NMSORT

A toolbox for spike sorting EMG

Najja J. Marshall

Contents

1 Toolbox Contents

Ephys custom class for working with electrophysiological data.

Spike custom class for working with spike events detected in electrophysiological data.

Neuron custom class for working with neural data (spikes and waveforms)

alignwaveforms aligns spike waveforms extracted from time series data

botm implements Bayes optimal template matching [**Franke et al, 2015**] for detecting and assigning spikes given a set of templates. Also implements subtractive interference cancellation for resolving overlapping spike waveforms.

brewermap (by Stephen Cobeldick) a function for generating attractive color schemes, based on the website Color Brewer.

brewermap_view a GUI for exploring color scheme options. See: MathWorks for details and the latest versions.

findpaths pathfinding algorithm for identifying connected nodes in an undirected graph.

findpulses detects pulse events in data. Used for spike detection.

getnoise identifies coincident quiescent segments in data across all channels.

histfun constructs a 2D histogram of functions. Used for visualizing many waveform shapes.

leave1wavefit tries fitting each template using the other templates to check if any one can be accounted for by a linear combination of the others

mixGaussVb (by Michael Chen) learns the cluster assignments for a Dirichlet Process Gaussian Mixture Model using the algorithm for variational Bayesian inference described in Bishop's *Pattern Recognition and Machine Learning* (PDF available online). `logMvGamma.m` and `logsumexp.m` are supporting functions. See: GitHub for the latest versions and code for all other algorithms described in PRML.

mutualinfo fast computation of lagged mutual information between pairs of spike trains.

noisecov estimates the spatiotemporal noise covariance matrix from a segment of quiescent time series data. [**Pouzat et al., 2002**]

rmvoutliers removes outliers from a set of waveforms

smooth1D 1-dimensional smoothing via FFT

spktrig extracts data snippets aligned to a set of spike indices

templatemanager GUI for refining final set of waveform templates

wavetemplate constructs the multi-channel template from a set of waveforms

waveclusent computes the multivariate-Gaussian entropy for clusters of waveforms

wavexcorr computes wave template weighted cross-correlation

plotwavetemplate plotting function for visualizing wave template

samplesort template for using the toolbox

2 Algorithmic Details

Our approach to spike sorting EMG can be broadly split into two parts: identify a set of well-isolated MUAPW templates, then use those templates to recover their spike times.

2.1 Pre-processing

Before any steps directly related to spike sorting occur, the data must be properly processed. This involves three steps:

1. Filter the data to reduce movement artifact and increase the temporal localization of each spike waveform.

2. Normalize each channel by the standard deviation of the noise, which is estimated as

$$\sigma_k = \text{median}\left(|\mathbf{x}_k|\right)/0.6745 \tag{1}$$

   where $\mathbf{x}_k$ is the data on channel k[1]. Normalization ensures that other functions in the toolbox generalize across data sets.

3. Identify concurrent segments of data across all channels that do not contain any spikes. These 'noise traces' are used to estimate the spatiotemporal noise covariance matrix C_n by computing the auto- and cross-correlation functions for all pairs of noise traces[2]. The noise covariance matrix is essential for the Bayes Optimal Template Matching (BOTM) algorithm, which is used to recover all spike times and resolve overlaps for a set of MUAPW templates[3].

2.2 Spike detection

We employ a two-step peak-detection algorithm for detecting spike events. In the first step, we take

$$\mathbf{y}_k = \Theta\left(|\mathbf{x}_k| - \alpha\sigma_k\right), \quad \alpha \in \mathbb{R}, \alpha > 0 \tag{2}$$

where $\Theta(\cdot)$ is the Heaviside function. All peaks in the rectified and thresholded data, $\mathbf{y}_k$ are then detected and every other time point set to zero. This transforms $\mathbf{y}_k$ via

$$\mathbf{z}_k[n] = \sum_{m=0}^{T} \delta[m]\mathbf{y}_k[n-m] \tag{3}$$

where $n = 0, \ldots, T$ denotes the sample points in the data and $\delta[n] = 1$ wherever $\mathbf{y}_k[n]$ has a peak. Thus, $\mathbf{z}_k$ consists entirely of a series of delta functions located at each stationary point in $\mathbf{x}_k$. Moreover, the amplitude of each delta function is equivalent to $|\mathbf{x}_k|$ at the same sample point. Therefore, after this first step, each MUAPW will register as a cluster of delta functions (located at each stationary point in the waveform) localized to a brief time window (the duration of the waveform).

