MemToolbox

Jordan Suchow*, Timothy Brady*, Daryl Fougnie, George Alvarez

Vision Sciences Lab, Harvard University

The MemToolbox is a collection of MATLAB functions for modeling visual working memory. In support of its goal to provide a full suite of data analysis tools, the toolbox includes implementations of popular models of visual working memory, real and simulated data sets, Bayesian and maximum likelihood estimation procedures for fitting models to data, visualizations of data and fit, validation tools, model comparison metrics, and experiment scripts.

Through a series of demonstrations, code for which can be found in the MemDemos folder, the following tutorial covers most of the toolbox's core functionality.

This tutorial uses a beta version of the toolbox and was last updated on 2/26/2013.

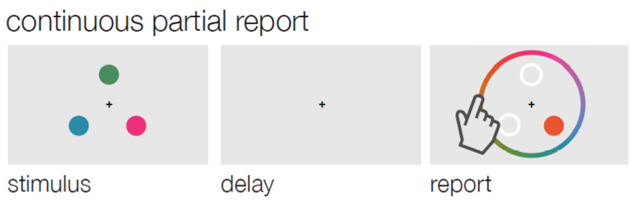
* = equal contribution

MemToolbox Tutorial

Table of Contents

Demo 1: Mixture modeling in just two lines of code

While the MemToolbox can be applied to data from a variety of tasks (e.g., Demo 9), to start the tutorial we will focus on only one of these tasks, a continuous partial report task with colors. In such a task, observers see the stimulus display, and then after a delay are asked to report the exact color of a single item. Error is quantified as the distance between the response and the correct answer, in degrees on the color wheel.



The simplest way to use the MemToolbox with such data is through `MemFit`, a function that houses much of the toolbox's functionality under one roof. To use `MemFit` with continuous report data, first specify your data as a vector of errors, one for each trial, in units of degrees on the color wheel. These errors should fall between –180 and 180.

```
>> errors = [-89, 29, -2, 6, -16, 65, 43, -12, 10, 0, 178, -42, 52, 1, -2];
```

Next, call `MemFit` on the error vector.

```
>> MemFit(errors);
```

The toolbox will now run through its analysis of your data, first showing you a histogram of the errors, and the name and parameterization of the model it is fitting to the data:

```
Error histogram:   -180 __________.'.__________ +180
          Model:   Standard mixture model
     Parameters:   g, sd
```

By default, the MemToolbox fits the standard mixture model of Zhang and Luck (2008) to the data. The error histogram is a simple text-based histogram — in this case, you can see that the data has a bump in the middle but is otherwise relatively flat. This visualization is useful to detect if your error histogram is wildly off from what you expect (for example, if you were to use the wrong units).

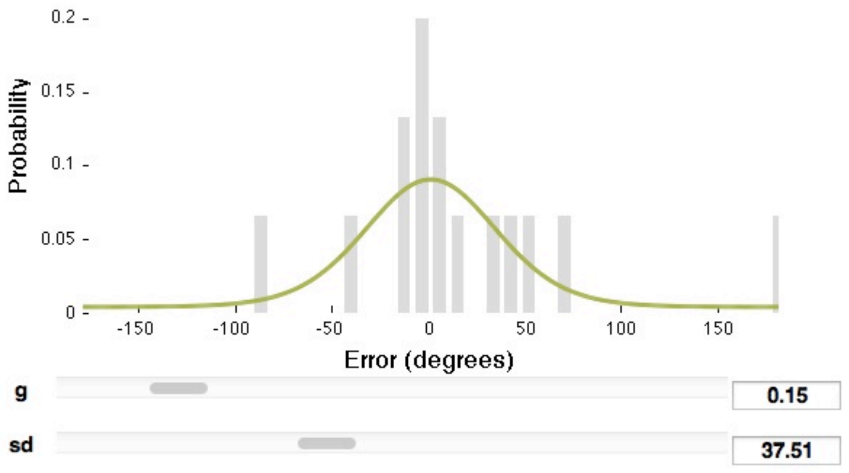
Next the MemToolbox will fit the model and return the maximum a posteriori and credible intervals of the model parameter values inferred from the data.

```
Just a moment while MTB fits a model to your data...

...finished. Now let's view the results:

parameter  MAP estimate  lower CI     upper CI
---------  ------------  --------     --------
g          0.151         0.027        0.623
sd         37.507        13.831       65.613
```

Then it will ask whether you'd like to see the fit:

```
Would you like to see the fit? (y/n):
```

If you respond in the affirmative (by entering the letter `y` and hitting the return key), you'll find an interactive visualization that lets you manipulate the values of the model parameters and see the impact on the predicted distribution of data. As you change the values, notice that the prediction line changes color — it gets less saturated (close to grey) as you choose values that make the data less likely.



Hooray! In just two lines of code, you've performed mixture modeling using the MemToolbox. For now, say no ('n') to the next question the toolbox asks you about seeing more detailed model fit information. We'll return to that in some of the later demos.

Demo 2: Choosing a different model

In the previous example, the toolbox picked a model on your behalf, but you can choose a different model if you would like. A full listing of available models can be found by entering `help MemModels` at the command line. One of the available models extends the standard mixture by allowing a bias term, such that the central tendency of the data isn't fixed at zero. You can access this model by using `WithBias()` together with the `StandardMixtureModel`:

```
>> model = WithBias(StandardMixtureModel);
```

To use this model, first specify your data:

```
>> errors = [-89, 29, -2, 6, -16, 65, 43, -12, 10, 0, 178, -42, 52, 1, -2];
```

Then call `MemFit`:

```
>> MemFit(errors, model);
```

The toolbox will then walk through the same analysis as before using the model that you picked.

Demo 3: Specifying your data as a structure

The two models we have seen cared only about the errors made by the participant on the task. Other models require auxiliary data to make their predictions. For example, the swap model advocated by Bays, Catalao and Husain (2009), which holds that participants sometimes mistakenly report the value of the other items in the display instead of the target item. This model requires not only the errors made on each trial but also the distances of the non-target items from the target value. (The swap model is available in the toolbox under the name `SwapModel`). Thus, to use this model, you need to tell the model the colors of the distractor items, relative to the target color. These data, the errors and the non-target values, should be bundled together into a structure:

```
>> data.errors = [-89, 29, -2, 6, -16 ...
>> data.distractors = [-10, 2, -100, 163, 42 ...
```

For example, this structure indicates that the distractor on the first trial was 10 degrees counterclockwise in color space from the target color, and that the observer chose a color 89 degrees counterclockwise of the correct value.

The help file for each model lists its required fields:

```
>> help SwapModel

  SWAPMODEL returns a structure for a three-component model with guesses and
  swaps. Based on Bays, Catalao, & Husain (2009) model. This is an extension
  of the StandardMixtureModel that allows for observers' misreporting
  incorrect items.

  In addition to data.errors, the data struct should include:
    data.distractors, Row 1: distance of distractor 1 from target
    ...
    data.distractors, Row N: distance of distractor N from target
```

Let's test the model. Rather than making you type out a long vector of errors and a full matrix of distractor values, we'll use one of the datasets that comes loaded with the toolbox. These can be accessed through the function `MemDataset`, which returns a structure array with fields containing different aspects of the data. For example, data set #3 includes a vector of errors and distractor values for each trial (in addition to the set size, `n`, and the delay duration, `time`). This is precisely what you need to use the swap model:

```
>> data = MemDataset(3)

data =
         errors: [1x4000 double]
    distractors: [2x4000 double]
```

```
               n: [1x4000 double]
            time: [1x4000 double]
```

Now, instead of feeding `MemFit` the error vector, you can give it the entire data structure at once:

```
>> MemFit(data, SwapModel);
```

The toolbox will now run through the usual analysis. You may notice some new things in the output of `MemFit`. We'll explain these in demo #4.

Demo 4: Digging deeper into MemFit()

Let's peek under the hood at the functions that underlie `MemFit`. By default, the MemToolbox uses Markov Chain Monte Carlo (MCMC) to do model fitting. MCMC samples parameter values in proportion to how well they describe the data and how well they match the prior. The distribution of these parameter values is used to estimate the true underlying model and to express our confidence in that estimate. In this demo, we'll show you how to read the various plots produced by `MemFit` that relate to MCMC. Once you've mastered the basics from demos 1–3, we recommend the following as a standard workflow when using the toolbox.

