MemToolbox2D Tutorial

John P. Grogan*, Sean J. Fallon, Nahid Zokaei, Masud Husain, Elizabeth J. Coulthard, & Sanjay G. Manohar

MemToolbox2D is a Matlab package for applying mixture models to two-dimensional data, such as from spatial working memory tasks. Please see the following paper for the full details (Grogan et al., 2019. A new toolbox to distinguish the sources of spatial memory error). It is based on the MemToolbox package for applying such models to one-dimensional data (Suchow, J. W., Brady, T. F., Fougnie, D., & Alvarez, G. A. (2013). Modeling visual working memory with the MemToolbox. Journal of Vision, 13(10):9, 1–8. doi:10.1167/13.10.9. MemToolbox.org).

This tutorial will explain the main functionality of the toolbox and provide example code that can be used to simulate and fit data. All the code is also available in `Demo2D_1.m` in the MemToolbox2D package. You will need the Statistics Toolbox and MemToolbox (MemToolbox.org) installed.

We have not changed any of the MemToolbox functions, instead creating new versions with '2D' appended to the file names so that MemToolbox2D will not overwrite or interfere with any of the functions from MemToolbox.

TUTORIAL

This section describes how to use the 2D fitting functions. It assumes some knowledge of MemToolbox (Suchow et al., 2013). You must download MemToolbox2D and MemToolbox (Suchow et al., 2013; MemToolbox.org), unzip them and add them to your Matlab path.

The functions are all adapted from the MemToolbox functions for 1D data, and so are used in a similar manner. The main difference is that all the data has twice as many rows, the first row being the x co-ordinates and the second the y co-ordinates. Whereas in the 1D fitting you would create a 'data' structure with vectors of 'errors' and 'distractors' which hold the angular distance between responses and targets, and distractors and targets, respectively, in the 2D version, each of these are matrices holding those differences in X and Y co-ordinates in separate rows.

These data are then used to estimate a set of probability density functions for different error types:

- Target errors are assumed to come from a 2D bivariate Gaussian distribution centred on the target location, with zero covariance, and the standard deviation is a free parameter estimated by the model.
- Non-target distances (from the targets) are used to estimate the probability that a target error came from a 2D bivariate Gaussian centred on the non-target location, with the same standard deviation parameter as the target distribution and zero covariance.
- Guesses are assumed to come from a uniform distribution spread over the entire screen.

The proportion of non-target and guess responses are estimated by fitting γ and β parameters, respectively, which also estimates the proportion of target responses (α) as they all sum to 1.

Below is a walkthrough of how to simulate and fit the model. The MemToolbox2D also contains a demo script that includes the code below, and other demonstrations.

Simulations

To simulate the model, you must specify the number of trials, and the number of items (stimuli) per trial, and use the function `GenerateDisplays2D` to generate the locations of the items, pick 1 stimulus per trial as a target, and calculate the distance between targets and distractors.
The default dimensions are 1366x768 (pixels) – if your screen dimensions are different, you **MUST** specify them when simulating and fitting your data.

```
numTrials = 300; % number of trials
itemsPerTrial = ones(1,numTrials)*3; % 3 items per trial
dimensions = [1366; 768]; % x and y dimensions in units (pixels here)
simulatedData = GenerateDisplays2D(numTrials, itemsPerTrial,...
                                    1,dimensions);
% make locations, pick target, calculate target-distractor distance
```

This is what the `simulatedData` structure looks like:

```
simulatedData =

  struct with fields:

              items: [6×300 double]
     whichIsTestItem: [1×300 double]
            targets: [2×300 double]
         dimensions: [2×1 double]
        distractors: [4×300 double]
                  n: [1×300 double]
```

Note that items, targets, and distractors all have two rows per item, for [x; y] co-ordinates.

