IEM Tutorial: fundamentals

First, let's go over a first set of basic concepts underlying the IEM method without all the overhead of processing EEG datasets. I've already set up data to be simple to use, and this is a high-fidelity fMRI dataset that has quite robust single-trial decoding performance. For our purposes, the dataset is basically identical to the EEG datasets we'll be working with later, but this one shoudl be a bit less cumbersome to deal with when

In this tutorial, we'll learn:

1. how to build a channel-based linear encoding model
2. how to use that encoding model, together with a behavioral task, to predict how modeled channels should respond
3. how to compute the contribution of each channel to each measurement (here, activity from EEG electrode)
4. how to *invert* that estimated *encoding model* to solve for channel responses on a separate set of data
5. how to think about these reconstructions

Load data

I've already minimally-processed this dataset so that we have a measured scalp activity pattern (alpha) on each trial. We'll cover ways to get this pattern later on. You can also try loading fMRI_basic.mat - the data is stored exactly the same way, and the datasets are interchangeable for our analyses here. The point is that any signal can be a good candidate for this analysis, so long as some of the assumptions (discussed below) more or less hold.

```
load('data/EEG_basic.mat'); % or fMRI_basic.mat, if interested - the data is stored hte same, just different source!
whos
```

```
  Name                Size              Bytes  Class      Attributes

  c_all             960x2               15360  double
  chan_labels         1x22               2566  cell
  data_all          960x20             153600  double
  delay_window        1x2                  16  double
  excl_all          960x1                 960  logical
  filter_band         1x2                  16  double
  r_all             960x1                7680  double
```

in your workspace, you should have:

- **data_all**: one activity pattern on each trial from each measurement (electrode or voxel) - n_trials x n_measurements
- **c_all**: condition label for each trial. first column is the polar angle remembered on that trial, in degrees (Cartesian coords, 45 = upper right); second column is the position bin (1-8; beginning at 0 deg and moving CCW, so bin 2 is centered at 45, 3 at 90, etc); n_trials x 2
- **excl_all**: logical vector, contains 1 for each trial that was marked by Foster et al (2016) for exclusion based on artifacts, etc; n_trials x 1
- **r_all**: label for each trial as to which run it came from; n_trials x 1
- **chan_labels** (for EEG): cell array of strings labeling each electrode; n_electrodes+2 x 1 cell array (we don't include EOG channels in data here)

This is all we need! we'll go over a few suggestions for how to process EEG data for best use with IEM methods a bit later.

Build encoding model

The central assumption in the IEM analysis framework is that we can model the activity of an aggregate neural measurement (EEG electrode; fMRI voxel) as a linear combination of a discrete set of modeled 'information channels', which we sometimes call 'basis functions'. For now, let's think of those information channels as a set of neural populations - we know, based on decades of vision science, that neural populations in the visual system are TUNED to particular features.

What are some examples of these?

- orientation
- motion
- spatial position <-- we'll focus on this one, since it's easiest
- color

A very important part of the IEM approach is to build SIMPLIFIED models - we know that spatial RFs in visual cortex tile the whole visual field, from fovea to periphery, etc. But do we need to model all known aspects of these neural populations? Especially for something like EEG, NO. We need to model the *relevant dimensions* for our task. In the datasets we're working with, stimuli were always presented along a ring at equal eccentricity - so we can only measure changes in neural responses as a function of polar angle - there is no point in trying to model different responses for different stimulus eccentrificites, since we never measured that! (see Sprague & Serences, 2013; 2014; 2016 for examples of this analysis across eccentricity as well)

The point is: *THE MODEL SHOULD REFLECT YOUR UNDERSTANDING OF NEURAL SENSITIVITY PROFILES AND THE CONSTRAINTS OF YOUR EXPERIMENTAL DESIGN*

At an extreme level: consider modeling sensitivity to color (e.g., Brouwer & Heeger, 2009) for the datasets we're using here. The stimuli we used never spanned that dimension, so such modeling is futile! We'll see below some of the mathematical/theoretical constraints on these models.

Ok - back to modeling.

The IEM framework we're using in this tutorial is LINEAR: that means, for each electrode, we're making the direct assumption that

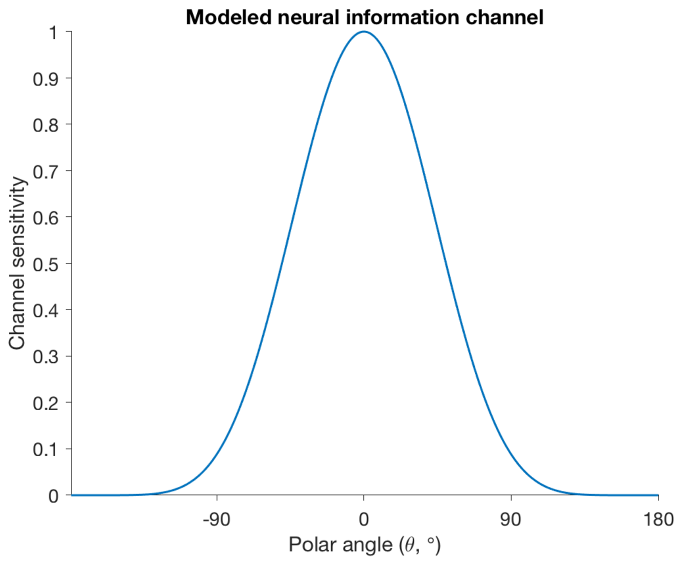
$$B = W_1C_1 + W_2C_2 + \cdots + W_nC_n$$

Where B is measured activity on each trial, C_i is the activity of the i'th modeled information channel on that trial, and W_i is the *weight* describing how strongly channel C_i contributes to the electrode in question.

