MET Analysis Kit (MAK)

Data Analysis Tutorial

VERSION 1.0, 07/07/2020

Written by Jackson Smith

Introduction

MAK is a function library written for Matlab. It is the offline counterpart to the Matlab Electrophysiology Toolbox (MET) and was initially written to analyse data that was collected with MET. A subset of the MAK library is devoted to pre-processing raw data collected with MET and a Blackrock Cerebus using Utah arrays; hence many of MAK's functions are designed to leverage Matlab's native ability to handle matrices of data, a natural data structure for storing multi-channel electrophysiology. However, some of MAK's functions are of more general use for analysing any pre-processed data with any type of probe. This tutorial is not a comprehensive guide to MAK. Instead, it gives a brief introduction to those functions which may be useful for a general data analysis.

Alphabetical list of functions

Run `$ doc <function name>` from the Matlab command line for more detailed inline documentation, which often includes functionality that is not discussed here. For a complete list of MAK functions, see `mak/readme.txt`.

`makbindvar.m`	– Group dependent variable based on binning paired, independent variable.
`makcf.m`	– Shorthand for cellfun(… , 'UniformOutput' , false).
`makconv.m`	– Fast and convenient convolution function.
`makddi.m`	– Disparity discrimination index.
`makfun.m`	– Apply any function to grouped data.
`makgabor.m`	– Evaluate 1D Gabor with a given set of coefficients over a set of values.
`makgaborfit.m`	– Find the least-squares, best-fitting Gabor for a set of tuning curves.
`makimat.m`	– Logical indexing matrices that access the upper-triangular half of a matrix.
`maklinfin.m`	– Bias-corrected linear Fisher Information.
`makmi.m`	– Non-parametric estimate of Mutual Information.
`makpspkern.m`	– Schall's spike convolution kernel, shaped like a post-synaptic potential.
`makrastplot.m`	– Quickly make a spike raster plot.
`makrccg2.m`	– Newer implementation of r_{CCG} that is optimised for speed.
`makroc.m`	– Area under ROC curve, Youden's J, bootstrap CI, and ROC curve itself.
`makxcorr.m`	– Fast cross-correlation function for older versions of Matlab.

Functions are discussed in the following sections. They are ordered by their perceived use. Some of the simpler functions are only demonstrated through examples written for other functions.

makfun

makfun is inspired by the *fun suite of Matlab functions, such as cellfun, arrayfun, and structfun. Each of these permits the user to apply any arbitrary function to the contents of some data structure; the function is specified by a function handle.

Example

Let T be an M x N cell array across M trials and N separate neurones. For the i^th trial and j^th neurone, T{i,j} contains a 1 x R vector of spike times in seconds; R can be different on each trial and for each neurone.

View the spike rasters for the j^th neurone across trials by:

```
% The curved brace returns a cell array that is a sub-set of T.
% This is very different from T{:,j} which returns all individual
% spike time vectors for neurone j in a comma-separated list.
$ makrastplot( T( : , j ) )
```

Let w = [w_1,w_2] define a spike-counting window with edges $w_1 < w_2$ in seconds. The width of window w is dw = w(2) - w(1). In order to find the firing rates from all neurones on all trials in two lines of code, we execute:

```
% Counts the spikes in T using an anonymous function
$ X = cellfun( @( s ) sum( w( 1 ) <= s & s <= w( 2 ) ) , T ) ;

% Change units from spikes to spikes/second, the firing rate
$ X = X ./ dw ;
```

The result is X, the M x N matrix in register with T containing the firing rates for each corresponding trial and neurone.

In a similar manner, makfun allows an arbitrary function to be applied to grouped data, where each group of data is passed into the function separately of one another. In practice, this is most useful for computing tuning curves.

In the most basic format, one calls:

```
[ y , n , u ] = makfun( <function handle> , x , <grouping data> )
```

Data in x is grouped and then each group is passed into the given function. This returns y, the output of the function as applied to each group, along with n the number of values in each group, and u the unique and ordered set of grouping values.

