MemToolbox tutorial

(This tutorial uses a beta version of the toolbox and was last updated on 8/11/2012.)

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.

Table of Contents

Demo 1: Mixture modeling in just two lines of code.

To start the tutorial we will assume you have collected data (or wish to collect data) using 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 responses and the correct answer in degrees on the color wheel.

The simplest way to use the MemToolbox 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);
```

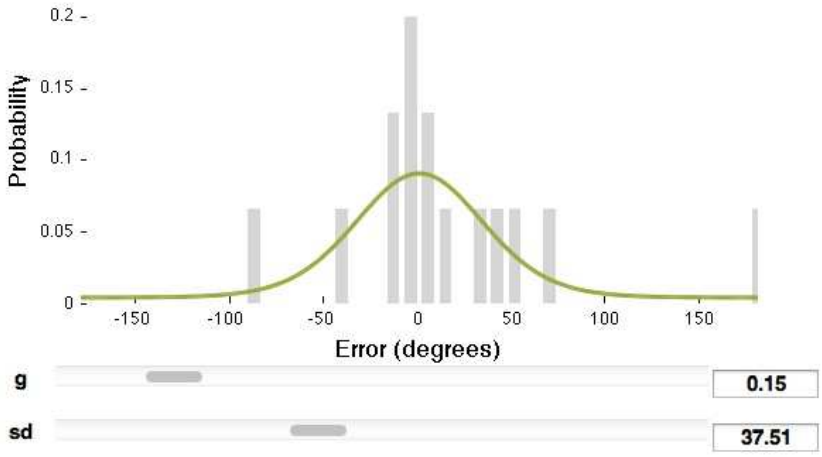
The toolbox will now run through its analysis of your data, showing you a histogram of the errors, the name and parameterization of the model it is fitting to the data, and the maximum a posteriori and credible intervals of the model parameter values inferred from the data. By default, the MemToolbox fits the standard mixture model of Zhang and Luck (2008) to the data.

```
Error histogram:    -180 _________.'_.________ +180
          Model:    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          0.151         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 and more gray 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 get to those 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, requires not only the errors made on each trial but also the values of the non-target items. (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. 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 ...
```

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!
```

