# CatContModel Introduction

*Kyle Hardman*

*Package version 0.7.2, February 2017*

## Contents

# Introduction

This package contains implementations of a few variants of psychological process models for delayed-estimation working memory experiments. Primarily, it implements the models used by Hardman, Vergauwe, & Ricker (2017). It also implements a version of the model used by Zhang & Luck (2008). It includes variations on the models that allow for linear (i.e. non-circular) data.

All of the models use Bayesian parameter estimation and model comparison methods. There is nothing inherently Bayesian about the models, but maximum likelihood estimation is, in my experience, relatively slow and unreliable. The unreliability only appears if the model is somewhat complex and if you know what to look for. The slowness of maximum likelihood only really appears once models become complex. The categorical models in this package are complex, by virtue of having a large number of parameters. As such, the choice of Bayesian methods is obvious.

There are two basic steps to using this package: 1) Running the parameter estimation and 2) analyzing the posterior distributions of the parameters. These steps each have many substeps, which are described in detail in the following sections.

I do not hold you responsible for having read and comprehended the Appendix of Hardman et al. (2017). However, there is a great deal of information there that I do not repeat here, so it would likely be of benefit for you to read it if there is information about the specification of the model that you do not understand.

# Parameter Estimation

## Data Format

This package works with two kinds of data: circular and linear. Circular data come from a circular study/response space, such as used by Hardman et al. (2017). Linear data, unlike circular data, do not wrap around. This package assumes that linear data is bounded within finite limits. See the Linear Data section for more information on using linear data.

The data must be stored in a `data.frame` where each row of the data frame has the data for a single study/response pair, with information about which participant and condition the study/response pair are from. The data should have 4 columns, in no particular order:

1. `pnum`: Character (or convertible to character). Participant number. You should have more than 1 participant, otherwise the hierarchical nature of the model will, at best, become dead weight and will, at worst, distort the results.
2. `cond`: Character (or convertible to character). Condition name. If you only have 1 condition, just give any constant value here.

3. `study`: Numeric. If the data are circular, the angle in degrees that the participant studied. If the data are linear, the studied value.
4. `response`: Numeric. If the data are circular, the angle in degrees that the participant responded with. If the data are linear, the responded value.

```
library(CatContModel)

data = read.delim("../../examples/betweenItem/betweenItem_data.txt")

data[1:5,]
```

```
##   pnum cond     study  response
## 1    1    1 142.89385 135.57130
## 2    1    1 204.04122  65.99424
## 3    1    1  10.72329  48.08573
## 4    1    1 196.48278 198.38269
## 5    1    1 196.55208 209.64022
```

Ideally, you have a completely within-participants design, where all participants do all conditions. This allows for maximal interpretability of the condition effect parameters. With a between-participants design, the model will run, but it is possible to have differences between conditions get soaked up, at least in part, by participant level parameters due to different participants being in different conditions. In theory, this is mitigated by the fact that there is a hierarchical prior on all of the participant level parameters. This should keep all participants somewhat clustered around one population grand mean, while the condition effects account for differences between conditions, but this is only in theory. If using a between-participants design, you could consider using a mixed between/within design, where all participants do 2 conditions out of 3 possible conditions, for example. I have tested a partially between-participants test case, and the model seems to work ok.

## Main Parameter Estimation Function

The behavior of the parameter estimation is controlled by a few configuration lists. You only need to supply the `iterations` and `modelVariant` members of the primary configuration list and your data. See the documentation for `runParameterEstimation` for more information about the arguments and further arguments you can use.

```
config = list(iterations=500, modelVariant="betweenItem")

results = runParameterEstimation(config, data)
```

Note that the number of iterations is set to a small value here because it is only an initial run for the purposes of tuning the Metropolis-Hastings acceptance rates. You almost certainly will not be able to call this function once and be done: You need to do the Metropolis-Hastings tuning procedure, which is described below. In the end, you will need to run thousands of iterations. Expect the parameter estimation to take a long time (hours for a typically sized data set).

The results object returned by `runParameterEstimation` is a named list with a bunch of stuff in it.

```
names(results)
```

```
##  [1] "posteriors"             "mhAcceptance"
##  [3] "mhTuning"               "priors"
##  [5] "constantValueOverrides" "pnums"
##  [7] "config"                 "data"
##  [9] "equalityConstraints"    "startingValues"
```

See the documentation for `runParameterEstimation` for more information about the contents of the results object. You will not necessarily need to directly use the contents of the results object, provided that the analyses you want to perform are basically identical to the analyses that I have already done and programmed into this package. If this is not the case, you will need to dig into the results object a little. That said, the analyses that are already programmed in are fairly general.

## Tuning Metropolis-Hastings Acceptance Rates

Many of the parameters of the model are updated with a Metropolis-Hastings (MH) step. The MH procedure works best when the acceptance rates of sampled parameters are in a moderate range. It is up to the user to verify that the acceptance rates for the parameters are in a reasonable range, at about a 0.4 to 0.6 acceptance rate. Once you have parameter estimation results, the following helper function can be used:

```
examineMHAcceptance(results)
```

```
##               paramGroup       mean       min      2.5%      median
## 1              catActive 0.02095667 0.0000000 0.0000000 0.006666667
## 12   catMu (Active only) 0.42093710 0.0000000 0.0000000 0.432800142
## 2  catMu (Total, ignore) 0.73555889 0.1200000 0.1398083 0.803666667
## 3                  catSD 0.50111667 0.1726667 0.2208000 0.493333333
## 4         catSelectivity 0.55308333 0.3453333 0.3561000 0.604833333
## 5                 contSD 0.47313333 0.2210000 0.2379417 0.470000000
## 6            contSD_cond 0.52816667 0.3163333 0.3269250 0.528166667
## 7              pCatGuess 0.47458333 0.3416667 0.3451500 0.497000000
## 8           pContBetween 0.53120000 0.4130000 0.4141083 0.503833333
## 9      pContBetween_cond 0.48366667 0.4246667 0.4276167 0.483666667
## 10                  pMem 0.44623333 0.4123333 0.4128083 0.452333333
## 11             pMem_cond 0.37366667 0.3670000 0.3673333 0.373666667
##        97.5%       max
## 1  0.1120333 0.1463333
## 12 1.0000000 1.0000000
## 2  1.0000000 1.0000000
## 3  0.7360833 0.7503333
## 4  0.6686333 0.6743333
## 5  0.7284917 0.7600000
## 6  0.7294083 0.7400000
## 7  0.5667083 0.5706667
## 8  0.7158750 0.7230000
## 9  0.5397167 0.5426667
## 10 0.4812417 0.4826667
## 11 0.3800000 0.3803333
```

Focus primarily on the mean acceptance rate. In my experience, the acceptance rates can be fairly accurately estimated from a short run of 500 to 1000 iterations. To tune the MH acceptance, the following process works well:

1. Run a short parameter estimation with 500 to 1000 iterations.
2. Check the MH acceptance rates.
3. Override the MH tuning (see below) for the parameters with acceptance rates that are not in the acceptable range.

