**This document outlines a proposal for estimating the neural activation profile in visual space that caused a certain pattern of voxel responses, by inverting the pRF model.**

Just to recapitulate, the pRF model basically says the following:

Here, *r(i)*(*t*) constitutes the neural population response at voxel *i* as a function of time *t.* *g*(∙) denotes a Gaussian function in visual space (which has coordinates x,y), modeling the receptive field, parameterized by 4 voxel-specific parameters: x0(i) and y0(i) denote the x- and y-coordinates of the mean location of voxel *i*’s receptive field, σ(i) represents its standard deviation and α(i) its amplitude. (Note that I’m using superscript indexation to accommodate the naming of the x0 and y0 parameters.) *s*(∙) denotes a function describing the stimulus intensity as a function of location in visual space, and as a function of time. We don’t need to worry about what this looks like - we just assume that since we designed the stimulus, we know what it looks like. Finally, *εi(t)* constitutes some noise in the response, i.e. the part of the response that cannot be explained by the stimulus and the pRF model.

In words, this equation says that to get the neural population response in a certain voxel at a certain time, we multiply its receptive field with the stimulus at that time, and then sum over visual space in the x- and y-directions. Note that this describes the *neural* response in this voxel - to get the actual BOLD response we still need to convolve this with the HRF, but this is a step which need not concern us for now. In what follows, I will assume the method for estimating pRF parameters to be known. In fact, let’s assume that we’ve already obtained these parameters for all the voxels in our dataset. Our goal in what follows is to estimate the stimulus *s* that evoked a new pattern of voxel responses (e.g. the voxel responses evoked on average by an attended stimulus), given the known model parameters.

## What is *s*?

The first question we need to ask ourselves is: how do we model the stimulus *s*? First of all, a bit of disambiguation: by “stimulus” in this case, I no longer mean just some bottom-up visual input, but this input combined with any top-down feedback that is also a function of visual space (e.g. spatial attention). So perhaps instead of “stimulus”, a better term at this stage would be something like “visual activation map”. For backwards compatibility however, we’ll keep calling it “stimulus”.

**Side note:** Later on, we might try to explicitly model the influence of attention in terms of a multiplicative effect. For now, however, I think it would be good to just have some method to “translate” an activation pattern in the brain to an activation map in visual space.

The trouble is that we do not necessarily have a good idea of what the stimulus will look like. In theory, we could divide visual space up into small “pixels” and estimate the stimulus intensity for each pixel. However, this would require estimating a huge number of variables and besides, it doesn’t seem very realistic to assume that each pixel might be activated independently (let alone that we could measure with such precision).

Instead, I propose that we use a *mixture of Gaussians* to model the stimulus. This would look like this:

Here, *g*(∙) once again denotes a Gaussian function with the same three parameters as before: the (x,y) coordinates of its mean and its standard deviation. However, to avoid confusion with the parameters of the pRFs, I’ve called them *a*, *b* and *τ*. Note that they are indexed by *j*, which runs from [1, *M*], where *M* is the total number of Gaussians included in the mixture. Finally *wj* constitutes the weight on the *jth* Gaussian in the mixture, i.e. its “contribution” to the mixture.

The advantage of this approach is that it allows you to model shapes that can be arbitrarily complex, while controlling the complexity of the shape with the value of *M*. Which brings us to the next point…

## How to choose M?

How many Gaussians should we use to model the stimulus? This is a typical machine learning question. Too many Gaussians, and we risk overfitting. Too few, and we are likely to be biased. The answer lies in cross-validation. That is, we can pick some value of *M*, and then use it to estimate the parameters of *s* on a subset (say 90%) of all voxels. Then, we can use the estimated parameters to try to predict the responses of the remaining 10% of voxels. We repeat this process for different separations of the data, choosing a different division of 90% and 10% each time, such that after 10 iterations, we’ve used each voxel exactly once in the prediction stage. Averaging across all 10 iterations, we can calculate the cost associated with the value of *M* that we picked at the start. Our job is to find the value for *M* that minimizes this cost. But what is that cost?

## The cost function

The cost function can be derived from the very first equation, if we assume that the noise term in this equation follows a normal distribution. If this is the case, then from the normal likelihood, we can derive the following cost function:

Where *N* is the number of voxels (in the training set). This may seem like a complicated expression, but really all we’re doing is computing the sum of squared differences between the observed voxel responses, and the responses predicted based on a stimulus defined by parameters . Note that the way we arrive at the predicted response of a voxel is no different from before: we still multiply the voxel’s pRF by the stimulus and then sum over visual space.

