different methods of detection can be defined by declaring 1) a Type that is the data structure for the algorithm, 2) a function “detect” that implements the algorithm 3) a function “threshold” to determine the threshold for comparison during the training period.

Often methods will need to be initialized during a calibration period, such as calculating the running sum of the last n samples in power detection. These initialization procedures are defined in “detectprepare” functions.

All datatypes are members of the abstract type Detect. They should have default constructors to be initialized as follows:

```
#Create instance of power detection for use in sorting
mypowerdetection=DetectPower()

#Create instance of median absolute value threshold detection for use in sorting
mymedian=DetectSignal()
```

2.2 Methods

2.2.1 Currently Implemented

Median Threshold

Each timepoint is compared to a threshold, where the threshold is generated as some multiple of the formula below:

$$\sigma_n = median\left\{\frac{|x|}{.6745}\right\}$$

Usually the threshold is set as 4 sigma.

```
type DetectSignal <: Detect
end
```

Reference:

Quiroga, R Quian and Nadasdy, Z. and Ben-Shaul, Y. Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. 2004

Nonlinear Energy Operator (NEO)

Also known as Teager energy operator (TEO). The NEO is calculated at each time point based on the current, future and past sample and compared to a threshold.

$$\psi[x(n)] = x^2(n) - x(n+1)x(n-1)$$

```
type DetectNEO <: Detect
end
```

References:

Mukhopadhyay S and Ray G C. A new interpretation of nonlinear energy operator and its efficacy in spike detection. 1998

Choi, Joon Hwan and Jung, Hae Kyung and Kim, Taejeong. A new action potential detector using the MTEO and its effects on spike sorting systems at low signal-to-noise ratios. 2006.

Power

This method looks at the local energy built up over n samples, where by default n=20.

$$P(t) = \left\{ \frac{1}{n} \sum_{i=1}^{n} (f(t-i) - \bar{f}(t))^2 \right\}^{1/2}$$

$$\bar{f}(t) = \frac{1}{n} \sum_{i=1}^{n} f(t-i)$$

Some value of power is chosen as a threshold.

```
type DetectPower <: Detect
        a::Int64 #sum of last n samples
        b::Int64 #sum of squares of last n samples
        c::Int64 #value of sample at t-n
end
```

Reference:

Kim and Kim, "Neural spike sorting under nearly 0-dB signal-to-noise ratio using nonlinear energy operator and artificial neural-network classifier," 2002.

2.2.2 Partially Implemented

Wavelet - Multiscale Correlation of Wavelet Coefficients

References:

Yang, Chenhui and Olson, Byron and Si, Jennie. A multiscale correlation of wavelet coefficients approach to spike detection. 2011.

Yuan, Yuan and Yang, Chenhui and Si, Jennie. The M-Sorter: an automatic and robust spike detection and classification system. 2012.

Yang, Chenhui and Yuan, Yuan and Si, Jennie. Robust spike classification based on frequency domain neural waveform features. 2013.

2.2.3 To Do

Amplitude detection - Multiple

Reference:

Kamboh, Awais M. and Mason, Andrew J. Computationally efficient neural feature extraction for spike sorting in implantable high-density recording systems. 2013.

EC-PC Spike Detection

Reference:

Yang, Zhi et al. A new EC–PC threshold estimation method for in vivo neural spike detection. 2012.

Dynamic Multiphasic Detector

Reference:

Swindale, Nicholas V. and Spacek, Martin A. Spike detection methods for polytrodes and high density microelectrode arrays. 2014.

Nonlinear Energy Operator - smoothed (SNEO)

Reference:

Azami, Hamed and Sanei, Saeid. Spike detection approaches for noisy neuronal data: Assessment and comparison. 2014.

Normalised cumulative energy difference (NCED)

Reference:

Mtetwa, Nhamoinesu and Smith, Leslie S. Smoothing and thresholding in neuronal spike detection. 2006.

Precise Timing Spike Detection

Reference:

Maccione, Alessandro et al. A novel algorithm for precise identification of spikes in extracellularly recorded neuronal signals. 2009.

