Bootstrap Bill

MATLAB concepts covered:

1. operations along columns of 2D matrix
2. use of 'repmat'
3. simple descriptive statistics: mean, std. error
4. plotting functions: plot, errorbar, polar
5. handle graphics
6. bootstrapping concepts: null hypothesis; permutation test; bootstrap confidence intervals

*See corresponding Matlab code in **BootstrapBill.m** for help on each step.*

Spike data were collected from an alert, fixating monkey. On each "trial" a circular field of random dots moving in a particular direction was presented for a period of two seconds. See movie *newsome-direction-MT.avi*.

The dots moved in one of eight possible directions evenly spaced around the circle at 45-degree intervals. The data consist of 5 repeats of each of the eight possible directions, presented in a block-wise random order (i.e. stimuli were selected randomly, but no stimulus was repeated until all 8 had been presented once). Why might one do it this way?

The values are the average spike rate of the neuron for the full 2 seconds of the stimulus presentation. Rows constitute observations (i.e. repeat trials of the same visual stimulus) and columns are the different directions of motion.

Step 1: Load and plot the data

```
load spikedata;
```

For convenience and clarity, it's often useful to extract parameters that we will probably use frequently. This prevents us from having to litter the code with unwieldy **`size(Data,1)`** and the like.

```
nRepeats = size(Data,1);        % # of stimulus repeats (= rows)
nDirs = size(Data,2);           % or, equivalently, length(Dirs)
```

Plot the data.

One immediately gets the idea that the cell fired at high rates for some stimuli (e.g. 0 degrees, which is rightwards, by convention) and barely responded at all to others (e.g. 180 deg., equals to the left). But you might notice that you can't always discern 5 separate data points for each direction of motion. Why might this be?

This is because some of the data points are essentially lying on top of each other due to the discrete nature of our sampling. One trick for fixing this is to randomly jitter the data points, in this case along the x-axis.

Step 2: Jitter the data and re-plot it

Add a small amount of random noise to the data points to jitter them along the x-axis so that they don't overlap.
You'll need to add different noise value to each x-value (direction) so you'll need to repeat the direction matrix to match the number of repetitions.
Hint:

```
help repmat
```

Once you've plotted your jittered data, type

```
hold on;
```

so we don't over-write the plot with subsequent plots.

Step 3: Means and standard errors

Plot the means and standard errors

Reminder: standard error = standard deviation/√n

As always, it might be a good idea to see if Matlab has any built-in functions that help you do this.

3b. Now, how about changing the color?

Step 4: Polar plots

This looks pretty good. But it's actually a bit misleading: the direction 315 degrees (down and to the right) is as similar to 0 degrees (right) as is 45 degrees (up and to the right), yet 315 is as far away from 0 on our plot as it can be. In other words, "direction" is a circular variable—it wraps around on itself. Because of this, there is no true zero point (Statisticians would say that it is an "interval scale" of measurement.) and any designation of high or low values is arbitrary. That is, the directions must be treated like categorical variables that do not permit of simple arithmetic operations. To get a sense of this, try to calculate the average of two directions: 45 and 315.

A more natural way to plot circular data is using a polar plot.

In a separate figure, make a polar plot of the data.

Try

```
help polar
```

Reminder: polar plots are drawn in polar coordinates, theta (angle) and rho (distance). Rho is just the distance from the origin, in this case, the firing rate. Theta is the direction *in radians*. To

convert from degrees to radians you just need to multiply deg * (pi/180). Or you can look to see if Matlab already has a function to do this.

Add "error bars" to show the standard errors. Does errorbar work here?

Step 5: The mean vector.

Just as a linear distribution has a mean and a variance, so does a circular distribution. We compute the "mean vector" of the circular distribution by treating each data point as a 2D vector with direction equal to the direction of motion and magnitude equal to the spike rate. We add up the vectors and then divide by the sum of the lengths of the individual vectors. There are at least two ways to add vectors: the geometric way, by putting them in tail-to-head order and drawing a line from the tail of the first vector to the head of the last one. The analytical way is to first decompose the vectors into their Cartesian components, then add them and convert back to polar coordinates. Luckily, MATLAB has two built-in functions that make this easy: 'pol2cart' and 'cart2pol'.

We have supplied you with a helper function 'DirCoM' to calculate the mean vector:

```
[Th,R] = DirCoM(deg2rad(Dirs),Y);
```

This will scale it and plot it on your polar plot:

```
ax = axis;  % [R_min R_max Th_min Th_max]
maxR = R * ax(2);
hp = polar([0 Th], [0 maxR], 'g-');
set(hp,'LineWidth',2);
```

The length of the mean vector, R, is a measure of the variability of our circular distribution. Actually, it is an inverse measure, since a big R (close to 1) indicates a narrow distribution (small circular variance). Indeed, 'circular variance' is defined as '1 - R'. So we can use R as a measure of 'strength of tuning': neurons with high R-values are highly selective for the direction of motion and those with low R-values are weakly selective.

Step 6: A permutation test for R

How can we tell if our neuron is "significantly tuned"? There are many ways to approach this problem. For example, there are parametric statistics for circular uniformity, such as "Rayleigh's Test" or, alternatively, we could treat the problem as a 1-way ANOVA. But let's suppose that all we have is a good random number generator and our knowledge of the experimental design. What might we do?

The first, and most important, step is to flesh out the null hypothesis (often symbolized as "H0"). Basically, the null hypothesis is always the party pooper: You didn't get a positive result from your experiment; it was all just random chance. What would this mean in the context of our direction-tuning experiment?

In interpreting our tuning curve, we are basically positing a causal effect between the visual stimulus (in this case, its direction of motion) and the number of spikes that the neuron produced.