Example

Let X be an M x N matrix of firing rates across M trials and N separate neurones that were recorded simultaneously such that X(i,:) is the population response on the i^th trial. Let s be the M x 1 vector of stimulus values on each trial, in register with rows of X; all values in s come from a set of V unique values. The average firing rate of each neurone in response to each unique stimulus value, the tuning curves, are found by:

```
$ [ y , n , u ] = makfun( @( x ) mean( x , 1 )' , X , s ) ;
```

y is the N x V matrix of average firing rates for each neurone (rows) in response to each stimulus value (columns); y(j,:) is the full tuning curve for the j^th neurone. n is the V x 1 vector containing the number of trials in each group, in register with columns of y. u is, in fact, a 1 x 1 cell array, and u{1} contains the V x 1 ordered vector of unique stimulus values, also in register with columns of `y`. Notice that the anonymous function is careful to operate across rows, only, and to return a column vector.

To plot the tuning curve for the `j`th neurone, run:

```
$ plot( u{ 1 } , y( j , : ) , 'o-' )
```

The reason that `u` is a cell array is to allow for multiple grouping factors.

Example

Let `c` be an `M x 1` logical vector containing `1`'s on trials in which the subject made a correct choice and `0`'s when the subject was incorrect. To compute tuning curves separately for correct and incorrect trials, run:

```
$ [ y , n , u ] = makfun( @( x ) mean( x , 1 )' , X , s , c ) ;
```

`y` is now the `N x V x 2` array of tuning curves, indexing neurones across rows, unique and ordered stimulus values across columns, and ordered trial outcome (`0` then `1` i.e. incorrect then correct) in dimension 3. `n` becomes the `V x 2` matrix of trial counts for each stimulus (rows) and outcome (columns). `u` is now the `1 x 2` cell array; `u{1}` still contains the `V x 1` ordered vector of unique stimulus values; `u{2}` now contains the unique and ordered set of outcome codes `[0,1]` in register with dimension 3 of array `y`.

In certain cases, the same groupings need to be applied multiple times. Grouping information can be returned by:

```
[ G , ng , n , U ] = makfun( <grouping data> )
```

`G` is a matrix of logical index vectors, where each column defines a separate group of data. `ng` is the number of grouping factors, `n` is the number of samples per group, and `U` is the cell array of ordered, unique values in each grouping factor. This grouping information can be applied with:

```
y = makfun( <function handle> , x , G , ng )
```

which achieves the same calculation as the basic function call defined above.

Example

Group trials by stimulus value:

```
$ [ G , ng , n , u ] = makfun( s ) ;
```

`G` is an `M x V` logical matrix where `G(:,v)` is the logical index vector that identifies all trials with the `v`th stimulus value. `n` and `u{1}` are the same `M x 1` and `V x 1` vectors as before.

Use this grouping information to compute tuning curves:

```
$ y = makfun( @( x ) mean( x , 1 )' , X , G , ng ) ;
```

`y` is the same `N x V` matrix of average firing rates as before.

Use the grouping information again to compute standard error of the mean:

```
$ e = makfun( @( x ) std( x , 0 , 1 )' , X , G , ng ) ;
$ e = e ./ sqrt( n') ;
```

Note that binary singleton expansion is implicit in recent versions of Matlab. In older versions of Matlab, run:

```
$ e = bsxfun( @rdivide , e , sqrt( n') ) ;
```

Plot the tuning curve of the `j`th neurone with SEM error bars:

```
$ errorbar( U{ 1 } , y( j , : ) , e( j , : ) , 'o' )
```

Further use grouping information to compute z-scores separately for each stimulus value:

```
Z = zeros( size( X ) ) ;

for  i = 1 : ng , g = G( : , i ) ;
  Z( g , : ) = zscore( X( g , : ) ) ;
end
```

Now find the `N x N` noise correlation matrix, pooling across all groups of trials:

```
$ C = corr( Z ) ;
```

The correlation matrix is mirrored across the diagonal. Unique noise correlation values can be pulled out of the upper-triangular half of `C` by first creating an `N x N` logical index matrix to do so:

```
$ I = makimat( N ) ;
```

We can now easily plot a histogram of noise correlations for all unique pairings of neurones, with no repeats:

```
$ histogram( C( I ) )
```

Suppose we find that the `n`th and `m`th neurones are a well-correlated pair, and want to examine this relationship by eye. We might try:

```
$ scatter( Z( : , n ) , Z( : , m ) )
```