So - let's look at an example 'channel' profile. We'll model this channel as a function of polar angle, $f(\theta)$, such that the channel is maximally sensitive at a particular feature value (polar angle), and insensitive at far away feature values. A Gaussian could do the trick, but a Gaussian will never reach 'zero' sensitivity - so instead we use rectified sin/cos functions:

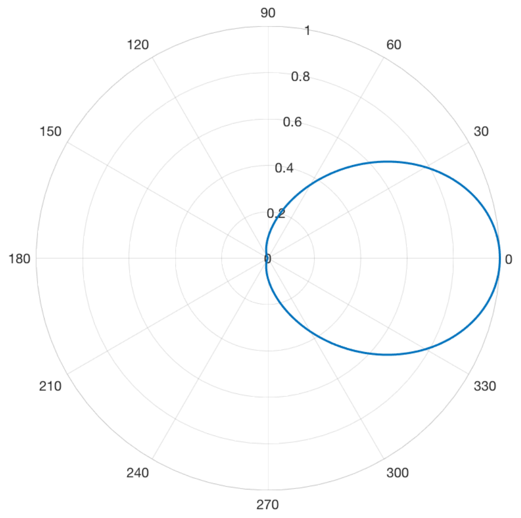
$$f_0(\theta) = \cos\left(\pi \frac{(\theta - \theta_0)}{2s}\right) \text{ where } \left|\theta\right| < s, 0 \text{ otherwise}$$

```
angs = linspace(-179,180,360);
this_chan_center = 0; this_chan_width = 180;

% quick utility function to compute angular distance (degrees):
ang_dist = @(a,b) min(mod(a-b,360),mod(b-a,360));

% the above function will make it easy to always compute the distance between two angles, with the max value being 180

this_basis_fcn = (cosd(180 * ang_dist(angs,this_chan_center) ./ (2*this_chan_width) ).^7) .* (ang_dist(angs,this_chan_center)<=this_chan_width);
plot(angs,this_basis_fcn,'-','LineWidth',1.5);
xlabel('Polar angle (\theta, \circ)');
ylabel('Channel sensitivity'); xlim([angs(1) angs(end)])
title('Modeled neural information channel')
set(gca,'FontSize',14,'TickDir','out','box','off','XTick',-180:90:180)
```



Try changing things like the width of the channel (this_chan_width) and its center location (this_chan_center) - it's important to notice that the channel exists in a circular space, so that when it's centered at, say, 180 deg, it peaks again on the opposite side of the space. You can try plotting it in polar coordinates too:

```
polarplot(deg2rad(angs),this_basis_fcn,'-','LineWidth',1.5);
```

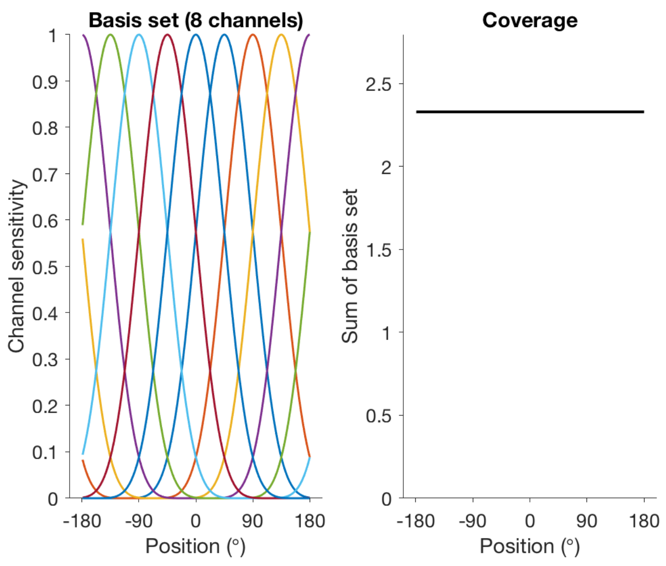


This basis function, or modeled neural information channel, tells us how we think some arbitrary population of neurons in the brain responds to a visual stimulus (spatial location). For today's purposes, we're treating channels as linear filters - the channel responds as a weighted sum of its 'input' (mask of stimulus location). Today we'll only be considering a single stimulus modeled as a point, so the shape of the channel response can be thought of as the response to each possible stimulus value around the screen.