Start by loading in a dataset, either yours or one from the toolbox:

```
>> data = MemDataset(1);
```

Now run `MemFit` using the `StandardMixtureModel`. This time, assign the output of `MemFit` to a variable and don't suppress printing with a semicolon.

```
>> fit = MemFit(data, StandardMixtureModel)
```

First, we see the usual histogram, model name, and parameterization:

```
Error histogram:   -180 _________.'._________ +180
          Model:   Standard mixture model
     Parameters:   g, sd
```

There should be no surprises here. In the off chance that you have specified your data in an incorrect format (e.g., by using radians instead of degrees or by coding errors in the range [0,360]), the toolbox will try its best to massage your data into the correct format, always throwing a warning letting you know what it has done. Then the toolbox announces:

```
Just a moment while MTB fits a model to your data...

   Running 3 chains...
```

This refers to the "chains" of Markov Chain Monte Carlo, which you can read about on Wikipedia (http://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo). The default MCMC algorithm used by MemToolbox uses multiple chains whose starting values are specified in the model file (and modifiable), and it continues running them until they have converged to a similar range of parameters. Every 200 steps, the toolbox will update you on its progress:

```
... not yet converged (200)
```

and then at some point, depending on the model, the chains will converge:

```
... chains converged after 400 samples!
```

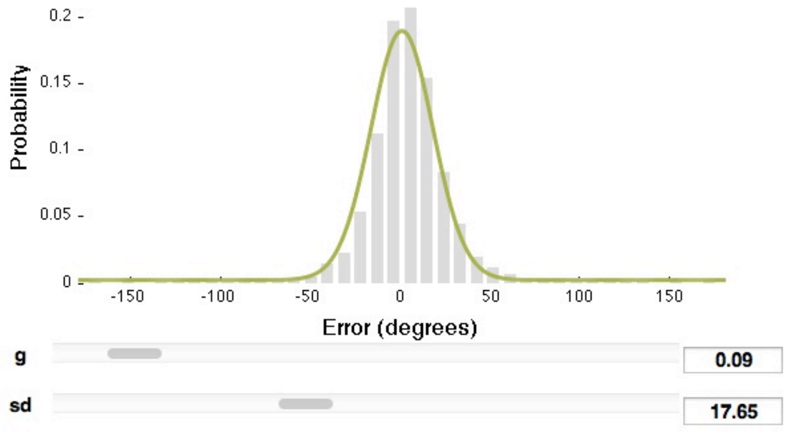
The final estimate is based on samples taken after convergence:

```
    ... collecting 4500 samples from converged distribution
```

The toolbox then analyzes these samples, returning the MAP (maximum a posteriori) estimate and credible intervals:

```
...finished. Now let's view the results: y


parameter   MAP estimate     lower CI       upper CI
---------   ------------     --------       --------
        g          0.093        0.078          0.106
       sd         17.638       16.940         18.377

Would you like to see the fit? (y/n): y
```

Depending on your interests, the maximum a posteriori (MAP) estimates and credible intervals might constitute an answer to your question. Even so, the next step is to visualize the model fit:

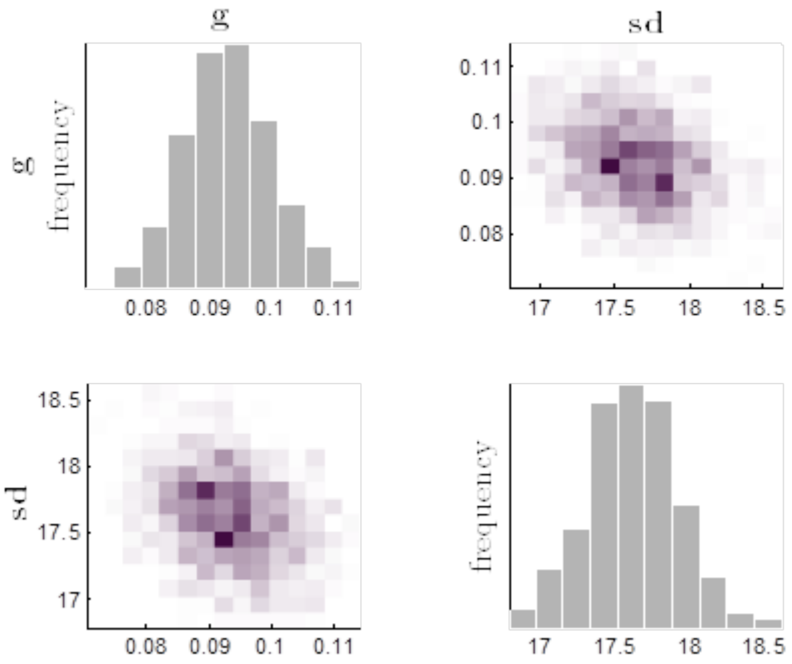


This is mostly useful for familiarizing yourself with the behavior of the model and for recognizing gross inconsistencies between the model and data. At first pass, this one looks okay.

Next, the toolbox will ask:

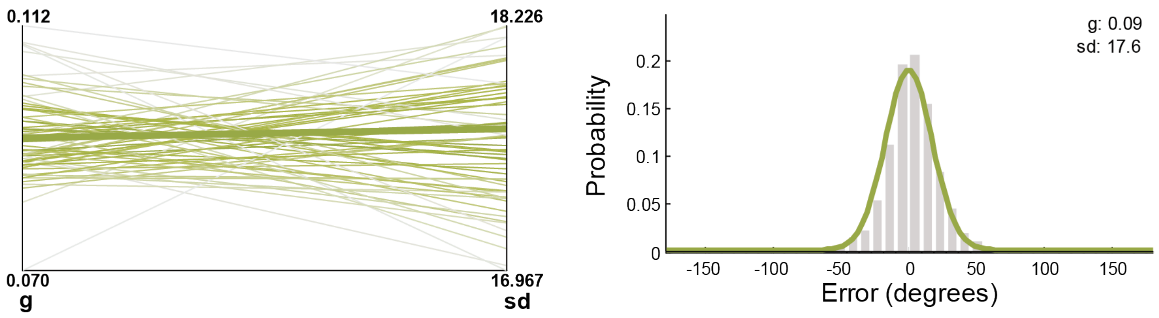
```
Would you like to see the tradeoffs between parameters, samples from the
posterior distribution and a posterior predictive check? (y/n): y
```

Answering in the affirmative will bring up three plots:



The first shows the full posterior distribution for each parameter and a heat map for each pair of parameters, showing how they covary. These plots are useful for diagnosing correlations between parameters and for understanding how much the data constrain the parameters. On the diagonal are plots that show the posterior for an individual parameter — e.g., the distribution for guess rate (`g`) is plotted in the top left corner. We can see that the data make us quite confident that the guess rate is between 0.08 and 0.11. On the off-diagonals are the correlations between the parameters – for example, the top right axis shows guess rate (y-axis) plotted against standard deviation (x-axis). Each row and each column corresponds to a parameter, e.g., the x-axis for all the plots in the second column corresponds to standard deviation.

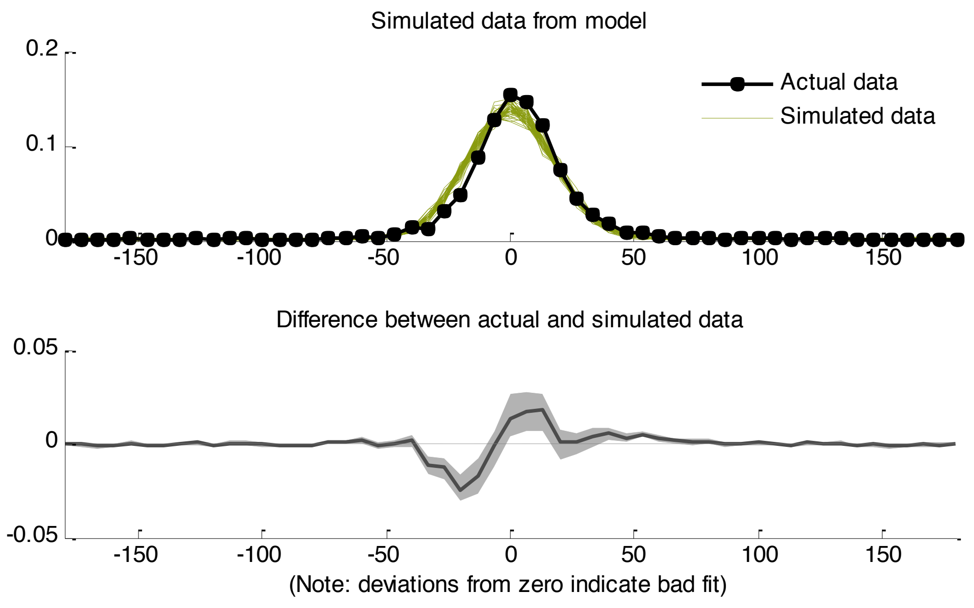
The off-diagonals show heatmaps to visualize correlations between parameters. For example, in the standard Zhang & Luck (2008) mixture model, there is a correlation between the guess rate parameter (`g`) and the standard deviation parameter (`sd`). The data is equally consistent with a slightly higher guess rate and slightly lower standard deviation, or with a slightly lower guess rate and slightly higher standard deviation. This is apparent in the negative slope of the heat map plots in the bottom left and upper right quadrant.



The second plot produced by `MemFit` shows the parameters of the model in a parallel coordinates plot (http://en.wikipedia.org/wiki/Parallel_coordinates), along with a visualization of the parameters fit to the data. Each line on the parallel coordinates plot corresponds to a reasonable set of parameters for fitting the data (e.g., a sample from the posterior on the parameters given the data). The lines are colored such that more likely parameter combinations get darker colors, and less likely parameter values appear more gray. The negative correlation between the parameters of this particular model is indicated by the fact that the lines tend to have slopes that differ from zero (e.g., high values on one parameter give low values on the other, and vice versa). A positive correlation would be shown by the lines being mostly parallel between the left and right sides, meaning high values of one parameter give high values of the other.

With only two parameters, the parallel coordinates plot does not provide as much information as the scatter plots and heat maps in the first figure. However, when more parameters are present, parallel coordinates plots allow you to simultaneously understand how all of the parameters relate to each other, rather than just how pairs of parameters relate; we will explore this later in the tutorial.

This plot is also interactive. Try clicking one of the lines with a high value of guess rate (`g`). The plot of the data on the right will redraw with the new parameter values you chose. Now try clicking on one of the lines with a low value of guess rate (`g`). Can you see why both provide equally good fits?



The third plot shows the *posterior predictive distribution*—the model's residual. The first two plots provide ways of examining which parameter values of the model provide the best fits. This figure instead shows whether or not the model—with its best parameter values—provides a good fit to the data. In particular, we simulate 'fake' data from the posterior of the model and ask whether it looks like the real data. If the model provides a good fit, then the data that is simulated using the best fit parameters (the posterior) will closely resemble the actual data. If the model is systematically wrong in some way, this should be (visually) evident as a mismatch between the simulated and actual data. The top of the plot shows the data in black, and the simulated data in green. They should be on top of each other if the model provides a good fit. The bottom of the plot shows the difference between the two, both on average (grey line) and with 95% confidence intervals (grey shading). Any regions where the gray shading does not include zero are regions where the model is systematically incorrect in its predictions.