## What is r(i)?

One thing remains to be cleared up before we talk about the full algorithm that’s going to estimate the stimulus parameters for us, and that is to talk about what we mean by “voxel responses”. When estimating pRF parameters, we simply take the raw BOLD response at each timepoint. Now, however, we’re interested in the (average) pattern of voxel responses associated with a certain *condition*, rather than at a particular point in time. In practice, however, this makes very little difference. We simply remove the dependency on *t* from our equations (as I’ve already done in fact from the 2nd equation onwards), and take *r(i)*to be the response of voxel *i* in the condition that we’re interested in. (So it’s as if we’re considering the voxel activations at just a single timepoint.)

Finally, to get the voxel responses in a particular condition, we can simply use the beta weights from our GLM analysis. So if we want to estimate the stimulus in the “attended” condition, we set . Or if we want to estimate the stimulus associated with the attentional *modulation*, we can set .

## An algorithm

Finally, let’s sum up everything in an algorithm that’s going to take as its input the voxel responses and the pRF parameters, and spit out an estimate of the stimulus. Such an algorithm, written in pseudocode, might look like this:

1. Define a vector M\_vals containing plausible values for M
2. For each k between [1, length(M\_vals)]:
   1. Set M = M\_vals(k)
   2. For each l between [1,10]:
      1. Separate the data into train and test sets, where a unique 10% of voxels are used for testing each time
      2. Find the values for the stimulus parameters that minimize the cost function L(), given the chosen value for M and the voxel responses in the train data. (Note: for this step I advise you to use fminunc in Matlab.)
      3. Given the estimated stimulus parameters, generate predictions for the responses of the voxels in the test set
      4. Compute the prediction error and store it in pred\_err(l)
   3. Compute mean(pred\_err)and store it in avg\_pred\_err(k)
3. Find the value of M that produced the smallest average prediction error, i.e. the smallest value in avg\_pred\_err
4. Set M to this optimal value
5. Find the values for the stimulus parameters that minimize the cost function L(), given the optimal value of M and using all the voxel responses
6. Return the estimated stimulus parameters

One more note: when splitting the data up into different sets of voxels, it would be good to make sure that the voxels in both the train and test sets are sampled approximately evenly in retinotopic space. That is, you don’t want to have (for example) only voxels in the test set that respond to peripheral visual locations, because the stimulus might not cover the periphery, and then it becomes impossible to predict the voxel responses of that particular test set.

## Implementation

When implementing this method, I would advise you to work from the middle of the algorithm “outwards”, going through the following steps (roughly):

1. Implement the mixture-of-gaussians model of the stimulus. Play around with the parameters, starting with a small value for M, and make some plots (using imagesc) of the shapes you’re able to create with this model.
2. Implement the cost function. Check that this cost function makes sense by putting in ridiculous parameters for the stimulus values. You should find that the cost is lower for values that are in a sensible range, and higher for values that are silly.
3. Experiment with using fminunc, and once you understand how it works, use it to estimate the stimulus parameters given some value of M that you choose as you like (I’d suggest a small value, certainly lower than 10). You can use any number of voxels at this stage - no need to implement the cross-validation loop yet. Plug the parameters you get from fminunc into your stimulus model (the mixture-of-gaussians function) and see what you get.
4. Once you’re satisfied that this works, you can implement the inner cross-validation loop (the loop over l). Think of a good way to split up your voxels into 10 groups (or fewer, if necessary). Check that voxels never get re-used for testing. Check that the predictions you get for the voxels in the test data make sense.
5. Finally, once you have this sorted out, you can implement the outer loop and the rest of the algorithm. It’s a good idea to plot the average prediction errors of the values for M that you tested; you should find that this function has a clear global minimum.

Don’t worry if you don’t understand all the different steps yet, some of them may become clear once you’ve completed the inner part of the algorithm and understood what it does. As you see I’ve made some suggestions for checking your work. In general, please play and experiment with this stuff as much as you can and satisfy yourself that you understand it. Also, you don’t have to try this on real data straight away - if you can think of some way to do it on fake data (where you know exactly what you’re dealing with because you’ve created the data yourself) that would be great.

## Gaussian product integral

Much of our computations in this method involve taking the integrals of products of Gaussians. Since a product of Gaussians is again Gaussian, we might expect this integral to take a simple form. In fact, it does. Let *f()* and *g()* be two Gaussian functions, then:

Where *μ*, *σ* and *a* denote the mean, standard deviation and amplitude of the respective Gaussians, and *Sfg* is a scaling factor:

And this scaling factor happens to be equal to the resulting Gaussian’s integral, i.e.:

Note that this is only true if we take the integral over the entire function. Integrals on a finite interval are not so tractable. Another helpful fact is that for a bivariate Gaussian, the following holds:

Thus, for two symmetric bivariate Gaussians (where symmetric means that they are circular in shape, and thus characterized by only one standard deviation per Gaussian), we can derive:

### New cost function expression

The cost function included the following term:

Using the expression we just derived, we can rewrite this term as follows:

So that the new expression for the cost function becomes:

Which should really speed up computation, as we’ve lost all dependency on the visual space coordinates *x* and *y*, so you no longer need to define a meshgrid and sum over pixels.