But, we also don't think the brain cares only about a single position. So, we need our channels to tile the feature space - here, polar angle. So let's make a set of n_chans basis functions, which model the sensitivity of each channel:

```
% can adjust these to see how differences in basis set change results below
n_chans = 8;       % DEFAULT: 8
chan_width = 180;  % DEFAULT: 180
chan_centers = linspace(360/n_chans,360,n_chans);

% let's write a function to create our basis function:
build_basis = @(eval_at,this_center,this_width) (cosd(180 * ang_dist(eval_at,this_center) ./ (2*this_width) ).^7) .* (ang_dist(eval_at,this_center)<=this_width);

% and let's build the basis set - all of our channels tuned to different positions
basis_set = nan(length(angs),n_chans);
for bb = 1:n_chans
    basis_set(:,bb) = build_basis(angs,chan_centers(bb),chan_width);
end

% plot the whole basis set
figure; subplot(1,2,1);
plot(angs,basis_set,'-','LineWidth',1.5);
xlabel('Position (\circ)'); ylabel('Channel sensitivity');
title(sprintf('Basis set (%i channels)',n_chans));
set(gca,'XTick',-180:90:180,'TickDir','out','Box','off','FontSize',14);

% and plot the sum of all channels - this tells us if there are any 'holes' in the coverage
subplot(1,2,2);
plot(angs,sum(basis_set,2),'k-','LineWidth',2);
xlabel('Position (\circ)'); ylabel('Sum of basis set');
```

```
title('Coverage');
set(gca,'XTick',-180:90:180,'TickDir','out','Box','off','FontSize',14);
tmpylim = get(gca,'YLim'); ylim([0 1.2*tmpylim(2)]); clear tmpylim;
```



Use encoding model to predict channel responses

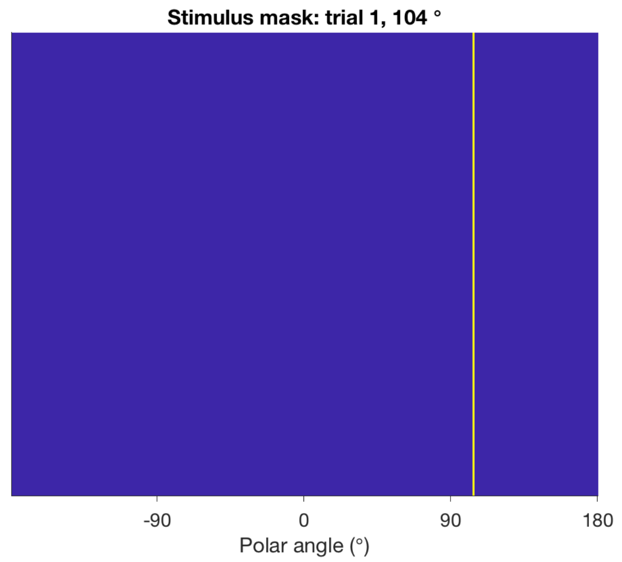
We now have our full encoding model in-hand (and try playing w/ those parameters and re-running analyses). Remember, we haven't touched data yet! This is purely an exercise in a computer at this point - we've built a model for how a neural population might respond to a particular feature of a visual stimulus (position), and then built a 'basis set' of several of those modeled neural populations. This is all under the assumption that the measured neural responses from EEG/fMRI in an electrode/voxel will be made up of this basis set: in the same way a periodic signal is made up of sines/cosines, the signal in our aggregate measurements is thought to be made up of activity in neural populations modeled by these basis functions.

Now, we need to use our encoding model to predict how each modeled population should respond on each trial of the experiment. (note that you don't actually need data for this step either!). We do this by computing a 'stimulus mask' - a vector describing the stimulus in feature space. Because our feature space is polar angle, the vector for each trial will be show the position of the stimulus in polar angle. Let's start with an example trial. Remember that 'c_all' contains the polar angle of each trial in its first column.

```
% quick: let's fix c_all(:,1) to be from -180:180:
c_all(c_all(:,1)>180,1) = c_all(c_all(:,1)>180,1)-360;
which_trial = 1;
c_all(which_trial,1)
```

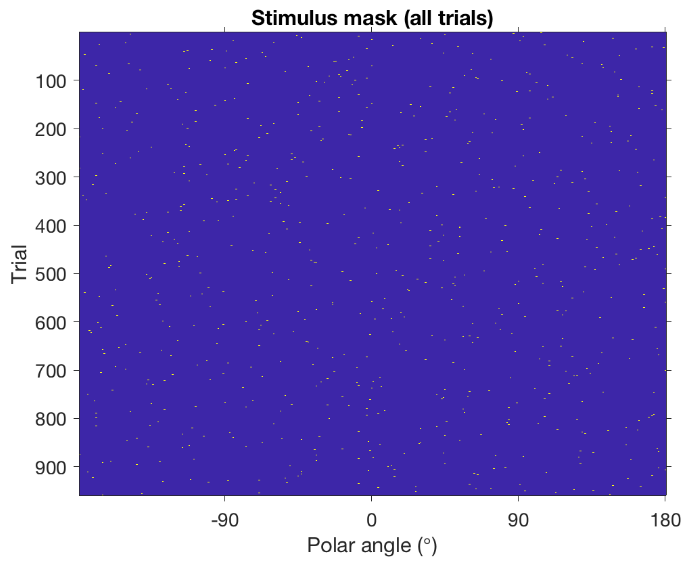
```
ans = 104
```

```
this_mask = zeros(length(angs),1);
this_mask(angs==c_all(which_trial,1))=1;
figure;imagesc(angs,1,this_mask.');
xlabel('Polar angle (\circ)');
set(gca,'XTick',-180:90:180,'TickDir','out','YTick',[],'Box','off','FontSize',14);
title(sprintf('Stimulus mask: trial %i, %i \\circ',which_trial,c_all(which_trial,1)));
```



Easy enough - we could, if we wanted, compute the stimulus mask in different ways: we could assume the representation is a fuzzy blob, or a wide strpe, etc. But, it's easier for now to just assume a single 'perfect' feature value. The stimulus mask is important, though, as it is used to predict channel responses, and so changes in the choice of mask can (subtly) impact results.

