FIXME: reference to 2nd reference paper needs to updated when accepted.

1. Introduction

Distributed networks of neurons, or cell assemblies, are widely assumed to be fundamental to brain functioning. A subset of these networks is thought to be formed by consistent timing of action potentials, or spikes, between neurons, a feature of spike recordings across species. The spiking between neurons of such networks can be synchronous or involve time, forming spike sequences when firing in a consistent order. Spike sequences can involve the same neurons and can occur within the same time window.

Finding networks defined by their spike timing consistency, or spike timing networks, is a tremendous challenge due to their possible complexity, as neurons can participate in multiple spike sequences at a continuum of between-spike time delays. Though many techniques exist to characterize spike timing networks, they typically suffer from several limitations. Namely, either, (1) the complexity of the identified networks is limited due to combinatorial explosion with increasing network size (e.g. template searching), (2) the networks are described only by the relatedness of their member neurons without describing spike sequences, (3) between-spike time delays greater than 0 are either discarded or not recovered, (4) temporal binning of spike times leads to a temporal precision vs detection trade-off, (5) networks with overlapping member neurons are not separated, or a combination of the above. These qualities are essential for the exact identification of neurons and their spike sequences, and to investigate their occurrence as a function of experimental variables.

This tutorial presents the procedure for an approach that can extract spike timing networks with arbitrary complexity of their spiking sequences, with sub-millisecond time delay precision, from neural recordings at arbitrary scale. This can be achieved by taking advantage of the fact that between-neuron spike timing consistency is reflected by phase differences over frequencies in the spectral covariance, i.e. the cross spectra, of the frequency-transformed neural spiking time series. The method behind our approach uses the structure in the cross spectra over neurons, frequencies, and trials (or epochs) of the neural recording, to extract spike timing networks. These networks describe the between-neuron spike timing consistencies of separate networks by time delays between neurons, forming spike sequences.

The extracted spike sequences given by the spike timing networks should be considered an aggregate spike sequence. That is, over the course of the recording, the involved neurons spike with the temporal relationship indicated by the sequence. This does not necessarily mean that the extracted sequences occur always (or at all) in full. For example, some trials might only contain a part of the sequence. As such, the extracted sequences are intended to be used as a first step in an analysis pipeline. They provide a template with high precision timing, which can be used for an informed search for identifying the sequences' exact occurrences on a trial-by-trial basis (e.g. by a template search, of which many exist).

Overview

In this tutorial you can find information on how you can extract spike timing networks from neural spike recordings using a technique denoted as SPACE. The main purpose of this tutorial is to provide a concrete step-by-step example of the procedure for extracting spike timing networks, with all the code necessary to go from the example dataset on disk to visualizing extracted networks. On occasion, it is thin w.r.t. to explaining what the parameters mean, what can, and what can't, be said about them, why certain choices are made, etc. This tutorial is, alas, not as polished as I wish it to be. As such, it is important to keep SPACE reference paper[2] mentioned below nearby, as it goes into more depth on some elements of extracting spike timing networks. When things are accidentally contradictory, the reference paper(s)[1,2] should be considered as the standard.

Spike timing networks will be extracted using the MATLAB toolbox *nwaydecomp* and *FieldTrip*, located at https://github.com/roemervandermeij/nwaydecomp and http://www.fieldtriptoolbox.org. The *nwaydecomp* toolbox contains several algorithms to find structure in neural recordings, of which this tutorial will use SPACE. One of the uses of SPACE is to extract spike timing networks. Other uses are described in other tutorials. If you use SPACE as described in this tutorial, please cite the following two reference papers (in addition to the FieldTrip reference paper, mentioned at its wiki page above).

1: van der Meij R, Jacobs J, Maris E (2015). *Uncovering phase-coupled oscillatory networks in electrophysiological data.* Human Brain Mapping

2: FIXME van der Meij R, Voytek B (submitted). *Uncovering neuronal networks defined by consistent between-neuron spike timing from neuronal spike recordings.*

Note that extracting spike timing networks can take several days, but with distributed computing (see below) can easily be sped up ~25-50x (i.e., roughly the number of random initializations). See *Box 2* below.

An example dataset

In this tutorial we will extract spike timing networks from a neural spiking recording from rat hippocampus and prefrontal cortex from a public data repository, http://CRCNS.org, recorded and kindly submitted by the Buzsáki lab (http://buzsakilab.com). The dataset we will work with can be found under the label *pfc-2*. The specific recording from this dataset is called *GG.069*. In short, rats performed an odor-based delayed matching-to-sample task, requiring them to run through either the left or right arm of a maze to obtain its reward. For more details regarding the dataset and tasks, see the documentation hosted on http://CRCNS.org.

2. Background: what are extracted spike timing networks?

What are decompositions?

Before introducing spike timing networks, which are extracted using a decomposition technique, it is useful to illustrate what a decomposition is. Decomposition techniques in the context of this tutorial are also known as dimensionality reduction, source separation, or feature extraction techniques. In the broadest sense, decompositions describe 'structure' in the data in a more parsimonious way. Consider the following toy example (see Figure 1). In part of an EEG experiment we obtain measurements from multiple EEG electrodes over the course of a few seconds. The numbers representing these recordings are arranged in a 2-way matrix (Fig 1A). Two distinct oscillations are present in this recording, a slow one and a fast one. Whereas the slow oscillation is strongest at the electrodes at the top, the fast oscillation is strongest at the middle electrodes. A decomposition technique uses the variability over the two dimensions (space and time), to separate these oscillations into what can be called *components*. Each of the components describes one of the oscillations, by two 1-dimensional *loading vectors* (Fig 1B). The spatial loading vector quantifies how strongly each electrode reflects, or loads, the time-course (the spatial pattern), and the temporal loading vector quantifies how strongly each time-point reflects the spatial pattern. Importantly, because the components describe the spatial and temporal patterns of the oscillations separately, they are easier to interpret and analyze than the original matrix. The components are also a parsimonious description of the original matrix, because they describe the same patterns with fewer numbers.

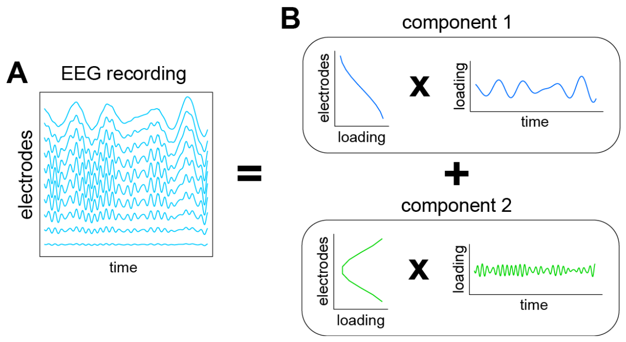

Figure 1: illustration of a decomposition technique

What are extracted spike timing networks?

Spike timing networks are networks identified by consistent between-neuron spike times. They can be extracted from neural spike recordings, in the form of spike trains from multiple epochs, using a decomposition technique denoted as SPACE. For the purpose of this tutorial, the epochs will be trials of an experiment, but these epochs can be any kind of meaningful temporal segmentation of a recording. Networks are not extracted from the raw spike trains, but from the cross spectra of frequency-transformed spike trains. The cross spectra describe between-neuron spike timing consistency by phase differences over frequencies. Each trial will have a neuron-by-neuron cross spectrum for every frequency (for more details see below).