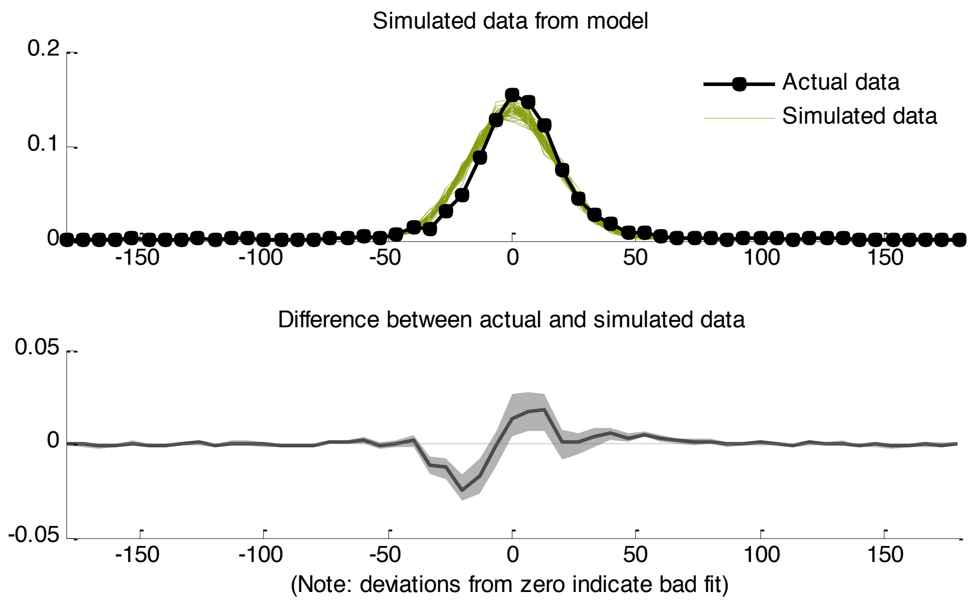
The current data shows an example of a poor fit — this observer's data is clearly shifted to the right of the model data. We'll explore this next in the section on model comparison.

Demo 5: Model comparison

In this demo we'll cover model comparison. Let's start by using the standard mixture model on the first built-in dataset, as in Demo 4:

```
>> data = MemDataset(1);
>> model1 = StandardMixtureModel();
>> MemFit(data,model1)
```

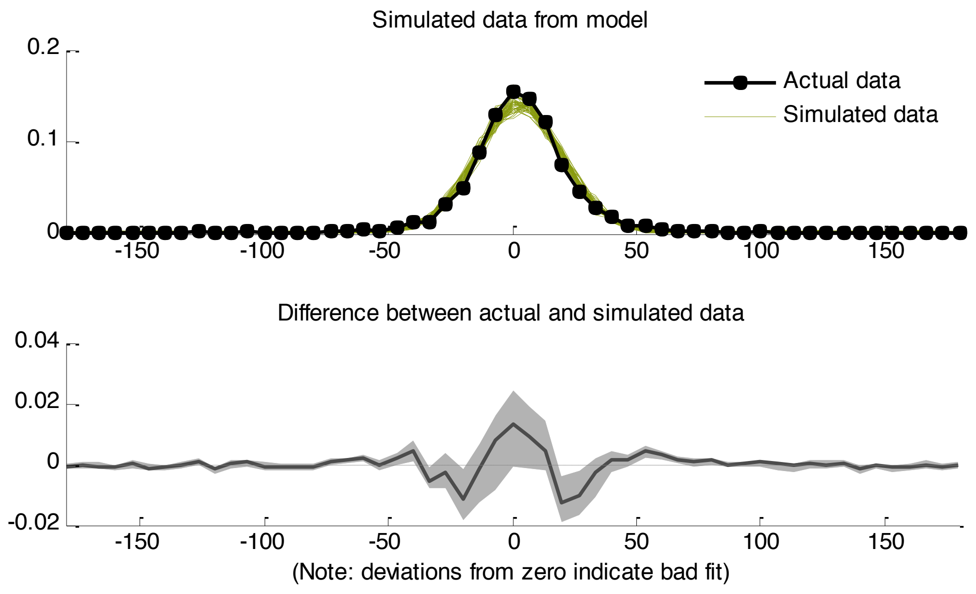
We can see that the residuals for this model show that it does not fit well:



In particular, it looks like the model overestimates counterclockwise error and underestimates clockwise errors. This particular subject had a clockwise-bias in their responses, and the `StandardMixtureModel` is always centered around zero. To fix this, we can use a different model. In particular, we can use the `StandardMixtureModel` model, but allow a bias parameter to shift the model's center point by using the `WithBias` wrapper function:

```
>> model2 = WithBias(StandardMixtureModel);
>> MemFit(data,model2)
```

This model fits much better than the one with no bias:



However, we can see that this model also seems to be systematically incorrect. In particular, the data appear to be ‘peakier’ than this model suggests they should be (for a peakier model, see the `VariablePrecisionModel`). Even so, this model still appears to be an improvement on the model that assumes no bias –the gray shading is much closer to encompassing the zero line than it was in the previous model. How can we quantify this? MemToolbox has built-in model comparison metrics. In particular, simply calling `MemFit` with more than one model automatically compares them:

```
>> MemFit(data, {model1, model2})
```

MemToolbox compares the models using six different metrics, including the log likelihood, AIC, AICc, BIC, Bayes factor (and posterior odds of the models) and DIC. It also provides (brief) ideas of what direction is preferable for each metric.

The MemFit command should give the following output:

```
You've chosen to compare the following models:

        Model 1:   Standard mixture model
     Parameters:   g, sd

        Model 2:   Standard mixture model with bias
     Parameters:   mu, g, sd

Just a moment while MTB fits these models to your data...
```

Initially the toolbox will show you the log-likelihood, AIC, AICc and BIC:

```
Computing log likelihood, AIC, AICc and BIC...

The log likelihood of the parameters given the data.


model  Log likelihood
-----  --------------
 1     -14429.61
 2     -14396.70
1:2        -32.91
Preferred model: 2 (Standard mixture model with bias)

The Akaike Information Criterion is a measure of goodness of fit that
includes a penalty term for each additional model parameter. Lower AIC
denotes a better fit. To compare models A:B, look at the difference AIC(A) -
AIC(B). Positive values provide evidence in favor of model B, negative in
favor of model A.

model  AIC
-----  ---
 1     28863.23
 2     28799.41
1:2        63.82
Preferred model: 2 (Standard mixture model with bias)
```

```
The corrected Aikaike Information is the same as the AIC, but it includes a
correction for finite data. It can be interpreted in the same way.

model  AICc
-----  ----
 1     28863.23
 2     28799.42
1:2        63.81
Preferred model: 2 (Standard mixture model with bias)

The Bayesian Information Criterion is similar to AIC, with different
assumptions about the prior of models, and thus a more stringent penalty for
more complex models.

model      BIC
-----      ---
 1     28875.32
 2     28817.55
1:2        57.77
Preferred model: 2 (Standard mixture model with bias)
```

You will then be asked if you wish to compare the models using DIC. Answer ‘y’.

```
Would you like to compute the DIC (note that this can be slow,
since it requires running MCMC on each model)? (y/n): y

Computing DIC...

Comparing 2 models by DIC:
- Sampling from model 1: Standard mixture model
- Sampling from model 2: Standard mixture model with bias

The Deviance Information Criterion is a
generalization of the AIC and BIC that includes a
penality for the effective number of parameters,
estimated from the dispersion in the posterior of
the models.

model      DIC
-----      ---
 1     28869.51
 2     28805.39
1:2        64.12
Preferred model: 2 (Standard mixture model with bias)
```

You will then be asked if you wish to compare the models using Bayes factors (‘y’):

```
Would you like to compute an approximate Bayes Factor? Note that the Bayes
Factor is heavily dependent on the prior in order to understand how flexible
each model is; it is thus important that before examining Bayes factors you
carefully consider the priors for your models. If you wish to specify a more
concentrated prior to be used for Bayes factor calculation but not for
inference, you can specify a model.priorForMC in addition to a model.prior.
Also note that Bayes Factor calculations are slow. (y/n): y

Computing Bayes Factors...
```

```
    Calculating posterior samples for model 1: Standard mixture model
    Calculating posterior samples for model 2: Standard mixture model with bias
    Calculating likelihoods for model 1: Standard mixture model
    Calculating likelihoods for model 2: Standard mixture model with bias

  When comparing models A:B, the log Bayes factor for model A is the change in
  log odds in favor of that model after seeing the data, with positive values
  ruling in favor of model A. Typically, a log Bayes factor between 0 and 0.5
  is not worth more than a mention, one between 0.5 and 1 is substantial
  support, one between 1 and 2 is strong support, and one above 2 is decisive.

  model      Log Bayes factor
  -----      ----------------
  1:2        -13.29

  Computed from the Bayes factor, this gives the posterior odds of each model,
  a measure of degree-of-belief in each after having seen the data.

  model      Posterior odds
  -----      --------------
   1     0.00
   2     1.00
  Preferred model: 2 (Standard mixture model with bias)
```

Note the warning you are given that you should consider the priors on your parameters (or specify new ones in the form of the `.priorForMC` function) in order to usefully compute Bayes factors. The prior conveys which parameter values are thought to be reasonable, and specifying it can be as straightforward as setting upper and lower bounds (for example, bounding the guess rate between 0 and 1). Both DIC and the Bayes factor have the disadvantage of depending strongly on the prior, since the only way to measure how flexible a model is without simply counting up the parameters is to quantify the range of data it could have predicted a priori. For the purposes of exploratory data analysis, it is common to use a noninformative or weakly informative prior that spreads the probability thinly over a swath of plausible parameter values. Following this tradition, the toolbox uses the Jeffreys prior, a class of noninformative priors that are invariant under reparameterization of the model. However, the Bayes factor heavily penalizes flexibility, and uses the prior to understand how flexible a model is. So, for example, the model with a bias could have predicted just about any graph of errors — as specified, a bias of 80 or 130 degrees is equally likely as a bias of only 5 degrees — so it is penalized heavily for its complexity. This is probably not a reasonable prior for a bias parameter, since in reality the bias parameter might be expected to vary only within a small range near zero. Thus, with the current priors the Bayes factor is overpenalizing the model with a bias.