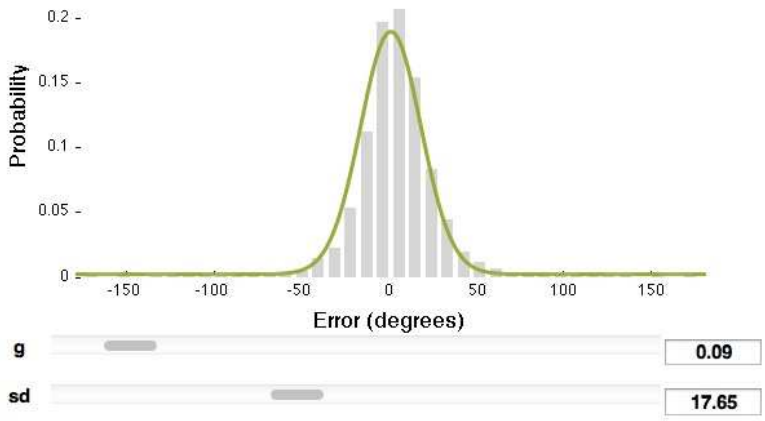
The final estimate is based on samples taken after convergence:

```
 ... collecting 3000 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:

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:



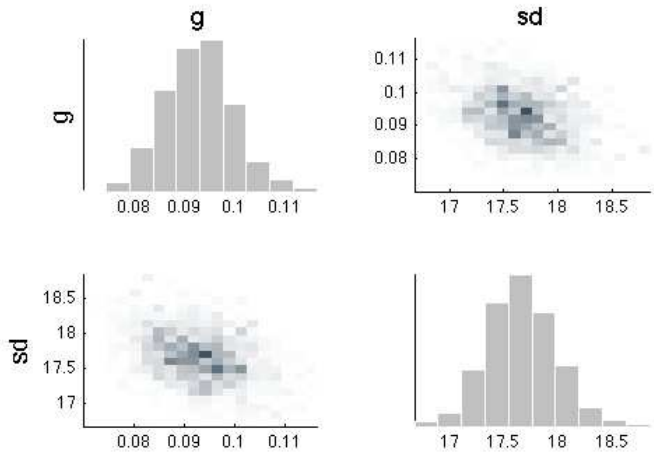
This is mostly useful for familiarizing yourself with the behavior of the model and for recognizing gross inconsistencies between the model and data. At first pass, this one looks okay.

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:

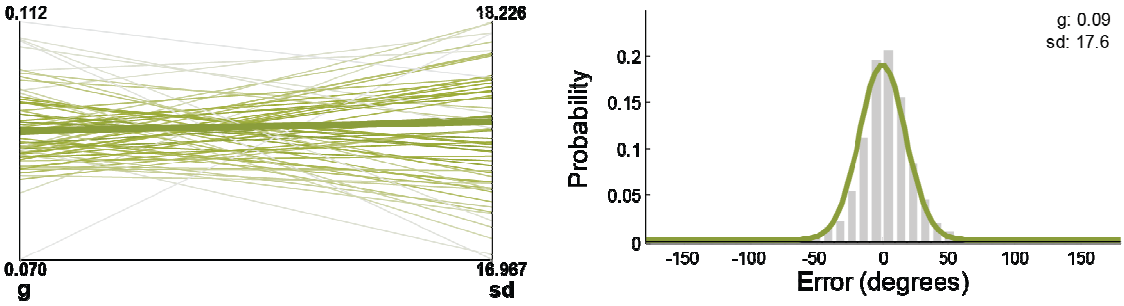
1. The first shows the full posterior distribution for each parameter and a heat map for each pair of them. This can be useful for diagnosing correlations between parameters and for understanding how much the data constrain the parameters:



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.

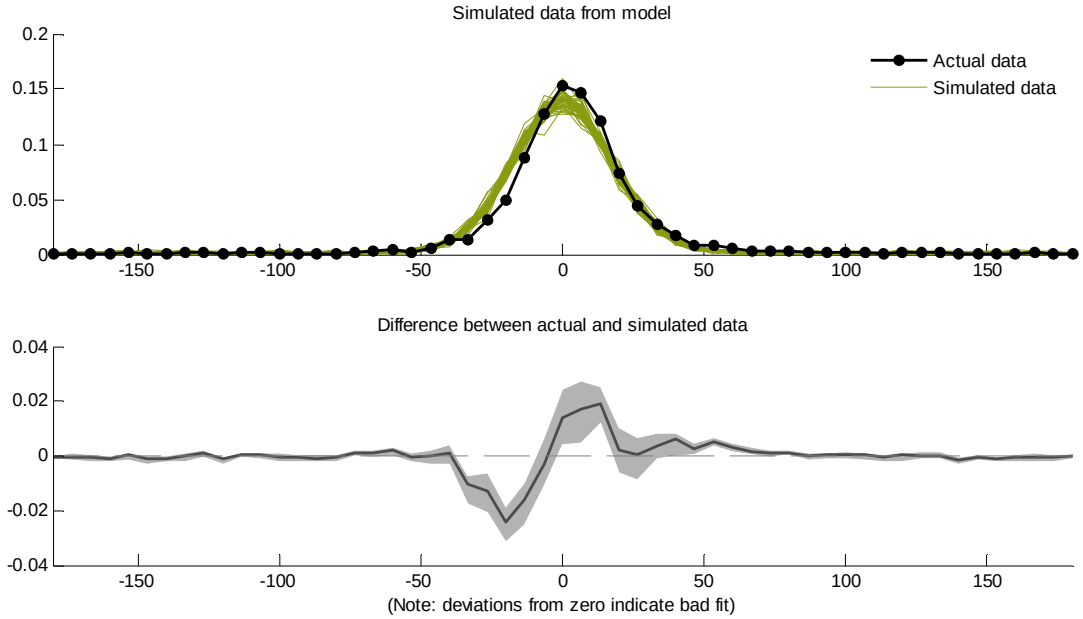
2. The second plot produced by MemFit shows the parameters of the model in a parallel coordinates plot (http://en.wikipedia.org/wiki/Parallel_coordinates), along with a visualization of their fit to the data. Each line on the parallel coordinates plot corresponds to a reasonable set of parameters for 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?

3) 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 (line) and with 95% confidence intervals (gray shading). Any regions where the gray shading does not include zero are regions where the model is systematically wrong.



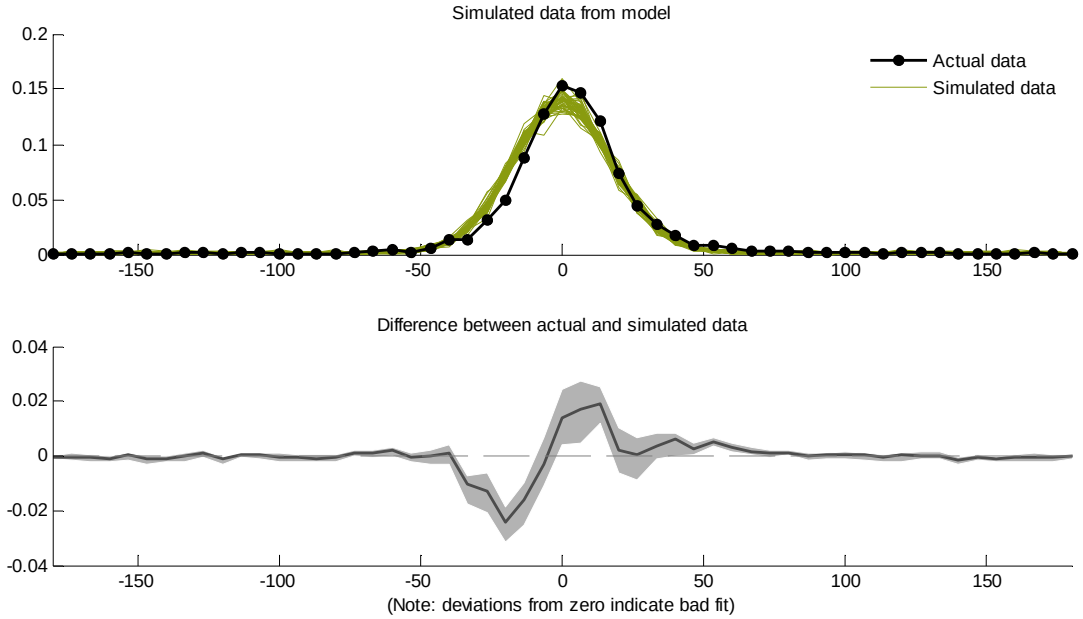
The current data shows an example of a poor fit – this observer's data is clearly shifted to the right of the model data. We'll explore this next in the section on model comparison.

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)
```