But this could be very messy to look at, making it difficult to see the correlation. One might try to plot the average response of neurone `m` as a function of the binned response of neurone `n`. Indeed, the average and SEM can be obtained by:

```
$ [ avg , err , bin ] = makbindvar( Z( : , n ) , Z( : , m ) ) ;
```

This bins trials using the responses of the `n`th neurone into `B` bins. Those groups of trials are then used to compute the average response and SEM for the `m`th neurone, separately for each group. This is returned in the `1 x B` vectors `avg` and `err`, each in register with vector `bin`, the centre of each of the `n`th neurone's response bins. Plot the relationship in responses of this correlated pair by:

```
$ errorbar( bin , avg , err , 'o' )
```

The uses of `makfun` are limited only by the analyst's imagination.

makcf

`cellfun` makes the default assumption that a scalar value will be returned from each execution of the function handle. It may be more desirable to return a larger data structure. `cellfun` allows this by calling `$ cellfun( <fun> , … , 'UniformOutput' , false )` in which case `cellfun` returns another cell array; each cell contains the non-scalar output from each execution of the function handle. For those who find this name/value pair to be too verbose, `makcf` provides a wrapper function for `cellfun` in which the `'UniformOutput' , false` name/value argument pair are always applied, implicitly. See `makconv` below for an example of its use.

makconv

Matlab's `conv` function is limited in that it can only convolve vector input arguments. `makconv` is far more efficient because it vectorises the convolution process. One may apply a single convolution kernel to an entire set of different signals in a single function call.

```
C = makconv( X , k , type )
```

`X` is an array with `T` rows and any size across remaining dimensions. Every column of `X` is a separate signal. `C` is the convolution of `X` with the vector `k`. `type` is a character code that specifies what kind of filter that kernel k is.

Example

Let `T` be an `M x N` cell array across `M` trials and `N` separate neurones. Each cell contains the `1 x R` vector of spike times in seconds fired from that neurone on that trial. Define millisecond-wide bins between times w_1 and w_2:

```
$ bins = round( w₁ : 0.001 : w₂ , 3 ) ;
```

Rounding to the nearest millisecond eliminates small errors that result from the finite-precision arithmetic of digital computers, even with the double floating point numeric type.

Bin all spike times into binary spike rasters with millisecond resolution:

```
$ X = makcf( @( t ) histcounts( t , bins )' , T ) ;
```

`makcf` returns `X` the `M x N` cell array. `X{i,j}` is the `B x 1` vector of `0`'s and `1`'s across `B` bins defining the spike raster on the i^{th} trial for the j^{th} neurone; bins containing a `1` show where a spike was fired, bins with `0`'s show where the neurone was silent. Our goal is to get a `B x N x M` array of spike rasters across individual bins (rows), neurones (columns), and trials (dimension 3). Start by performing the equivalent of a high-dimensional transpose using Matlab's `permute` function:

```
$ X = permute( X , [ 3 , 2 , 1 ] ) ;
```

`X` is now the `1 x N x M` cell array of `B x 1` spike raster vectors, with neurones indexed across columns and trials indexed across dimension 3. Collapse the cell array into a `B x N x M` double array of spike rasters:

```
$ X = cell2mat( X ) ;
```

We are now ready to convolve these spike rasters in order to estimate the instantaneous firing rate at each millisecond time point. First we define a symmetrical convolution kernel that is Gaussian shaped with a 10ms time constant, making sure that the cropped function integrates to 1:

```
$ kgauss = normpdf( -0.04 : 0.001 : +0.04 , 0 , 0.01 ) ;
$ kgauss = kgauss ./ sum( kgauss ) ;
```

Next, we use Schall's causal kernel, which is shaped like a post-synaptic potential with a 1ms rising constant and a 20ms decay constant:

```
$ kpsp = makpspkern ;
```

Convolve each spike raster separately using each type of kernel. Notice that the kernel types are given as character codes so that the convolved output is in temporal alignment with the spike rasters:

```
$ Cgauss = makconv( X , kgauss , 's' ) ; % 's'ymmetric kernel
$ Cpsp   = makconv( X , kpsp   , 'c' ) ; %    'c'ausal kernel
```