Repeat steps 1 to 3 until the acceptance rates are in the acceptable range.

### Adjusting MH Tuning

Tuning the MH acceptance rate is accomplished by adjusting the standard deviation of candidate distributions for the parameters. A larger standard deviation will result in fewer acceptances because more candidate values will be far from good parameter values. Thus, if the acceptance rate is too high, you need to increase the MH tuning standard deviation to get fewer good candidates. If the acceptance rate is too low, you need to decrease the MH tuning standard deviation to get more good candidates.

You can find the values of the tuning parameters that were used for an estimation run by examining `results$mhTuning`. Both `pMem_cond` and `condSD` had acceptance rates on the low end, so examine their tuning parameters and adjust them down. Smaller steps means more acceptance.

```
results$mhTuning$pMem_cond
```

```
## [1] 0.15
```

```
results$mhTuning$contSD
```

```
## [1] 2
```

```
mhTuning = list()
mhTuning$pMem_cond = 0.12 #Set some new values that are lower than before.
mhTuning$contSD = 1.6

results = runParameterEstimation(config, data, mhTuningOverrides=mhTuning)
```

Note that it is not possible to tune the acceptance rate for the `catActive` parameters because they only take on the values 0 and 1. They tend to have a really low acceptance rate, but that's just how it is.

For these new results, examine the MH acceptance rates and adjust the MH tuning values, and then rerun parameter estimation. Keep doing this until the MH acceptance rates are all in the acceptable range (0.4 to 0.6) before going on.

## Main Parameter Estimation

Once the MH tuning is complete, you can go on to the main parameter estimation. The parameter estimation is slow, taking on the order of seconds per iteration, while a relatively large number of iterations are required. I would say about 10,000 iterations is probably sufficient for publication quality results. You can do less for exploratory work, maybe 3,000 to 5,000 iterations, and get somewhat reliable results. Publishable results, however, really should be based on at least 10,000 iterations. Results can be excessively variable with substantially smaller numbers of iterations. The more iterations you run, the more precise your results will be.

In order to get these somewhat large number of iterations (at least with respect to how slow each iteration is), it can be helpful to run parallel parameter estimation chains and combine them together. On a 4 core CPU, you can run 4 chains in parallel, which can speed up the process dramatically. However, you must also remove burn-in iterations from the beginning of each chain. I would say that a burn in of at least 500 iterations is warranted, based on my experience, but that 1000 burn-in iterations is safer (see Verifying Convergence below). Given that there is a long burn-in period, you should probably run at least a few thousand iterations in each of the individual parallel chains so that the burn-in doesn't take too many iterations out of each chain. These ideas are addressed in the following sections.

Once you have figured out the MH tuning, you can do something like this in each of a few parallel R sessions (or just one if you aren't doing it in parallel):

```
config = list(iterations=3000, modelVariant="betweenItem")

mhTuning = list()
```

```
mhTuning$pMem_cond = 0.12
mhTuning$contSD_cond = 1.6

results = runParameterEstimation(config, data, mhTuningOverrides = mhTuning)

#The only thing that needs to be different in each instance is the name of the results file
saveRDS(results, file = "results_1.RDS")
```

I will show you how to combine these parallel chains in the Combining Results from Parallel Chains section.

## Verifying Convergence

It is important for the purposes of analysis that the posterior distribution of the parameters has converged. Many of the parameters in the model are estimated with the Metropolis-Hastings procedure, which are often somewhat slow to converge. As a result, it is important to find out when the posterior chains have converged so that the pre-convergence burn-in iterations can be removed.