SPACE describes the systematic variability of the cross spectra by multiple spike timing networks, each network consisting of three parameter vectors (Fig 2): the *neuron profile* (1-by-Nneuron), the *time profile* (1-by-Nneuron), and the *trial profile* (1-by-Ntrial). The neuron profile describes how strongly each neuron is part of the network, by a single number per neuron. In the example below, there are roughly three neurons with high loadings, the rest are low. As such, this network reflects spike timing consistency between these three neurons. The time profile describes the spike sequence of the network, by a single number per neuron indicating its time delay with respect to the other neurons. Because the neuron profile of this network only strongly involves three neurons, only these coefficients of the time profile are meaningful. The time profile of these three neurons directly reflects the timeline of the sequence of their spikes. The trial profile of a network describes how strongly the network is present in each trial, by a single number per trial. This can, in principle, be used an index of 'network activity', providing a first estimate of spike time consistency differences between conditions. However, as is presented in SPACE reference paper[2], the trial profile can be very sensitive to differences in firing rate between neurons/trials. This also the case for the example dataset, and as such, it will not be treated in depth on this tutorial. For an idea on what can be done with the trial profile, see the other SPACE tutorials. Each spike timing network also has a *frequency profile* (1-by-freq), but for the purpose of this tutorial it is not important. It plays a key role in the extraction of oscillatory networks/components from EEG/MEG/etc, which is the focus of other tutorials. See SPACE reference paper[2] for a longer description, with more examples, on what is meant with spike timing networks and on how to interpret the parameter vectors.

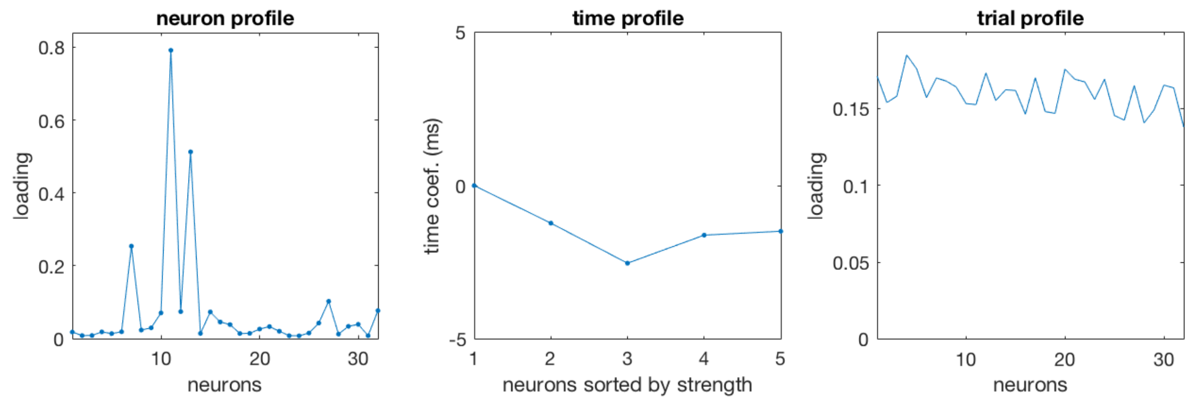

Figure 2: example spike timing network extracted from the example dataset

Spike timing networks can be extracted without strong constraints

Spike timing networks, or components of the cross spectra, are extracted using SPACE, which is inspired by a decomposition technique named PARAFAC/2[3-5]. Common decomposition techniques, like Principal and Independent Component Analysis (PCA and ICA) require constraints to extract components. For example, PCA can only extract components under the constraint that each (loading vector of a) component is orthogonal (uncorrelated) to all the others. In the case of ICA, components need to be statistically independent, which is an even stronger constraint. Spike timing networks can be extracted without constraints that strongly impact their interpretation (for the rationale behind this for PARAFAC, and SPACE by extension, see [6]). In more formal terms, SPACE does not require constraints such as the above to ensure uniqueness of the extracted networks. This is described in more detail in SPACE reference paper[1].

Two SPACE models

SPACE can extract networks according to two models: SPACE-time, and SPACE-FSP (for Frequency-Specific Phase). These two 'sub-SPACEs' extract components that are very similar, but differ in how they describe between-neuron/sensor/electrode phase coupling.

SPACE-time is the model used in this tutorial, and will simply be referred to as SPACE. It describes phase differences over frequencies of the cross spectra, by time delays between neurons. This is possible, because time differences in the time domain translate to phase differences in the frequency domain, linearly increasing with frequency. That is, a 1ms time delay equals 1/20th of a cycle at 50Hz, 1/10th at 100Hz, 1/5th at 200Hz, etc. SPACE-time uses this property to find the time delays between neurons that explains the most variance in the cross spectra. When the cross spectra are not computed from frequency-transformed neural spiking time series, but from frequency-transformed potentials (local field potentials, electrocorticography, EEG, etc), the extracted networks are not spike timing networks, but phase-coupled oscillatory networks. Such networks are the focus of a different tutorial.

SPACE-FSP described the phase differences over frequencies not by a time delay per neuron/electrode/sensor, but by a phase per frequency. It is the focus of another tutorial, which extracts so-called 'rhythmic components' from EEG/MEG recordings.

3. Background: two key issues when extracting spike timing networks

Making sure spike timing networks are extracted accurately

To extract networks, the iterative SPACE algorithm needs to be started randomly multiple times. Some of these random starts result in accurate networks, some result in inaccurate networks. Which random start should we choose? Because SPACE is an (alternating) Least Squares algorithm, we should *always* choose the random start with the highest amount of explained variance of the data. How do we know that there isn't another random starting point, which results in an even higher explained variance? We can never be certain, but by using the following rationale we can be reasonably certain. First, we sort random starts by their explained variance, and inspect whether explained variance has plateaued out. If so, we compare the networks coming from random starts at this plateau. If they are (nearly) identical, we can be reasonably certain that we have found the 'most accurate' random start. All others need to be discarded.

Phrased differently, the SPACE algorithm can converge unto *local minima* of its loss function. To avoid this, we use multiple random starts and determine whether we have reached the *global minimum* using the rationale above.

In this tutorial, we will use software that takes care of the random initialization, and we will determine afterwards whether we have accurately extracted our networks using several statistics. Importantly, the number of random initializations needs to be guessed. Because SPACE is computationally costly, and because this cost increases roughly linearly with the number of initializations, it is important to be conservative in the amount that we choose. As a rule of thumb, the 'most accurate' random start is usually reached with 10-100 initializations.

Determining the number of networks to extract

The number of networks to extract from the data cannot be obtained analytically and needs to be determined empirically, similar to ICA and PARAFAC. This issue is completely separate from the random initializations described above, which holds for every step of what is discussed now. There are many different ways we can determine the number of networks; some are currently implemented. In this tutorial, rather than obtaining an estimate of the true number of networks that exist in the data, which might capture networks that are only active very briefly, we will estimate the number of *reliable* networks. Reliable in this context refers to being consistent over an odd-even split of spikes. In this tutorial we will do this by splitting the recording into two splits, extracting the same number of networks from the full data and from both splits, and assess the similarity of the latter with the former (see below). If their similarity is sufficient, we increase the number of networks to extract. If it is not, we lower the number of networks. We do this until we have found the largest number at which networks are still reliable. The result of this procedure is that we will only extract networks that are strongly present in both splits of the data, which may or may not be desired. In this tutorial we will use software that does the split-reliability determination automatically, and we will judge its success afterwards. Note that for each step, it is necessary to extract accurate spike timing networks, and, as such, each step requires multiple random starts (for each split, at each number of networks to extract, etc).

Other implemented options for determining the number of networks are to increase the number to extract until they no longer increase the explained variance by a certain degree, or to simply extract a certain number of networks, and judge their reliability afterwards. Each method has its advantages and disadvantages. Whatever method is used, it is important to distinguish between networks that are driven by structure in the data (both neural and artefactual), and those that are driven by noise.

(Note: networks that are extracted depend on the other networks (similar to ICA/PARAFAC, dissimilar to PCA without 'rotation'). That is, the first network when extracting two networks is not necessarily the same as the first network when extracting three networks. In practice though, they are often very similar.)

