# SUMMARY

# The purpose of this code is to take some example event templates selected by the user, and give the probably that each time bin in the data came from the distribution defined by the event templates under gaussian assumptions (we found that almost all real world data does not match the Poisson ideal of shot noise due to additional noise sources).

# INSTALLATION

# Simply download the code in the github folder:

# <https://github.com/swenceslaoe/Final_ASAP4_paper_detection_code>

Run DEMO.m. You should put break points in, such as at the very end of detect\_events, to study how the data is transformed as it passes through the code, and read the internal documentation to have a deep understanding of how the probabilities are being calculated in order to better interpret them.

On a mid2012 MacBook Pro with the following technical specifications:

Processor: 2.7 GHz Quad-Core Intel Core i7

Memory: 16BG 1600 MHz DDR3

Startup Disk: 4TB SSD

Graphics: NVIDIA GeForce GT 650M 1 GB

Demo.m has a run time of approximately 18 seconds from the time parpool is booted up and ready to run. Note that parpool allows the code to run on all 4 cores of the above Processor in parallel, but if you’re computer does not have multiple cores or if you do not have the parallel computing toolbox, you can change the parfor loop in the detect\_events code to a regular for loop. For large datasets, a dedicated data processing computer with a large multi-core CPU or GPU is strongly recommended.

# Dependencies

-----------------------------------------------------------------------------------------------------

MATLAB Version: 9.10.0.1739362 (R2021a) Update 5

MATLAB License Number: STUDENT

Operating System: Mac OS X Version: 10.15.7 Build: 19H2026

Java Version: Java 1.8.0\_202-b08 with Oracle Corporation Java HotSpot(TM) 64-Bit Server VM mixed mode

-----------------------------------------------------------------------------------------------------

MATLAB Version 9.10 (R2021a)

Parallel Computing Toolbox Version 7.4 (R2021a)

Signal Processing Toolbox Version 8.6 (R2021a)

Statistics and Machine Learning Toolbox Version 12.1 (R2021a)

# DESCRIPTION OF CODE

The program is setup to require everything to be normalized to 1. Templates need to start at 1. The length of the templates you feed into the program are taken to be the window size for computing the log likelihood ratio. The mean of the template distribution the data is being compared to at each point is the mean of all of the templates at that point. The std of the template distribution at that point is the std of all of the template spikes at that point.

# The program takes mean of the template, finds peak to determine if the indicator is upwards or downwards.

# Enter function filter\_for\_detection.

[smooth\_wave,data,fF,fflow\_all,Flowpass,ASAPtime,base,Noise\_std\_all,B\_vec] = filter\_for\_detection(data,template,fs,cutoff,updown,varargin);

User can select the lowpass filter frequency they want, default is 20hz. Can also select no filter. The signal is divided by this lowpass filtered version of itself, to get a df/F trace.

Smooth wave is defined as a lowpass filtered version of the trace, filtered at 25% of the sampling rate. The mean of the original signal is added, and then smooth\_wave is divided by its own mean.

Data – The original data the user fed in.

fF – matrix of filtered traces. Traces are highpass filtered at Fpass, or 20hz default if nothing is specified. This is divided by fflow (data lowpass filtered with 2’nd order butterworth at 20hz).

Fflow – Lowpass filtered data at default 20hz, 2’nd order butterworth.

Fpass – User specified pass frequency.

ASAPtime – Time bins in units of 1/fs for the data.

Base – A vector of ones, since the data is centered at 1.

Base\_subtracted – take fF, subtract base from it. Now it’s just fF, but centered at 0.

Chopped\_signal – Take base\_subtracted, chop cutoff from either end, which is a user specified number of data points to drop off the ends. Could also just set it to 0 so nothing is chopped.

Noise\_std – Calculated 1 of 3 ways.

**First attempt**: Fits a mixed gaussian model to the chopped\_signal, with the assumption that there are two gaussians present. One is the signal, one is the noise. This tends to work well for traces were lots of activity is present, otherwise it tends to only find a single gaussian. The standard deviation of the noise is set to be the std. of the first gaussian.

**Second attempt:** If the standard deviation of the GMM fit is greater than the standard deviation of the whole chopped\_signal, then something bad happened with the fit because this makes no sense. The program then reverts to what we call the “flippy technique”.