In the second step, $\mathbf{z}_k$ is smoothed with a Gaussian filter and the peaks in the resulting signal are detected. The indices of these peaks are returned as the location of spike events.

2.3 Waveform alignment

The two-step process described above detects positive and negative spikes. Since eqs. 2 and 3 transform every stationary point in the data into a pulse, each MUAPW will register a series of pulses centered at each of its stationary points. The smoothing of $\mathbf{z}_k$ prevents each of these pulses from being detected as a separate spike event. Instead, the singular spike event that is detected will tend to coincide with the largest amplitude deflection of the MUAPW. Thus, under ideal conditions, every detected spike generated by the same MU will be aligned to the same point: the largest amplitude deflection of its waveform. However, noise and tonic spike discharge can distort waveform shapes such that their largest peak differs across spike events. Consequently, additional alignment is necessary after spike detection.

To perform the alignment, `alignwaveforms.m` creates a window centered around each spike event. All spike indices are then shifted so that the maximum squared amplitude of their corresponding waveforms are centered within the window. Since the shifting operation can introduce new, possibly larger extrema into the window, this process proceeds for several iterations or until some minimum fraction of all waveforms are aligned. If during this procedure one spike event is shifted so that it falls within some small time window of another, then one of the two spikes will be deleted. In other words, whenever shifting creates a refractory period violation, one of the two offending spikes is removed. For this reason, *it is recommended to use as small a time window as possible for alignment* – just long enough to capture the most salient waveform features, but not too long so as to increase the likelihood of refractory period violations. If too long a window is used for alignment, then small spikes can be re-aligned to center around larger, nearby waveforms. However, after the short wave snippets have been aligned, a longer window can be used to extract the full MUAPWs.

2.4 Clustering

After the alignment step, principal components analysis is used to reduce the dimensionality of the waveforms. A clustering algorithm is then run on the waveform features to assign each waveform to a distinctive unit. *Note that detection, alignment, and clustering are all performed on each channel, separately.*

We leverage Dirichlet process Gaussian mixture models (DP-GMM) for clustering. In standard GMMs, the data are modeled as arising from a mixture of Gaussian distributions:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \Lambda_k), \quad \pi_k \in [0,1] \tag{4}$$

where π_k are mixing coefficients that describe how much of each Gaussian contributes to the data[4]. The mixing coefficients can also be expressed in terms of latent cluster assignments, which permits an alternative formulation for eq. 4:

$$p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z}), \quad p(z_k = 1) = \pi_k \tag{5}$$

A central problem of fitting data to mixture models is determining how many latent factors to incorporate (i.e. choosing the "right" K). One approach to this problem is to try several different values of K and choose the one that minimizes some model selection criterion (e.g. BIC) that quantifies goodness of fit while incorporating a penalty that scales with the number of parameters.

Instead, we leverage a class of methods called Bayesian nonparametrics to learn the cluster assignments without specifying the number of latent factors *a priori*[5]. In this approach, the generative model defines a joint distribution over the observations, $\mathbf{x}$, cluster assignments, $\mathbf{z}$, and cluster parameters, $\boldsymbol{\Theta}$:

$$p(\mathbf{x}, \mathbf{z}, \boldsymbol{\Theta}) = \prod_{k=1}^{K} G_0(\theta_k) \prod_{n=1}^{N} F(x_n|\theta_{z_n})p(z_n), \quad \theta_k = \{\boldsymbol{\mu}_k, \Lambda_k\} \tag{6}$$

Using this model, we would like to compute the posterior distribution of the cluster assignments

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{\sum_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})} \tag{7}$$

in order to know which cluster we should assign to each observation. From eq. 6, we have

$$p(\mathbf{x}|\mathbf{z}) = \int_{\boldsymbol{\Theta}} \left[\prod_{n=1}^{N} F(\mathbf{x}|\theta_{z_n}) \prod_{k=1}^{K} G_0(\theta_k) \right] \, \mathrm{d}\boldsymbol{\Theta} \tag{8}$$