4. Extracting spike timing networks: from raw data to networks

Procedure

We will extract spiking networks from the example dataset using the following steps:

1) Obtain spike trains per trial of the experiment
2) Calculating the input for SPACE, cross spectra (or, rather, their square root)
3) Determine an appropriate neuron-wise normalization of the cross spectra
4) Determine the number of networks to extract, and extract them, using *nd_nwaydecomposition*
5) Determine the appropriateness of the number of networks extracted, and determine whether the amount of random initializations was sufficient

Obtaining spike trains per trial

First, download the *pfc-2* dataset from http://CRCNS.org. Once unpacked, the file *GG.069_Behavior.mat* contains everything we need. We are going to segment the recording into trials, which are laps of the rat running through the maze. Spike times are going to be converted to spike trains, neuron-by-time binary matrices, which will be sparse (for memory usage). Sparse matrices only store their non-zero parts. It is important that the spike trains are created at the maximal sampling rate, in this case 20kHz.

The following code will generate a cell-array of spike trains matrices, one cell per trial (40 trials in total). We base our trial boundaries on the linearized distance of the rat from the start to the end of the maze, for which we will apply two corrections for this dataset. We also create two bookkeeping variables to be used later, for keeping track of the type of each trial and the ID of each neuron.

```
dat = load('GG.069_Behavior.mat');
lindist  = dat.whlrld(:,7);                % linearized rat distance in maze
fsample  = 20000;                          % sampling rate in Hz of recording, see documentation
begtrial = round(dat.SessionNP(:,2) * (fsample./512)); % convert to lindist sampling rate
endtrial = round(dat.SessionNP(:,3) * (fsample./512)); % convert to lindist sampling rate
ntrial   = numel(begtrial);
data     = cell(1,ntrial);
% correct end of trial to last known position
for itrial = 1:ntrial
  currldist = lindist(begtrial(itrial):endtrial(itrial));
  nzind     = find(currldist,1,'last');
  endtrial(itrial) = begtrial(itrial) -1 + nzind;
end
% ensure start/end of trial are closest to beg/end of maze
for itrial = 1:ntrial
  currldist = lindist(begtrial(itrial):endtrial(itrial));
  endtrial(itrial) = begtrial(itrial) -1 + find(currldist>=currldist(end),1);
  begtrial(itrial) = begtrial(itrial) -1 + find(currldist>currldist(1),1);
end
begtrial = begtrial .* 512; % convert back to timestamp sampling rate (20kHz)
endtrial = endtrial .* 512; % convert back to timestamp sampling rate (20kHz)
% create spike trains
for itrial = 1:ntrial
  % fetch timestamps and neuren IDs for current trial
  ind = dat.spiket >= begtrial(itrial) & dat.spiket <= endtrial(itrial);
  currts = dat.spiket(ind);
  currts = currts - begtrial(itrial) + 1; % convert to trial specific samples
  neurid = dat.spikeind(ind);
  % create sparse spike train matrix
  nsample = (endtrial(itrial)-begtrial(itrial))+1;
  data{itrial} = sparse(neurid,currts,ones(size(currts)),dat.numclus,nsample);
end

% bookkeeping variables
trialtype = dat.SessionNP(:,4); % 1 (right) or 2 (left) lap
label     = []; % neuronIDs
for inneuron = 1:dat.numclus
  label{inneuron} = ['neuron' num2str(inneuron)];
end

% select neurons with mean spike rate above 1Hz
trialtime = cellfun(@size,data,repmat({2},[1 ntrial])) ./ fsample;
nspikes   = cellfun(@sum,data,repmat({2},[1 ntrial]),'uniformoutput',0);
nspikes   = cat(2,nspikes{:});
spikerate = mean(bsxfun(@rdivide,nspikes,trialtime),2);
selind = spikerate>1;
for itrial = 1:ntrial
  data{itrial} = data{itrial}(selind,:);
end
label = label(selind);
```

Creating input for SPACE: cross spectra (and their square root)

SPACE finds spike timing networks by describing structure in the cross spectra (spectral 'covariance matrices'), computed from the spike train matrices at different frequencies. The crucial principle here, is that time differences in the time domain translate to phase differences in the frequency domain, linearly increasing with frequency. That is, a 1ms time delay equals 1/20th of a cycle at 50Hz, 1/10th at 100Hz, 1/5th at 200Hz, etc. As such, by calculating cross spectra, we convert the differences in spike times between two spikes of a neuron-pair are converted to phase differences at multiple frequencies, and summed over time. SPACE uses the above property to find the time delays between neurons that explains the most variance in the cross spectra.

To compute the cross spectra, we first convolve the spike train matrices of each trial with complex exponentials (untapered 'wavelets') at different frequencies but of constant length. The result is a complex-valued matrix, per trial, at each frequency of the complex exponentials. We then take the cross product over time for each of these matrices, resulting in a neuron-by-neuron cross spectrum matrix for each trial at each frequency. These cross spectra describe the consistent time delays between spikes of pairs of neurons by the phases of their off-diagonal elements (the cross spectra 'interaction terms'). The magnitude of these off-diagonal elements are a function of the amount of spikes at a consistent time delay of each neuron pair. Importantly, these magnitudes also depend on the firing rates of the involved neurons. These firing rates are reflected by the diagonals of the cross spectra (cross spectral 'power').

The time domain length and the frequency of the complex exponentials used determine how sensitive the cross spectra are to consistent vs non-consistent time delays. The time domain length of the complex exponentials determines which between-spike time delays can contribute to the cross spectra, and it should be chosen based on the expected range of time delays. The optimal length for an expected range is a trade-off. For a longer explanation, see SPACE reference paper[2]. We will use complex exponentials of 20ms, to be sensitive to time delays between 0 and 10ms. Once a time domain length is chosen, the frequencies of the complex exponentials follow. In our case, for 20ms, the first frequency will be 1/0.020 = 50Hz (one over the length in seconds). The other frequencies are integer multiples of this frequency, and we will 20 frequencies in total. Thus, our frequencies are 50Hz to 1000Hz in steps of 50Hz. Using frequencies based on this rule is crucial, as it will greatly determine how sensitive the cross spectra are to non-consistent time delays. For a longer explanation, see SPACE reference paper[2].

The code below will compute the cross spectra from the spike trains matrices we obtained above. Even though *only the cross spectra are used in estimating the parameters* of the spike timing networks, for purposes irrelevant to this tutorial, SPACE requires the square root of the cross spectra as input. We will still talk about the cross spectra as the input to SPACE whenever possible, because it is the most convenient when thinking about spike timing networks.

Three practical things to note. (1) The square root of the cross spectra are the matrices that are obtained after convolving the spike train matrices with the complex exponentials, having dimensions neuron-by-time. These, however, tend to be very big. Since only the cross products of the matrices are used, it is wiser to replace them with a matrix that has the same cross product, but that is much smaller (which we do below). This is explained in *Box 1* in more detail. (2) Below we use a function (*sconv2.m*) for convolving spare matrices that can be obtained from the MATLAB File Exchange (*https://www.mathworks.com/matlabcentral/fileexchange/41310*). *(3)* In the other tutorials, the time dimension of the neuron-by-time matrices is referred to as a taper dimension, reflecting the fact that the time points are the result of a time window of data multiplied with a wavelet/complex exponential/other kind of taper. For simplicity, the dimension is still called 'taper' even after replacing the neuron-by-time matrices with a smaller matrix that has the same cross product.