Flippy technique: This assumes that the signal creates an asymmetric gaussian distribution (called skew). normpdf is used to create a guassian based on the mean and std dev of the GMM fit to the first gaussian (which for upward indicators would be noise). For downward indicators, the right side must be pure noise. For upward indicators, the left side must be pure noise. This half of the distribution is then “flipped” to the other side, and the std of the flipped distribution must be the std. of the noise.

**Third attempt:** Occasionally, attempt 2 somehow fails, and the std. of the flipped dist. Is greater than the std of the total data. In this case, the noise\_std is just assigned the value of the std of the chopped\_signal, and we move on.

Note that how the Noise\_std is chosen will effect the numbers the log likelihood vector puts out. Smaller Noise\_std values will result in larger log likelihood values, and vice versa.

B\_vec – Mean of base with length of the template. Background vector.

For tr = 1:size(data,2) % Enter loop for each trace in data

Template\_mean – the mean of all of the templates. The average template.

Signal\_std – If there are more than 2 templates, and the data is at least 1000 data points long, then it is std(template\*base(1000,tr)); Otherwise it is std(template\*base(length(ff),tr)); If there are less than 3 template traces, the standard deviation of the templates is just the standard deviation of fF, written as std(fF(:,tr))\*ones(1,length(template));

ENTER THE GENERATION OF THE PROBABILITY VECTOR IN FUNCTION sig2prob

Each time point is processed in a loop, per trace.

template\_mean – Rewritten as the mean of the template traces, but with the t’th point of the base vector added to it (t being the time bin in the data we are on), with the first data point of the mean of the template traces subtracted. This serves the purpose of allowing the template average to “float” on the slowly evolving background, whatever that is. This means that for comparison purposes, the template is always mounted on 1, since that is what base is. Subtracting the first point of the mean of the templates allows the template to rise (for upward indicators) or fall (for downward indicators) from the mean of the background, meaning the background at time t is the start point for the template.

Signal\_std – If there are fewer than 10 templates, it is set to the Noise\_std. This is a rather hacky fix, and 10 was chosen arbitrarily. The issue was that for small template numbers (which you shouldn’t do anyways), the signal\_std was dramatically inaccurate, and it caused issues with accurate detection. The real solution is just to pick a large number of template traces, such that the addition of another n template traces does not significantly (statistically speaking) change the signal\_std anymore.

If more than 10 traces are present, signal std is the std of (template \* base) for that trace.

Another arbitrary fix: Any signal\_std timepoints (every timebin in the template average has an std now) that is less than Noise\_std/100, is assigned the value of Noise\_std. This was done to deal with situations where all the templates fed in were normalized to say 1 at the peak, which resulted in situations where there was no std at that timebin, so all datapoints were infinitely far in probability space.

**Integration portion**

Points where the template mean is above the base at that point have flipped integration limits from points where the mean is below the base at that point.

**Integration works as follows for the signal**: For points where the template\_mean is above base (the baseline), a distribution defined by having mean template\_mean at that point, and std signal\_std at that point, will be integrated from -Inf to the value of the data point.

For points where the template mean dips below base (the baseline), the same thing is done, but then 1-the integral is taken. This effectively flips the integration so that we are now integrating a distribution defined by having mean template\_mean at that time point, and std signal\_std at that time point, from the data point to Inf.

Combined, the points above and below the base should give an integral value for each timebin of the template. The product of these integral values is the probability that that stretch of data came from the distribution defined by the templates.

**Integration works as follows for the noise**: For points where the template\_mean is above base (the baseline). From integrating the distribution with mean base and Noise\_std from the data point to Inf.

For points when the template mean is below base (baseline), the distribution with mean base and Noise\_std is integrated from -Inf to the data point.

Thus probability values for length(template\_mean) data points are obtained, and all multiplied together to get the probability that the stretch of data came from the noise distribution.

Some additional lines take care of situations where MATLAB rounds to 0, which introduces Inf values later when we divide probability of signal by probability of noise. If these lines still fail, the following is implemented:

tiny\_numbers\_integrals – I coded up a way of doing Riemann sums to allow MATLAB to deal with lots of decimal places. For extreme cases e.g. if your signal is 100 std from the mean, it still fails and will return a 0. For our typical data this hasn’t been a problem, but for some extreme data this could happen, and break points in the likelihood vector will appear where values go to infinity.

Ratio – The final log likelihood ratio that gets plotted. This is the log of the signal\_likelihood/noise\_likelihood. The log is taken because it makes the trace much easier to view.

This is repeated until an entire data vector has been analyzed, minus length(template) data points at the end so we don’t have to deal with padding when we go off the end of the vector.