Both `C*` output arguments are `B x N x M` arrays of spike trains that have been convolved across the time domain (rows). `makconv` uses a Fourier based approach to convolution, because convolution in the time domain equals point-by-point multiplication in the Fourier domain. But this requires use of the inverse Fourier transform. It is possible for small negative values to appear when non-negative spike rasters are convolved with non-negative kernels, as a result of finite-precision error. To eliminate this, simply apply Matlab's `max` function:

```
$ Cgauss = max( 0 , Cgauss ) ;
$ Cpsp   = max( 0 , Cpsp   ) ;
```

Compute the average firing rate across trials, separately for each neurone.

```
$ Agauss = mean( Cgauss , 3 ) ;
$ Apsp   = mean( Cpsp   , 3 ) ;
```

Each `A*` is now the `B x N` matrix of average firing rates for each type of convolution kernel.

makxcorr

In older versions of Matlab, the cross-correlation function `xcorr` was implemented inefficiently using for-loops. `makxcorr` uses a vectorised, Fourier-based approach which is far more efficient. Recent versions of Matlab have re-implemented `xcorr` along the same lines, thus no appreciable reduction in execution time is now obtained with `makxcorr`. However, in complete defiance of Matlab's own column-major indexing convention, `xcorr` returns all pairwise cross-correlations from a set of signals in a row-major order. In comparison, `makxcorr` returns data in a more intuitive format for neuroscientists:

```
Y = makxcorr( X )
```

If `X` is an `M x N` matrix of `N` separate signals across `M` samples then `Y` is the `2M-1 x N x N` array of all auto- and cross-correlations. Think of the ith row of `Y(i,:,:)` as being a single `N x N` correlation matrix at the ith lag. If one prefers to think of their correlation matrices in terms of rows and columns then `Y` can easily be re-arranged with the `permute` function:

```
$ Y = permute( Y , [ 2 , 3 , 1 ] ) ;
```

Returning an `N x N x 2M-1` array where the `N x N` correlation matrix at the ith lag is obtained by `Y(:,:,i)`.

makrccg2

An implementation of Wyeth Bair's $r_{CCG}(\tau)$ noise correlation metric that is optimised for speed. Let `X` be a `B x N x M` array of `B` millisecond-wide spike-counting bins (rows) for a set of `N` neurones (columns) recorded simultaneously over a group of `M` trials (dimension 3). r_{CCG} is found for all pairs of neurones by calling:

```
R = makrccg2( X )
```

which returns the `B x N x N` array `R` of correlation values across integration widths from `0ms` up to `B-1ms` (rows) for every pair of neurones (columns & dimension 3). For the `w`th integration width:

```
squeeze( R(w,:,:) )
```

returns the `N x N` square correlation matrix. Alternatively:

```
permute( R , [ 2 , 3 , 1 ] )
```

rearranges `R` into an `N x N x B` array of `B` square correlation matrices, one for each integration width.

`makrccg2` is suited for resampling analyses because the spike raster data `X` can be computed once and then repeatedly resampled, calling `makrccg2` on each version of the resampled data.

MAK has an older implementation of r_{CCG} called `makrccg` (without the '`2`'). This implementation takes spike counts as the input and bins them before computing r_{CCG}. Hence, it may be more convenient in some cases, but it is inefficient for any resampling analysis. Additionally, `makrccg` can return the normalised cross-correlograms for each pair of input signals. Beware that the auto-correlation r_{CCG} values from `makrccg` are used to return a different quantity and, hence, are not equal to `1`.

makroc

The receiver operating characteristic (ROC) curve is an essential tool in signal detection theory. Area under the ROC curve (AROC) is the probability that an ideal observer can correctly classify a randomly sampled value as having come from either a distribution of true-positive values or from a distribution of false-positive values. In practice, this is used to quantify the psychophysical performance of an ideal observer given distributions of spike counts, for direct comparison with a subject's performance. It can also be used to quantify the behavioural correlation of residual variance in the spike counts. `makroc` efficiently computes AROC for a series of neural signals across a set of `M` trials.