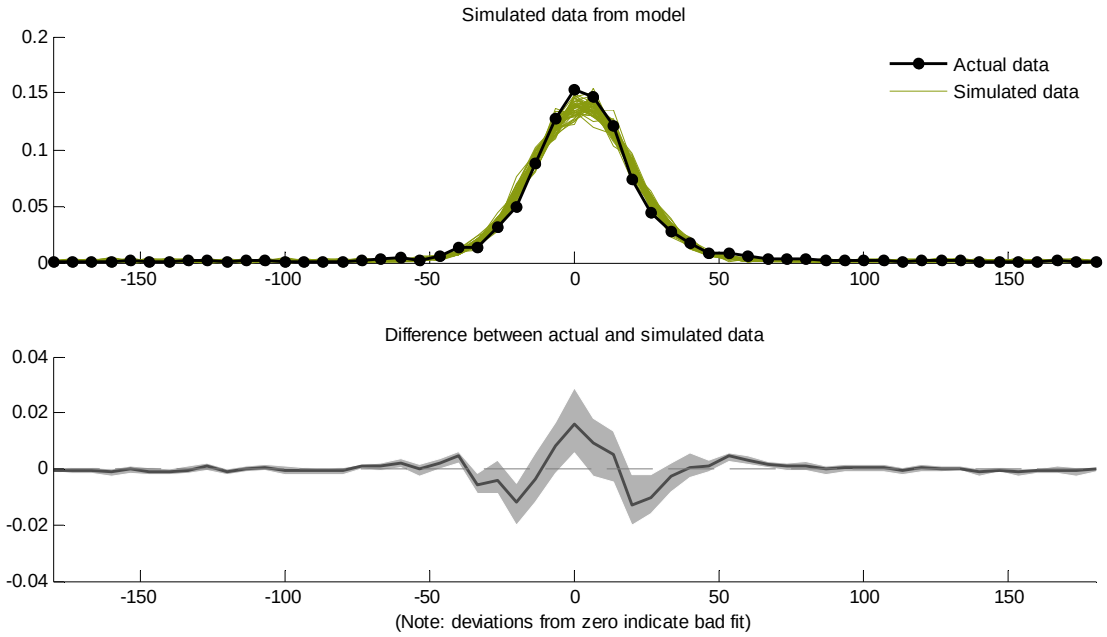
We can see that the residuals for this model show that it does not fit well:



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:

```
>> 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. Even so, this model still appears to be an improvement on the model that assumes no bias. 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})
```

This 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, initially showing you the log-likelihood, AIC, AICc and BIC:

```
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...

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. Answer ‘y’.

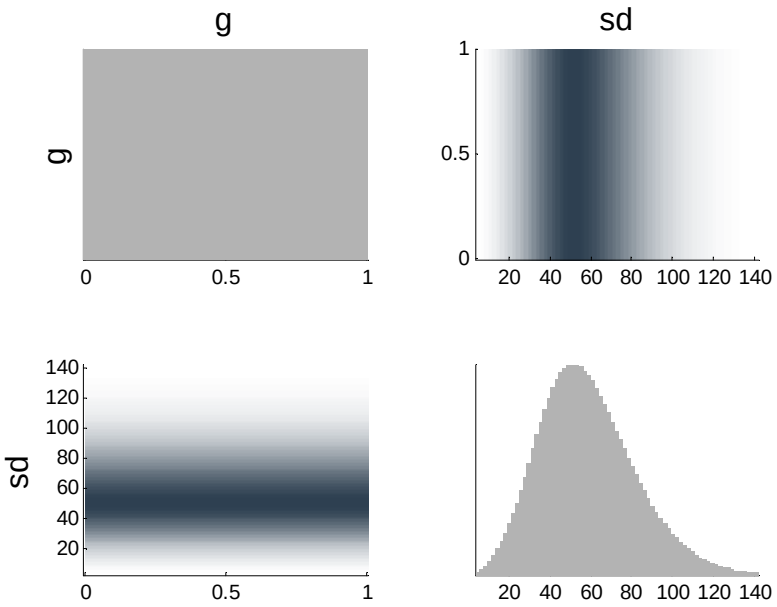
```
Would you like to compute an approximate Bayes Factor? Note that
the Bayes Factor is heavily dependent on the prior in order to understand
how flexible each model is; it is thus important that before examining Bayes
factors you carefully consider the priors for your models. If you wish to
specify 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

  Posterior odds of models:
    0.00   1.00


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 specify priors on your parameters (in the form of the `.priorForMC` function) in order to usefully compute Bayes factors. This is because the automatic Occam's razor of the Bayes Factor heavily penalizes flexibility, such as the fact that 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. This is probably not a reasonable prior for a bias parameter, and so the Bayes factor is overpenalizing the model with a bias. In reality the bias parameter might be expected to vary only within a small range near zero.

To visualize what the priors you are using are, you can use the `PlotPrior` function. For example, the default prior for the `StandardMixtureModel()`:

```
>> PlotPrior(model1)
```



Demo 6: Using the parallel toolbox to speed up the 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: Prettier plots of the posterior

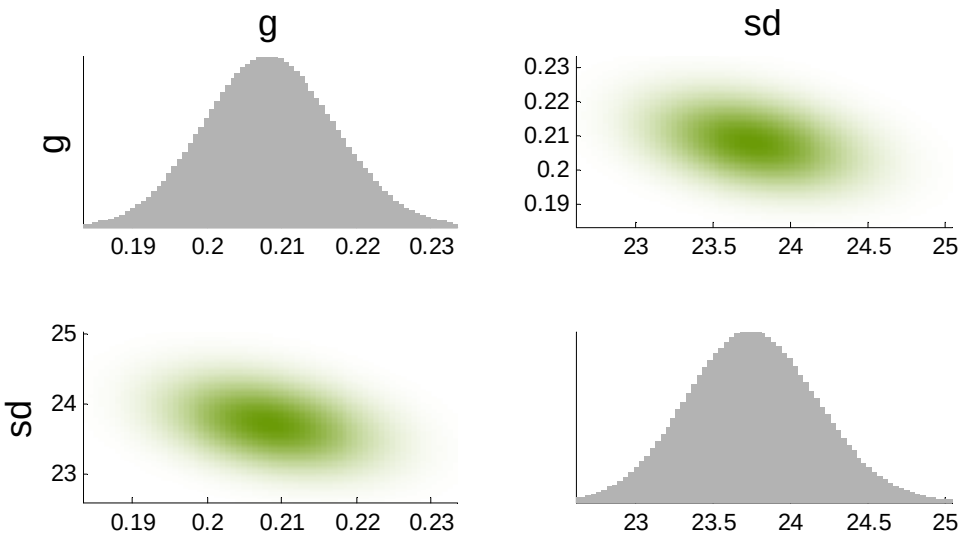
Using MCMC means that we are sampling from the posterior rather than evaluating it directly. Thus, even though the true posterior is almost always a smooth function of the parameters, the plots obtained from the standard `MemFit` call will not always be smooth. We can fix this by using `GridSearch` rather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