Specifying more reasonable priors for parameters is up to the user of the toolbox, since it depends heavily on your particular paradigm. For example, if you were running a standard color working memory experiment with set size 3, it might be reasonable to specify a prior like so:

```
model = StandardMixtureModel();
model.priorForMC = @(p) (betapdf(p(1), 1.25, 2.5) * ... % for g
                         lognpdf(deg2k(p(2)), 2.5, 0.5)); % for sd
```

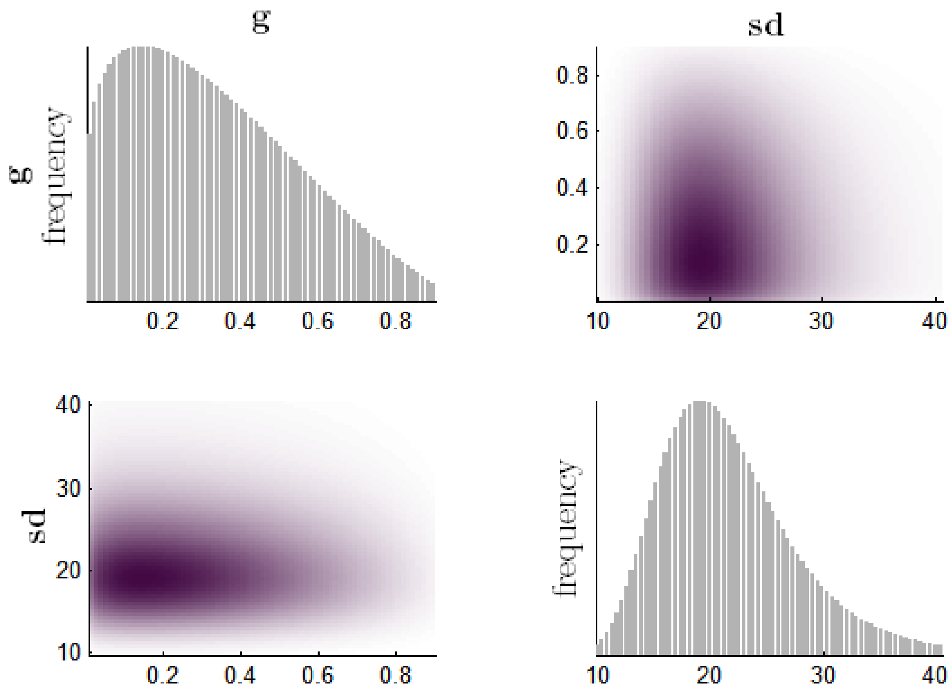
which says something like: in advance of the data, it is reasonable to expect a guess rate around 0.20 (and it would be surprising if it was less than 0.05 or more than 0.80), and a standard deviation of ~10–30 degrees would be reasonable as well. To visualize what the priors you are using are, you can use the `PlotPrior` function. For example, to see the default prior for the `StandardMixtureModel()`:

```
>> PlotPrior(model)
```

or, to visualize the priorForMC we specified above:

```
>> PlotPrior(model, 'UseModelComparisonPrior', true)
```

which would show you the prior we just specified:



These plots are interpretable in the same way as the plots of the posteriors – the diagonals show the posterior for each parameter, and the off-diagonal shows the heatmap of parameter correlations. You may also find the `PlotPriorPredictive` function useful; it shows you what kinds of data you would expect from the current prior.

Bayes factors can, of course, be computed with very diffuse priors like those used by default in the toolbox. However, this almost certainly overpenalizes models with more parameters and should be interpreted with this in mind.

Demo 6: Using the parallel toolbox to speed up analysis

The MemToolbox is compatible with the Parallel Computing toolbox. Using it can achieve a big speedup. To use it, run the command `matlabpool open`. You can automate this within your program by opening the pool if it isn't already open:

```
if(~(matlabpool('size') > 0))
 matlabpool open;
end
```

If your machine has multiple processors or multiple cores, then all of the fitting functions — `MCMC`, `MLE`, etc. — will work many times faster once you run `matlabpool`.

Demo 7: Cleaner plots of the posterior

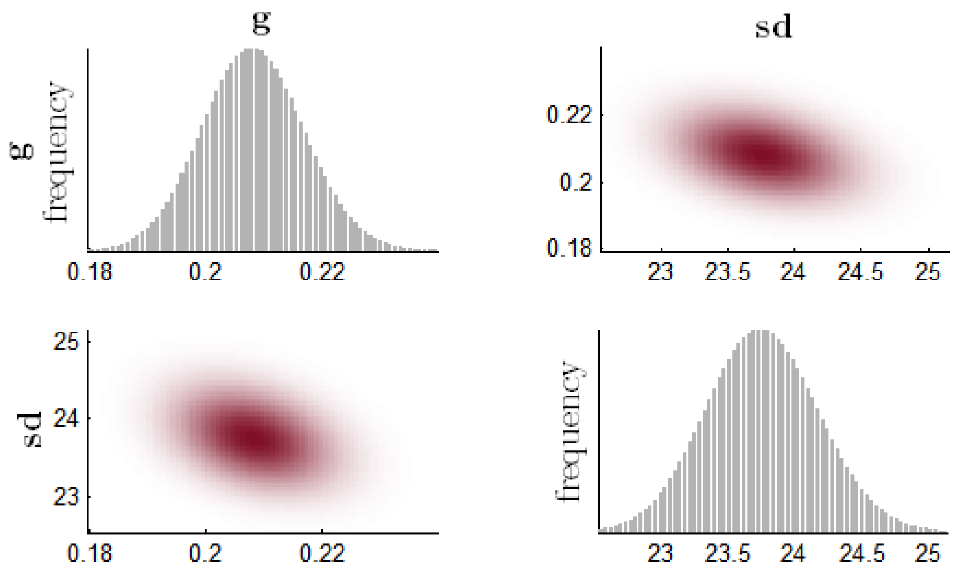
Using MCMC means that we are sampling from the posterior rather than evaluating it directly. Thus, even though the true posterior is almost always a smooth function of the parameters, the plots obtained from the standard `MemFit` call will not always be smooth. We can fix this by using `GridSearch` rather than MCMC to refine our graph. In particular, given the results of a standard call to `MemFit`:

```
>> data = MemDataset(3);
>> model = StandardMixtureModel();
>> fit = MemFit(data, model);
```

We can make a more refined version of the first figure from Demo 4:

```
>> fullPosterior = GridSearch(data, model, ...
     'PosteriorSamples', fit.posteriorSamples)
>> PlotPosterior(fullPosterior, model.paramNames)
```

Then, rather than the noisy sampling-based-plot, we can get a full posterior:

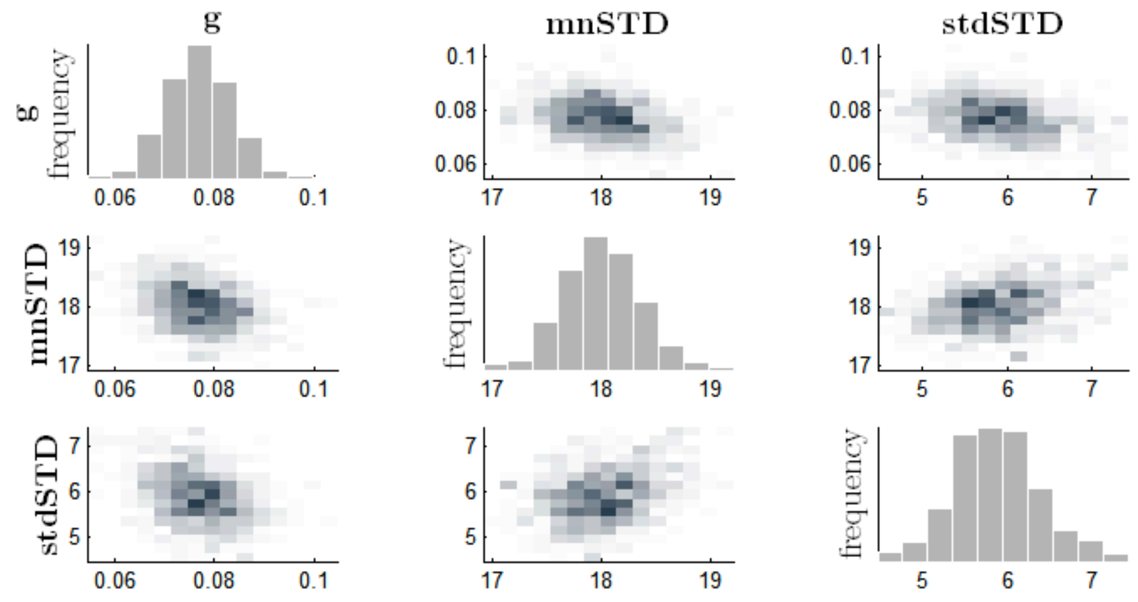


This example also gives you a little bit of a peak under the hood of `MemFit` – in particular, the separate fitting functions (try `help MemFitting`) and plotting functions (`help MemPlots`) of MemToolbox.