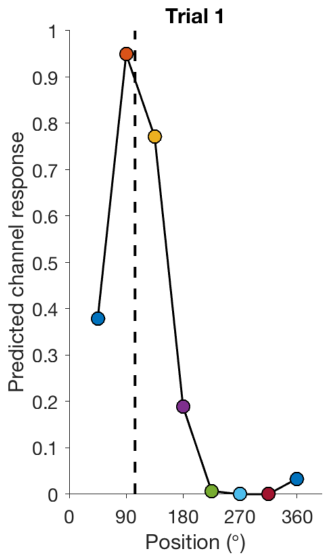
```matlab
% make stimulus mask for all trials
stim_mask = zeros(size(c_all,1),length(angs));
for tt = 1:size(c_all,1)
    stim_mask(tt,angs==c_all(tt,1)) = 1;
end

% plot it - now we have a mask of the stimulus value on each trial
figure;
imagesc(angs,1:size(c_all,1),stim_mask); axis ij;
ylabel('Trial')
xlabel('Polar angle (\circ)');
set(gca,'TickDir','out','FontSize',14,'XTick',-180:90:180);
title('Stimulus mask (all trials)')
```

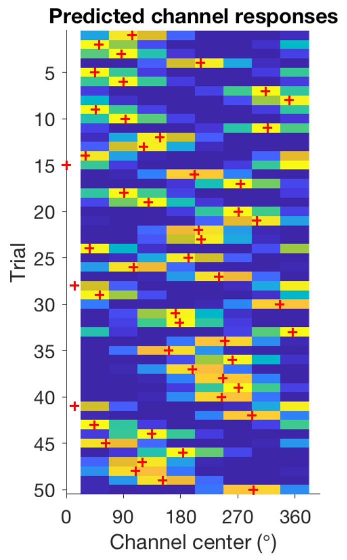


Next, let's use our stimulus mask (n_trials x n_angs) and our basis set (n_angs x n_chans) to predict channel responses.

We want one number for each channel for each trial - let's store it as a matrix called C that's n_trials x n_chans. Because the predicted channel response for each channel is just the weighted sum of the stimulus mask by the channel sensitivity profile (basis_set), we can pretty easily compute C with matrix math. If S is the stimulus mask and F is the basis set (with dims above), we can do: $C = SF$

```
% I'm going to be verbose with variable names, but we often call pred_chan_resp X (convention for regression desgin matrix) or C (in the math descriptions of IEM in papers)
pred_chan_resp = stim_mask * basis_set;

% let's plot an example trial (as above) and the full predicted channel response matrix
figure;
subplot(1,2,1);
chan_colors = lines(n_chans);
hold on;
plot(chan_centers,pred_chan_resp(which_trial,:),'k-','LineWidth',1.5);
for cc = 1:n_chans
    plot(chan_centers(cc),pred_chan_resp(which_trial,cc),'ko','MarkerSize',10,'MarkerFaceColor',chan_colors(cc,:))
end
plot([1 1]*mod(c_all(which_trial),360),[0 1],'k--','LineWidth',2);

title(sprintf('Trial %i',which_trial));
xlabel('Position (\circ)'); ylabel('Predicted channel response');
set(gca,'TickDir','out','FontSize',14,'XTick',0:90:360);

subplot(1,2,2); hold on;
imagesc(chan_centers,1:size(pred_chan_resp,1),pred_chan_resp);
% add a red + on each trial showing the feature value
plot(mod(c_all(:,1),360),1:size(c_all,1),'r+','LineWidth',1.5,'MarkerSize',5);
axis ij;
xlabel('Channel center (\circ)');
title('Predicted channel responses');
ylabel('Trial');
ylim([1 50]+[-.5 0.5]); % zoom in on a subset of trials
set(gca,'TickDir','out','FontSize',14,'XTick',0:90:360);
```





Try changing properties of the basis set and seeing how predicted channel responses change. You should notice that wider basis functions (channels) result in a 'wider' predicted channel response function - that is, more correlated predicted channel responses. That can cause some issues later, and we'll see that below.

Use predicted channel responses to fit encoding model

As described above, the central premise of the IEM approach we're using here is that the activity of each signal (electrode) is a combination of the activity of a discrete set of modeled channels. For a single measurement B, that looks like:

$$B = W_1 C_1 + W_2 C_2 + \cdots + W_n C_n$$

You might recognize this as a linear regression problem - we know B - the activity in an electrode on each trial - and C - the predicted response of each channel on each trial - and we want to solve for W. We can do this most simply with matrix math. We rewrite this as:

$$\boldsymbol{B} = \boldsymbol{C}\ \boldsymbol{W}$$

Where B is n_trials x 1, C is n_trials x n_chans, and W is n_chans x 1.