Then you define the model you wish to use, and set your parameters for the model (see `model.paramNames` for the order):

```
model = SwapModel2D(); % the model to simulate
params = [0.1, 0.1, 20]; % gamma, beta, SD
```

Then you use `SampleFromModel2D()` to simulate responses from the model, and calculate the distance from targets to responses, which is stored as `simulatedData.errors`:

```
simulatedData.errors = SampleFromModel2D(model, params,...
                          [1 numTrials], simulatedData);
% simulate model to generate responses, calculate distance to targets
```

Fitting

To use a Maximum Likelihood fitting, you call `MLE()` which takes the data and model:

```
fit = MLE(simulatedData, model); % fit the model to simulated data
```

and returns the parameters as a vector in the same order as in `model.paramNames`, in this case, guessing, misbinding and imprecision:

```
fit =

    0.1153    0.0889   19.5683
```

You can also use `MemFit2D()` to use a Markov Chain Monte Carlo (MCMC) method to fit the data, which will return the posterior distribution of parameters and credible intervals, and can also display figures of the model fit and posterior distributions:

```
fit2 = MemFit2D(simulatedData, model, 'Verbosity', 0);
% Bayesian MCMC fitting, without asking about plotting
```

MemFit2D will use the parallel computing toolbox (if installed) to parallel process the MCMC chains, speeding up the fit:

```
fit2 =

  struct with fields:

       posteriorMean: [1×3 double]
     posteriorMedian: [1×3 double]
        maxPosterior: [1×3 double]
       lowerCredible: [1×3 double]
       upperCredible: [1×3 double]
    posteriorSamples: [1×1 struct]
```

Biases

Wrapper functions can be used with the models, for example to correct for radial biases in the data; these allow simulation and fitting of data:

```
params2 = [.1 .1 20 .2]; % 4th parameter is radial bias
model = WithRadialBias2D(SwapModel2D); % add radial bias to SwapModel
simulatedData.errors = SampleFromModel2D(model, params2,...
                          [1 numTrials], simulatedData);
                         % simulate responses
fit3 = MLE(simulatedData, model); % fit
```

2AFC

The models also work with two-alternate forced choice data (2AFC), by using the wrapper function `TwoAFC2D`() around your main model. Here, you simulate a set of stimuli as before, and simulate whether the agent responds correctly when an item is presented a certain distance from its original location (`simulatedData.changeSize`) and the agent must say whether it has moved or not:

```
model = TwoAFC2D(StandardMixtureModel2D);
% use TwoAFC2D wrapper on model without misbinding (for speed)
dimensions = [1366; 768]; % x and y dimensions in units (pixels here)
simulatedData = GenerateDisplays2D(numTrials, ...
                   itemsPerTrial,2,dimensions); % mode = 2 for 2AFC
simulatedData.afcCorrect = SampleFromModel2D(model, [.1 20],...
                              [1 numTrials], simulatedData);
% simulate which 2AFC choices were correct - note only 2 params
```

The TwoAFC2D wrapper tells the sampler to simulate whether the person got each `changeSize` stimulus correct or not. It is fit in the same way as the other models:

```
fit2AFC = MLE(simulatedData, model);
% ensure model has TwoAFC2D() wrapper
```

Stimulus constraints

You can set stimulus constraints when generating stimuli, for example to prevent stimuli appearing too close to each other, or to screen edges or centre:

```
% min distances between [stimuli, stimuli & edges, stimuli & centre]
minDists = [88 29 88];
% pixel equivalent of [3 3 1] visual degrees at 40cm viewing distance
model = SwapModel2D();

% simulate with those constraints
simulatedData = GenerateDisplays2D(numTrials, itemsPerTrial,...
                        1, dimensions, minDists);
simulatedData.errors = SampleFromModel2D(model, params,...
                    [1 numTrials], simulatedData);% simulate responses

fit5 = MLE(simulatedData, model);
```