The following code will use the spike trains obtained above to generate a neuron-by-freq-by-trial-by-time' 4-way array of the square root of the cross spectra. The resulting array of square roots we will also call the Fourier array (reminiscent of the fact that the neuron-by-time matrices can be considered 'fake' Fourier coefficients).

```
% one variable taken from above
data % cell-array of spike train matrix per trial
% set cross spectra choices
timwin  = 0.020; % complex exponential length in seconds
freq    = 50:50:1000; % frequency in Hz
fsample = 20000; % sampling rate of the spike trains in Hz
% set n's
ntrial  = numel(data);
nneuron = size(data{1},1);
nfreq   = numel(freq);
% pre-allocate
fourier = NaN(nneuron,nfreq,ntrial,nneuron,'single');
fourier = complex(fourier,fourier);
% convolve with complex exponential, obtain cross spectrum, obtain square root
for itrial  = 1:ntrial
  for ifreq = 1:nfreq
    disp(['working on trial #' num2str(itrial) ' @' num2str(freq(ifreq)) 'Hz'])
    % construct complex exponential
    timeind = (-(round(timwin .* fsample)-1)/2 : (round(timwin .* fsample)-1)/2) ./ fsample;
    wlt     = exp(1i*timeind*2*pi*freq(ifreq));
    % get spectrum via sparse convolution
    spectrum = sconv2(data{itrial},wlt,'same');
    % obtain cross spectrum, csd
    csd = full(spectrum*spectrum');
    % normalize csd by number of time-points (for variable length epochs)
    csd = csd ./ (size(data{itrial},2) ./ fsample);
    % obtain square root of cross spectrum
    [V L] = eig(csd);
    L     = diag(L);
    tol   = max(size(csd))*eps(max(L));
    zeroL = L<tol;
    eigweigth = V(:,~zeroL)*diag(sqrt(L(~zeroL)));
    % save in fourier
    currm = size(eigweigth,2);
    fourier(:,ifreq,itrial,1:currm) = eigweigth;
  end
end
```

The resulting Fourier array is single precision to further reduce memory usage. SPACE will convert sections of the array to double precision where needed.

Box 1: SPACE only uses the cross spectra: an opportunity for memory usage reduction

Though SPACE takes as input a 4-way array containing frequency- and trial-specific neuron-by-time (or taper) matrices, it only uses their neuron-by-neuron cross products. SPACE inherits this from PARAFAC2. This leads to an important trick, which is carried out by the relevant functions, but is also of use for the end-user that computes and stores the square root of the cross spectra (also called Fourier coefficients here). This is because the neuron-by-time matrices can become prohibitively large. Because SPACE only uses the neuron-by-neuron cross products of the cross spectra, we can replace the neuron-by-time matrices (F) by a different matrix, as long as the cross product of this matrix is equal to the cross product of the original (FF*; where * is the complex conjugate transpose).

We can do this using the Eigendecomposition of the neuron-by-neuron matrix FF*. The Eigendecomposition of a symmetric matrix FF* can be written as follows:

FF* = VDV*

Where V is a matrix containing the Eigenvectors of FF* as columns and D is a diagonal matrix

containing the corresponding (real-valued) Eigenvalues.
From this it follows that:
$FF^* = VDV^* = VD^{0.5}D^{0.5}V^* = (VD^{0.5})(D^{0.5}V)^*$
As such, the neuron-by-time matrix F can be replaced by the product of the Eigenvectors of the Eigendecomposition of FF* and the square root of its eigenvalues: $VD^{0.5}$. Because $VD^{0.5}$ will never bigger than neurons-by-neurons, memory usage can be reduced (as well as computation time).

Normalizing cross spectra for extracting spike timing networks:

Now that we have computed our cross spectra, and transformed them the Fourier array above, we can estimate the number of spike timing networks to extracted. Before we estimate the number of networks to extract however, we need to find an optimal neuron-wise normalization (see SPACE reference paper[2] for more detail). This is necessary, because the neuron profiles of spike timing networks describe both the off-diagonal elements of each cross spectrum (i.e. between-neuron spike pairs), and their diagonal elements, i.e. power (reflecting the total number of spikes of neurons). Typically, the firing rates of neurons can be very different, resulting in large differences in cross spectrum power. Additionally, the power of each cross spectrum is typically much larger than its off-diagonal elements, because only a small subset of spikes of a neuron have a consistent temporal relationship with those of other neurons. Combined, this can lead to neuron profiles that are more driven by power (i.e. firing rates), than by spike time consistency, and thus mostly consist of one neuron (with a high loading). To increase sensitivity to consistent spike timing, we can normalize power differences between neurons to decrease their impact. This comes at a downside however, as it has an impact on the estimation of the number of networks to extract. When normalizing the power too strongly (such as transforming the cross spectra into coherency matrices), the order in which networks are extracted becomes more variable over random initializations, causing the split reliability estimation procedure to stop prematurely when extracting less than the total number of networks present in the data. This is likely a consequence of reducing the difference in explained variance between networks (which is due to certain noise sensitivities, see SPACE reference paper[2]).

The consequence of the above, is that we aim to find a normalization that, (1) is strong enough to lead to networks that are driven by spike time consistency instead of power, and (2) is weak enough for the split reliability estimation to result in a useful number of networks. We do this by progressively increasing the normalization strength of the cross spectra until we see evidence of success: neuron profiles having a strong weighting for more than one neuron. Then, we estimate the number of networks using the split-reliability procedure. If the procedure extract very few reliable networks, we can reduce the normalization strength until we extract more.

We do the above in practice below. We will normalize the cross spectra, extracted a certain number of networks, and inspect the results. The normalization we will apply will be one in which the total power of all cross spectra, summed over trials and frequencies, is equal to its Nth-root. N is the factor we increase to increase normalization strength. Below we will only inspect the networks several normalizations.

The code below normalizes the cross spectra such that their total power is equal to its Nth-root, in this example, it's 8th root (square root of square root of square root). The normalization is performed on the Fourier array without computing cross spectra, as this is more convenient than re-computing the cross spectra and its square root.

```
N = 8; % an example N
[nneuron,nfreq,ntrial,ntime] = size(fourier);
% compute sum of power
power = zeros(nneuron,1);
for itrial = 1:ntrial
  currfour = double(squeeze(fourier(:,:,itrial,:)));
  power = power + nansum(nansum(abs(currfour).^2,3),2);
end
% compute scaling such that power is its kth root
scaling = (power .^ (1/N)) ./ power;
for itrial = 1:ntrial
  for ifreq = 1:nfreq
    currfour  = double(squeeze(fourier(:,ifreq,itrial,:)));
    nonnanind = ~isnan(currfour(1,:));
    currfour  = currfour(:,nonnanind);
    currfour  = bsxfun(@times,currfour,sqrt(scaling));
    % save in fourier
    fourier(:,ifreq,itrial,nonnanind) = single(currfour);
  end
end
```

The above we do for N = 1 (no normalization), N = 8, N = 16, N = 32, and N = 64. These levels are chosen to illustrate the progressive effect of normalization strength.

For extracting networks below, we will use 5 as the number to extract, at 25 random initializations, which is a guess. We won't know how many we should extract until we go through the split-reliability procedure. However, for the purpose of finding the proper normalization it is fine to guess. The reason is simple, we only want to see whether we can extract networks that are useful to us: i.e. reflect spike time consistency by having a neuron profile loading more than one neuron strongly. Even if the networks are not split-reliable, they will provide an idea on the effect of the normalization. Ideally, we would do a split-reliability approach for each normalization, but this is can be time intensive, and will, from experience, rarely result in a different decision.

We will extract networks using the function *nd_nwaydecomposition*. This function handles the multiple random initializations of the SPACE algorithm, and contains algorithms for determining the number of networks to extract (see *Background*). To extract networks from the cross spectra, we first put the above Fourier array in a MATLAB structure with additional information required for network extraction. We'll also use the bookkeeping variables here so that they can be passed on into the output. The 'dimord' field here contains labels of the individual dimensions in the 'fourier' field. It is important when using a Fourier array computed by FieldTrip functions and can be ignored here (but needs to be added nonetheless).