Finally the ratio vector is shifted in time by W(1) to account for the shift induced by this convolution-like process.

**End sig2prob, back to detect\_events**

**Determine log likelihood ratio threshold for significance**

The user can manually input a ratio threshold to use. Note that because all of the signals have a probability distribution, and all of the noise has a probability distribution, that the threshold is effectively a line between these two. Everything to one side gets called a signal, everything to the other side gets called noise. This means that any given threshold will have associated true positive and false positive rates, as well as true and false negative rates. The threshold is input by giving the program a desired false positive value, which it then converts into a log likelihood threshold value.

**determine\_threshold** – A function that takes the statistics of the noise and the templates, and assigns a threshold at the place where the two distributions first intersect. This would mean that for identically shaped distributions, with just the mean offset for one, this function would find the point where the rate of false positives is equal to the rate of false negatives. If you want to bias it one way or the other, you want to put in a desired FP rate.

Temp – The distribution defined by having mean(base) and std Noise\_std is integrated from -Inf to template\_mean. So one value for each value of template mean.

The same is then done for the noise detecting itself.

These values are then plugged into the Schnitzer equations for getting S\_mean or B\_mean.

**If no value is put in for ratio\_thresh**

The probability of every template is compared against the mean of all of the templates, to get a log likelihood probability for the template encountering each of the templates that make it up.

We now have a number of likelihood values equal to the number of templates we used. Sometimes a huge template, or a tiny one would be included, which would bias the standard deviation by a lot for this vector. To get rid of outliers, we bootstrap. 3 numbers are grabbed at random from this vector 100,000 times. The standard deviation of each of these 3 numbers is taken, so we have 100,000 std values. We then take the median (50’th percentile) of these std values as the true std value.

If there are fewer than 3 templates being used (which you should never do anyways), the signal distribution std is just set to be the Noise\_std. The issue with having them be the same is that when bleaching happens, or illumination changes, the Noise\_std will change.

The mean of the noise in probability space is obtained by shuffling the data vector, and assuming it is all noise.

D’ formula is applied once we have B\_mean, S\_mean, and standard deviations for both, to get a d’ for the templates applied to a given data vector.

These values are used to find the point where FNR==FPR, and sets it there. If the user input a desired FPR, it finds the ratio\_thresh needed to give that desired rate. Note that since templates tend to be more obvious versions of a signal, the actual d’ will often be less (unless truly representative templates are chosen), meaning the threshold will be on the conservative side.

**End no value is put in for ratio\_thresh**

Back to detect\_events

find\_spikes – Finds all of the points in ratio that are above the threshold we determined. Every segment above threshold is considered an electric event. The peak in the log likelihood ratio for the duration that it remains above threshold is considered the peak of the event, e.g. an action potential.

Raw peak locations (raw\_locs in the code) are determined by searching the raw data +/- 2 data points from the ratio peak location for the highest (or lowest for a downward indicator) point in the raw data.

Filtered peak locations (locs in the code) are determined by doing the same, but on the vector smooth\_wave.

spike\_peaks – the peaks from the ratio data.

Back to detect\_events

**get\_spike\_info** – This function pulls lots of different stats and parameters out of the data. It is technically less important because it does not do the detection, and you can just ignore the output from this function and use the spike timings to compute your own metrics. 4-11hz is used as the bandpass frequency limits for pulling out theta. 1’st order butterworth bandpass used. Then Hilbert transform done on that data, and the absolute value taken to get the amplitude, and the angle taken to get the phase. Gets fed spike\_peaks! These are labeled ratio etc. in the spikes\_struct.

Back to detect\_events

After this, lots of the computed metrics are saved in a structure called spikes\_struct, and a lot of relevant metrics are plotted and saved as an image.

**End tr loop, this loop runs once for each data trace**

**End code description.**

**Gaussian Log Likelihood Probability Calculation Example**

Consider data points x to x+length(template). We want to know what the probability that these data points make up a spike is, and we want to do it in a way that takes into account the shape of the spike.

The probability that data point x came from the template is the gaussian integral from -Infinity to that data point, integrating the gaussian distribution with mean of the mean template trace at the very first entry (template(1)), and a std equal to the standard deviation at that point.

The probability that the x+1 data point came from the template is the same thing, only at the second entry. So the gaussian integral from -Infinity to the x+1 data point value, using the average of the second template entry as the mean of the gaussian, and the std of all of the template spikes at the second entry as the std.