Solving the posterior distribution under this model is intractable because we need to compute the marginal likelihood (denominator of eq. 7), which requires summing over every partition of the data into K classes. The Bayesian nonparametric approach addresses the problem of choosing K by assuming that there exists an infinite number of latent clusters, but that only a finite number of them are used to generate the data. In practice, this is achieved by specifying a prior over infinite groupings in such a way that it favors assigning data to a small number of groups. The prior used for this is the Dirichlet distribution.

Whereas most familiar distributions are defined over a vector space (e.g. $\mathbb{R}^n$, $\mathbb{Z}$), the Dirichlet is a distribution defined over other distributions. A Dirichlet process (DP) is parameterized by a base distribution, G_0, and concentration parameter, α. The reason for using a Dirichlet prior is that repeated draws from a Dirichlet process exhibit a natural clustering property with α serving as a type of inverse variance (small values of α increase the likelihood of drawing a repeated value). The complete model is thus

$$\begin{aligned} G &\sim \mathrm{DP}(\alpha, G_0) \\ \theta_{z_n} &\sim G \\ x_n &\sim F(\,\cdot\,|\,\theta_{z_n}) \end{aligned}$$

for which posterior inference is now tractable. This is still very technical and the mathematical details beyond the scope of this document, but a popular method and the one used in this toolbox is variational inference. Briefly, the idea is to pick a family of distributions that approximates the conditional posterior,

$$q(\mathbf{z}) \approx p(\mathbf{z}|\mathbf{x})$$

then find a member of q that best matches p by minimizing the KL divergence

$$D_{\mathrm{KL}}(q||p) = \sum_{\mathbf{z}} q(\mathbf{z}) \log \frac{q(\mathbf{z})}{p(\mathbf{z}|\mathbf{x})}$$

To perform inference we borrow code from the Pattern Recognition and Machine Learning Toolbox [PRMLT], which includes code for implementing all the algorithms described in Bishop[4]. This particular inference algorithm treats α as a hyperparameter to be learned alongside K. Critically, whereas the DP-GMM models the data as arising from a finite number of infinite factors, it is, of cousre, infeasible to represent infinite latent factors on a computer. Consequently, it is necessary to truncate the Dirichlet distribution by providing an upper limit on the number of factors, κ. The beauty of DP-GMM is that the number of learned latent factors can be less than κ. In other words, some of the latent factors are allowed to be unused if a smaller number adequately model the data. In general, κ can be fairly generous, but a larger truncation will typically lead to longer runtimes.

2.5 Identification of unique templates

By this point, the user will have, at most, $M\kappa$ total clusters, where M is the number of recorded channels. So long as κ exceeds the true number of MUs detectable in the recording, many clusters should be well isolated, containing only the waveforms belonging to one unit. However, depending on the signal-to-noise on each recorded channel, the number of active MUs and the rate of coincident spiking, some clusters are likely to contain noise events or waveforms formed by the superposition of several MUs. Moreover, variability in waveform shapes can lead to inconsistent alignment for a given unit, which will cause its waveforms to be split into multiple clusters. Aside from these issues related to clustering within each channel, there remains the additional hurdle that a given MU is likely to have a distinctive waveform signature across several channels and so we will need to pool our clusters across channels in order to construct each MU's multi-channel waveform template. In summary, we need to identify a set of multi-channel templates that represents a unique set of MUs (i.e. while rejecting noise and duplicate clusters).

Constructing multi-channel templates from single-channel clusters requires solving two problems. First, we need to identify which clusters should be grouped across channels. We can then take the mean waveform shape from each cluster as the signature of the MU on that channel. Second, for a given MUAPW, we need to determine how to align its mean shapes across channels relative to each other. We address both problems with a simple solution. Rather than stitch together clusters across channels, we instead leverage the fact that each cluster of waveforms has a corresponding set of spike indices. Thus, we can use each set of clustered spike indices to take a spike-triggered average of the data across all channels. This will provide a set of multi-channel templates

$$\mathbf{w}_i = [\mathbf{w}_{i,1}^{\mathrm{T}}, \mathbf{w}_{i,2}^{\mathrm{T}}, \ldots, \mathbf{w}_{i,M}^{\mathrm{T}}]^{\mathrm{T}}, \quad i = 1, 2, \ldots, M\kappa \tag{9}$$

where $\mathbf{w}_{i,k}$ is the waveform of template i on channel k.