We do the following for normalization strength, N = 1, N = 8, N = 16, N = 32, and N = 64.

```
% create input structure for nd_nwaydecomposition
fourierdata = [];
fourierdata.fourier   = fourier;               % square root of cross spectra
fourierdata.freq      = freq;                   % frequencies in Hz
fourierdata.label     = label;                  % neuron IDs
fourierdata.dimord    = 'chan_freq_epoch_tap'; % req. for SPACE due to other uses
fourierdata.trialinfo = trialtype;              % left/right trial codes
fourierdata.cfg       = [];

% extract spike timing networks
cfg = [];
cfg.model              = 'spacetime';  % the model for extracting spike timing networks
cfg.datparam           = 'fourier';    % field containing square root of the cross spectra
cfg.Dmode              = 'identity';   % necessary, see background/SPACE ref papers
cfg.ncomp              = 5;            % number of networks to extract
cfg.numiter            = 1000;         % max number of iterations
cfg.convcrit           = 1e-6;         % stop criterion of algorithm: minimum relative
                                       %    difference in fit between iterations
cfg.randstart          = 25;           % number of random initializations
nwaycomp = nd_nwaydecomposition(cfg,fourierdata);
```

This typically take less than 2 hours on a single core machine, per normalization. See *Box 2* for how to speed this up using distributed computing.

The output structure *nwaycomp*

First, a description of the output MATLAB structure *nwaycomp*, which contains the following:

```
nwaycomp  =
         label: {1x32 cell}                   % neuron IDs we gave
        dimord: 'A_B_C_S_D'                   % see below
          comp: {{1x5 cell}  {1x5 cell} ...} % each cell contains a network, a 1x5 cell-array
        expvar: 13.3700                       % explained variance of the Fourier array
     tuckcongr: [6x1 double]                  % N/A
       scaling: [33.0676 33.0476       ...]  % relative network ‘strengths’
    randomstat: [1x1 struct]                  % details of random start procedure
  splitrelstat: [1x1 struct]                  % details of split-reliability procedure (if used)
     trialinfo: [40x1 double]                 % left/right trial codes we passed on
          freq: [50:50:1000]                  % frequencies used
           cfg: [1x1 struct]                  % configuration options used
```

The function of some of these fields will become clear in the sections that follow. The two most important fields in the output are the *comp* field and the *dimord* field. The *comp* field is a cell-array, which in each cell contains a 1x5 cell-array containing the network-specific parameters. Which parameters are in which cell is described by the *dimord* (dimension order) field. This field is a crucial field in the output of most FieldTrip functions, and describes the content of each dimension in the most important output field (*comp*, in this case). A, B, C, S, and D describe the neuron profile (A), the frequency profile (B), the trial profile (C), the time profile (S), and the matrices D, respectively. The field naming, the lettered labeling, and their order, follows the application-agnostic form presented in the SPACE reference papers[1,2]. The D cells can be ignored. Another field of interest is the *scaling* field. This contains a number, in the units of the input data, which can be used to judge the strength of a network relative to the other networks. The networks in the *comp* field are ordered by their value in the *scaling* field.

To determine whether our normalization has the desired effect, we visualize the neuron profiles given by N = 1, N = 8, N = 16, N = 32, and N = 64.

```
% plot extracted neuron profiles for all Ns
% first, we assume the Fourier arrays for all Ns are gathered in a cell-array
nwaycompallN{1} % N = 1
nwaycompallN{2} % N = 8
nwaycompallN{3} % N = 16
nwaycompallN{4} % N = 32
nwaycompallN{5} % N = 64
figure
N = [1 8 16 32 64];
nneuron = numel(nwaycompallN{1}.label);
for iN = 1:5
  % plot neuron profiles (contained in the 1st cell)
  subplot(1,5,iN)
  hold on
  for inetw = 1:5
    plot(nwaycompallN{iN}.comp{inetw}{1},'marker','.');
  end
  ylim = get(gca,'ylim');
  set(gca,'ylim',[0 ylim(2)*1.05],'xlim',[1 nneuron])
  xlabel('neurons')
  ylabel('loading')
  title(['neuron profiles N = ' num2str(N(iN))])
  axis square
end
```

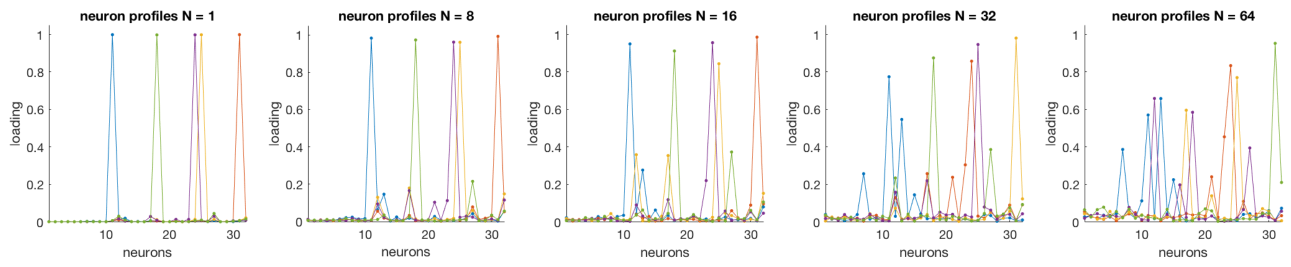

Figure 3: neuron profiles at different normalization strengths

Viewing the neuron profiles, we see that without normalization (N=1), almost all 'networks' have only one neuron that has a strong loading. I.e., these networks will not be useful, as it is unlikely that the time profile of the neurons with low neuron profile loadings are reliable, and, as such, these networks likely only reflect structure in the cross spectra resulting from power (i.e., firing rates). We see that, the stronger the normalization, the more each network has strong loadings for several neurons. (*Note: the absolute value of the loadings should not be interpreted, but the within-network between-neuron loading ratios can be.*) With a normalization of N = 32, for this dataset, neuron profiles start to have a loading ratio above 1:2 for the strongest two neurons. This is a decent rule of thumb to use for selecting a normalization to proceed with, and estimate the number of reliable networks that can be extracted using it. If we go a bit further, for N = 64, the strongest two neurons of several networks are even very close together. Ideally, this is what we achieve, under the assumption that there exist neurons in the data that have between-neuron spike time consistency.

How to move forward from this? It is easy, pick one of the above levels, perform a split-reliability procedure, and increase the normalization if a number of networks is extracted that is useful. If a lot of networks are extracted, we can increase the normalization strength, to optimize sensitivity to spike time consistency. If too few reliable networks (or none at all) are extracted, we can decrease the normalization strength. This comes at the cost of the neuron profiles, time profiles, and trial profiles to be more sensitive to noise (because firing rate differences will have a bigger influence). In our case, if we try a split-reliability procedure on N = 64, we will extract 0 reliable networks. If we use N = 32, as we will see below, we will extract 4 networks. *(Note: we can, of course, try N's which are not powers of 2, to have smaller differences between normalization strengths.)*

Extracting spike timing networks

We will now use a split-reliability procedure for estimating the number of reliable networks in the data. We do this by setting *cfg.ncompest = 'splitrel'*. For other approaches, see the function documentation. We determine the split-reliability by independently extracting networks from the full data and from two splits of the data, one split containing only the odd numbered spikes, and one split containing only the even numbered spikes. One can also use other/more splits (see the function documentation). Reliability is then quantified using a coefficient akin to the Tuckers congruence coefficient[6]. This coefficient is computed between networks extracted from the full data and networks extracted from each split, per network, per parameter, and ranges from 0 to 1. A coefficient of 1 indicates that the parameter of two compared networks is identical. Using *cfg.ncompestsrcritval* we set how conservative we want to be in the procedure. It is the minimum coefficient at which networks from the splits are considered similar to the networks from the full data. We set it at 0.7, which can be loosely interpreted as requiring networks to be 70% similar. Using the other *cfg.ncompest** options we can determine other aspects of the procedure.