To solve for W, we need to multiply $\boldsymbol{B}$ on the left by the inverse of $\boldsymbol{C}$. But $\boldsymbol{C}$isn't square! That's ok - it's common in matrix math to use the Moore-Penrose pseudo-inverse to perform matrix divisioin:

$$\widehat{W} = (C^T C)^{-1} C^T B$$

In MATLAB:

```
which_meas = 7; % which electrode or voxel we're looking at
this_data = data_all(:,which_meas); % NOTE: we're including bad trials right now!!! (and no CV)
this_w = inv( pred_chan_resp.' * pred_chan_resp ) * pred_chan_resp.' * this_data;
```

However, for this to work, there are some cosntraints on C:

- To compute a unique solution for W, it's necessary for the predicted responses for each channel to be sufficiently different (as an extreme example, if the predicted responses for two channels were always identical, it's impossible to decide which weight to assign each of them). When might this happen?
- There must be at least n_chans unique sets of predicted channel responses; this typically means there must be n_chans or more unique stimuli which result in unique predicted channel responses.
- If both of these requirements are fulfilled, your Cmatrix will be 'full-rank' - it will have rank equal to the number of columsn (the rank of the matrix is the number of linearly-independent columns; if rank < n_cols, one column can be expressed as a combination of another, and so its impossible to uniquely estimate a solution for W)

Let's check our C:

```
rank(pred_chan_resp)
```

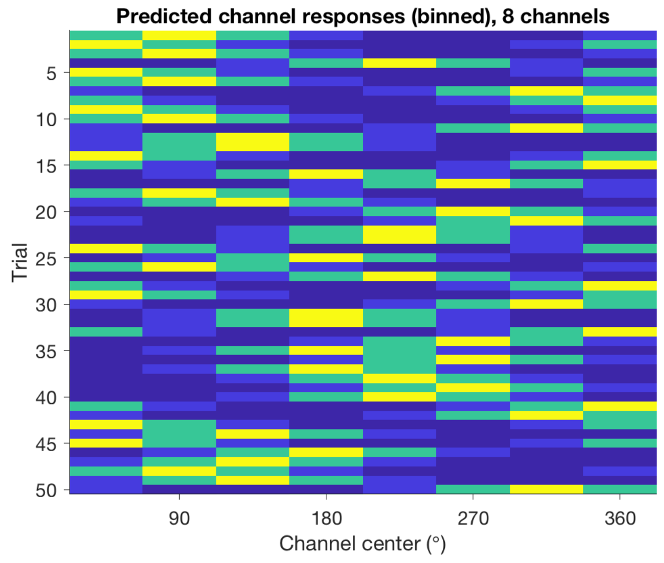
```
ans = 8
```

(We'd have seen errors or warnings in MATLAB if our C wasn't full-rank - but this is *always* something important to keep track of!)

Brief aside: restrictions on encoding models

Because this experiment involved a full uniform distribution of positions, it's actually pretty challenging to adjust the basis set to parameters that result in rank-deficient C. But, we can illustrate a quick example if we instead predict channel responses based on the stimulus position bin (0, 45, 90, etc). This will result in only 8 unique predicted channel response patterns. Under what conditions would we expect to see a rank-deficient predicted channel response (C) matrix?

```
% need to make a new stimulus mask, and a temporary set of basis functions for testing this
stim_mask_bin = zeros(size(c_all,1),length(angs));
bin_centers = linspace(0,315,8); % the bin centers
bin_centers(bin_centers>180) = bin_centers(bin_centers>180)-360;
for tt = 1:size(c_all,1)
    stim_mask_bin(tt,angs==bin_centers(c_all(tt,2))) = 1;
end

n_chans_bin = 8; % try adjusting this
chan_centers_bin = linspace(360/n_chans_bin,360,n_chans_bin);
chan_width_bin = 180; % and this

basis_set_bin = nan(length(angs),n_chans_bin);

for bb = 1:n_chans_bin
    basis_set_bin(:,bb) = build_basis(angs,chan_centers_bin(bb),chan_width_bin);
end

% predict channel responses:
pred_chan_resp_bin = stim_mask_bin * basis_set_bin;

% plot predicted channel responses
figure;
imagesc(chan_centers_bin,1:size(c_all,1),pred_chan_resp_bin);
xlabel('Channel center (\circ)'); ylabel ('Trial'); title(sprintf('Predicted channel responses (binned), %i channels',n_chans_bin));
ylim([1 50] + [-0.5 0.5])
set(gca,'XTick',0:90:360,'TickDir','out','FontSize',14,'Box','off')
```



```
% Check the rank:
rank(pred_chan_resp_bin)
```

```
ans = 8
```

Under what scenarios does the rank not equal the number of channels?

What if you can only acquire a very small number of trials - what does that do to the number of channels that can be independently estimated?

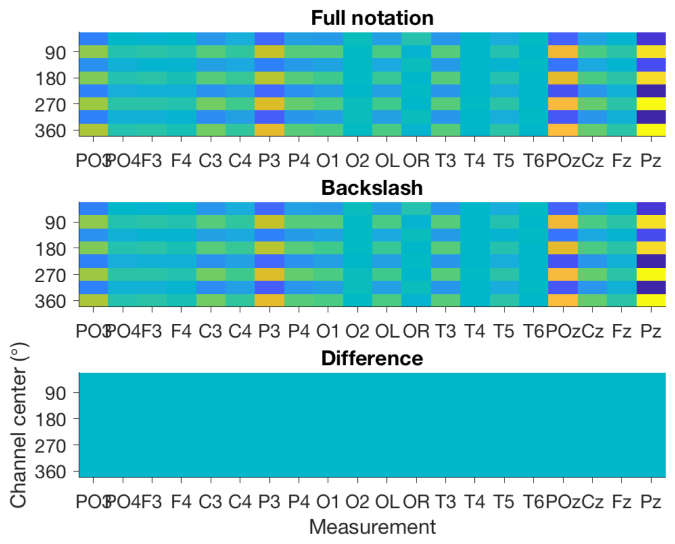
Back to business: fitting models across all voxels

Ok - now that we've learned how to check to be sure we've built a 'good' and 'fittable' model, we can move on.