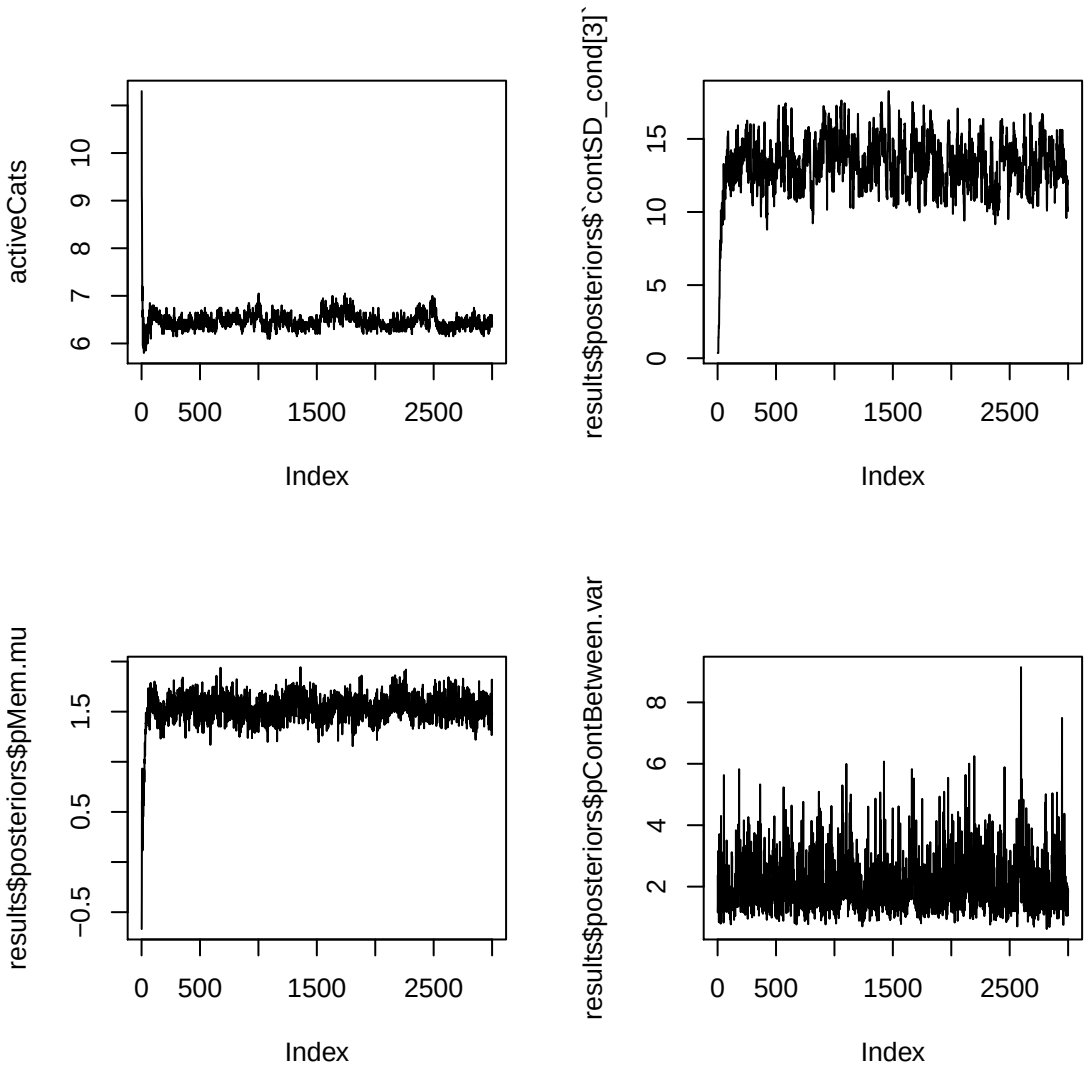
### Plotting Chains

In order to know how many burn-in iterations to discard, it is necessary to examine parameter chains to verify that the Markov chain has converged. To start with, I usually just examine plots of chains to see when the mean of the chain becomes constant. For this model with categories, I have found that the average number of active categories across all participants is a reasonably good indicator of convergence, but I also examine other parameters.

```
par(mfrow=c(2,2))

#Average number of active categories
post = convertPosteriorsToMatrices(results, "catActive") #This will be discussed later
activeCats = apply(post$catActive, 3, mean) * results$config$maxCategories
plot(activeCats, type='l')

#Some other parameters
plot(results$posteriors$`contSD_cond[3]`, type='l')
plot(results$posteriors$pMem.mu, type='l')
plot(results$posteriors$pContBetween.var, type='l')
```




You can see from all of there parameter chains that convergence was fairly quick in this case. It seems like the chains are basically converged past 200 iterations, although convergence doesn't seem to be totally complete at that point.

This is just a small sampling of parameters, you may want to check others as well. However, know that it is unlikely that one parameter will converge if another parameter has not converged because most of the parameters are at least a little correlated.

### Removing Burn-In iterations

Once you have examined convergence, you will almost certainly have found that there are some pre-convergence iterations that should be removed as burn-in. This number is typically 500 to 1000 in my experience, but it depends on the convergence plots you get. You can remove burn-in iterations with the `removeBurnIn` function.

```
results = removeBurnIn(results, burnIn=500)
```

#### Geweke Post Burn-In Convergence Diagnostic

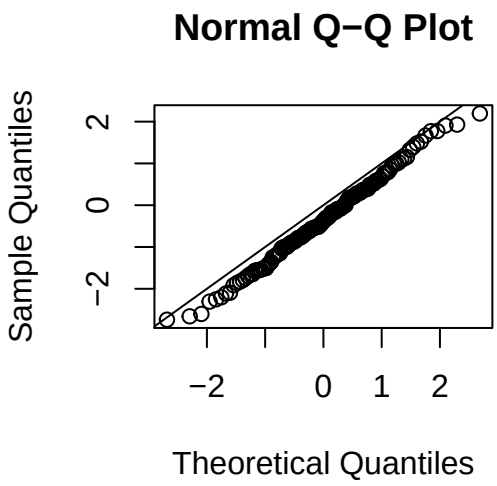
once you have removed burn-in iterations, you should verify that you have removed enough iterations and that the remaining chains have converged. One way to do this is to use the Geweke convergence diagnostic, one implementation of which can be found in the `coda` package. To use the Geweke diagnostic, you must have one large matrix of all parameter chains. The helper function `convertPosteriorsToMatrix` makes this matrix.

```
library(coda)

pmat = convertPosteriorsToMatrix(results)
pmat = mcmc(pmat) #convert to coda format
```

With this parameter matrix, you can use the Geweke diagnostic from the `coda` package. The Geweke diagnostic calculates a z statistic for each parameter. Under the hypothesis that the chains have converged, the z statistics follow a standard normal distribution. We can examine how well the z statistics follow a standard normal by making a QQ plot.

```
gr = geweke.diag(pmat, 0.3, 0.3)
qqnorm(gr$z) # The z-scores should follow a standard normal distribution.
abline(0,1)
```




Note that if a few participant-level parameters do not converge, that may not be a big deal. Inferences are generally based of estimates of the populations of participant-level parameters, which should be reasonably robust to the non-convergence of individual participant parameters. This means that you should focus on convergence of the condition effects and the population mean parameters (e.g. for `pMem`, `pMem_cond[2]` and `pMem.mu`; see the Parameter Names section for more information).

```
gr$z[c("pMem_cond[2]", "pMem.mu")]
```

```
## pMem_cond[2]      pMem.mu
##     1.667749    -1.743873
```

### Combining Results from Parallel Chains

Imagine that you have saved results from a number of different R sessions. You can load the data into a single R session as follows:

```
results_1 = readRDS("results_1.RDS")
results_2 = readRDS("results_2.RDS")
results_3 = readRDS("results_3.RDS")
# Load more results objects...

results_1 = removeBurnIn(results_1, 500)
# Remove burn in for all results...
```

**Note**: You must remove burn-in iterations *before* merging results. Once you have done that, you can combine together the parallel chains with `mergeResults`.

```
results = mergeResults(results_1, results_2, results_3)
```

This final results object can now be used for final analyses.

### Continuing Parameter Estimation

Since it can take hours do to a single parameter estimation run, I made a way to continue sampling from the end of a completed chain, which is done with the `continueSampling` function, which returns a list containing the `oldResults`, the `newResults`, and the `combinedResults`. In this example, you would get 1000 more iterations from the current results chain.

```
# Assume that the "results" object already exists
continueList = continueSampling(results, iterations=1000)
results = continueList$combinedResults
```

A perhaps even more interesting way to use `continueSampling` is as a way to regularly save progress while sampling. Instead of sampling 1000 iterations at once, this example samples 10 blocks of 100 iterations, saving to a file as it goes.

```
for (i in 1:10) {
    continueList = continueSampling(results, iterations=100)
    results = continueList$combinedResults
    saveRDS(results, file="results/loopUpdatedResults.RDS")
}
```

## Examining Results

As these are complex models, there are a variety of ways in which the results can be examined. I have implemented several basic analyses, most of which are presented below. For all of the functions below, more information can found in the documentation of the function, including, typically, function arguments that I leave at the default values here.

### Color Generating Function

This section is an aside about prettifying plots: Feel free to skip it.

This model was designed to work with color delayed estimation tasks, in which case the study and response angles are not as good of a description of the stimuli as what the study and response *colors* were. To add color information to the plots, you may add a function named `colorGeneratingFunction` to the results object that takes an angle in degrees as a argument and returns a color. The result of this is that plots made by this package will plot color information, where appropriate. You will be able to see this in the following plots. If the color generating function is not set, the plots will still work, just without color information.