Thus one way to think of H0 is as the negation, or breaking, of this proposed connection: the direction of motion of the visual stimulus presented on a given trial had *nothing to do* with the spike rate of the neuron on that trial. Sometimes the neuron fires a lot, sometimes a little, and we just happened to have sampled this noisy spike rate distribution in a certain way that just happened to put high spike rates in certain "bins" (columns in our data matrix) and low rates in others.

Armed with this idea, we can create a rigorous test of H0 by "breaking the association" between spike rate and visual stimulus. In other words, we simulate data sets using the random sampling strategy that we just proposed and then look at how our experimentally determined value ('R' in this case) compares to R-values generated according to H0. One way to do this is to take our original data set and *randomly assign spike rates to different values of the visual stimulus*—that is, we shuffle the original spike counts ignoring the visual stimulus that actually elicited them. We then compute 'R' for the permuted data set, repeat this a few thousand times and then see where in this distribution of H0 R-values our real R-value falls. This allows us to directly answer the question of how often we would expect to get our real R-value by chance.

So, your goal is: Using R as a measure of goodness-of-tuning, code up a "permutation test" to answer the question: Under the null hypothesis, how likely would we be to obtain an R-value as large as or larger than the one we obtained in our experiment. See steps below.

In our first crack at the bootstrap, we will preserve the association between stimulus and response by sampling with replacement within columns and then permuting the columns.

Step 6a: Generate one set of random indices to sample with replacement from within columns of our original data set. That is, generate the random indices that you would use in one iteration of your bootstrap.

Step 6b: Generate 5000 sets of these and “prove" that your code gives whole numbers on the interval [1 nRepeats] with equal probability**.**

Step 6c: Permute the data, breaking the association between stimulus and response. Create a loop and use your method of indexing to sample randomly from within each column *and* permute the columns.

Each iteration of the loop should give you an array the same size as your original data. The data should be shuffled within each column and the columns should be shuffled. Use this new data to get an R value. Repeat this 5000 times to generate 5000 bootstrapped measurements of R.

What does the distribution of R values look like?

How might you get a p-value from this distribution?

Reminder: a p-value, in this case, is the probability of obtaining an R value as extreme as your original R value assuming H0 (your permuted data) is true.

Step 7: Bootstrapping for confidence intervals

Another, more common, use of so-called "bootstrapping" techniques is to compute confidence intervals (CIs) on experimentally measured parameters. With standard statistical techniques, this is only possible for a handful of things we might wish to know about a population sample, such as the mean. Using the bootstrap, however, we can determine CIs for anything we can compute on our data set. All we do is sample from our original data set **with replacement** to obtain a new estimate of our desired parameter. Again, we do this lots of times (thousands) to see the kind of variability we can get in our estimate, *given our own data.* That is, using the variability in the data we have collected, we can determine the variability in our estimate of our parameter. In this case, **we do not want to break the association between stimulus and response**; rather, we want to ask how our estimate of R might change as a function of different re- samplings of the original data. In essence, we are letting our experimental data serve as the model of the universe of possible responses, and repeatedly re-sampling the data to obtain so-called "bootstrap samples." You do this with replacement so that your universe of possible firing rates doesn't get smaller every time you sample it.

Step 7a: Sample the data such that you

1) preserve the association between stimulus and response
2) allow a sampled firing rate to stay in the pool of possible firing rates (i.e. sample with replacement)

The end result of each bootstrap iteration should be the same size as the original data set. Use this new data to get a new R value on each iteration. This should look somewhat different from your bootstrap loop in step 6. As you write your code, check to make sure your indices are on the right interval.

What does the distribution look like?

Bonus Step 7b: Use MATLAB's bootstrp function

As for many things that we want to do a lot, MATLAB has a function to do bootstrapping. It's called bootstrp without the 'a'. Use the function RvalBS (written by Rick) as the second input to bootstrp.

Step 8: Bootstrapping pairs of data

This is reasonable, but the preferred bootstrap approach is to bootstrap pairs of data. That is, we don't assume that we're always going to get 5 repetitions of each direction. We generate paired data by pairing up each response with the direction of motion that elicited that response. Then we throw all of these pairs into one big pot and, again, sample with replacement to generate a bootstrap sample of the same size as our original one. In this case, however, our 40 values will not necessarily be evenly distributed across the eight different directions.

Step 8a: Use repmat (and some other stuff) to generate a 40 x 2 array of the paired data.

Step 8b: Use this data do the bootstrap by pairs and find the 95% confidence interval for R. Don't forget to check and visualize your data along the way!

Compare the distributions using the different methods. What are the assumptions underlying each version?

Bonus Questions
According to Efron & Tibshirani (the Bible for bootstrappers), you can get a good estimate of the standard error from a few hundred bootstrap samples, but it takes a few thousand samples to get a good estimate of confidence intervals. Why do you think this is so?

Extra Credit: Let's suppose we are cpu-limited and can only do a few hundred bootstrap repetitions, so we are confident of our std. error. Suppose further that we have reason to believe that our bootstrap statistic is normally distributed. How might we take advantage of this to estimate our confidence intervals?

Reference for Bootstrapping:
Efron, B., and R. J. Tibshirani (1993) An Introduction to the Bootstrap (London: Chapman & Hall/CRC)

Revisions:
RTB wrote it, 23 May 2011 originally as pRasterTool.m, converted to DirBS.m for teaching purposes: direction tuning curve and bootstrap test for significance
AS edited 4 Aug 2011.
JW edited 8 Aug 2011
RTB updated 18 Aug 2012 and changed name of corresponding m-file to BootstrapBill.m