Let `X` be the `M x B x N` array of `N` neural signals recorded across `M` trials and measured in a series of `B` time bins. For example, `X` could be a rearranged version of the convolved spike trains found in the example for `makconv`, above:

```
% Re-arrange the causal, PSP-shaped convolution of spike trains
X = permute( Cpsp , [ 3 , 1 , 2 ] ) ;
```

This re-arrangement is necessary because `makroc` operates across rows. Suppose that the stimulus was the same on all `M` trials but that the subject's choice was variable. Let `choice` be `M x 1` logical vector that codes the subject's choice as `1` on any trial when the choice was congruent with the neurones' preferred stimulus or `0` on any trial when the choice was incongruent. The choice probability can be found by calling:

```
CP = makroc( X , choice )
```

where `CP` is the `B x N` matrix of AROC values at each time point (rows) for each neurone (columns). We can plot the histogram of choice probabilities across neurones at the `i`th time point with:

```
histogram( CP( i , : ) )
```

Alternatively, the average choice probability across neurones as a function of time is:

```
plot( 0 : B - 1 , mean( CP , 2 ) )
```

`makroc` can return other useful values in addition to area under the ROC:

```
[ AROC , Y , CI , T , F , TH ] = makroc( X , p , a )
```

Youden's J statistic (`Y`) is the threshold value that provides maximum discrimination between true- and false-positives. 95% bootstrap confidence intervals (`CI`) can be estimated for each value in `AROC`. `T` and `F` are required to plot the ROC curves, as each is the cumulative probability that the true- (`T`) or false- (`F`) positive distributions have values less than or equal to each possible threshold value in `TH`. For the ith time bin and jth neurone, the ROC curve is plotted by:

```
% Slice out ROC curve data for a single time point and neurone
 t =  T( : , i , j ) ;
 f =  F( : , i , j ) ;
th = TH( : , i , j ) ;

% NaN placeholders exist for repeats of each threshold values
reps = isnan( th ) ;

% Non-nan values
 t( reps ) = [ ] ;
 f( reps ) = [ ] ;
th( reps ) = [ ] ;

% Plot CDF functions separately for true- and false-positives
figure
plot( th , [ t , f ] )

% At last, plot the ROC curve, with diagonal chance line
figure
plot( [ 0 , 1 ] , [ 0 , 1 ] , f , t )
axis tight square
```

makmi

The non-parametric, shuffle-corrected mutual information between two quantities can be computed with `makmi`. Let `X` be the `M x B x N` array of firing rates from `N` neurones recorded across `M` trials and measured in a series of `B` time bins. And let `s` be the `M x 1` vector of stimulus values on each trial. If there are `V` unique stimulus values then we can rank each stimulus value from `1` to `V` by:

```
[ ~ , ~ , srank ] = unique( s ) ;
```

The mutual information is estimated non-parametrically by binning the data to compute joint probabilities. Estimate the shuffle-corrected information using `V` bins:

```
MI = makmi( srank , X , [] , V )
```

where `MI` is the `B x N` matrix of mutual information between the stimulus value and the firing rate at each point in time (rows) for each neurone (columns).

Shuffle-correction is done by estimating the chance amount of information from randomly re-shuffled data; the process is repeated 30 times by default and the average shuffled-information is subtracted from the raw, empirical value. Given the finite amount of data and the finite number of repeats, it is possible that the shuffle-corrector exceeds the raw, empirical value by some small amount, producing negative values. These are easily replaced with zeros by:

```
MI = max( 0 , MI ) ;
```

A by-product of estimating the shuffle-corrector is that a distribution of chance information values is built up. This can be used to estimate the significance (p-value) of the shuffle-corrected mutual information:

```
[ MI , ~ , PVAL ] = makmi( srank , X , reps , V )
```

where `PVAL` is the estimated significance of each value in `MI`. The null hypothesis is that the measured information is the amount expected by chance. If the p-values are small then we can reject this in favour of the alternative hypothesis that the measured information exceeds what is expected by chance.