Since the spike-triggered average is taken for each of the $M\kappa$ clusters, many templates will be duplicates of the same unit. Conversely, clusters that consisted primarily of noise or poorly aligned waveforms are unlikely to generate templates that look like any other. We leverage both of these points to reduce our full set of $M\kappa$ templates. First, we compute the similarity between each template as

$$S_{ij} = \frac{\min(\mathbf{w}_i^{\mathrm{T}}\mathbf{w}_i,\ \mathbf{w}_j^{\mathrm{T}}\mathbf{w}_j)}{\max(\mathbf{w}_i^{\mathrm{T}}\mathbf{w}_i,\ \mathbf{w}_j^{\mathrm{T}}\mathbf{w}_j)} \max\left[\sum_{k=1}^{M}(\mathbf{w}_{i,k} \star \mathbf{w}_{j,k})(t)\right] \tag{10}$$

That is, we take the maximum cross-correlation between the templates weighted by the minimum ratio of their energies. The assumption being made here is that a pair of duplicate templates will nearly line up on top of each other modulo a single temporal shift. The values S_{ij} are then folded into an adjacency matrix and a path finding algorithm used to identify all connected nodes in the graph, where a node is said to be connected to another if $S_{ij} > \beta$. Each set of "connected nodes" corresponds to a set of templates that are all similar to each other according to eq. 10. From each set of similar templates, the one that has the lowest entropy is returned to the user for the final step in the algorithm.

2.6 Recovery of spike times and resolution of overlaps

Given our final set of N templates, the final step is to recover their spike times and resolve instances of overlapping waveforms. To do this we leverage the Bayes optimal template matching (BOTM) algorithm[3]. BOTM integrates matched filtering and linear discriminate analysis to determine whether a spike has occurred and, if so, which template to assign it to. To resolve overlapping waveforms we leverage subtractive interference cancellation, as described in[3].

3 Using the Toolbox

The toolbox includes several custom objects to accelerate data exploration, standardize across data sets, and streamline the pipeline.

3.1 Loading your data

The very first step that must be done is to load your data into the Ephys object:

```
% Fs: sample rate (Hz)
% X: data matrix (observations x channels)

EMG = Ephys(Fs, X);
```

3.2 Quick start

listing 1 provides a guide for implementing the main components of the algorithm (as described above). Recommended values are provided for all necessary constant parameters and, wherever possible, sequential steps are condensed by stacking methods calls (e.g. lines 5 and 13-16). This outline is for optimizing speed. For a more detailed explanation of the various steps and examples of various options the toolbox provides, see listings 2 to 5.

```
 1 % filter & normalize
 2 [EMG,sigma] = EMG.filt('bandpass',2,[500 2000]).normalize;
 3 
 4 % identify noise trace
 5 [xNoise,noiseLim] = getnoise(EMG.data, EMG.Fs, 'normalized',true);
 6 
 7 % estimate noise covariance matrix
 8 Cn = noisecov(EMG.range(noiseLim).data, EMG.Fs, 20e-3);
 9 alpha = 0; % set small (e.g. 1e-8) if Cinv is poorly conditioned
10 Cinv = (Cn + alpha*eye(size(Cn,1)))^(-1);
11 
12 % detect, align, cluster
13 Spk = EMG.detect_spikes...
14     .align(EMG.data, 3e-3, 5e-4)...
15     .get_waveforms(EMG.data, 5e-3)...
16     .cluster(3, 25);
17 
18 % get templates
19 waveforms = spktrig(EMG.data, cat(1,Spk.indices{:}), round(EMG.Fs*20e-3));
20 
21 % manually select final templates using the template manager GUI
22 % export results as "templates"
23 templatemanager(EMG.data, EMG.Fs, Cinv, waveforms)
24 
25 % get final spike times
26 load('templates')
27 spikes = botm(EMG.data, EMG.Fs, w, Cinv, 'verbose',true);
```

Listing 1: quick start template

3.3 Template manager GUI

Everything up to manually selecting the final set of templates (line 23, listing 1) can be done with minimal oversight. After the clustering step, the final set of templates is obtained using the `templatemanager` GUI. fig. (1) provides a screen shot of the GUI with each major component labeled. A description of each component is provided below.