Running the split-reliability procedure can take quite some time depending on the number of networks, the number of neurons, and the number of trials. For the current dataset, it should take about 2-8 hours on a single core of a CPU, depending on the hardware. Using a distributed computing setup, this can be reduced by a factor of roughly the number of initializations (see *Box 2*).

First, we create the two splits, one with the odd numbered spikes, the other with the even numbered spikes. Then, we calculate the 4-way square root of cross spectra, identically to the above, which is not shown below.

```matlab
% obtain splits
nneuron  = size(data{1},1); % data is the cell-array of spike trains we created above
ntrial   = numel(data);
dataodd  = cell(1,ntrial);
dataeven = cell(1,ntrial);
for itrial = 1:ntrial
  % get spike time stamps/indices
  nsample = size(data{itrial},2);
  [neurid, ts] = ind2sub(size(data{itrial}),find(data{itrial}));
  oddind  = 1:2:numel(ts);
  evenind = 2:2:numel(ts);
  % create new sparse spike train matrices
  dataodd{itrial}  = sparse(neurid(oddind), ts(oddind), ones(size(oddind)), nneuron,nsample);
  dataeven{itrial} = sparse(neurid(evenind),ts(evenind),ones(size(evenind)),nneuron,nsample);
end

%%% Calculate Fourier array %%%
```

Having obtained the Fourier arrays for the two splits, we first combine the them into a new Fourierdata structure. Then, we perform another SPACE decomposition, but this time with the split reliability approach. To be less conservative, the similarity coefficients mentioned above are averaged over splits, before comparison to the criterion of 0.7. We will also ignore the trial profiles, because we wish to find networks even if they occur on few trials, and the frequency profiles, because they contain no useful information for spike timing networks. We do this by setting the criterion as a 1x5 vector, with each element being the parameter-specific criterion, following the order A-B-C-S-D mentioned above (D we will also ignore). We will also use a higher number of random initializations, to be a little more certain that we have found our global minimum.

```
% create input structure for nd_nwaydecomposition
fourierdata = [];
fourierdata.fourier   = fourier;               % square root of cross spectra of all spikes
fourierdata.fouriersplit{1} = fourierodd;      % square root of cross spectra of odd spikes
fourierdata.fouriersplit{2} = fouriereven;     % square root of cross spectra of even spikes
fourierdata.freq      = freq;                  % frequencies in Hz
fourierdata.label     = label;                 % neuron IDs
fourierdata.dimord    = 'chan_freq_epoch_tap'; % req. for SPACE due to other uses
fourierdata.trialinfo = trialtype;             % left/right trial codes
fourierdata.cfg       = [];                    % req. for SPACE due to other uses

% extract spike timing networks
cfg = [];
cfg.model                 = 'spacetime';         % the model for extracting spike timing networks
cfg.datparam              = 'fourier';           % field containing Fourier of full data
cfg.ncompestsrdatparam    = 'fouriersplit';      % field containing Fourier of splits
cfg.Dmode                 = 'identity';          % necessary, see background/SPACE ref papers
cfg.ncompest              = 'splitrel';          % estimate number of networks using splits
cfg.ncompestsrcritjudge = 'meanoversplitscons'; % take mean of similarity coef. over splits
cfg.ncompeststart         = 10;                  % start from 10
cfg.ncompeststep          = 5;                   % increase number in steps of 5
cfg.ncompestend           = 50;                  % extract no more than 50
cfg.ncompestsrcritval     = [.7 0 0 .7 0];       % split-reliability criterion for each parameter
cfg.numiter               = 1000;                % max number of iteration
cfg.convcrit              = 1e-6;                % stop criterion of algorithm: minimum relative
                                                 %    difference in fit between iterations
cfg.randstart             = 50;                  % number of random initializations
cfg.ncompestrandstart     = 50;                  % number of random init. for split-rel. proc.
nwaycomp = nd_nwaydecomposition(cfg,fourierdata);
```

Box 2: using a distributed computing system to run random initializations in parallel

To greatly speedup the extraction of spike timing networks it is possible to run the random initializations in parallel using a distributed computing system. Currently, two systems are supported, MATLABs Parallel Distributing Toolbox and Torque. The support for Torque depends on the FieldTrip *qsub* module (which needs to be on the MATLAB path). Other systems that are supported in *qsub* are implicitly supported as well.

Distributed computation of random initializations is specified with the following options:

```
cfg.distcomp.system          = string, distributed computing system to use, 'torque' or
                              'matlabpct' ('torque' requires the qsub FieldTrip module on
                               path, 'matlabpct' implementation is via parfor)
cfg.distcomp.timreq          = scalar, (torque only) maximum time requirement in seconds of
                               a random start (default = 60*60*24*1 (1 days))
cfg.distcomp.memreq          = scalar, (torque only) maximum memory requirement in bytes of
                               a random start (default is computed)
cfg.distcomp.inputsaveprefix = string, (torque only) path/filename prefix for temporarily
```

```
                               saving input data with a random name (default, saving is
                               determined by the queue system)
cfg.distcomp.matlabcmd       = string, (torque only) command to execute matlab (e.g.
                              '/usr/local/MATLAB/R2012b/bin/matlab') (default = 'matlab')
cfg.distcomp.torquequeue     = string, (torque only) name of Torque queue to submit to
                               (default = 'batch')
cfg.distcomp.mpctpoolsize    = scalar, (matlabpct only) number of workers to use (default is
                               determined by matlab)
cfg.distcomp.mpctcluster     = Cluster object, (matlabpct only) Cluster object specifying
                               PCT Cluster profile/parameters, see matlab help PARCLUSTER
```

Did we use a sufficient number of random initializations to accurately extract networks?

Before we interpret our networks, we need to determine whether we should have used more random initializations. We do this by (1) judging whether the explained variance of random initializations has plateaued out, and (2) whether the random initializations at this plateau resulted in the same networks (see *Background*). Whether the *number* of networks is appropriate we will investigate in the next section.

We will now plot the information we need to make our decision in most cases. The necessary information is contained in the *nwaycomp.randomstat* field. This field is a MATLAB structure that contains details from the random start procedure. For those fields containing information per random initialization, the initializations are sorted by explained variance. We will use *randomstat.expvar*, containing the explained variance of each random initialization, *randomstat.congrall*, containing the congruence between initializations per network, per parameter, and *randomstat.congrglobmin*, containing the congruence between those initializations no more than 0.1% (absolute) away from the one with the highest explained variance. The *randomstat.globmininit* field describes which initializations these are. Congruence is computed using a coefficient similar to the split-reliability coefficient, going from 0 to 1 (perfectly similar). If the above does not provide sufficient detail for judging similarity between random initializations, *randomstat.congrcumul* contains the cumulative congruence between initializations of an increasing set of initializations (the first one, the first two, the first three, etc). This is field is, in fact, the one we will use instead of *randomstat.globmininit*, as for this dataset it turns out that the crude absolute criterion (0.1%) included nearly all random initializations.

```
% plot explained variance in first panel
subplot(1,3,1)
plot(1:50,nwaycomp.randomstat.expvar,'d-')
axis([1 50 0 nwaycomp.randomstat.expvar(1)+5])
xlabel('random start #'); ylabel('%')
title('explained variance of each initialization')
% plot congruence for the first 5 initialzations
subplot(1,3,2)
imagesc(squeeze(nwaycomp.randomstat.congrcumul(5,:,:)))
caxis([.7 1]); axis xy; axis square; ylabel('network number'); colorbar
set(gca,'ytick',1:4,'xtick',1:5,'xticklabel',{'A','B','C','S','D'})
title('congruence between first 5 initializations')
% plot congruence for all initialzations
```

```
subplot(1,3,3)
imagesc(nwaycomp.randomstat.congrall)
caxis([0 1]); axis xy; axis square; ylabel('network number'); colorbar
set(gca,'ytick',1:4,'xtick',1:5,'xticklabel',{'A','B','C','S','D'})
title('congruence between all initializations')
```