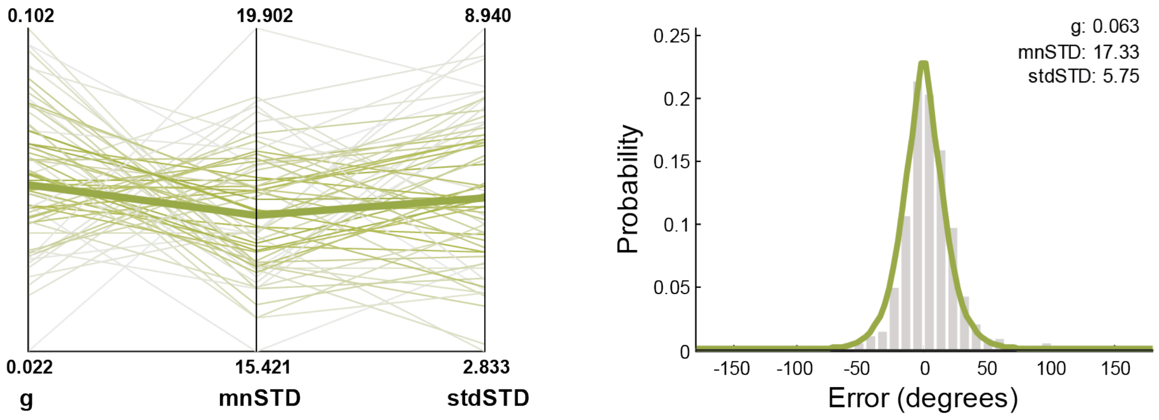
We first fit the data using `GridSearch` and obtain the posterior. We call this `fullPosterior`, since it came from `GridSearch` (although in reality it is only proportional to the posterior, it is not the true posterior distribution); the results from MCMC we generally call `posteriorSamples`. Then we call the plot function `PlotPosterior`, which takes either a `fullPosterior` or `posteriorSamples` as the first parameter, and the names of the model's parameters as the second parameter. More advanced uses of the MemToolbox will generally require separate calls to the MemFitting and MemPlots functions.

Demo 8: Understanding models with more parameters

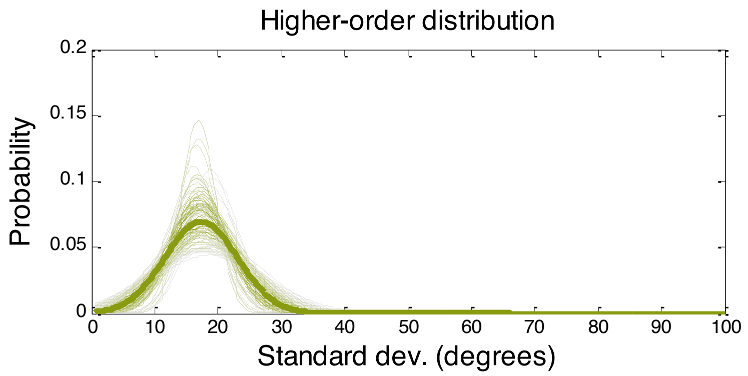
So far we have focused on understanding the posterior of models with only 2 parameters, like that of the standard mixture model. In such models, a heatmap of the two parameters values provides a full understanding of the posterior distribution. For more complex models, understanding the distribution from just the 2-dimensional heatmaps can be more challenging. For example, the default `VariablePrecisionModel` has three parameters, controlling the guess rate (`g`), the mean of the precision (`mnSTD`) and the standard deviation of the precision (`stdSTD`). All of these parameters are slightly correlated with each other:



For example, you can see that high values of guess rate give rise to low values of the mean precision and low standard deviations of the precision. Sometimes understanding the full posterior distribution in this format can be complex. In these cases, the parallel coordinates plot is often helpful, as it shows the correlations between all parameters and the predicted distribution for particular parameter settings.



This example also gives you a peak at another feature of the MemToolbox – custom model plots. When you call `MemFit` with the `VariablePrecisionModel`, it not only creates the graphs you are used to seeing -- the posterior plot, the paralell coordinates plot, and the posterior predictive – it also creates another, fourth plot. This is because the `VariablePrecisionModel` includes a function called `.modelPlot`, which is automatically called by `MemFit`. In this case, the custom model plot shows the higher-order distribution on precision, with the MAP parameters displayed in solid green and other samples from the posterior shown with their color indicating their likelihood:

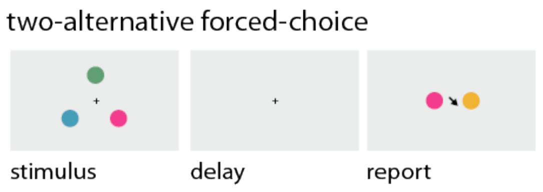


This helps visualize the meaning of the higher-order parameters. Other models include different custom model plots; for example, the swap model shows a distractor-centered error plot (as in Bays et al. 2009 Figure 2), to allow visualization of whether observers are frequently choosing distractor items.

Demo 9: Fitting 2AFC data

Throughout the tutorial we have assumed that you are making use of a continuous report task. However, the MemToolbox also fully supports two-alternative forced-choice tasks for all models.

For example, in a two-alternative forced-choice, observers might be cued to a particular item (for example, with an arrow pointing to the relevant item) and then asked to pick which of two colors the item at that location was.



To make use of such data, we can specify `data.changeSize` and `data.afcCorrect` rather than `data.errors`. `data.changeSize` indicates the difference between the study and test item, in degrees; `data.afcCorrect` indicates whether the observer got the trial correct (1) or incorrect (0).

For example, lets make a fake dataset where we sometimes test observers with a very distant color (exactly the opposite side of the color wheel), and observers get 100% of these trials correct; and we sometimes test observers with a relatively close color (20 degrees away on the color wheel), and observers get exactly 50% of those trials correct:

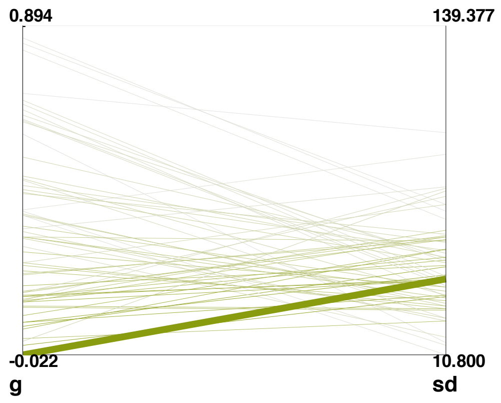
```
>> data.changeSize = [180 180 180 180 180 180 20 20 20 20 20 20];
>> data.afcCorrect = [1 1 1 1 1 1 0 0 0 1 1 1];
>> MemFit(data)
```

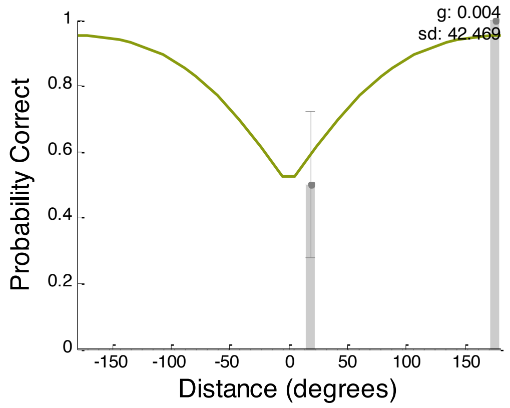
When we run this code, `MemFit` will recognize that we passed in 2AFC data and will automatically adjust the default model to make use of 2AFC data:

```
Warning: It looks like you passed in 2AFC data. Trying to fit with
TwoAFC(StandardMixtureModel()).

Mean percent correct: 0.75
               Model:   2AFC Standard mixture model
          Parameters:   g, sd

Just a moment while MTB fits a model to your data...

...finished. Now let's view the results:

parameter  MAP estimate        lower CI      upper CI
---------  ------------        --------      --------
        g         0.001        0.010         0.817
       sd        42.207        19.006        90.096
```

The MAP estimates should look intuitively reasonable – since we got all the trials with 180 degree differences correct, we probably aren't guessing much; and since we got half of the trials at 20 degrees correct, our standard deviation is probably around 40 degrees. However, having only 6 trials of each type, our credible intervals are very large: it's still reasonable to believe our guess rate might be as high as 0.80, and our SD could be anywhere from 19 to 90 degrees.

The data from this model can also be visualized; the default is to show a bar plot with error bars, along with the model fit:

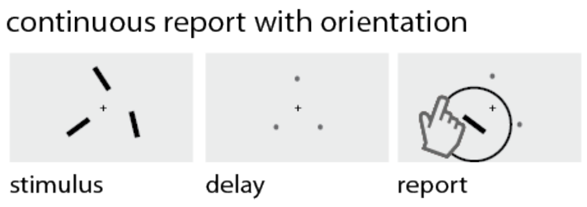




The MemToolbox supports converting any model to a 2AFC model using the `TwoAFC()` wrapper function. Thus, to make a model for a 2AFC task with a `SwapModel`, you could use `TwoAFC(SwapModel)` as your model. However, for models that depend on information about particular displays (like the swap model), the 2AFC wrapper is somewhat slow, since it must compute the likelihood for each display separately.