The resolution of the returned p-values is limited by the number of repetitions used to estimate the shuffle-corrector. This value can be set in `reps`. `30` repetitions give a resolution of 1/30 (minimum p-value step of `0.033`); `1000` repetitions give a resolution of 1/1000 (min p-value step `0.001`); `2000` repetitions give a resolution of 1/2000 (min p-value step `0.0005`); and so on. Increasing the number of repetitions gives both a more accurate shuffle-corrector and p-value, at greater computational cost.

makddi

As demonstrated, mutual information can be used to quantify the sensitivity of a neurone's firing rates to changes in the value of a stimulus feature. A traditional way to quantify the sensitivity of a neurone to binocular disparity is the disparity discrimination index (DDI). Let `X` be an `M x N` matrix of firing rates from `N` neurones across `M` trials, while `s` is the `M x 1` vector of disparity values that were presented on each trial, sampled from a unique set of `V` disparities. DDI is found by:

```
ddi = makddi( X , s ) ;
```

Returning `ddi`, the length N vector of DDI values for each neurone.

maklinfin

Bias-corrected linear Fisher information in the population firing rates of simultaneously recorded neurones. Suppose that the firing rates of `N` neurones are recorded in response to either stimulus values s_1 or s_2. Each stimulus value is presented in `T` trials, for a total of `2T` trials. That is, half of the trials show stimulus value s_1 and the other half shows s_2. Let the average population response be the `N x 1` vectors `F1` and `F2`, with average firing rates in response to s_1 and s_2, respectively. Similarly, let `C1` and `C2` be the `N x N` square noise-correlation matrices of the population in response to each stimulus value. The difference between the two values is $ds = s_1 - s_2$. Empirical Fisher information (`I`) and its estimated variance (`V`) are returned by function call:

```
[ I , V ] = maklinfin( T , N , ds , {F1,F2} , {C1,C2} , 'normal' )
```

Alternatively, the information in an equivalent population of de-correlated neurones *i.e.* neurones with statistically independent noise can be estimated by:

```
Ishuffle = maklinfin( T , N , ds , {F1,F2} , {C1,C2} , 'shuffle' )
```

In practice, empirical data is almost never balanced with the same number of trials per stimulus value. If the imbalance is small and there are many trials, then one approach is to randomly discard the excess trials for one stimulus value before computing firing rate vectors and noise correlation matrices. If there are precious few trials then another approach is to resample trials, re-computing `F*` and `C*` each time and then re-running `maklinfin` to build up a distribution of Fisher information values; the average resampled value then becomes the estimated Fisher informaion.

Be aware that the bias-corrected linear Fisher information becomes unstable if the following inequality is violated:

```
T ≥ ( N + 5 ) / 2                                                [1]
```

Therefore, one must have at least `4` trials in order to estimate the information in a single pair of neurones (`N = 2`). If there are too few trials to satisfy inequality `[1]` for the number of neurones in the population then the population itself could be re-sampled a number of times. The resampled population size must then satisfy the inequality:

```
2T - 5 ≥ N                                                       [2]
```

Where the number of trials limits the size of the population. Information is computed for each re-sampled population and the average value becomes the estimated information.

makgaborfit

The 1-dimensional Gabor function is good at characterising certain kinds of tuning curves *e.g.* binocular disparity. Given a set of measured tuning curves, `makgaborfit` can return the coefficients for the least-squares, best-fitting Gabor fit separately to each tuning curve.

Example

Let `Y` be the `V x N` matrix of average firing rates for `N` neurones across `V` stimulus values. `x` is the `V x 1` vector of unique, ordered stimulus values, in register with rows of `Y`. Fit a 1D Gabor to the tuning curve of each neurone:

```
$ [ C , r2 ] = makgaborfit( x , Y ) ;
```

`C` is the `6 x N` matrix of best-fitting coefficients such that `C(:,i)` is the set of coefficients describing the Gabor that best fits the i^{th} neurone's tuning curve. `r2` is the `1 x N` vector of R^2 values i.e. coefficients of determination for each fit.

If `E` is the same size as `Y` and contains corresponding SEM then the empirical data can be plotted next to the best-fitting Gabor of the i^{th} neurone with the following:

```
% Create a new figure and axes ready for multiple plots
$ figure , hold on

% Plot empirical firing rates with SEM at each stimulus value
$ errorbar( x , Y( : , i ) , E( : , i ) , 'o' )

% Sample the Gabor at some relatively high resolution in the
% stimulus space
$ s = x( 1 ) : 0.001 : x( end ) ;
```

```
% Plot best-fitting Gabor next to the empirical tuning curve
$ plot( s , makgabor( C( : , i ) , s ) )
```