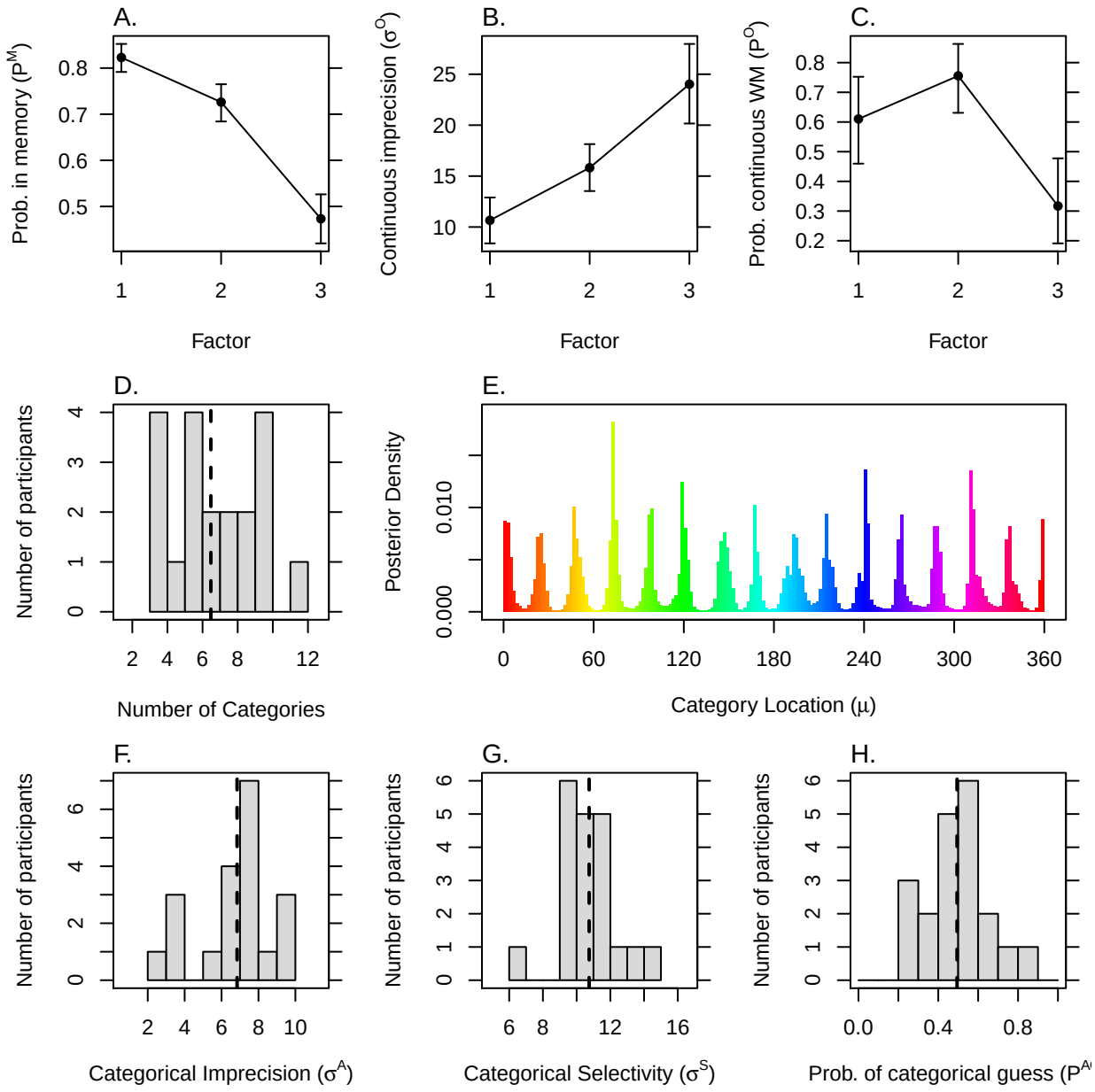
```
results$colorGeneratingFunction = function(x) {
    hsv((x %% 360) / 360, 1, 1)
}
```

Obviously, you should use a color generating function that is the same as what you used to generate your stimuli rather than this example one.

## Parameter Summary Plot

You can make a plot that summarizes the parameters of the model with `plotParameterSummary`. This is a good place to start.

```
par(cex=0.75)
plotParameterSummary(results)
```




## Posterior Means and Credible Intervals

The posterior means and credible intervals for parameters with condition effects are plotted with `plotParameterSummary`. You can access the data used to make these plots by calling the `posteriorMeansAndCredibleIntervals`.

```
posteriorMeansAndCredibleIntervals(results)
```

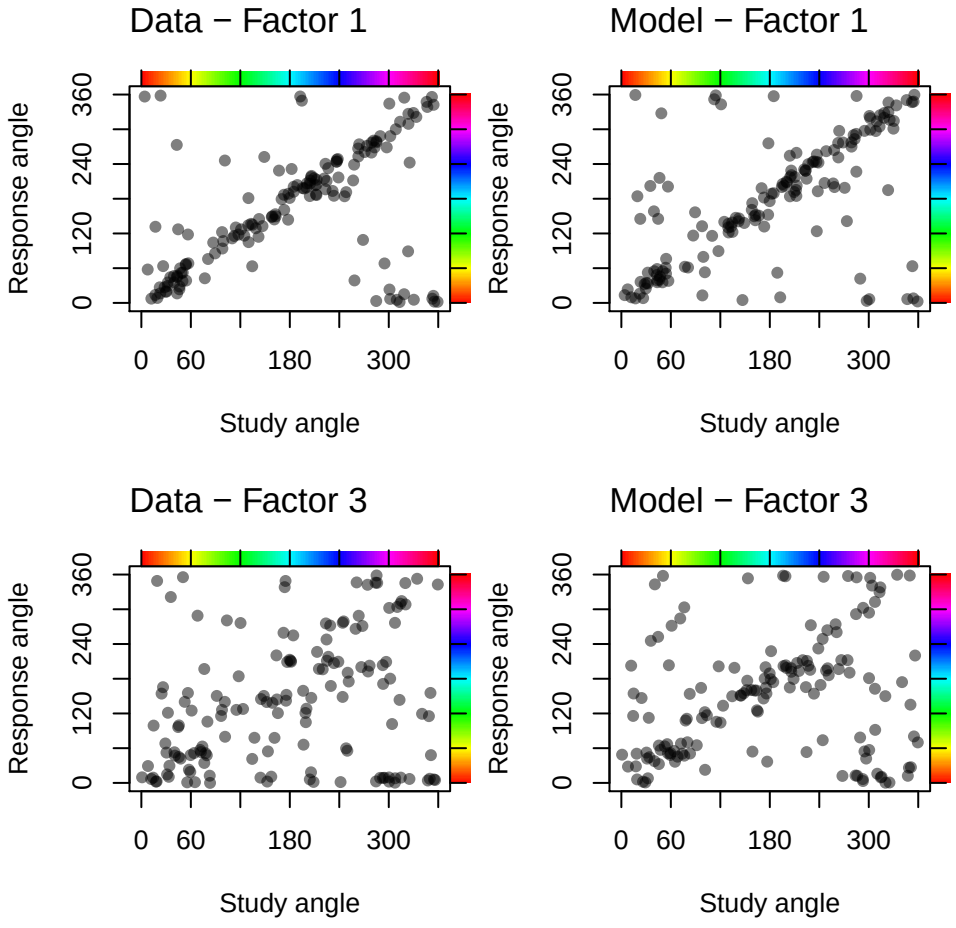
```
##             param cond       mean      lower      upper
## 12          catSD   NA  6.8432872  5.7739668  8.0017865
## 11 catSelectivity   NA 10.7416145  8.7470298 12.9016253
## 8           contSD    1 10.6581103  8.3952686 12.8938609
## 9           contSD    2 15.8189533 13.5320409 18.1357009
```

```
## 10          contSD    3 24.0205645 20.1641968 27.9752135
## 7        pCatGuess   NA  0.4997873  0.3902108  0.6026831
## 4    pContBetween    1  0.6102319  0.4597356  0.7523319
## 5    pContBetween    2  0.7554765  0.6309029  0.8629899
## 6    pContBetween    3  0.3170152  0.1911335  0.4775591
## 1            pMem    1  0.8228625  0.7916117  0.8523486
## 2            pMem    2  0.7262034  0.6842209  0.7649441
## 3            pMem    3  0.4732404  0.4199671  0.5262246
```