Spatial Interpolation Detection

Reference:

Muthmann et al. Spike Detection for Large Neural Populations Using High Density Multielectrode Arrays. 2015.

Summation

Reference:

Mtetwa, Nhamoinesu and Smith, Leslie S. Smoothing and thresholding in neuronal spike detection. 2006.

Wavelet - Continuous Wavelet Transform

References:

Nenadic, Zoran and Burdick, Joel W. Spike detection using the continuous wavelet transform. 2005.

Benitez, Raul and Nenadic, Zoran. Robust unsupervised detection of action potentials with probabilistic models. 2008.

Wavelet - Stationary Wavelet Transform

Reference:

Kim, Kyung Hwan and Kim, Sung June. A wavelet-based method for action potential detection from extracellular neural signal recording with low signal-to-noise ratio. 2003.

Wavelet - Wavelet Footprints

Reference:

Kwon, and Oweiss. Wavelet footprints for detection and sorting of extracellular neural action potentials. 2011.

Kwon, Ki Yong and Eldawlatly, Seif and Oweiss, Karim. NeuroQuest: a comprehensive analysis tool for extracellular neural ensemble recordings. 2012.

CHAPTER 3

Alignment

3.1 Overview

When a candidate spike is detected, it is usually because some metric computed from the running voltage signal crossed a threshold. Consequently, a piece of the voltage signal surrounding that threshold event is sectioned out and passed to the alignment step for subsequent analysis. Background noise and other factors may not cause the same part of an extracellular potential to generate the threshold crossing event each time: as a result just taking a window around threshold crossing may make the time varying voltage pattern of a given spike forward or backward shifted compared to a previous spike from the same neuron. This can be problematic from feature extraction steps, which may not be immune to phase shifts such as this. Therefore, it can be important to apply an alignment meature to every candidate spike that is detected.

Each alignment method needs 1) Type with fields necessary for algorithm 2) function "align" to operate on sort with type field defined above 3) any other necessary functions for alignment algorithm

All datatypes are members of the abstract type Align. They should have default constructors to be initialized as follows:

```
myalign=AlignMax() #align based on the signal with the higest voltage

myalign=AlignFFT() #upsample the signal with an FFT, then perform alignment based on␣
↪the maximum
```

3.2 Methods

3.2.1 Currently Implemented

Maximum Index

Upsampling via fft

3.2.2 Partially Implemented

3.2.3 To Do

Upsampling via cubic splines

Center of mass

CHAPTER 4

Features

4.1 Overview

The output of our electrophysiology system is a time varying voltage signal. We expect extracellular potentials generated by nearby neurons to have some protypical features or shape that is different from noise, and that one neuron would also have differences from another. We can chose to make these comparisons looking at the time varying voltage trace that we are given, or we could perform some transformation of this signal to try to accentuate what we expect to be discriminatory components of the signal.

Each feature extraction method needs 1) Type with fields necessary for algorithm 2) function "feature" to operate on sort with type field defined above 3) any other necessary functions for extraction algorithm

Sometimes during calibration periods these features need to be calculated a little differently to make more or less data available to the subsequent calibration of clustering or reduction steps. These methods can be defined in "featureprepare" methods.

All data types are of abstract type Feature. they should have default constructors that are initialized as follows:

```
myfeature=FeatureTime() #use the time varying voltage signal as is

myfeature=FeatureCurv() #use the curvature of the voltage signal
```

4.2 Methods

4.2.1 Currently Implemented

Temporal

The simplest feature to use is the plain temporally varying voltage data with no transformation applied. This is invoked with:

```
myfeature=FeatureTime()
```

Curvature

The curvature at time t can be found as the following:

$$curv(t) = \frac{V''}{(1 + V'^2)^(3/2)}$$

where V' and V" are the first and second derivatives at time t. This is invoked as:

```
myfeature=FeatureCurv()
```

Reference:

Horton, P. M. and Nicol, A. U. and Kendrick, K. M. and Feng, J. F. Spike sorting based upon machine learning algorithms (SOMA). 2007.