Demo 10: Fitting orientation data

Sometimes you may wish to use continuous report data on dimensions that span only 180 degrees. For example, oriented lines that do not have a preferred direction. MemToolbox supports this via the `Orientation()` wrapper function. In particular, for such data you should specify your `.errors` as ranging from -90 to 90, rather than -180 to 180, and then wrap your chosen model in `Orientation()` before calling any of the fitting functions:

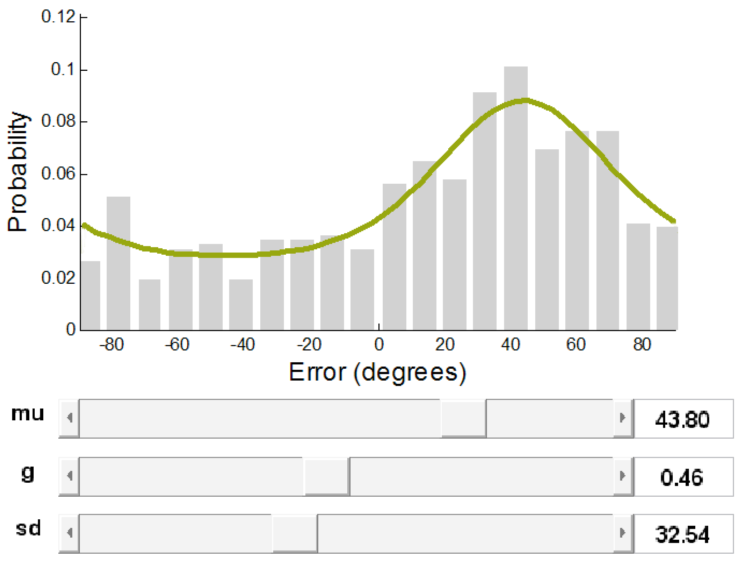


```
>> data.errors = [-12 3 38 27 -29 21 -22 52 -2 -19 21 17 38 6 34 25 44 ...];
>> MemFit(data, Orientation(WithBias(StandardMixtureModel), [1,3]))
```

This will fit a model that wraps from -90 to 90 degrees, and will adjust the first and third parameters of the model to account for this wrapping. These parameters have values in degrees, and so must be adjusted to account for the different wrapping . The model will have “(for orientation)” added to the end of its name.

```
Model:   Standard mixture model with bias (for orientation)
```

The graphs of the model will also be adjusted to range from -90 to 90 degrees:



If you specify data that appears to be orientation data, but the model you include is not wrapped with the `Orientation()` wrapper function, `MemFit` will raise a warning but will proceed anyway.

Demo 11: Fitting models that require data from multiple set sizes

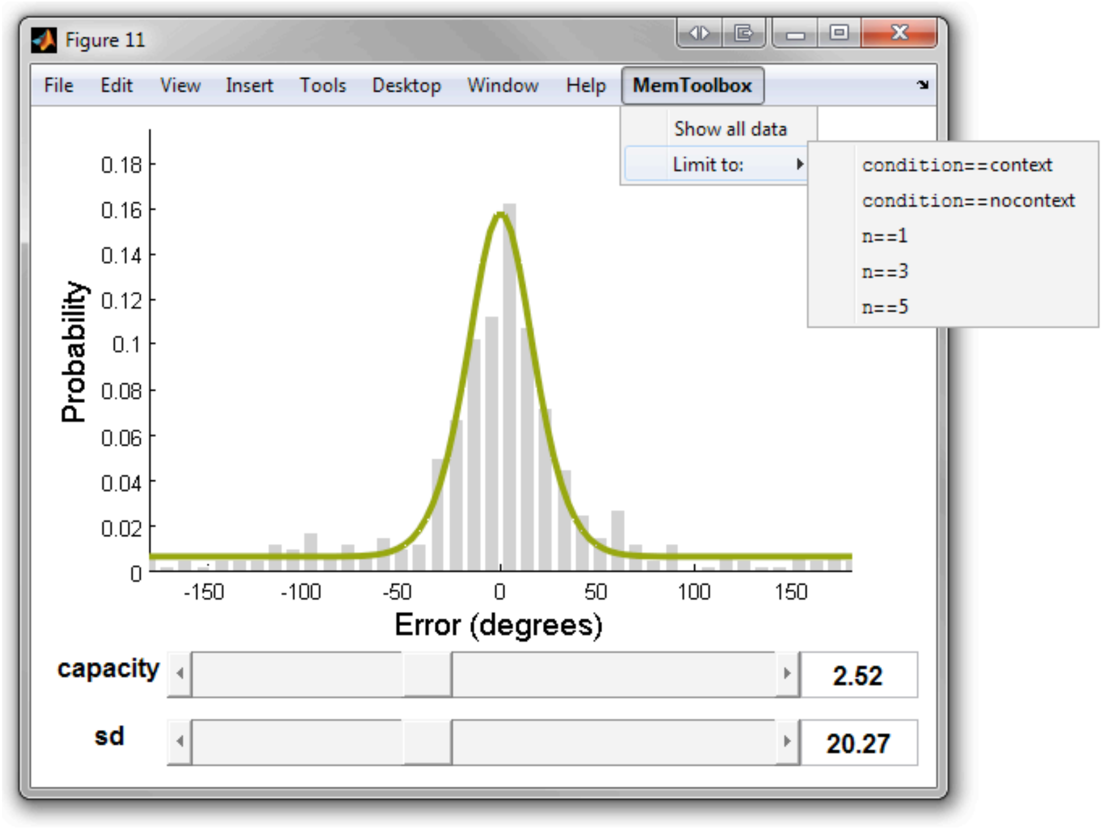
Several of the models in the MemToolbox fit data across multiple set sizes with a likelihood function that depends on the number of items on the display. For example, the slots+averaging model of Zhang and Luck (2008) postulates that there are a fixed number of memory slots available to observers, and thus is committed to the idea that the guess rate and standard deviation should be predictable across set size, based on the number of slots present and the amount of information contained in a slot. To use such models, you must provide a data struct that contains a set size field, `n`. This field says, for each trial, how many items were present. For example:

```
>> data.errors = [-12 3 38 27 -29 21 -22 52 -2 -19 21 17 38 6 34 25 44 ...];
>> data.n = [2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 ...];
>> MemFit(data, SlotsPlusAveragingModel)

Error histogram:   -180 __________.'-__________ +180
          Model:   Slots+averaging model
     Parameters:   capacity, sd


parameter  MAP estimate  lower CI     upper CI
---------  ------------  --------     --------
capacity          2.519     2.325        2.713
      sd         20.265    17.898       24.015
```

This takes the data from all the set sizes and fits a single combined model indicating the best fit for the number of slots, capacity, and for the information content of each slot, sd.

The standard plotting functions of the MemToolbox are useful in this case, but, by virtue of the fact that they plot all of the data together (collapsing across set size), they do not provide the best visualizations of this kind of data. Rather than asking if the model fits the average data across all of the set sizes, it is often more informative to look at the fits and model residuals for each set size individually.

For this reason, all of the plotting functions of the MemToolbox which depend on the data create figures that have an additional menu added to them, labeled **MemToolbox**:



This menu allows you to limit the data being shown to a particular set size (for example, to see what the model predicts at set size 1 vs what it predicts at set size 5).

This menu is automatically generated from your data struct. Thus, any field of the data struct that is the same size as `.errors` (or `.afcCorrect`, for 2AFC data) will automatically be added to this menu so that you may see the fits for a particular value of the parameter.

Demo 12: Analyzing multiple subjects

Typically, the question of interest in working memory research is not about a single observer, but a population: Do those who napped have higher working memory capacities? Is guess rate greater at set size 6 than at set size 3?

When aggregating results from multiple subjects we typically fit a model separately to the data from each subject. To do so in MemToolbox, pass multiple datasets to `MemFit`:

```
  >> datasets{1} = MemDataset(1);
  >> datasets{2} = MemDataset(2);
  >> model = StandardMixtureModel();
  >> fit = MemFit(datasets, model)
```

This is equivalent to simply looping over each subject and calling `MemFit`:

```
    for i=1:length(datasets)
      params(i,:) = MemFit(datasets{i}, model);
    end
```

The toolbox also provides two helper functions that provide an alternative route to fitting data from multiple participants. The `FitMultipleSubjects_MAP` function computes the MAP estimate for each subject and then also then calculates means and standard errors for each parameter for you; there is a similar `FitMultipleSubject_MLE` function that calculates maximum likelihood parameters rather than maximum a posterior parameters (i.e., it does not consider the prior at all).

In addition to fitting separate models for individual subject, the MemToolbox also supports a better technique, although one that is more computationally intensive: fitting a single hierarchical model of all the subjects together. This treats each of the subjects' parameters as samples from a normally-distributed population and uses the data to infer the population mean and SD of each parameter, plus the best fit subject-specific parameters.

Hierarchical modeling can be performed by passing multiple data sets, one per subject, to the `MemFitHierarchical` function:

```
  >> data1 = MemDataset(1);
  >> data2 = MemDataset(3);
  >> model = StandardMixtureModel();
  >> fit = MemFitHierarchical({data1,data2}, model)
```

Peeking under the hood of `MemFitHierarchical`, you will see that it is also possible to use the `Hierarchical` model wrapper function directly. For example, you can create a hierarchical model for 3 subjects using the standard mixture model, you can call:

```
  >> data = {MemDataset(1), MemDataset(2), MemDataset(3)};
  >> hModel = Hierarchical(data, StandardMixtureModel());
```

You can then fit this data with the MAP and MLE functions, or call MCMC on it, e.g.,

```
>> params = MAP(data, hModel);
```

The parameters are “flattened” into a single vector to fit the model; to get back parameters for each subject and for the group mean and group standard deviation, you will want to use the `OrganizeHierarchicalParams` function:

```
>> fit = OrganizeHierarchicalParams(hModel, params);
fit =
    paramsSubs: [3x2 double]
     paramsStd: [0.041056 3.3635]
    paramsMean: [0.14089 18.959]
```

This workflow – calling `Hierarchical()` with the model and data, then fitting it with `MAP`, and then converting the parameters back to understandable fits – is what is done by `MemFitHierarchical`. However, doing it one step at a time allows you to choose to use `MLE` or `MCMC` instead of `MAP`, and also allows you to use hierarchical models in doing model comparison (by passing two hierarchical models to `MemFit`, for example).