### Posterior Predictive Distribution

You can examine data generated from the fitted model by sampling from the posterior predictive distribution. In the following plots, a single participant's real data is plotted alongside data generated from the model. The sampled data are returned invisibly in a data frame.

```
sampled = posteriorPredictivePlot(results, pnums=4, conditions=c(1,3))
```




```
sampled[1:5,]
```

```
##       study   response        type cat pnum cond
## 1 264.14618 297.213237  continuous   4    4    1
## 2  17.08058   7.791241  continuous   3    4    1
## 3  53.72036  60.726529  continuous   1    4    1
```

```
## 4 101.25249  52.952561   unifGuess   0    4    1
## 5  39.80503  54.753006 categorical   2    4    1
```

You can use a vector of `pnums`, which allows you to make plots that include data from any number of participants at once. The different participants are all included in the same plot, which can help you to see overall patterns of fit.

```
# This example is not run
posteriorPredictivePlot(results, pnums=1:length(results$pnums), alpha=0.1)
```

## Condition Effects

If you have more than one condition in your data, one important question is whether the primary parameters of the model change as a function of conditions. You can examine results of hypothesis tests about whether conditions differ with the following function:

```
condEff = testConditionEffects(results)
```

```
condEff
```

```
##           param  cond bfType           bf
## 7        contSD 1 - 2     01 1.231439e-32
## 9        contSD 1 - 3     01 7.980125e-11
## 11       contSD 2 - 3     01 1.188508e-05
## 8        contSD 1 - 2     10 8.120580e+31
## 10       contSD 1 - 3     10 1.253113e+10
## 12       contSD 2 - 3     10 8.413908e+04
## 13 pContBetween 1 - 2     01 9.635861e-12
## 15 pContBetween 1 - 3     01 4.844472e-07
## 17 pContBetween 2 - 3     01 2.957922e-11
## 14 pContBetween 1 - 2     10 1.037790e+11
## 16 pContBetween 1 - 3     10 2.064209e+06
## 18 pContBetween 2 - 3     10 3.380752e+10
## 1          pMem 1 - 2     01 1.478308e-05
## 3          pMem 1 - 3     01 2.545842e-25
## 5          pMem 2 - 3     01 3.765154e-13
## 2          pMem 1 - 2     10 6.764490e+04
## 4          pMem 1 - 3     10 3.927973e+24
## 6          pMem 2 - 3     10 2.655934e+12
```

The null hypothesis in each test is that the listed conditions do not differ from one another. The `bfType` column indicates whether the Bayes factor is in favor of the null (`bf01`) or the alternative (`bf10`) hypothesis. Note that both values are provided for each test.

## Testing Categorical Responding

One research question is whether there is any categorical responding present in the data. One quick way to check this is to see if the parameters that reflect the proportion of categorical responding indicate that no categorical responding is present. These parameters are `pContBetween` and `pCatGuess`. If `pContBetween` = 1 and `pCatGuess` = 0, then there is no categorical responding. That is effectively what the `testCategoricalResponding` function tests, although the probabilities that are tested by default are 0.99 and 0.01 for reasons discussed in Hardman et al. (2017). See the documentation for `testCategoricalResponding` for more information.

```
testCategoricalResponding(results)
```

```
##          param cond H0_value         bf01         bf10 success
## 1 pContBetween    1     0.99 6.867015e-08 1.456237e+07    TRUE
## 2 pContBetween    2     0.99 4.506523e-07 2.219006e+06    TRUE
## 3 pContBetween    3     0.99 7.442996e-12 1.343545e+11    TRUE
## 4    pCatGuess    1     0.01 1.307582e-14 7.647703e+13    TRUE
## 5    pCatGuess    2     0.01 1.307582e-14 7.647703e+13    TRUE
## 6    pCatGuess    3     0.01 1.307582e-14 7.647703e+13    TRUE
```

Note that the test is done in each condition individually. The null hypothesis for each test is that the parameter value is equal to the null hypothesized value, `H0_value`.

## Testing Parameter Means

The test of categorical responding uses a more general function that can test hypotheses about the means of any model parameters (other than `catActive` and `catMu`). The test that is performed is on the population mean in a condition of the experiment.

```
testMeanParameterValue(results, param = "pMem", cond = 1, H0_value = 0.5)
```

```
##   param cond H0_value         bf01        bf10 success
## 1  pMem    1      0.5 6.195486e-11 16140784697    TRUE
```

## Model Fit Statistics (WAIC)

Models are often compared to one another with model fit statistics like AIC and BIC. For the kinds of Bayesian hierarchical models fit with this package, an appropriate fit statistic is WAIC. WAIC is conceptually similar to AIC, but cannot be directly compared with AIC values (although they do often end up being fairly similar in magnitude). You can caluate WAIC for a fitted model as follows.

```
waic = calculateWAIC(results)
```

```
waic
```

```
##       stat       value
## 101 WAIC_1  88643.3744
## 102 WAIC_2  88652.5814
## 103    P_1    268.1501
## 104    P_2    272.7536
## 105   LPPD -44053.5371
```

This can be used to compare model variants (e.g. the between-item and within-item variants). It can also be used to examine the effects of changing constraints (i.e. constant parameter values or priors) within a model variant.

## Testing Main Effects and Interactions

See this section on testing main effects and interactions.

# Advanced Use

## Working With Parameters

### Parameter Names

The parameters are named as follows:

1. Probability parameters:

- `pMem`: The probability that the tested item is in memory.
- `pBetween`: The probability that a memory response is a between-item response.
- `pContBetween`: The probability that a between-item memory response is continuous in nature.
- `pContWithin`: The proportion of a within-item memory response that is continuous in nature.
- `pCatGuess`: The probability that a guess is from one of the color categories (rather than from a uniform distribution).

2. Standard deviation parameters:

- `contSD`: The standard deviation in degrees of continuous memory responses.
- `catSD`: The standard deviation in degrees of categorical memory responses and of categorical guesses.
- `catSelectivity`: A standard deviation in degrees related to how the probability that a study item is assigned to different categories. See the article.