Above we fit an encoding model to a single measurement (single EEG electrode). This is a *univariate* operation - meaning we can fit each measurement individually and get the same answer as if we fit to all measurements at the same time. The encoding model's job is to describe activity in a measurement, and that requires no information about other measurements. So, we can fit the encoding model to all measurements very easily using the same linear algebra, just bigger data matrix:

```

% fit to data_all, not this_data
this_w_all = inv( pred_chan_resp.' * pred_chan_resp ) * pred_chan_resp.' * data_all;

% MATLAB suggests we switch to backslash notation, which gives an identical result:
this_w_all_backslash = pred_chan_resp \ data_all;

% check that these give the same result (W is now n_chans x n_measurements)
figure;
subplot(3,1,1);
imagesc(1:size(data_all,2),chan_centers,this_w_all); axis ij;
title('Full notation');
set(gca,'TickDir','out','YTick',0:90:360,'Box','off','FontSize',14);
if size(this_w_all,2)==20 % if EEG dataset, label w/ electrode names
    set(gca,'XTick',1:size(this_w_all,2),'XTickLabel',chan_labels);
end

subplot(3,1,2);
imagesc(1:size(data_all,2),chan_centers,this_w_all_backslash); axis ij;
title('Backslash');
```

```
set(gca,'TickDir','out','YTick',0:90:360,'Box','off','FontSize',14);
if size(this_w_all,2)==20
    set(gca,'XTick',1:size(this_w_all,2),'XTickLabel',chan_labels);
end


subplot(3,1,3);
imagesc(1:size(data_all,2),chan_centers,this_w_all-this_w_all_backslash); axis ij;
title('Difference');
xlabel('Measurement');
ylabel('Channel center (\circ)');
set(gca,'TickDir','out','YTick',0:90:360,'Box','off','FontSize',14);
if size(this_w_all,2)==20
    set(gca,'XTick',1:size(this_w_all,2),'XTickLabel',chan_labels);
end


match_clim(get(gcf,'Children'));
```



What happens to the fit weight matrix as you vary channel properties? # of channels, width, etc? If we *really* cared about the *encoding* aspect of the analysis, we'd add some additional constraints to model-fitting, including regularization constraints to minimize effect of channel overlap, etc. But, because the goal is to *invert* the model to reconstruct channel responses, we forego this for now. See Sprague & Serences, 2013 and Vo et al 2017 for examples of using regularization to compute voxel-level spatial sensitivity profiles with a linear channel model and regularization. That's all we'll address about this for now.

Reconstructing channel responses

So far, we've learned how to design an encoding model, and how to fit it to a set of data. This encoding model describes how a modeled set of neural information channels contribute to each measurement.

Now, we're going to put that model to work. Our model - $\hat{\boldsymbol{W}}$ - describes the relationship between our data - $\boldsymbol{B}$ - and our predicted channel responses on each trial - $\boldsymbol{C}$- under our linear model:

$$\boldsymbol{B} = \boldsymbol{C}\hat{\boldsymbol{W}}$$

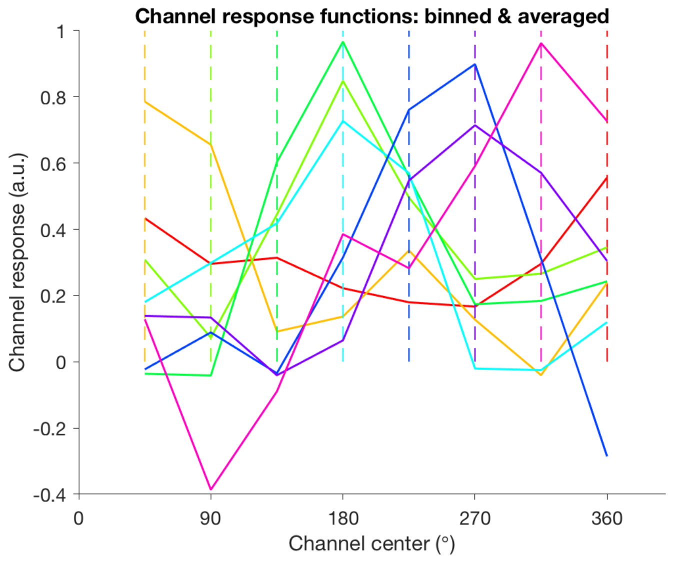
Our goal with this analysis is to recover the channel responses most likely to give rise to an observed activation pattern - $\boldsymbol{B}_{\text{new}}$. Because we know $\hat{\boldsymbol{W}}$ and we know $\boldsymbol{B}_{\text{new}}$, we can just solve for $\boldsymbol{C}_{\textbf{new}}$:

$$\boldsymbol{C}_{\text{new}} = \left(\hat{\boldsymbol{W}}^T\hat{\boldsymbol{W}}\right)^{-1}\hat{\boldsymbol{W}}^T\boldsymbol{B}_{\text{new}}$$

$$\boldsymbol{C}_{\text{new}} = \boldsymbol{B}_{\text{new}}\hat{\boldsymbol{W}}^T\left(\hat{\boldsymbol{W}}\,\hat{\boldsymbol{W}}^T\right)^{-1}$$