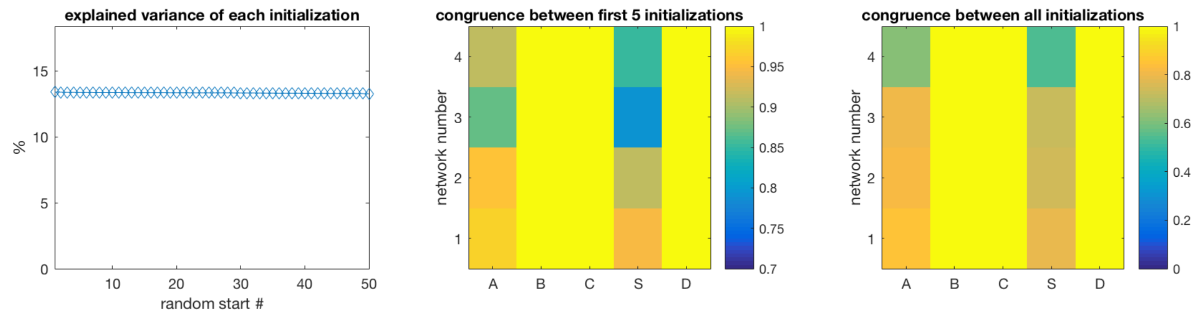

Figure 4: details of the random initialization procedure. Note the different color scales on the two rightmost panels.

The first panel of the above figure shows the explained variance of each initialization. At this zoom-level, explained variance appears to have plateaued out, and the 'most accurate' random start appears to have been obtained, and covers all 50 initializations. The second and third panel shows the Tuckers congruence coefficients (0 <-> 1) for each parameter, for each network. The second panel for only those initializations at the first 5 initializations, the third panel for all initializations.

In the second panel, for nearly all networks and all parameters the coefficients are around .8+, and only one is around .75. As such, we can all the 4 networks were very similar over the first 5 initializations (the third panel shows this is not the case for all initializations). Combined with the explained variance plateauing out, we can be confident we have reached the 'most accurate' random start. If we want additional certainty, we can increase the number of random initializations to say, 100, and rerun the procedure.

Did the network number estimation procedure succeed?

The next step is to determine whether we extracted an appropriate number of networks, Nnetwork. We will use the field *nwaycomp.splitrelstat*, containing details of the split-reliability procedure, similar to *nwaycomp.randomstat*. Of its fields, we first look at *splitrelstat.stopreason*, which indicates why the procedure stopped. If this says *'split-reliability criterion'*, which it does in this case, we can continue. This field can also mention whether the procedure failed by reaching the maximum Nnetwork we set in *cfg.ncompestend*, or whether none of the networks reached the split-reliability criterion ($N_{reliable}$<1; which was the case for normalizations at the 64th root).

Now that we know the procedure stopped because no additional reliable networks could be extracted, we look into *splitrelstat.splitrelsucc* and *splitrelstat.splitrelfail*. These contain the split-reliability coefficients for the Nnetwork at which all networks were reliable, and for the next Nnetwork, at which at least one parameter set of one network was unreliable.

```
% first plot for the Nnetwork that succeeded
subplot(1,2,1)
imagesc(nwaycomp.splitrelstat.splitrelcsucc)
caxis([0 1]); axis xy; axis square; ylabel('network number'); colorbar
set(gca,'ytick',1:4,'xtick',1:5,'xticklabel',{'A','B','C','S','D'})
title('split-reliability coeff. at N = success')
% then plot for the Nnetwork that failed
subplot(1,2,2)
imagesc(nwaycomp.splitrelstat.splitrelcfail)
caxis([0 1]); axis xy; axis square; ylabel('network number'); colorbar
set(gca,'ytick',1:5,'xtick',1:5,'xticklabel',{'A','B','C','S','D'})
title('split-reliability coeff. at N = fail')
```

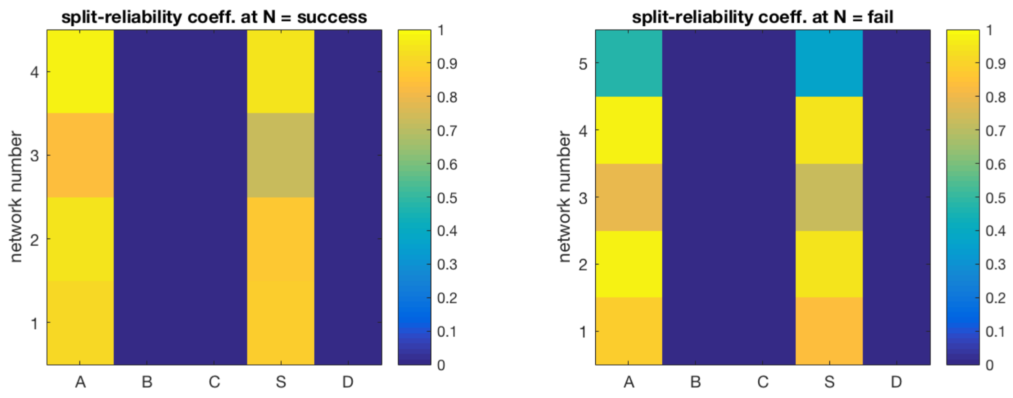

Figure 5: details of the split-reliability procedure

In this figure the coefficients for the frequency profiles (B), trial profiles (C), and matrix D, are set to NaN, because we chose to ignore them via *cfg.ncompestsrcritval*.

In the first panel, visualizing the split-reliability coefficients at the highest successful Nnetwork (4), we can observe that the split-reliability coefficients for the neuron and time profiles are above 0.7 (as expected given our settings). As such, the networks extracted from the full data match those extracted from only the even numbered and the odd numbered spikes. The second panel shows the coefficients at the Nnetwork that failed (5). When we compare the first panel to the second, we see little difference for the first 4 networks, but the 5th network has split reliability coefficients around .4. As such, it did not pass our criterion, and can be assumed to be not split reliable.

The field *splitrelstat* contains additional fields in case we wish to be more rigorous. For example, in the field *splitrelstat.randomstatsucc/fail*, we can find the *randomstat* fields for both splits of the data for the Nnetwork at which the procedure failed/succeeded. Another example is the *splitrelstat.allcompsrc* field, which contains the split-reliability coefficients for all Nnetwork's at which it was computed.