This pattern continues until you reach data point x+length(template), meaning you have taken gaussian integrals for a number of data points. How many data points? The length of your template, whatever that is. You now have a vector of numbers with length equal to the length of your template. Each number is a probability. We are assuming independence due to noise, so to get the total probability that the data points we are looking at came from the template, we multiply each of the probabilities together. Why is that? Because we are assuming they are independent.

Take coin flips for instance. What is the probability of getting heads twice in a row? It is (1/2)\*(1/2), or ¼ which is 25%. That’s because the events are independent. The chance of getting heads the first time is 50%. To get heads a second time, you have to get it the first time too. Thus you have a 50% chance of getting to the point where you have a 50% chance of getting the second heads. Put another way, 50% of 50%, or 25%. This is because the probability of each event occurring is completely independent of the events that came before and after it. Since our data comes from light which is poisson, and is further noised by camera read noise etc., we assume each of our data points is independent.

The same process is repeated, only using points x to x + length(template) from the trace used as baseline. The integral here is taken from the data point to infinity. The mean of each distribution is taken to be the value of the base at that point, and the std is taken to be the std of the entire highpass filtered ff vector, with the first and last 0.1 seconds cut off to get rid of filtering artifacts. Note that this overestimates the std of the baseline distributions, because some of the points contributing to the ff vector are actually spikes. This will make the program more conservative, by making it appear more likely that data came from the baseline (noise) distribution. If you have some data from the same imaging conditions that you know doesn’t have spikes, you could put that in instead and get a better estimate. Assuming you don’t though, we have to estimate it.

For each length(template) set of data points, as you can see we get a set of numbers (length(template) of them) that they came from the templates, and the same size set that they came from the baseline. We multiply all of the template probabilities by each other and get a single number, and we multiply all of the baseline probabilities by each other and get a single number. With the signal likelihood in the numerator, we take the ratio of these two numbers. To keep it from blowing up, we take the log of this ratio since it linearizes exponentials.

We now have a single number describing the likelihood that the data points x to x + length(template) constitute a spike. We then slide over 1, so that now we are grabbing x+1 to x+1 + length(template) data points, and repeat the procedure. This continues until we hit the end. At the very end we can’t take probabilities when there aren’t more data points, so we stop length(template) data points short of the end.

Thus each entry of ff has a corresponding number in the log likelihood ratio, except for the last length(template) points. We then do detection on this log likelihood vector, which looks a lot like the ff vector except spike shapes have been exaggerated. If you want to get technical, we are convolving the data ff with our log likelihood filter, which is effectively a special kind of bandpass filter.

**Determining Significance Threshold in Log Likelihood Space: Building the Distributions**

Now we have the log likelihood ratio of the data. Great! At what point will we set our cutoff threshold in log likelihood space though, in order to say a spike is a spike? To determine this, we first need to figure out what the log likelihood ratio looks like when it encounters signals, and what it looks like when it encounters noise. Then we can set a threshold in a probabilistic way, such that we can say “by setting it here, there is an x percent chance of getting a false positive (call noise a spike), and a y percent chance of getting a false negative (call a spike noise)”.

We take the mean of base, multiplied by a vector of ones of length(template) to serve as the background here. The standard deviation of the background is the standard deviation of the highpass filtered trace, with the ends (0.1 seconds) chopped off.

Each template spike is then compared to the mean of all of the templates and this background vector, giving a single probability ratio value for each template vector. I found that a couple of template vectors were often much bigger than the others, which gave a couple of outlier points when calculating the log likelihood ratio of each template vector. Specifically, these outliers would be well above the mean. This ends up giving the Log likelihood of the signal distribution an artificially large standard deviation. To deal with outliers, I used bootstrapping, which is the statistically proper way to do it.

100,000 samples with replacement, 3 at a time, are drawn from template probability vector. Described above. Each of these 100,000 clusters of 3 values has a std. The chance that each of these 3 values has an outlier in it is small, so most of the standard deviation values calculated from these clusters of three should not contain the outliers. Since this was done 100,000 times, we have 100,000 bootstrapped std values. The median value is taken to be the std of the log likelihood signal distribution, as it should ignore the impact of outliers while being greater than the std values that coincidentally included only low values. The median mean is taken to be the true mean of the log likelihood signal distribution, as it is less effected by outliers now.

Now we have a very good estimate of the mean and standard deviation of the log likelihood signal distribution.

We only have a mean for the noise distribution though, which we got from computing the log likelihood for a vector with std = std of the ff vector, and mean equal to the mean of the base vector, which is the baseline and is very close to the mean of the ff vector. How do we get a standard deviation?