Discrete Derivatives

Slopes are calculated at each time t over different time scales:

$$dd_\delta(n) = s(n) - s(n - \delta)$$

By default, delta values of 1,3, and 7 are used. We therefore have more features than in the original temporal waveform. This can be invoked as

```
myfeature=FeatureDD()
```

References:

Gibson, Sarah and Judy, Jack W. and Markovi{'{c}}, Dejan. Technology-aware algorithm design for neural spike detection, feature extraction, and dimensionality reduction. 2010

Karkare, Vaibhav and Gibson, Sarah and Markovi{'c}, Dejan. A 130-W, 64-channel neural spike-sorting DSP chip. 2011

4.2.2 Partially Implemented

Integral transform

References:

Gibson, S. and Judy, J. and Markovic, D. Comparison of spike sorting algorithms for future hardware implementation. 2008

Zviagintsev, A. and Perelman, Y. and Ginosar, R. Algorithms and architectures for low power spike detection and alignment. 2006.

Principle Components Analysis

Reference:

Adamos, Dimitrios A and Kosmidis, Efstratios K and Theophilidis, George. Performance evaluation of PCA-based spike sorting algorithms. 2008.

Jung, Hae Kyung and Choi, Joon Hwan and Kim, Taejeong. Solving alignment problems in neural spike sorting using frequency domain PCA. 2006.

4.2.3 To Do

Discrete Derivatives - DD2 extrema

Reference:

Zamani, Majid and Demosthenous, Andreas. Feature extraction using extrema sampling of discrete derivatives for spike sorting in implantable upper-limb neural prostheses. 2014.

Discrete Derivatives - Feature extraction using first and second derivative extrema (FSDE)

Reference:

Paraskevopoulou, Sivylla E. and Barsakcioglu, Deren Y. and Saberi, Mohammed R. and Eftekhar, Amir and Constandinou, Timothy G. Feature extraction using first and second derivative extrema (FSDE) for real-time and hardware-efficient spike sorting. 2013.

Frequency - Raw

Reference:

Yang, Chenhui and Yuan, Yuan and Si, Jennie. Robust spike classification based on frequency domain neural waveform features. 2013.

Frequency - PCA

Reference:

Jung, Hae Kyung and Choi, Joon Hwan and Kim, Taejeong. Solving alignment problems in neural spike sorting using frequency domain PCA. 2006.

Fuzzy Score

Reference:

Balasubramanian, Karthikeyan and Obeid, Iyad. Fuzzy logic-based spike sorting system. 2011.

Projection pursuit method based on negentropy maximization

Reference:

Kim, Kyung Hwan and Kim, Sung June. Method for unsupervised classification of multiunit neural signal recording under low signal-to-noise ratio. 2003.

SVD

Reference:

Oliynyk, Andriy and Bonifazzi, Claudio and Montani, Fernando and Fadiga, Luciano. Automatic online spike sorting with singular value decomposition and fuzzy C-mean clustering. 2012.

Wavelet - Discrete Wavelet Transform

Reference:

Quiroga, R Quian and Nadasdy, Z. and Ben-Shaul, Y. Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. 2004.

Wavelet - Wavelet packet decomposition

Reference:

Hulata, Eyal and Segev, Ronen and Ben-Jacob, Eshel. A method for spike sorting and detection based on wavelet packets and Shannon's mutual information. 2002.

Zero-crossing features

Reference:

Kamboh, Awais M. and Mason, Andrew J. Computationally efficient neural feature extraction for spike sorting in implantable high-density recording systems. 2013.

CHAPTER 5

Reduction

5.1 Overview

Feature spaces from raw signals can often be high dimensional; dimensionality reduction allows the sorting algorithm to only compare a subset of the possible dimensions. This can be useful in terms of speed and accuracy during the subsequent clustering step. The difficulty is that we often do not know the most "discriminatory" dimensions to use in clustering. The methods below outline methods to select dimensions of a given high dimensional feature space to use for later clustering.