3. Additional category parameters:

- `catMu`: The location of a color category. Each participant has several of these parameters.
- `catActive`: Whether a color category is active. Each participant has several of these parameters.

Each participant has 1 of each of the probability and standard deviation parameters. They have several of the additional category parameters, the number of which is controlled by the `maxCategories` setting of the configuration that is used for parameter estimation. The number of category parameters is constant, but the number of categories actually used by participants is controlled by the `catActive` parameters.

Different model variants have different parameters. Parameters that are not used by a model variant are set to constant values.

- `betweenAndWithin` uses all parameters.
- `betweenItem` does not use `pBetween` or `pContWithin` (`pBetween` is effectively set to 1).
- `withinItem` does not use `pBetween` or `pContBetween` (`pBetween` is effectively set to 0).
- `ZL` uses only `pMem` and `contSD` (`pBetween` and `pContBetween` both effectively set to 1, `pCatGuess` effectively set to 0).

The names of the participant level parameters are formatted as follows: `parameter[pnum]`, where `pnum` is the participant number of the participant associated with that parameter. For example, for the participant with `pnum` 7, the `pMem` parameter is named `pMem[7]`.

Each participant has several `catMu` and `catActive` parameters. Their names are formatted as follows: `catMu[pnum,catIndex]`, where `catIndex` is 0-indexed. For example, the first (i.e. 0-th) `catMu` parameter for the participant with `pnum` 7 would be `catMu[7,0]`.

Population level (hierarchical) parameters have `.mu` or `.var` appended to them. For example, `pMem.mu` is the mean of the participant-level `pMem` parameters.

The condition effects names have `_cond[c]` appended to them, where `c` is replaced with the name of a condition. For example, the `pMem` condition effect for the condition named "ss0" would be `pMem_cond[ss0]`.

### Parameter Chains

The raw parameter chains are stored in `results$posteriors`, which is a named list:

```
names(results$posteriors)[1:5]
```

```
## [1] "pMem[1]"         "pBetween[1]"     "pContBetween[1]" "pContWithin[1]"
## [5] "pCatGuess[1]"
```

These first 5 are participant level parameters where the value in brackets is the participant number. You can access individual parameter chains the same way you would access any named list element:

```
pMem1 = results$posteriors[["pMem[1]"]]
pMem1[1:10] #first 10 iterations of the chain
```

```
##  [1] 1.155852 1.127039 1.127039 0.981843 0.981843 1.001151 1.001151
##  [8] 1.099387 1.099387 1.099387
```

For convenience, it is possible to get the posterior chains for participant level parameters in matrices by calling `convertPosteriorsToMatrices`. The results is a named list of matrices (or arrays, for the `catMu` and `catActive` parameters). Each column of the matrices is a single participant and each row is a single iteration of the chain. The `catMu` and `catActive` arrays are 3 dimensional. The first dimension is participant, the second is category within participant, and the third is iteration.

```
post = convertPosteriorsToMatrices(results)
names(post)
```

```
##  [1] "pMem"           "pBetween"       "pContBetween"   "pContWithin"
##  [5] "pCatGuess"      "contSD"         "catSelectivity" "catSD"
##  [9] "catActive"      "catMu"
```

```
post$contSD[50:52, 1:3]
```

```
##             1       10       11
## [1,] 7.005831 10.98431 8.171866
## [2,] 7.005831 11.06504 8.408084
## [3,] 6.405204 11.06504 8.408084
```

### Parameter Transformations

If working with parameters directly, it is extremely important that you remember that most of the parameters must be transformed to the proper space. For example, the probability parameters exist on the interval `(-Inf, Inf)` and are transformed to probabilities in the interval $[0, 1]$ with the logit transformation.

Critically, you must apply condition effects to participant level parameters before transforming parameters. This means that, for example, to find out what a given participant's probability of having the tested item in memory is in a given condition of the experiment, you must do the following:

1. Retrieve the participant level parameter chain for `pMem`.
2. Retrieve the condition effect chain for `pMem` in the condition of interest.
3. Add the two chains together elementwise.
4. Transform the resulting chain from the latent space to the manifest space with the correct transformation function.

In the following code, the process is done for participant number 4 in condition 3:

```
pMem_part = results$posteriors[["pMem[4]"]] #step 1
pMem_cond = results$posteriors[["pMem_cond[3]"]] #step 2
pMem_latent = pMem_part + pMem_cond #step 3: add the chains
```

```
#step 4 in two parts: getting the transformation function, then doing the transformation
transformationFunction = getParameterTransformation("pMem", results)
pMem_manifest = transformationFunction(pMem_latent)
```

Now you can work with the manifest `pMem` values, like getting their posterior mean or credible interval.

The aforementioned process is performed by the helper function `getParameterPosterior` which can be used as follows:

```
pMem_manifest = getParameterPosterior(results, param="pMem", pnum=4, cond=3)
```

### Helper Function for Parameters for an Iteration

Sometimes you might want all of the transformed parameters for one iteration, in which case you may want use the `getTransformedParameters` function, which provides the transformed parameters for a given participant, condition, and iteration.

```
param = getTransformedParameters(results, pnum=2, cond=1, iteration=7)
param
```

```
## $pMem
## [1] 0.7447417
##
## $pBetween
## [1] 1
##
## $pContBetween
## [1] 0.8018793
##
## $pContWithin
## [1] 1
##
## $pCatGuess
## [1] 0.3432819
##
## $contSD
## [1] 5.474441
##
## $catSelectivity
## [1] 10.76143
##
## $catSD
## [1] 7.107651
##
## $catMu
##  [1] 240.47286 100.46622 314.82192  73.30608 286.56208 214.63248 153.05836
##  [8] 267.67146  22.14887 335.70319
```

Note that inactive categories have already been removed from `param` which is why it does not include the `catActive` parameters (but see the `removeInactiveCategories` argument to control this behavior).

## Priors

See the Priors section of the appendix of Hardman et al. (2017) for a conceptual overview of the priors used by the model. All parameters have default priors that are basically fine to use.

You can change the priors on various parameters in the model with the `priorOverrides` argument of `runParameterEstimation`. It should be a list mapping from the name of the prior parameter to the value of that parameter. The only real complexity here is the naming of the parameters, which I will get to.

All of the parameters have default priors. The values for the default priors can be found by running a parameter estimation without specifying any prior overrides and then examining the `results$priors` list. This also allows you to examine the names of all of the priors.

### Hierarchical Priors

All participant-level parameters have a hierarchical prior such that the mean and variance of those parameters is estimated. The priors for the participant-level parameters that you can control are thus the priors on the mean and variance of the participant-level parameters. The naming of these parameters is as follows:

- `P.mu.mu` - The prior mean of the participant mean.
- `P.mu.var` - The prior variance of the participant mean.
- `P.var.a` - The prior shape ($\alpha$) of the participant variance.
- `P.var.b` - The prior rate ($\beta$) of the participant variance.