Often it is desirable to compare parameters across conditions to determine whether the population mean in one condition is different than the population mean in another condition. Within the hierarchical modeling framework, this question can be addressed by simply testing whether the posteriors for the means of the populations overlap for the two conditions. In other words, we can ask "does the uncertainty about the population mean in condition 1 overlap with the uncertainty about the population mean in condition 2?" If they do not overlap, we would conclude that we have high confidence that the population means are different. To ask about the uncertainty in parameters, we have to use `MCMC` on the hierarchical model. Using `MCMCSummarize`, we can then get the upper and lower credible intervals for each parameter’s mean:

```
>> posteriorSamples = MCMC(data, hModel);
>> lowerCredible = MCMCSummarize(posteriorSamples, ‘lowerCredible’);
>> upperCredible = MCMCSummarize(posteriorSamples, ‘upperCredible’);
>> lowerMean = OrganizeHierarchicalParams(hModel, lowerCredible)
lowerMean =
    paramsSubs: [3x2 double]
     paramsStd: [0.0286 2.3228]
    paramsMean: [0.0152 14.1632]

>> upperMean = OrganizeHierarchicalParams(hModel, upperCredible)
upperMean =
    paramsSubs: [3x2 double]
     paramsStd: [0.5417 24.5397]
    paramsMean: [0.3389 38.4790]
```

Thus, for example, the 95% credible interval on guess rate for the population (`paramsMean`) lies between 0.0152 and 0.3389. By comparing these values across conditions, we can examine whether there are reliable differences in a particular parameter.

Demo 13: Comparing data across conditions

Comparing data across conditions can be done much the same way as analyzing multiple subjects: independently fit models for each condition and then look at how the parameters differ. The MemToolbox offers several helpful functions to facilitate this process. In particular, the function `SplitDataByCondition()` takes a data struct with a .condition field as input, and returns a cell array of data structs, one per condition. Thus, if you had data like:

```
>> data.errors = [-89, 29, -2, 6, -16, 65, 43, -12, 10, 0, 178, -42, 52, 1];
>> data.condition = [1 1 1 1 1 1 1 2 2 2 2 2 2 2];
>> [datasets, conditionOrder] = SplitDataByCondition(data);
```

Gives:

```
>> conditionOrder
conditionOrder =
     1     2

>> datasets{1}
ans =
       errors: [-89 29 -2 6 -16 65 43]
    condition: [1 1 1 1 1 1 1]

>> datasets{2}
ans =
       errors: [-12 10 0 178 -42 52 1]
    condition: [2 2 2 2 2 2 2]
```

You can then loop over these datasets, fitting each separately. The function `SplitDataByField` is the more general version of `SplitDataByCondition`: rather than `SplitDataByCondition`, you could call:

```
>> [datasets, conditionOrder] = SplitDataByField(data, 'condition');
```

which would give you equivalent results (and also allow you to split by `n` or any other dimension of your data).

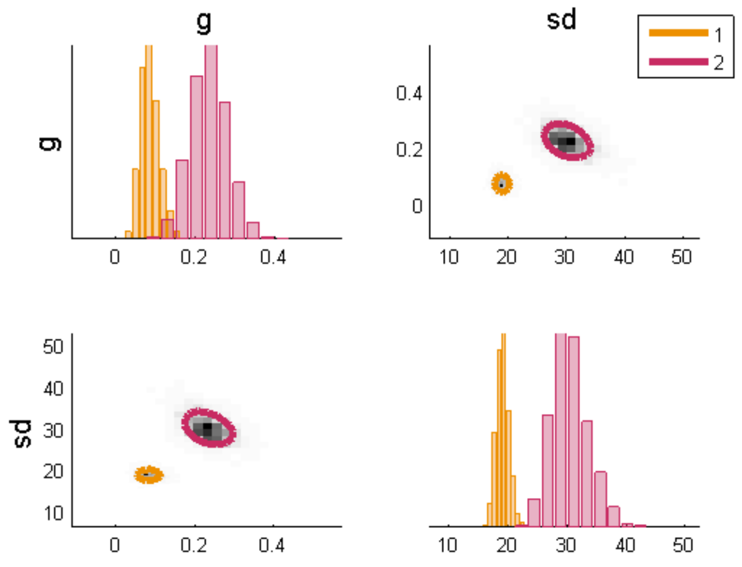
Having split up the datasets and fit them independently, for example, like so:

```
>> for i=1:length(datasets)
>>   posteriorSamples{i} = MCMC(datasets{i}, StandardMixtureModel);
>> end
```

You might then want to compare not only the best fit parameters for each model, but also the full posterior distributions. To do this, you can make use of the `PlotPosteriorComparison()` function, which plots two posterior distributions:

```
>>PlotPosteriorComparison(posteriorSamples, model.paramNames)
```

For example, if we use data with 200 trials/condition where condition 1 has from `g`=0.1 and `sd`=20, whereas condition 2 has `g`=0.3, `sd`=30, we might get something like:

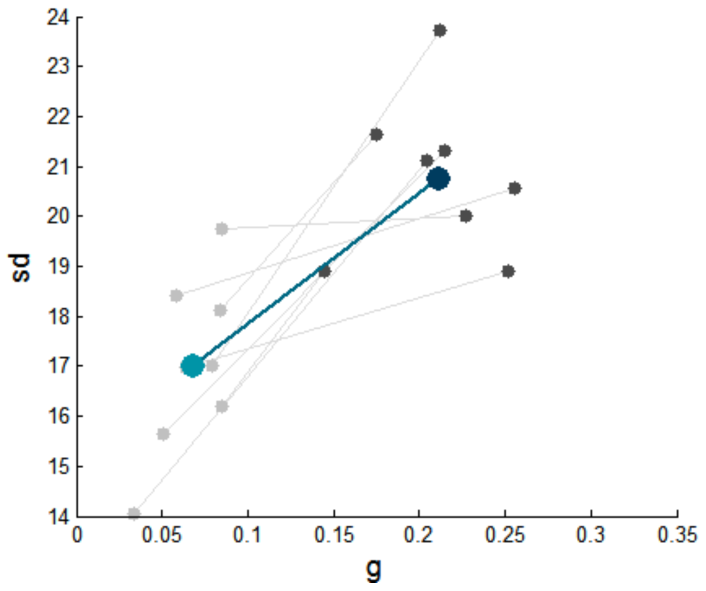


Finally, you might wish to compare many subjects' best-fit parameter values for different conditions. The MemToolbox includes one function designed for visualizing how parameter values differ between different conditions. For example, imagine you have calculated `MAP` parameters for 2 conditions for a set of 8 subjects:

```
>>for sub=1:8
>>[datasets, conditionOrder] = SplitDataByCondition(data{sub});
>>  for cond=1:length(datasets)
>>    params{cond}(sub,:) = MAP(datasets{cond}, model);
>>  end
>>end
```

Now you wish to visualize how the `g` and `sd` parameters differ between these conditions. The MemToolbox includes a function called `PlotShift` that is designed for this scenario. Similar to the heatmap plots in the posterior distribution (showing `g` on one axis and `sd` on the other), the `PlotShift` function shows you how the best fit parameters for `g` and `sd` differ between conditions using lines drawn between subjects' points from the two conditions:

```
>>PlotShift(params{1}, params{2}, model.paramNames);
```



The light gray dots show the parameter values for each subject in condition 1; the dark gray dots show the parameter values for each subject in condition 2. The blue dots show the means per condition. Lines connect points that derive from the same subjects. The direction of the lines in this figure conveys that the guess rate and standard deviation values were higher for condition 2 for each participant.

This visualization uses a format much like that used by `PlotPosterior` and `PlotPosteriorComparison` to see how the parameter values compare *across* subjects,. This allows you to examine whether the parameters seem to be correlated with each other within a condition (for example, the light gray dots show that both sd and g seem to be higher in the same subjects), and how the variance across conditions relates to the variance within a condition (e.g., the guess rates for condition 1 are all smaller than the guess rates for condition 2, and seem to be more tightly clustered than in condition 2).

Demo 14: Fitting a model with one set of data and evaluating it with another

Throughout the tutorial, we have fit models to the same data we have then evaluated the models with. However, by using the fitting and evaluation functions separately, it is relatively straightforward to fit a model with one set of data and examine it's fit on a different set. For example, you might wish to fit the models that generalize across set size (e.g., `SlotPlusResourcesModel` vs `ContinuousResourceModel`) using only set sizes 1 and 3, and then ask how well the results generalize to set size 5.