1. Working list of "good" templates. This is the final set that will be exported. Multiple templates can be selected at once.

2. "Remove": removes currently selected template(s) from the working set, which returns it to its original class (matched or unmatched tabs – see 9 and 10). "Exile": places the template(s) aside to a relegated class. Think of this as a trash bin, where you might place templates that are unequivocally bad so that you don't accidentally reconsider them later.

3. Plot of working set of good templates (1).

4. Check this box to only plot the templates that are selected in (1).

5. Similarity threshold on eq. 10 used to determine duplicate templates. This number (β) must be between -1 and 1 (though values between 0.7 and 0.9 are likely to be most useful). If $\beta = 1$ that means that two templates must be literally identical for them to be grouped together. The smaller β, the less similar two templates are allowed to look for them to be considered the same. Try adjusting this to get a good initial guess at the templates, then manually add or remove clusters as needed. But note: *changing this value will reset all progress.*

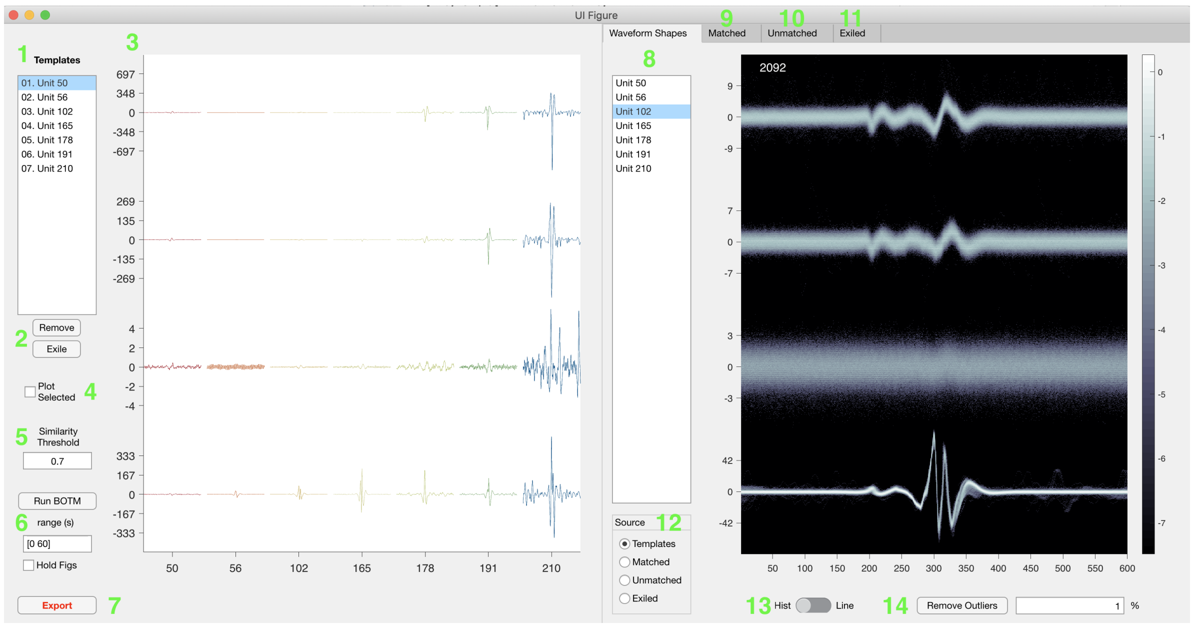

Figure 1: templatemanager GUI

6. Controls for running BOTM on a segment of the data using the working set of good templates (1). The edit box sets the time limits over which to run the BOTM algorithm. Press the button to run, which will plot the template fits to that section of the data, the reconstruction, residual, and residual energy. Check the "hold figs" box to overwrite the auto-generated figures the next time the "Run" button is pressed, rather than generate new figures.