Each reduction method needs 1) Type with fields necessary for algorithm 2) function "reduction" to operate on sort with type field defined above 3) any other necessary functions for alignment algorithm

There can also be a "reductionprepare" function for the calibration step if necessary.

All datatypes are members of the abstract type Reduction. They should have default constructors to be initialized as follows:

```
#Create instance of reduction type specifying that no reduction will be used
myreduction=ReductionNone()

#Create instance of reduction type specifying to select dimensions based on maximum␣
↪difference test
myreduction=ReductionMD()
```

5.2 Methods

5.2.1 Currently Implemented

No Reduction

Dimensionality reduction is not necessary for spike sorting, though it may produce more accurate results depending on the selected feature space and clustering method. No reduction is chosen by invoking:

```
myreduction=ReductionNone()
```

Maximum Difference

The maximum difference test will find the dimensions in the feature space that were most likely to have the largest difference between them during a calibration period. By default, 3 dimensions are chosen. It can be implemented as follows:

```
myreduction=ReductionMD()
```

Reference:

Gibson, Sarah and Judy, Jack W. and Markovi{‘{c}}, Dejan. Technology-aware algorithm design for neural spike detection, feature extraction, and dimensionality reduction. 2010

5.2.2 Partially Implemented

Principle Components Analysis

Refs:

Adamos, Dimitrios A and Kosmidis, Efstratios K and Theophilidis, George. Performance evaluation of PCA-based spike sorting algorithms. 2008.

Jung, Hae Kyung and Choi, Joon Hwan and Kim, Taejeong. Solving alignment problems in neural spike sorting using frequency domain PCA. 2006.

5.2.3 To Do

Laplacian eigenmaps

Reference:

Chah, E. and Hok, V. and Della-Chiesa, A. and Miller, J J H. and O’Mara, S. M. and Reilly, R. B. Automated spike sorting algorithm based on Laplacian eigenmaps and k-means clustering. 2011.

Projection Pursuit based on Negentropy maximization

Reference:

Kim, Kyung Hwan and Kim, Sung June. Method for unsupervised classification of multiunit neural signal recording under low signal-to-noise ratio. 2003.

Shannon mutual information

Reference:

Hulata, Eyal and Segev, Ronen and Ben-Jacob, Eshel. A method for spike sorting and detection based on wavelet packets and Shannon’s mutual information. 2002.

SVD

Reference:

Oliynyk, Andriy and Bonifazzi, Claudio and Montani, Fernando and Fadiga, Luciano. Automatic online spike sorting with singular value decomposition and fuzzy C-mean clustering. 2012.

Uniform Sampling

Reference:

Karkare, Vaibhav and Gibson, Sarah and Markovic, Dejan. A 130-W, 64-channel neural spike-sorting DSP chip. 2011

CHAPTER 6

Clustering

6.1 Overview

The final step of spike sorting is determining if the possiblely reduced signal represented in some feature space belows to an extracellular potential, and if so to which neuron it belongs. In offline analysis, the number of possible clusters can be determined by examining all of the data; in realtime recordings, the number of clusters needs to be determined on the fly or during a calibration period.

Each clustering method needs 1) Type with fields necessary for algorithm 2) function "cluster" to operate on sort with type field defined above 3) any other necessary functions for clustering algorithm

If the clustering step needs to perform a different function during the calibration period, it can be defined in a "clusterprepare" method.

All datatypes are members of the abstract type Cluster. They should have default constructors to be initialized as follows:

```
mycluster=ClusterOSort() #Assign a candidate spike to a cluster based on the␣
↪euclidean distance between the spike and the mean of the cluster
```

6.2 Methods

6.2.1 Currently Implemented

OSort / Hierachical Adaptive Means

These algorithms characterize clusters by their centroid in feature space, and assign a spike to a particular cluster based on minimizing the euclidean distance between the spike and the existing clusters. If the distance between the spike and existing clusters is above some estimate of the noise between clusters, then a new cluster can be created on the fly.