Where "P" is the name of the participant-level parameter (e.g. "pMem"). These priors default to fairly uninformative values. Changing them will probably not have much of an effect unless you use highly-informative values.

### Condition Effects

The condition effect parameters account for differences in participant-level parameters between task conditions. The priors on the condition effect parameters are Cauchy distributions with location 0 and some scale value. In particular, the default scales are 2 for the probability parameters (e.g. `pMem`, `pContBetween`) and 5 for the standard deviation parameters (e.g. `contSD`, `catSelectivity`). Note that the scale of the parameters should be thought of with respect to the untransformed space (see the Condition Effects section of the Appendix of Hardman et al. (2017) for information about the transformation).

The naming of these priors is formatted like `P_cond.loc` and `P_cond.scale`, where `.loc` is the location and `.scale` is the scale. NB: The software allows you to set the prior location of condition parameters to a non-zero value, but I would strongly recommend against doing this. Leave the prior location at 0.

For a given parameter (e.g. `pMem`), you would change the prior scale depending on your prior beliefs about the magnitude of the differences between conditions for that parameter. For example, if you expect there to be very small differences in `pMem` between conditions of your experiment, you would set `pMem_cond.scale` to a small value, like `0.5`.

Note that the cornerstone parameterization of the condition effects means that the condition effect for the cornerstone condition is set to 0. This can be thought of as setting the prior scale to 0 for that condition effect. This cannot be modified.

### Example

In this example, two prior overrides are set:

- The prior mean on the mean `contSD` is set to 10.
- The prior scale on the condition effects on `pMem` is set to 0.2.

Any priors not given in `priorOverrides` are set to default values.

```
priorOverrides = list()
priorOverrides[["contSD.mu.mu"]] = 10
priorOverrides[["pMem_cond.scale"]] = 0.2

results = runParameterEstimation(config, data, priorOverrides = priorOverrides)
```

## Controlling the Number of Categories

One of the issues with this model is that it is eager to use a lot of categories. However, it is reasonable to think 1) that people do not have a huge number of categories and 2) that those categories are not extremely close to one another. As discussed in the Category Center and Active Parameters subsection in the Appendix of Hardman et al. (2017), with a uniform prior on the `catMu` and `catActive` parameters, the maximum number of parameters was always used for all participants, which is unreasonable.

There are two ways to control the number of categories used by participants: Setting the maximum number of categories and setting a prior on category locations.

### Maximum Number of Categories

The simpler way to limit the number of parameters is to set the `maxCategories` setting in the configuration of the parameter estimation. Participants will not be able to have more categories than this number.

```
#assume the config list is already partially made
config$maxCategories = 12 #set the maximum number of catgeories.
results = runParameterEstimation(config, data)
```

There are a few guidelines to follow here. 1. If you set `maxCategories` to a very large value (like 100), it will make the parameter estimation take a really long time. This is because every category has parameters that are estimated regardless of whether the category is active or not. Specifically, even for inactive categories, the `catActive` parameter is consistently checking whether the category should become active and the `catMu` parameter is wandering around. 2. If you set `maxCategories` to the exact number of categories that you want to be the maximum, the effective maximum will be slightly lower than `maxCategories` because even categories in good locations will occasionally be deactivated. Thus, you should set `maxCategories` to a value slightly larger (maybe 20%) than the value you want as the effective maximum.

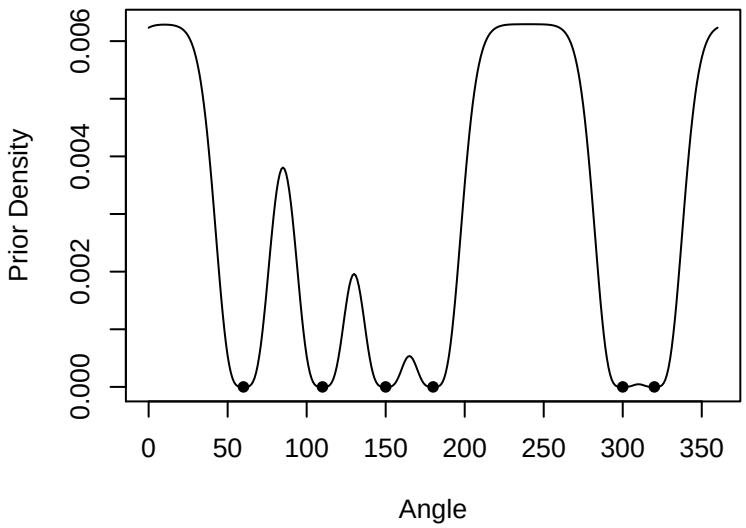
### Prior on Category Locations

The more complex way to control the number of categories is to set a prior which penalizes categories which are too close to one another. This prior discussed in the Category Center and Active Parameters section of the Appendix of Hardman et al. (2017). The parameter that control this is called $\sigma_{0\mu}$ in the article and `catMuPriorSD` in the package. You can set its value by providing a list with it as a prior override, as shown in the code snippet below.

```
priors = list(catMuPriorSD = 12)
results = runParameterEstimation(config, data, priorOverrides=priors)
```

The mathematical behavior of the prior is described in the article. Below is a plot of what this prior looks like for the given category centers. The solid points are at the category centers.

```
par(mar=c(5,4,1,1), cex=0.8)
catMus = c(60, 110, 150, 180, 300, 320)
plotCatMuPrior(12, catMus)
```




This function can be thought of as the prior density for a new category that is trying to be added to the existing categories. In the vicinity of existing categories, the prior density for the new category is quite low, which makes it less likely that the new category will go in that location. However, far from any existing category, such as near 240 degrees, the prior density is relatively high and locally uniform, which makes it relatively easy for a new category to be placed there.

Note that this prior is on both `catMu` and `catActive`. The effect it has on `catActive` is to make it more likely that an active category that has moved too near another category will be deactivated, if active, along with making it less likely that an inactive category would become active when it is near an active category. The effect it has on `catMu` is that it makes it less likely that an active category would move nearer to another active category. It has no effect on inactive `catMu`s (inactive `catMu` parameters move around with a random walk, looking for a good place to become active).

## Setting Parameters to Constant Values

It is possible to set any parameters of the models to constant values. This prevents those parameters from being estimated, but there are reasons for doing this. For example, you may not have enough data to estimate the locations of categories. You might also have very strong beliefs about where the categories are, such as if you have orientation data and believe that participant categories are at the canonical compass points. You might also want to make a reduced model in which some parameters are set to constant values. For example, you might want to disallow categorical guessing, so you set `pCatGuess` to 0.

The way you set parameters to constant values is to provide a list containing constant values for parameters and give it as the `constantValueOverrides` argument to `runParameterEstimation`.

```
constantParam = list()

#Set pMem for participant with pnum 3 to 1.5 (in the latent space, so pMem = 0.82)
constantParam[["pMem[3]"]] = 1.5

#Set the contSD condition effect for condition A to -1
constantParam[["contSD_cond[A]"]] = -1

#Provide the contant value overrides to the parameter estimation
runParameterEstimation(config, data, constantValueOverrides = constantParam)
```

Note that the `constantValueOverrides` must specify parameters for each participant individually. Also note that values must be provided in the latent (untransformed) space.