7. Export good set of templates. Saves the templates listed in (1) as a rank-3 tensor (samples $\times$ channels $\times$ units) to a file. After loading this file to the work space, the templates can be passed immediately to the BOTM function (line 66, **??**) to get the final spike times.

8. Waveform shape inspector. Visualizes the waveform shapes for the selected template (only one at at time). Spike count displayed in top left of axes. Helpful for checking the isolation quality of a template and removing outliers. Can either plot the waveforms as a histogram (default) or as lines (much slower).

9. Matched tab. List of templates that were "matched" with at least one other, based on the similarity threshold. Sometimes one of these are actually better isolated than their counterpart, which is automatically placed in the working set (1).

10. Unmatched tab. List of templates that had no match with any others. Depending on your similarity threshold, this will contain mostly noisy templates.

11. Exile tab. A special list of templates that are set aside to prevent further consideration.

12. Sets the source for the waveform shapes listbox. Change this value to, for example, quickly inspect the waveform shapes in the "duplicates pile".

13. Toggle waveform shapes axes between histogram and line view.

14. Remove outliers. Can be adjusted to remove outliers from a set of waveforms. Adjust this value (between 0 and 100) and press the button to remove that percentage of outliers from the set. This can be adjusted up or down from its current value (and always reset to 0 to recover the original set of waveforms). Typically, values between 0 and 25% are most useful.

3.4 Pre-processing

```
1  %% Pre-processing
2  
3  % 2nd order bandpass filter between 500 and 2000 Hz
4  FILT_ORD = 2;
5  FILT_CUT = [500 2000];
6  EMG = EMG.filt(’bandpass’, FILT_ORD, FILT_CUT);
7  
8  % 2nd order highpass filter at 500 Hz
9  % EMG = EMG.filt(’high’,2,500);
10 
11 % normalize
12 [EMG,sigma] = EMG.normalize;
13 
14 % inspect the filtered data
15 PLOT_RANGE = [0 60]; % sec
16 EMG.range(PLOT_RANGE).plot
17 
18 % only plot channel 1
19 EMG.chan(1).range([0 300]).plot
20 
21 % identify noise trace
22 [xNoise,noiseLim] = getnoise(EMG.data, EMG.Fs, ’normalized’,true);
23 
24 % check that it worked
25 EMG.range(noiseLim).plot
26 
27 % if not, specify your own noise trace
28 % MY_NOISE_RANGE = [0 1];
29 % noiseLim = EMG.range(MY_NOISE_RANGE).data;
30 
31 % final template duration (sec)
32 TEMPLATE_DUR = 20e-3;
33 
34 % estimate noise covariance matrix
35 Cn = noisecov(EMG.range(noiseLim).data, EMG.Fs, TEMPLATE_DUR);
36 
37 % invert noise covariance matrix
38 % if Cinv is poorly conditioned, add "a little identity" to Cn make it
39 % invertible. Set alpha to something small (e.g. 1e-8)
40 alpha = 0;
41 Cinv = (Cn + alpha*eye(size(Cn,1)))^(-1);
```

Listing 2: data pre-processing

listing 2 elaborates on data pre-processing. The `Ephys` object includes a `filt` method that can be used to apply a Butterworth, differentiating, boxcar, or Gaussian filter to the data. Lines 6 and 9 show how to apply a band- or high-pass filter. See m-file for additional options.

The `range` method allows the user to plot the data falling with a specified time range (line 16). The `chan` method can also be used to restrict to a specific channel or set thereof (line 19).

The function `getnoise` attempts to automatically identify a coincident segment of noise across all channels (line 22) . The outputs are the multi-channel noise matrix (`xNoise`) and the identified time limits (`noiseLim`). The success of this step can be determined by inspecting the noise trace as shown in line 25.