As you have noticed, the log likelihood ratio relies on order as well, meaning things shaped like the template will be more likely, although high values in general help boost your log likelihood ratio. Length(ff) samples were drawn, with replacement, from ff. This means that we have all of the same data points, they are just shuffled in terms of their order, as would be expected for noise. The shuffled points still have the same mean and std.

These shuffled points are treated as a data vector, and the log likelihood ratio is computed for the entire shuffled vector. Any spikes in the data should have been broken up, so we get a vector of log likelihoods from the data, except that order has been shuffled. The histogram of this data has a mean and standard deviation. The mean will be extremely close to the mean of the noise distribution we calculated before, effectively being the same number. We have an std now though, which is the std of the log likelihood noise distribution.

**Determining Significance Threshold in Log Likelihood Space: Determining Significance**

Now that we have means and stds for both the signal and the noise distributions in log likelihood space, we can pick a threshold log likelihood value, and compute the false positive (FP) and false negative (FN) rates, both of which we want to minimize. The false positive rate is the gaussian integral of the noise distribution we created from the threshold chosen, called ratio\_thresh in the code, to infinity. This gives an estimated rate of false detection.

The false negative rate is computed by taking the gaussian integral of the signal distribution we just determined, from -Infinity to the threshold chosen, called ratio\_thresh.

The optimal point for minimizing both FPs and FNs is the point at which the two distributions meet each other. The code will automatically calculate this point, and you can tell it to set the threshold there by simply inputting any string e.g. ‘middle’, for the desired\_FP variable.

Alternatively, you can tell it to set the threshold given a desired false positive rate. For example, if you want 1 FP per hour and you are sampling at 1000hz, you would input 1/(1000\*60\*60) for desired\_FP, since the rate is per sample. This has the benefit of allowing you to put statistically relevant guarantees on your probability of false detection. If your data is noisy though, setting the ratio\_thresh to a level where you are actually getting only one FP per hour will result in a lot of false negatives. Both rates are printed on the graph the code generates in the top right panel.

**Determining Significance Threshold in Log Likelihood Space: Picking Out Detected Spikes**

Whenever the log likelihood ratio goes above ratio\_thresh, a spike segment is started, and whenever it dips below ratio\_thresh the segment is ended. A time interval (min\_int) can be set such that if peaks occur within that interval of each other, the largest is taken to be the spike and the rest are ignored. All data points in the highpass filtered data that correspond to positions in the log likelihood vector that were above threshold are taken, broken up into the same spike segments. The position of the max of each spike segment in the ff data is found. M points before this max are grabbed, and N points after it are grabbed, and together the vector is called a spike. M+N+1 is equal to length(template). The mean of template is taken, and the position of the peak is found by max value. The number of points before this peak is M, and the number after it is N. Thus the spikes you detect will be of the same duration as your template, with the peak in the same relative position.

This could use some refining, as we are vulnerable here to missing spikes that occur very close together in time. For instance, if a massive burst occurs, and the log likelihood ratio stays above threshold for 200ms, the entire episode will be recorded as a spiking event. The max will be found and a spike detected relative to that, but there may be multiple spikes within the event we are not seeing. For this reason, the variable spikes\_struct that is output contains all of the data points from ff that occur in each spike segment, as well as the spike determined from the methods above and the indices for the detected data points.

**How the threshold is set in log likelihood space if you don’t manually set it**

The log likelihood ratio is calculated, but instead of feeding it data the individual traces from the template matrix are fed in as data. Since we assumed these were all real spikes, we now know what the log likelihood ratio should look like when we hit a spike. Specifically, we have a mean and a standard deviation that we can define a signal distribution for in log likelihood space.

We do that again, but this time feeding it the average of the baseline trace, which we know is noise. This gives us a number, which is the mean of the noise distribution in log likelihood space. Importantly, I assume the standard deviation of this distribution is the same as the signal one. Some of the rationale for why this had to be done is in the comments in the code, but it means that the templates should come from data that was recorded under similar conditions as the data it is being compared to. For example, you wouldn't want to use templates from 100mW of power on a sample where 25mW of power was used, as it would greatly underestimate the noise.

Now that we have a known signal and a known noise mean, we can compute d' on them. We can also set a threshold in the log likelihood ratio that will give us known detection rates, given that we know where the signal and the noise distributions are.

Questions? Email swenceslaoe@hotmail.com