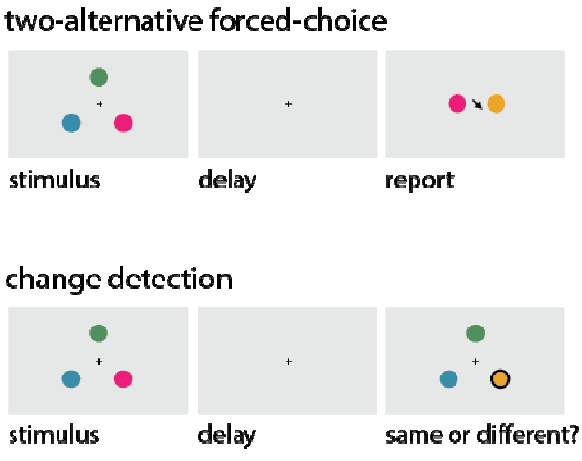


This example also gives you a little bit of a peak under the hood of `MemFit` – in particular, the separate fitting functions (try `help MemFitting`) and plotting functions (`help MemPlots`) of MemToolbox.

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: Fitting 2AFC or change detection data

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



For example, in a change detection task observers see the stimulus display, then after a delay are asked to report whether the test display is the same or different, with a cue to which item may have changed. 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.

These two tasks are formally identical, since in both cases observers are effectively being asked whether their memory representation more closely matches the original item or a new, different

item. To make use of such data, we can specify `data.changeSize` and `data.afcCorrect` rather than `data.errors`. `data.changeSize` indicated 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:

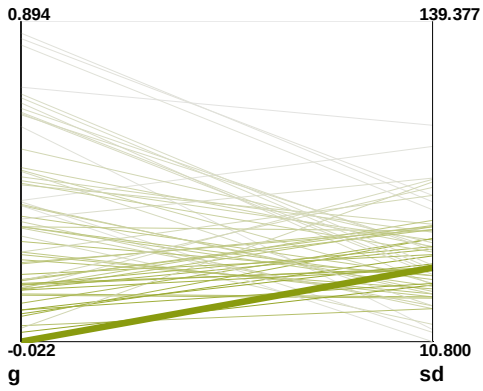
```
>> data.changeSize = [180 180 180 180 180 180 20 20 20 20 20 20];
>> data.afcCorrect = [1 1 1 1 1 1 0 0 0 1 1 1];
>> MemFit(data)
```

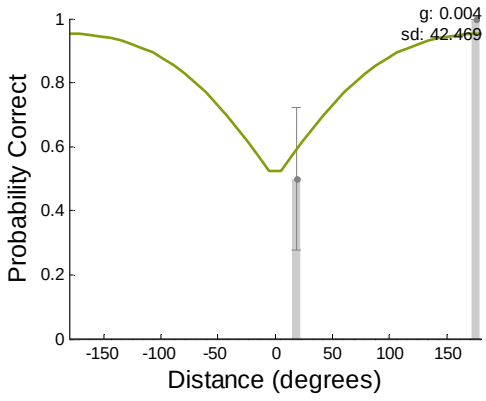
When we run this code, MemFit will recognize that we passed in 2AFC data and will automatically adjust the default model to make use of 2AFC data:

```
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
```

In addition, the MAP estimates will come out 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 2-AFC/change detection model using the `TwoAFC()` wrapper function. Thus, to make a model for a change detection 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 rather slow, since it must compute the likelihood for each display separately.

Demo N-1: Customizing plots.

Demo N: Creating your own model.