5. The extracted spike timing networks

Now that we are confident that we extracted reliable spike timing networks, it is time to visualize them.

```
% plot spike timing networks
nnetwork = numel(nwaycomp.comp);
nneuron  = numel(nwaycomp.label);
figure('numbertitle','off')
for inetwork = 1:nnetwork
  % plot neuron profile (contained in the 1st cell)
  subplot(3,nnetwork,inetwork + (nnetwork*0))
  plot(nwaycomp.comp{inetwork}{1},'marker','.');
  ylim = get(gca,'ylim');
  set(gca,'ylim',[0 ylim(2)*1.05],'xlim',[1 nneuron])
  xlabel('neurons')
  ylabel('loading')
  title(['neuron profile network ' num2str(inetwork)])
  axis square

  % plot time profile (4th cell)
  subplot(3,nnetwork,inetwork + (nnetwork*1))
  % set t=0 arbitrarily to strongest neuron
  [dum ind] = sort(nwaycomp.comp{inetwork}{1},'descend');
  tp = nwaycomp.comp{inetwork}{4}(ind(1:5))*1000;
  plot(tp-tp(1),'marker','.');
  set(gca,'ylim',[-5 5],'xlim',[1 5])
  xlabel('neurons sorted by strength')
  ylabel('time coef. (ms)')
  title(['time profile network ' num2str(inetwork)])
  axis square

  % plot trial profile (3rd cell)
  subplot(3,nnetwork,inetwork + (nnetwork*2))
  plot(nwaycomp.comp{inetwork}{3});
  ylim = get(gca,'ylim');
  set(gca,'ylim',[0 ylim(2)*1.05],'xlim',[1 nneuron])
  xlabel('neurons')
  ylabel('loading')
  title(['trial profile network ' num2str(inetwork)])
  axis square
end
```

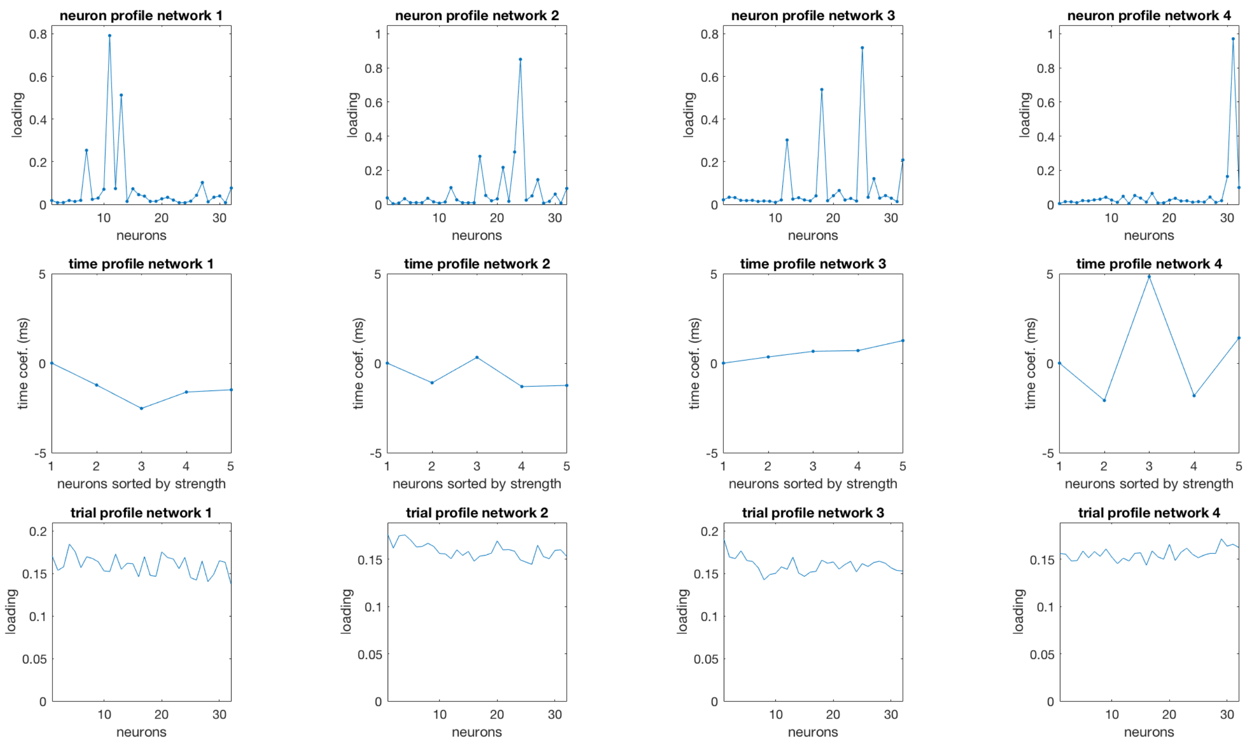

Figure 6: all extracted spike timing networks

What does these networks reflect? If we look at the first network, we see 3 neurons with high loadings. Let's call them neurons 1, 2, and 3, in their order of loading strength. The time profile is viewed from the perspective of the strongest neuron in the neuron profile. Given their time coefficients, the likely spike sequence is in this case 3-2-1 (neuron 3 has the lowest time coefficient, then neuron 2, followed by neuron 1). The precise time line of the sequence is given by the time coefficients themselves. From the perspective of the first neuron in the sequence (neuron 3), this is 0m-1.31ms-2.53ms. When neuron 3 spikes, sometimes neuron 2 spikes 1.31ms (on average) later, and sometimes neuron 1 spikes 1.22ms (on average) after that (2.53 - 1.31ms = 1.22ms). The 'on average' is crucial here: the time line reflects an aggregate time line, aggregated over the entire recording. Timing of individual sequences can vary, but on average they follow the time line above. The aggregate nature of the spike sequence also means that the sequence can piecewise present in some (or all) trials, e.g. consisting of only neurons 1 and 2, or 1 and 3, or 2 and 3.

The neuron profiles and time profiles of the other networks can be interpreted similar the network 1. Network 4 likely only reflects a 'network' consisting of two neurons.

(*Note: the absolute value of the neuron profile loadings should not be interpreted, but the within-network between-neuron loading ratios can be. The time profile has a similar constraint: the absolute value of the time coefficients are not meaningful, only the time coefficients <u>relative</u> to those of the other neurons in the same networks. For more details, see SPACE reference paper*[2].)

The trial profiles are more difficult to interpret, and less useful for this specific example dataset (see SPACE reference paper[2], and the tutorial for extracting *rhythmic components*).

Using the extracted spike timing networks

The spike timing networks we extracted are a statistical description of the spike timing relationships in the recordings. In principle, they can be used as is, in a descriptive manner. Their most useful application however, is arguably as a first step prior to an in-depth analysis of the spike trains. What the networks provide is a highly precise spike timing template that is extracted without prior information, and that is not dependent on any binning, pattern-selection, or other method to reduce sequence complexity to make template searching feasible. This is especially useful in the case of large-scale recordings, where searching for networks without prior knowledge is less tractable. As such, the extracted spike timing networks provide a very convenient exact (w.r.t. to the average temporal relationships) template to use for investigating spike time sequences. For example, does spike timing consistency vary between experimental conditions? Is spike sequence partiality related to task performance, e.g. full sequences on hit trials, partial sequences on miss trials? Do they occur more often in some conditions than others? Answering these questions requires a further analysis dependent on other techniques, such as template searching, which is beyond the scope of this tutorial.

References

1. van der Meij, R., Jacobs, J. & Maris, E. Uncovering phase-coupled oscillatory networks in electrophysiological data. *Hum Brain Mapp* **36**, 2655-2680 (2015).

2. FIXME van der Meij R, Voytek B (submitted). *Uncovering neuronal networks defined by consistent between-neuron spike timing from neuronal spike recordings*

3. Harshman, R.A. Foundations of the PARAFAC procedure: model and conditions for an 'explanatory' multi-mode factor analysis. *UCLA Working Papers in Phonetics* **16**, 1-84 (1970).

4. Kiers, H.A.L., Ten Berge, J.M.F. & Bro, R. PARAFAC2 - Part I. A direct fitting algorithm for the PARAFAC2 model. *Journal of Chemometrics* **13**, 275-294 (1999).

5. Carrol, J.D. & Chang, J. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika* **35** (1970).

6. Bro, R. PARAFAC. Tutorial and applications. *Chemometrics Intell. Lab. Syst.* **38**, 149-171 (1997).