To facilitate this, the MemToolbox has two functions designed for creating "hold-out" sets of data: `GetDataByField()` and `RemoveDataByField()`. For example, if you have a data struct that contains errors from set sizes 1, 3, and 5, and want to fit using only the data from set sizes 1 and 3, you might call:

```
>>dataWithout5 = RemoveDataByField(data, ‘n’, 5);
```

Then you could fit both models to get posterior distributions:

```
>>continuousResourceSamps = MCMC(dataWithout5, ContinuousResourceModel);
>>slotPlusResourceSamps   = MCMC(dataWithout5, SlotsPlusResourcesModel);
```

And then show a posterior predictive distribution for each model for the data from set size 5, to examine which provides a better fit:

```
>>dataSetSize5 = GetDataByField(data, ‘n’, 5);
>>PlotPosteriorPredictiveData(ContinuousResourceModel,
                               continuousResourceSamps, dataSetSize5);
>>PlotPosteriorPredictiveData(SlotsPlusResourcesModel,
                               slotPlusResourceSamps, dataSetSize5);
```

or perhaps compute some model comparison metrics by passing in these samples.

Demo 15: Customizing plots and figures

Many of the figures used by MemToolbox come with a default color scheme and positioning. In addition, most of the functions create new figure windows when you call them (with the notable exception of `PlotModelFit`, which plots into the current axis by default; `PlotModelFitInteractive`, however, creates a new figure by default).

These options are customizable. All of the plotting functions take an optional parameter ‘`NewFigure’`, which, if set to false, causes them to plot into the current figure window and/or current axis rather than creating a new figure. In addition, plotting functions take optional arguments to indicate the color to plot the model (default: green), ‘`PdfColor’`; and all plotting functions that show data accept an argument indicating how many bins to separate the data into, ‘`NumberOfBins’`.

Finally, the toolbox often makes use of a special function for arranging the figures on the screen. This function, `subfigure()`, is the figure-equivalent of `subplot()`. It divides the screen into grids and places figures in particular positions based on this grid. Thus, to divide the screen into a 2 x 2 grid and place the figure with handle figHand into the top left quadrant, you can call:

```
>>figHand = PlotData(data);
>>subfigure(2,2,1,figHand);
```

Demo 16: Advanced MCMC diagnostics

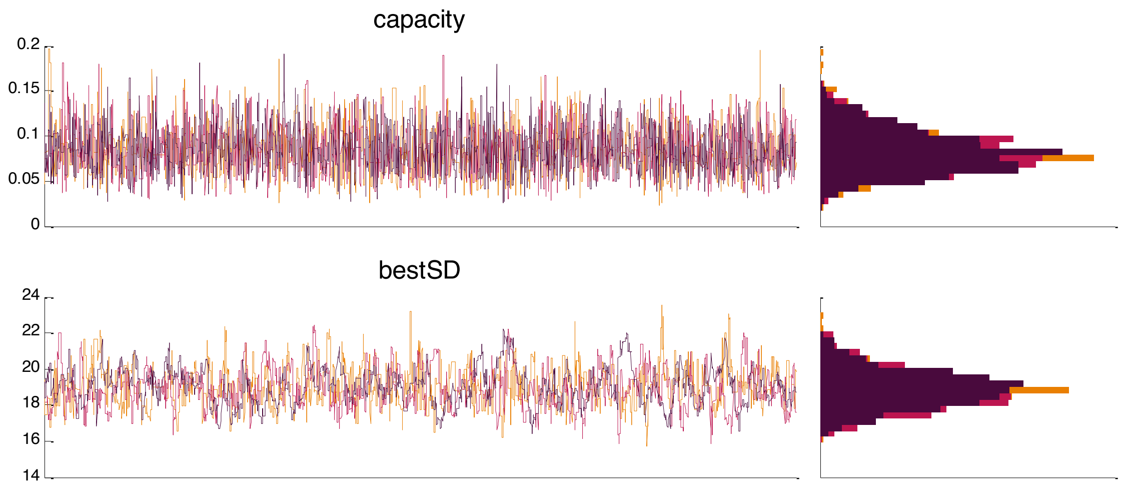
The `MCMC` function of the MemToolbox is designed to require as little intervention as possible. It automatically detects correlations between parameters and adapts the proposal distribution (which is always a multivariate normal); decreases the distance of proposals if too few are accepted; and detects convergence between different chains (and thus convergence to the posterior distribution) using the method of Gelman and Rubin (1992).

Nevertheless, it is entirely plausible that sometimes you will run into a model where the standard MCMC method does not work as expected. One relatively simple solution to many problems with convergence is to turn off the automatic convergence detection and simply run a large number of burn-in samples. You can do this by passing the following optional parameters (for 5,000 burn-in samples and 15,000 samples after burn-in):

```
    ... 'ConvergenceVariance', Inf, 'BurnInSamplesBeforeCheck', 5000,
            ‘PostConvergenceSamples', 15000, ...
```

This may be useful if the variance calculation is wrong for one of your models. For example, if you have data where the bias (shift) parameter is near 180 degrees, then if some chains settle near -180 and some near +180, this should be considered low-variance, but MCMC will consider it high variance. In this case it may be useful to turn off the automatic convergence detection and simply run a large number of MCMC samples.

The toolbox also includes a PlotConvergence function, which takes a posteriorSamples struct (from MCMC) and plots each MCMC chain separately, so you can check for convergence by eye. The different chains appear in different colors.



If you collected 3000 samples post convergence, for example, with 3 chains, the left-side plots will show lines with 1000 points in them, one for each chain. These plots show the trace of the chain – what value each parameter had at each time point. These plots should show that the different chains are directly on top of each other; the chains should have converged such that they generally agree on the range of the parameter and how often they visit each value. The right-side plots show a histogram of all the values from the left side, indicating how often each chain visited each value. These histograms should also not be separated according to which chain they came from. In addition, the trace plots, on the left, should not show a large degree of autocorrelation or a tendency to get “stuck” in a particular value for long stretches (horizontal lines). If they do, this indicates the chain was frequently getting stuck on a particular value and so might not give a good sense of the posterior distribution.

This plot thus allows you to detect if your model is not properly converging. If this is the case, you may consider either reparameterizing the model (see Demo 17, below), or simply running more chains and/or more burn-in samples.

Demo 17: Creating your own model

Creating your own model in MemToolbox is relatively straightforward. The main requirements are to (1) specify the parameters and their ranges, and (2) implement a likelihood function that says how likely the data is given a particular parameter setting. These requirements are nearly the same as those to use the built-in Matlab `mle()` or `fminsearch()` functions. There is an example model in the `MemTutorials` folder, `TemplateModel,`with a thorough set of comments that may be used as an example.

A model struct requires the following (standard) fields:

`.name` – the name of the model
`.paramNames` – a list of parameters
`.lowerbound` – the lowest possible value for each parameter
`.upperbound` – the highest possible value for each parameter
`.start` – some reasonable initial values for the parameters
`.pdf` – the likelihood function for the data, given the parameters

In addition, the model requires one field that is unique to MemToolbox:

`.movestd` – the amount the search technique should move for each parameter

Finally, it is possible to specify a number of optional fields:

`.prior` – a prior for the parameter settings
`.priorForMC` – a prior used only in computing Bayes Factors
`.generator` – to produce samples from the model, given some parameters

For a more thorough discussion of each of these fields and for examples of possible settings for them, please see the example `TemplateModel` included in the `MemTutorials` folder:

```
>> edit TemplateModel.m
```

Demo 18: Checking a model and sampling data

The MemToolbox includes a number of tools to check models and ensure they are being properly fit and to ensure that the correct models are being recovered by the model comparison metrics. Many of these functions are located in the `MemTests` folder. For example, the function `TestSamplingAndFitting` takes as input a model and the parameter values for that model, plus a list of how many trials to simulate data from. It will then return the best fit parameters and credible intervals for fitting the model on simulated data. You could use this to calculate, for example, whether the model's credible intervals are well calibrated; or whether the model seems to properly recover its parameters:

```
>>model = WithBias(StandardMixtureModel);
>>paramsIn = {0, 0.1, 10}; % mu, g , K
>>numTrials = [10 100 1000];
>>numItems = [3 5]; % simulate half trials at set size 3 and half at 5
>>[paramsOut, lowerCI, upperCI] = ...
        TestSamplingAndFitting(model, paramsIn, numTrials, numItems);
```

The function `TestAllModels` runs code much like this on each of the included models to ensure the parameters of these models are being properly recovered. Similarly, `TestModelComparison` allows you to see under what circumstances the correct model can be reliably recovered from data.

These functions make use of a number of built-in functions that can be helpful to your own model simulations. For example, `SampleFromModel()` takes as input a model and the parameters of that model, and generates data from this model. Thus, to generate 500 samples from the `StandardMixtureModel` with parameters `g`=0.1 and `sd`=20, you could run:

```
  >> simulatedData.errors = SampleFromModel(StandardMixtureModel, [0.1 20], [1
500])
```

Some models require extra information in order to make it possible to sample from them – for example, you cannot sample from a `SwapModel` without saying what the distractors should be for the data you wish to generate. To help create display information, the toolbox provides a `GenerateDisplays` function that takes as input the number of trials to generate and the number of items per trial (as either a scalar or vector whose length is equal to `numTrials`). It also takes a parameter that says how to choose the colors/items on each trial, although currently the only option is to generate them randomly and independently. This information can then be passed to `SampleFromModel` to generate data:

```
>> numTrials = 100;
>> itemsPerTrial = [3 3 3... 4 4 4... 5 5 5];
>> simulatedData= GenerateDisplays(numTrials, itemsPerTrial);
>> params = [0.1, 0.1, 20];
>> errors = SampleFromModel(SwapModel, params, [1 numTrials], simulatedData)
>> simulatedData.errors = errors;
```

We now know this data contains 100 trials sampled from a `SwapModel` with a swap rate of 0.10 and a guess rate of 0.10; we could thus use this as ground truth for checking that parameters are properly recovered or that the `SwapModel` is preferred to the `StandardMixtureModel` when doing model comparison with this data.