WORK IN PROGRESS

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 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 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. Though higher order statistical analyses of spike timing networks do not require these qualities, they are essential for the exact identification of neurons and their spike sequences, and investigating their occurrence as a function of experimental variables.

This tutorial presents an approach that allows for extracting 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. These sequences can be considered an aggregate spike sequence of a network, and should be used in conjunction with other techniques for identifying their exact occurrences on a trial-by-trial basis (e.g. by an informed template search).

Overview

In this tutorial you can find information on how you can extract spike timing networks from neural spike recordings, what these networks are, and how they can be interpreted.

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 neural networks formed by consistent between-neuron spike timing from neural spike recordings.*

Note that extracting spike timing networks can take several days, but with distributed computing (see below) can easily be sped up ~25x (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 *EE.165*. In short, rats performed an odor-based delayed matching-to-sample task, requiring it 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 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-dimensional 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 are 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.

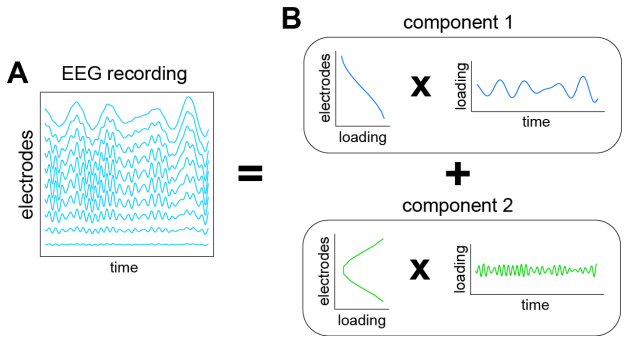

Figure 1: illustration of a decomposition technique

What are 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 neural spiking time series. 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 trial, 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-neuron), the *time profile* (1-by-neuron), and the *trial profile* (1-by-trial). The neuron profile describes how strongly each neuron is part of the network, by a single number per neuron. 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. Lastly, the trial profile 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, allowing for comparing neural activity between conditions at the level of networks instead of at the level of individual neurons. 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.

TO BE ADDED

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 components. This is described in more detail in the two SPACE reference papers.

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 $1/20^{th}$ of a cycle at 50Hz, $1/10^{th}$ at 100Hz, $1/5^{th}$ 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, electroencephalography, etc), the extract 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 frequency 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 magnetoencephalographic 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 components 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 initializing, 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. In this tutorial, we begin with 50 random initializations, and then move forward from there. 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 on a couple of trials, 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 biggest 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, 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 networks when extracting two is not the same as the first networks when extracting three. 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/Fourier coefficients
3) Determine appropriate neuron-wise normalization of the cross spectra
4) Determine the number of components to extract, and extract them, using *nd_nwaydecomposition*
5) Determine the appropriateness of the number of components 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 *EE.165_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, 42 trials. 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('EE.165_Behavior.mat');
begtrial = round(dat.SessionNP(:,2) * 20000); % convert time in sec to timestamp index
endtrial = round(dat.SessionNP(:,3) * 20000); % convert time in sec to timestamp index
ntrial   = numel(begtrial);
data     = cell(1,ntrial);
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

% create 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 spike rate above 1Hz
totaltime = sum(cellfun(@size,data,repmat({2},[1 ntrial]))) ./ 20000;
nspikes   = cellfun(@sum,data,repmat({2},[1 ntrial]),'uniformoutput',0);
nspikes   = sum(cat(2,nspikes{:}),2);
spikerate = nspikes ./ totaltime;
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 matrix'), 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 $1/20^{th}$ of a cycle at 50Hz, $1/10^{th}$ at 100Hz, $1/5^{th}$ at 200Hz, etc. SPACE uses this 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. We then compute the cross products over time for each of these matrices, resulting in a cross spectrum matrix for each trial at each frequency. The resulting cross spectra describe the consistent time delays between spikes by the phases of their off-diagonal.

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 the second SPACE reference paper mentioned above. We will use complex exponents 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 the second SPACE reference paper mentioned above.

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 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 new 'time points' are the result of a time window of data multiplied with a wavelet/complex exponential/other kind of taper.

The following code will use the spike trains obtained above to generate a 4-way neuron-by-freq-by-trial-by-time' array of the square root of the cross spectra:

```
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 (see also *Box 1*). 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 Fourier coefficients. This is because the neuron-by-time matrices can become prohibitively large. Because SPACE only uses the neuron-by- neuron cross products of these matrices, 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 extract spike timing networks. We will do this using the function *nd_nwaydecomposition*. This function handles the multiple random initializations of the SPACE algorithm, and contains algorithms for determining the number of components to extract (see *Background*).

Before we estimate the number of networks to extract however, we need to find an optimal neuron-wise normalization (see the second SPACE reference paper 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 power. Additionally, the power of each cross spectrum is usually 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 another neuron. Combined, this can lead to neuron profiles that are more driven by power, than by spike time consistency, and thus mostly consist of one neuron. 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, causing the split reliability estimation procedure to stop prematurely. This is likely a consequence of reducing the difference in explained variance between networks (which is due to certain noise sensitivities, see the second SPACE reference paper).

The consequence of the above, is that we aim to find a normalization for which the split reliability estimation still extracts 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 components using the split-reliability procedure. If the procedure extract very few reliable components, we can reduce the normalization strength until we do.

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 the amount of normalization. Below we will only inspect the cumulative congruence of an optimal normalization, a sub-optimal normalization, and no normalization. In reality, it requires some guess work to obtain a starting point, but the decision procedure is the same.

The following 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 it 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 XXX XXX. 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 still needs to be added).

```
% 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       = [];                    % req. for SPACE due to other uses
```

Now, we can finally extract networks. We will use 5 networks below (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 do the following for every normalization mentioned above.

```
% 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. See *Box 2* for how to speed this up using distributed computing.

We will 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 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. We will interpret the computed coefficients afterwards. 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 as 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. This should take about 0.5 to 1 day on a single core of a CPU, depending on the hardware. Using a distributed computing setup, this can be reduced by a ~20x (i.e., 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.

```
% 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 average over splits, before comparison to the criterion of 0.7.

```
% 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;                   % split-reliability criterion
cfg.numiter             = 1000;                 % max number of iteration
cfg.convcrit            = 1e-6;                 % stop criterion of algorithm: minimum relative
                                                %    difference in fit between iterations
cfg.randstart           = 25;                   % number of random initializations
cfg.ncompestrandstart   = 25;                   % 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
```

TO BE CONTINUED