To account for electrode drift, the centroids that define each cluster are continuously updated with new samples.

References:

Rutishauser, Ueli and Schuman, Erin M. and Mamelak, Adam N. Online detection and sorting of extracellularly recorded action potentials in human medial temporal lobe recordings, in vivo. 2006.

Paraskevopoulou, Sivylla E. and Wu, Di and Eftekhar, Amir and Constandinou, Timothy G. Hierarchical Adaptive Means (HAM) clustering for hardware-efficient, unsupervised and real-time spike sorting. 2014.

6.2.2 Partially Implemented

CLASSIT

Reference:

Gennari, John H and Langley, Pat and Fisher, Doug. Models of incremental concept formation. 1989.

6.2.3 To Do

BIRCH

Reference:

Zhang, T. and Ramakrishnan, R. and Livny, M. BIRCH: An Efficient Data Clustering Method for Very Large Databases. 1996.

DBSCAN

Reference:

Haga, Tatsuya and Fukayama, Osamu and Takayama, Yuzo and Hoshino, Takayuki and Mabuchi, Kunihiko. Efficient sequential Bayesian inference method for real-time detection and sorting of overlapped neural spikes. 2013.

Mahalanobis Clustering

References:

Kamboh, Awais M. and Mason, Andrew J. Computationally efficient neural feature extraction for spike sorting in implantable high-density recording systems. 2013.

Aik, Lim Eng, Choon, Tan Wee. An Incremental clustering algorithm based on Mahalanobis distance. 2014.

Time varying dirichlet process

Reference:

Gasthaus, Jan and Wood, Frank and Gorur, Dilan and Teh, Yee W. Dependent dirichlet process spike sorting. 2009.

CHAPTER 7

Benchmarking

7.1 Overview

The SpikeSorting.jl package provides several ways to calculate objective metrics of a method's performance. Given "gold standard" data, most likely in the form of simulations, the quality metrics of spike sorting performance as outlined here:

http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3123734/

can be calculated. Briefly, we determine the correctly classified spikes (true positives), as well as erroneously identified spikes (false positives) and spikes which are not identified (false negatives). False positives are further divided into false positives that are attributable to 1) overlap of potentials from different neurons in the same window of time, 2) clustering a spike from one neuron as another neuron and 3) classifying noise as a spike. Additionally, false negatives are either attributed to 1) an error in the detection step (e.g. too high of a threshold) or 2) an error due to overlap of multiple potentials from different neurons.

The total number of detected spikes will be equal to the sum of the true positives and false positives.

The total number of "gold standard" spikes will be equal to the sum of the true positives and false negatives.

7.2 Calculating Quality Metrics

Quality metrics are calculated with 1) an array of arrays of time stamps marking when neurons fire in the gold standard data set 2) an array of voltage vs time of the gold standard signal 3) an instance of the sorting method of interest

Example code is as follows:

```
#time_stamps is the Array of arrays of neuron firing times
#fv is the array of voltage vs time of extracellular signal

detect=DetectPower()
cluster=ClusterOSort(100,50)
align=AlignMin()
```

(continues on next page)

(continued from previous page)

```
feature=FeatureTime()
reduce=ReductionNone()
thres=ThresholdMean(2.0)
s1=create_multi(detect,cluster,align,feature,reduce,thres,1)

cal_time=180.0 #calibration time in seconds
sr=20000 #sample rate

ss=SpikeSorting.benchmark(fv,time_stamps,s1[1],cal_time,sr)
```

The benchmark function will print the quality metrics as well as return the following tuple:

1. detected spike times
2. array with each integer indicating the total number of true spikes for a neuron
3. total number of true positives
4. array of the false positives due to clustering, overlap and noise
5. array of false negatives due to overlap and detection

7.3 Optimizing sorting method based on results

Coming soon!

7.4 Under the hood

7.4.1 Matching clusters to neurons

7.4.2 Attributing to overlap