Because of our linear assumptions this is an easy problem to solve! Let's look at an example, we'll take ~50% of trials, randomly chosen, as data used to estimate the model & use the remaining data to reconstruct:

```
rng(3728291); % so we get the same answer everytime
tmprnd = rand(size(c_all,1),1);
trn_proportion = 0.5;

% select trn_proportion of trials to train with (but don't keep excluded trials)
trnidx = tmprnd <= trn_proportion & ~excl_all;

% for the test data, pick the complement:
tstidx = tmprnd > trn_proportion & ~excl_all;

% estimate encoding model using 'training' trials
trn = data_all(trnidx,:);
trnX = pred_chan_resp(trnidx,:); % because pred_chan_resp is a 'design matrix' in a linear model, it's often called 'X'

this_W = trnX \ trn;  % estimate W

% use W to compute channel responses:
tst = data_all(tstidx,:);

this_chan_resp = tst * this_W.' * inv(this_W * this_W.'); % estimated channel responses on each trial!
% above is equivalently written as: this_chan_resp = tst/this_W - we'll use this format later on!


% plot average reconstructions for each position bin
figure; hold on;
% get a variable we can use to index into position bins
pu = unique(c_all(:,2));
this_c = c_all(tstidx,:); % label for each trial

% colors for plotting
pos_colors = hsv(length(pu));
bin_centers_plot = mod(bin_centers,360); bin_centers_plot(bin_centers_plot==0)=360;
for pp = 1:length(pu)

    % pick the trials in this bin
    thisidx = this_c(:,2)==pu(pp);

    % plot channel responses
    plot(chan_centers,mean(this_chan_resp(thisidx,:),1),'-','LineWidth',1.5,'Color',pos_colors(pp,:));

    % plot position bin center - this should match the 'peak' of the reconstruction
    plot(bin_centers_plot(pp)*[1 1],[0 1],'--','LineWidth',1,'Color',pos_colors(pp,:));
end
xlabel('Channel center (\circ)');
ylabel('Channel response (a.u.)');
title('Channel response functions: binned & averaged')
set(gca,'TickDir','out','FontSize',14,'XTick',0:90:360);
```



Ok - let's recap what we just did.

First, we divided data into **training** and **test** sets (or model-estimation and reconstruction sets). Next, we estimated the encoding model ($\hat{W}$) using the training set. Then, we *inverted* that encoding model to compute channel responses, given that encoding model, during each trial in the held-out test set of data. This step, whereby we *invert* the encoding model, is where the magic happens. This is the first (and only) step where we're using *multivariate* information across all measurements (electrodes). What's actually happening is we're projecting activity from 'measurement space' (one number for each measurement: electrode, voxel, etc) into a modeled 'channel space' (one number for each modeled information channel). If our encoding model is well-fit, the recovered response pattern across channels - the **channel response function** - should refelct a peak in activity near the relevant location on that trial.

Above, as a quick first-pass at visualization, I took the average of the `this_chan_resp` variable (C_{new} in our written notation) across all trials within each position bin. If our channel response functions are tracking the correct location, we'd expect to see a shift in the peak of the channel response function across the bins (indicated by the vertical dashed lines). To me, this looks quite good! Try changing different aspects of the analysis, including:

- # of channels and their width (scroll up, then click "run to end")
- proportion of trials used for training set (`trn_proportion` above)

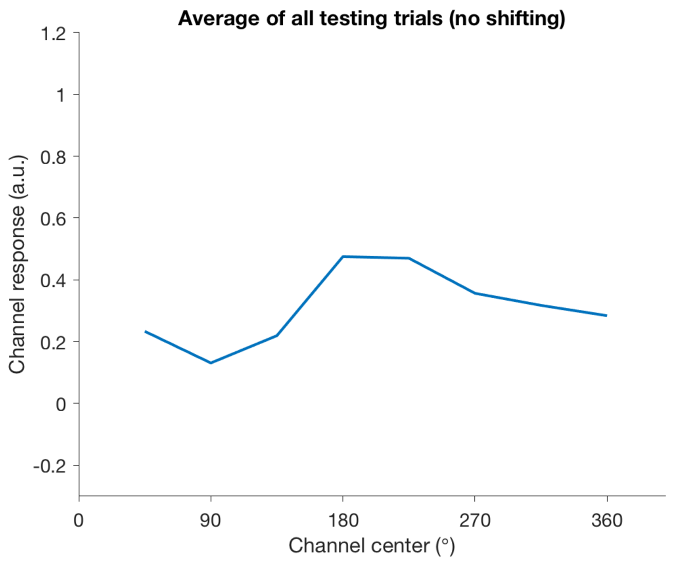
As discussed above, when building/training/testing IEMs, there are a few restrictions to keep in mind:

- In order for $\hat{W}$ to be pseudo-invertible, you must have more measurement dimensions (here, electrodes) than modeled channels. Try changing the number/width of channels above to unsolvable regimes and see what happens (MATLAB will often let you do it! but don't do that!)
- In order for the reconstruction to be valid, the data used to estimate the model must be *entirely independent* of the data used for reconstructing channel responses. Otherwise, you're just showing that you've overfit your data, and your conclusions aren't valid.