The noise segments are used to estimate the noise covariance matrix (line 35), whose inverse is needed for the BOTM algorithm. Using a longer segment of noise (e.g. 1 second) than is needed for the final template duration provides a better estimate of the covariance matrix. If the covariance matrix is poorly conditioned, taking its inverse will throw a warning. This can be ameliorated by add a bit of the identity to the covariance matrix before taking its inverse. Do this by setting α (line 40) to something small, like 10^{-8}). Increase as needed until line 41 proceeds without warning.

3.5 Detection, alignment, and clustering

```
%% Detection
% detect positive and negative spikes with a threshold of 4 sigma
Spk = EMG.detect_spikes;

% check initial detection
EMG.range([0 60]).plot('Spk',Spk)

% check only on one channel
CHAN = 1;
EMG.chan(CHAN).range([20 40]).plot('Spk',Spk(CHAN))

% use a different threshold per channel
THR = 5;
Spk(CHAN) = EMG.chan(CHAN).detect_spikes('thresh',THR);

% use an asymmetric threshold of -4 sigma
Spk = EMG.detect_spikes('sym',false, 'thresh',-4);

% use a wider filter width (if spikes are more multiphasic)
Spk = EMG.detect_spikes('minWid',2e-3);

%% Alignment
% align waveforms using a 3 ms window and 0.5 ms refractory period
Spk = Spk.align(EMG.data, 3e-3, 5e-4);

% extract 5 ms long waveforms for subsequent clustering
Spk = Spk.get_waveforms(EMG.data, 5e-3);

% plot all waveforms after alignment
figure
Spk.plot_hist

% restrict limits of histogram
figure
Spk(CHAN).plot_hist('lim',[-500 500])

%% Clustering
% cluster with 3 features and truncate the Dirichlet to 25 max clusters
Spk = Spk.cluster(3, 25);

% plot waveform clusters, limited to 25 plots per page (ppp)
Spk(CHAN).plot_hist('ppp',25)

% plot all waveforms without labels
Spk(CHAN).clr_labels.plot_hist

% plot PC 1 vs. time from channel
Spk(CHAN).plot_feat([0 1])
```

Listing 3: spike detection, waveform alignment and clustering

listing 3 elaborates on spike detection, alignment, and clustering. By default, the detect_spikes method uses a symmetric detection threshold as described in section 2.2. The detection results can be inspected by adding the Spike object as an argument to the plot method (line 6). This will plot the data and overlay a red marker at every index where a spike was detected. If detection found more or fewer spikes than desired, a smaller or bigger threshold can be used (lines 14 & 15). A single (i.e. asymmetric) threshold can also be used if desired (line 17). And the user can adjust the width of the Gaussian filter used in the second step of the spike detection algorithm (smoothing $\mathbf{z}_k$, defined in eq. 3). The Gaussian filter is set to a width of half the minimum spike width (minWid, line 20).

The align method takes two inputs for alignment: a window size and a refractory period. Any time a refractory period violation is created through the course of alignment, one of the two offending spikes is removed (as discussed in section 2.3). After alignment, the waveforms can be extended to a longer duration using the get_waveforms method (line 27). All aligned waveforms can be plotted as a heatmap using the plot_hist method (line 31), which will plot all channels in one figure as subplots. Select channels can be plotted instead and the limits of the heatmap restricted using an optional argument (line 35).

The cluster method runs the DP-GMM on the set of waveform features stored in the Spike object. This method takes two inputs: the number of features to be used and a truncation on the maximum number of allowed clusters. Typically using three or four features is sufficient. If there are lots of multi-unit clusters, a larger truncation can be used (e.g. 50, 75, 100). More features and higher truncations will add to runtimes. The clustered waveforms can be inspected using the plot_hist method, which will automatically separate each cluster into its own subplot. If a large truncation point was used, a limit can be placed on the number of subplots per page (line 42); if there are more clusters than this limit, the function will automatically create new figures as needed. After waveforms have been labeled via the clustering step, the unlabeled, aligned waveforms can be plotted just as in line 35 by first calling the clr_labels method (line 45). Finally, the waveform features can be plotted with the labels using the plot_feat method. This method can take a 2- or 3-D vector of inputs specifying which features to plot against each other. The value 0 corresponds to time. All other nonzero values correspond to that number principal component.