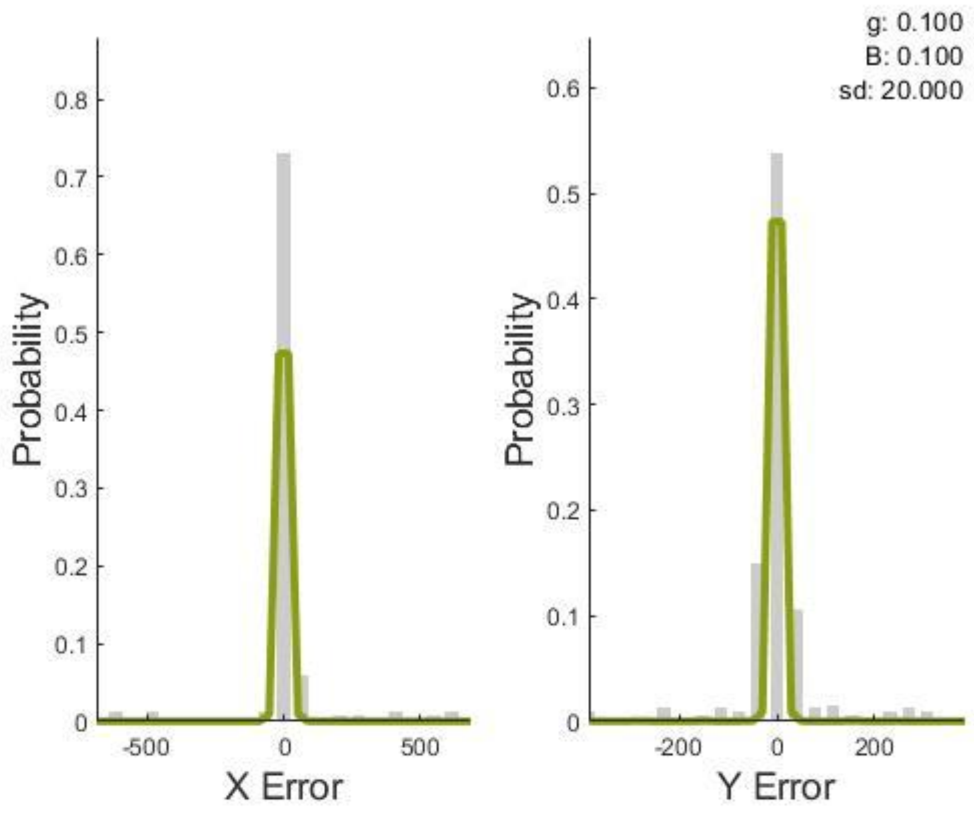
Screen boundaries

The default behaviour of the models when responses are generated outside the screen dimensions is to resample those responses from the same distributions that generated them – 'edge-resampling' from the paper. This can be overridden to give 'edge-constraining' if desired. These two methods give very similar results (see the paper for details):

```
model = SwapModel2D(1); % set boundary to 1 when calling the model
simulatedData = GenerateDisplays2D(numTrials, itemsPerTrial,...
                        1, dimensions); % simulate as normal
simulatedData.errors = SampleFromModel2D(model, params,...
                    [1 numTrials], simulatedData);% simulate responses
fit6 = MLE(simulatedData, model); % fit as normal
```

Other functions

All the other functions from MemToolbox either have 2D versions created for them (e.g. `SampleFromModel2D`) or work the same as in the 1D toolbox (e.g. `MCMC`). Plotting functions (e.g. `PlotPosteriorPredictiveData2D`, `PlotModelFit2D`) have been updated and will look different as they show the x- and y-errors separately:



Data Fitting

In order to fit the models, as a minimum you need a structure with the field '`error`', which is the distance between response and target co-ordinates as a matrix with the first row containing X-co-ordinate distances, the second row Y co-ordinate distances, and one column per trial.

If you are fitting models with misbinding, you also need `data.distractors` which is the distance between target and distractor locations, again with x- and y- distances on separate rows. There should be one pair of rows for each distractor. It is important that you do not use the locations of the distractors here, or the distance between responses and distractors, by mistake.

If your screen dimensions are not 1366*768, you **MUST** enter those as `data.dimensions`, in the form [x-dimension; y-dimension]. Some models require extra information (e.g. target locations for bias wrapper functions), and will return errors if they are missing. Read the help for the models you wish to fit to see what information they require.