Aligning all trials

As plotted above, reconstructed channel response functions from each trial show remarkably different profiles depending on the represented feature value. This means that if we were to average all trials, as-is, we'd be looking at basically nothing:

```
figure;
plot(chan_centers,mean(this_chan_resp,1),'-','LineWidth',2);
xlabel('Channel center (\circ)');
ylabel('Channel response (a.u.)');
title('Average of all testing trials (no shifting)');
set(gca,'XTick',0:90:360,'FontSize',14,'TickDir','out','Box','off')
ylim([-0.3 1.2]); % make our y-axis match, approximately, that above
```



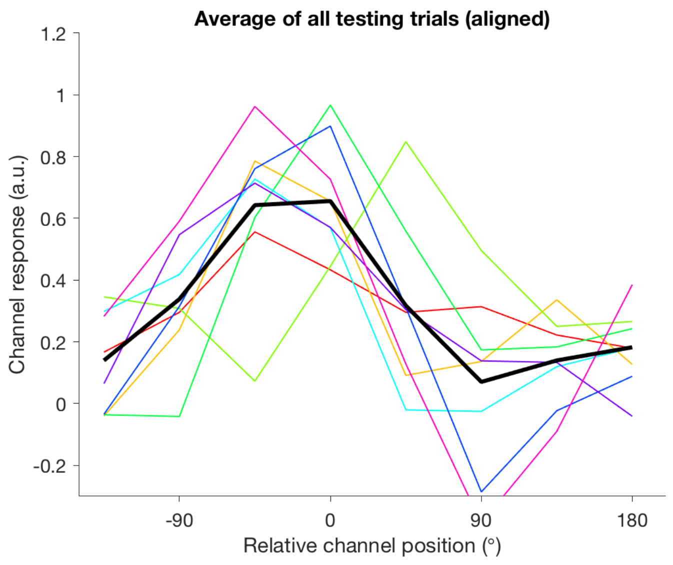
Any structure we see here is more or less an accident (see the advanced tutorial for a few possible causes and example code for addressing them).

But, all is not lost! We know the stimulus value on each trial, so we can manipulate our reconstructions to align all trials to the same position. Here, I'm going to illustrate this procedure based on the binned position information, but see the advanced tutorial for an example of how to do this using the 'raw' feature values.

IMPORTANT: the below analysis will *only* work with 8 modeled channels due to the way the data is binned. You can experiment with changing channel width, but if you change the # of channels the aligned CRFs will be nonsensical!!!!! Note that this is only an issue for this type of alignment in channel space, but full feature reconstructions are not subject to this limitation (see Advanced tutorial)

```
targ_pos = 180;
[~,targ_idx] = min(abs(chan_centers-targ_pos));
rel_chan_centers = chan_centers - targ_pos; % x values for plotting - relative channel position compared to aligned

% see note above: this scheme will only work when using 8 channels!!!
if n_chans == 8

    % create a new variable where we'll put the 'shifted' versions of channel response functions
    chan_resp_aligned = nan(size(this_chan_resp));


    % loop over all trials
    for tt = 1:size(this_chan_resp,1)
        % because we're in a circular space, we can easily just shift the values in this_chan_resp left or right so that they 'line up' across trials
        % we'll use circshift to do so. that means we need to know the direction & how much to shift by
        % + is right, - is left
        shift_by = targ_idx - this_c(tt,2); % when targ_pos is higher, we need to shift right
        shift_ang = targ_pos - mod(bin_centers(this_c(tt,2)),360);
        chan_resp_aligned(tt,:) = circshift(this_chan_resp(tt,:),shift_by);
    end

    % plot it - plot all trials aligned, sorted by bin in their bin color (liek above)
    % then add the mean across all trials on top

    figure; hold on;

    for pp = 1:length(pu)
```

```
        % pick the trials in this bin
        thisidx = this_c(:,2)==pu(pp);

        % plot channel responses
        plot(rel_chan_centers,mean(chan_resp_aligned(thisidx,:),1),'-','LineWidth',1,'Color',pos_colors(pp,:));

    end
    % plot mean
    plot(rel_chan_centers,mean(chan_resp_aligned),'k-','LineWidth',3);


    xlabel('Relative channel position (\circ)');
    ylabel('Channel response (a.u.)');
    title('Average of all testing trials (aligned)')
    ylim([-0.3 1.2])
    set(gca,'XTick',-180:90:180,'FontSize',14,'TickDir','out','Box','off')
else
    figure;
    text(0.5,0.5,'USE 8 CHANNELS!!!!','FontSize',18,'FontWeight','bold','HorizontalAlignment','center','VerticalAlignment','middle','Color','r')
    axis off;
end
```



And there you have it! This is the essentials of performing an IEM analysis on a simplified dataset (one measurement per trial). After finishing this tutorial, you have all the skills necessary to move on to a meatier dataset - one with 250 samples per second on each trial.

Here, we didn't actually reconstruct each and every trial - we only split the data into two sets. You can try, as an exercise, running a full cross-validation procedure where you update the training set on each iteration to leave out a different set of trials for reconstruction.