3.6 Identification of unique templates

```
1  % spike triggered extraction of waveforms across channels
2  spkIdx = cat(1,Spk.indices{:});
3  waveforms = spktrig(EMG.data, spkIdx, round(EMG.Fs*TEMPLATE_DUR));
4  
5  % compute template cross-correlation and plot similar templates
6  w = wavetemplate(waveforms);
7  [wXC, enerRat] = wavexcorr(w);
8  grp = findpaths(wXC .* enerRat > 0.85);
9  plotwavetemplate(w(:,:,grp{1}))
10 
11 % compute waveform entropy
12 wEnt = waveclusent(waveforms,3);
13 [~,bestUnit] = min(mean(wEnt(grp{1},:),2));
14 N_PLTS = 2;
15 for ii = 1:N_PLTS
16     subplot(1,N_PLTS,ii)
17     histfun(waveforms(grp{1}(ii),:),'plot',true,'nbins',5e3,...
18         'count',length(spkIdx{grp{1}(ii)}));
19 end
20 
21 % manually select final set of templates
22 % export final templates as "templates"
23 templatemanager(EMG.data, EMG.Fs, Cinv, waveforms)
24 
25 % check if any template is likely a linear combination of others
26 load('templates')
27 leave1wavefit(w, EMG.Fs, xNoise, Cinv)
```

Listing 4: getting the final set of templates

listing 4 elaborates on the process of identifying the set of unique templates that will be used to obtain the final set of spike times. After the clustering step, the `indices` method can be used to return the set of spike indices across all clusters and channels (line 2). These can then be used to take a spike-triggered extraction of the data across channels to get the multi-channel set of waveforms for each set of spike indices (line 3).

Lines 6-9 show how the waveform similarity metric (eq. 10) is used to identify duplicate templates. `wavexcorr` computes the template cross-correlation for all pairs of templates as well as their energy ratios. The product of these two matrices is a matrix whose elements are S_{ij} (eq. 10). Lines 8 and 9 plots one set of grouped templates based on a similarity threshold of 0.85.

Lines 12 and 13 demonstrate computing the multivariate Gaussian entropy for each cluster of waveforms and using this quantity to identify the best isolated cluster from a group. Lines 14-19 show how to plot the set of waveforms across all channels for a cluster as a heatmap.

Note: lines 5-19 are built into the `templatemanager` GUI and these quantities automatically computed given the requisite inputs (line 23), so these lines are not needed for every sort.

After exporting a set of templates, `leave1fit` can be used to check whether any one template can be explained as a linear combination of the others. This is done by embedding each template in the noise trace and running the BOTM algorithm using the remaining set of templates. This can be visually diagnostic if it is unclear if a template is that of a unique MU or simply the result of a linear combination of some other MUs.

3.7 Recovery of final spike times

```
load('templates')
spikes = botm(EMG.data, EMG.Fs, w, Cinv, 'verbose',true);
```

Listing 5: inferring spike times for templates

After settling on a final set of templates, the BOTM algorithm is used to obtain their spike times. The `botm` function allows for several plotting options (all used by the template manager), including overlaying the template fits, reconstruction, and residual on top of the raw data as well as plotting the residual energy of either the entire data or restricted to regions where spikes were detected. See the mfile for more information.

References

[1] Quiroga, R. Q., Nadasdy, Z. & Ben-Shaul, Y. Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. Neural computation **16**, 1661–1687 (2004). 4

[2] Pouzat, C., Mazor, O. & Laurent, G. Using noise signature to optimize spike-sorting and to assess neuronal classification quality. Journal of neuroscience methods **122**, 43–57 (2002). 4

[3] Franke, F., Quiroga, R. Q., Hierlemann, A. & Obermayer, K. Bayes optimal template matching for spike sortingcombining fisher discriminant analysis with optimal filtering. Journal of computational neuroscience **38**, 439–459 (2015). 4, 8

[4] Bishop, C. M. Pattern recognition and machine learning (springer, 2006). 5, 7

[5] Gershman, S. J. & Blei, D. M. A tutorial on bayesian nonparametric models. Journal of Mathematical Psychology **56**, 1–12 (2012). 6