Typically, you will set all parameter values for all participants to the same value, so there is a helper function called `setConstantParameterValue` for that purpose. In addition, setting the category parameters (`catMu` and `catActive`) to constant values is complex, so `setConstantCategoryParameters` helps with that.

A simple usage of each of the helper funcions is shown below. See the documentation of those functions for more information on usage.

```
#Set pMem to 0.8 for all participants
pMemConstant = setConstantParameterValue(data, param="pMem", value=0.8)

#For pnums 1 and 2, set the locations of the first 4 categories to the compass points.
catParam = data.frame(pnum = rep(c(1,2), each=4),
    catMu = rep(c(0, 90, 180, 270), 2),
    catActive = 1 )

categoryConstant = setConstantCategoryParameters(data, catParam,
    maxCategories = config$maxCategories)


constantParam = c(pMemConstant, categoryConstant)

runParameterEstimation(config, data, constantValueOverrides = constantParam)
```

Setting parameters to constant values is a totally general and flexible system, which also means that it is possible for you to do stupid things with it, like set a variance to 0, so be careful.

See also the Constraining Parameter Estimation section for information on adding or removing effects of task condition on parameters.

## Factorial Designs

This package supports for contraining parameter estimation and performing hypothesis tests for factorial designs. Both fully-crossed and non-fully-crossed designs are supported. Nested designs can be worked with, but are not super easy to work with.

I will start by working with an example of a design with two fully-crossed factors, inventively named "letters" and "numbers". The letters factor has the levels "a" and "b" and the numbers factor has the levels "1", "2", and "3". Note that factor levels should not be numeric, so converting numbers to strings is a good idea.

With factorial designs, before running parameter estimation, you will need to add additional information to the `config` list: A `data.frame` called `factors` and (optionally) a `list` called `conditionEffects`. The `factors` data frame specifies the factor levels for the experimental conditions.

```
#config being the configuration list given to runParameterEstimation()

config$factors = data.frame(letters = c('a',  'a',  'a',  'b',  'b',  'b'),
    numbers = c('1',  '2',  '3',  '1',  '2',  '3'),
    cond =    c('a1', 'a2', 'a3', 'b1', 'b2', 'b3'),
    stringsAsFactors = FALSE) #I like everything to be strings, not factors.

config$factors[3,]
```

As can be seen in this example, condition "a3" is associated with level "a" of the letters factor and level "3"

of the numbers factor. This information is needed by the package so it can know what conditions in the data go with what factor levels. If you have only one factor, you do not need to provide the factors data frame, but you may. It is not necessary to include the factors in the data set: You still only need the `cond` column in the data.

Now that the model has this information about factors, you can start doing two things: constraining parameter estimation and performing hypothesis tests. Note that the discussion of factors naturally lends itself to factorial designs, the following subsections also apply to one-factor designs.

## Constraining Parameter Estimation

You can control which parameters of the models vary as a function of which factors. By default, the "primary" parameters of the models vary as a function of all factors. For example, for the Between-Item model variant, `pMem`, `contSD`, and `pContBetween` vary by task condition. The less critical parameters `catSD`, `catSelectivity`, and `pCatGuess` do not vary by condition, altough it is conceptually possible for them to do so and the model allows that. You can selectively control this behavior by creating a member of the `config` list given to the parameter estimation function, an example of which follows:

```
#config being the configuration list given to runParameterEstimation()

config$conditionEffects = list(
    pMem = "numbers",                  # only use effect of numbers, not letters
    pContBetween = "letters",          # only use effect of letters, not numbers
    contSD = "all",                    # use all effects (letters and numbers)
    catSD = c("numbers", "letters"),   # Same as "all" in this case
    catSelectivity = "none"            # use no effects
)
```

This continues from the above example, where the two factors are called `numbers` and `letters`. As you can see, `conditionEffects` is a list that maps from the name of a parameter to a character vector (often with only 1 element) that gives the name(s) of the factor(s) to use. In addition, the special value `"all"` means to use all factors and the special value `"none"` means to use no factors. Note that this means that you can't have factors that are named `"all"` or `"none"` in the `config$factors` data frame. By default, all parameters that you do not explicity specify, like `pCatGuess` in this example, are set to `"none"`. Thus, it is not neccessary to state `catSelectivity = "none"`, but it is allowed.

To be clear about how this works, I will explain how the above code snippet affects the values of parameters. For `contSD` and `catSD`, for which both factors are used, each cell of the design (each condition) will have its own condition effect. Thus, `contSD` and `catSD` are allowed to freely vary. In contrast, `catSelectivity` and, implicitly, `pCatGuess` will not be allowed to vary with task condition at all, so each condition will have the same value for those parameters (with each participant having their own parameter value, of course). The last two parameters, `pMem` and `pContBetween`, each have an effect of one factor. What happens in that case is that the parameter will have the same value across all levels of the unused factor. In the case of `pMem`, that has an effect of the `numbers` factor, `pMem` will be different in each level of the `numbers` factor, but will be unaffected by the level of the `letters` factor. For a given condition, the value of `pMem` will only depend on which level of the `numbers` factor that condition is in, but the value will not depend on the level of the `letters` factor that condition is in.

Changing which parameters vary by task condition allow a researcher to decide which parameters they are most interested in and allow those to vary by task condition, but restrict the other parameters to be the same across task conditions. This idea naturally leads to comparing models which differ in terms of which parameters are affected by which factors. You can do nested or non-nested model comparisons between models fitted with different configurations using the WAIC statistic, which is discussed in the Model Fit Statistics section. For example, you could fit two models, one with `pMem` either varying with a factor or constant across all levels of the factor. Then you would calculate WAIC for both models to compare the

different beliefs about how `pMem` varies.

Note that although this example is for factorial designs, you can use the same procedure with one factor designs. In that case, it is most simple if you just use `"all"` or `"none"` to specify which factors to use because there is only one factor.

Contraining condition effects in this way works regardless of whether the design is fully-crossed or not.

## Hypothesis Tests of Main Effects and Interactions

It is possible to perform hypothesis tests related to main effects and, if you have more than one factor, interactions between factors. You can do these tests for any parameter for which condition effects were estimated (see the Constraining Parameter Estimation section). Note that this section does not only apply to factorial designs as you may be interested in testing the main effect in a single-factor design.

These tests work with fully-crossed designs (all combinations of factor levels contain data; also called balanced designs in R lingo) and non-fully-crossed designs (some combinations of factor levels do not contain data; also called unbalanced designs). Note an important caveat which is that non-fully-crossed designs have complex interpretation issues, even of main effects, and have choices that you can make about how to perform the tests that affect their interpretation. Fully-crossed designs do not have these issues. If you have a fully-crossed design, everything is easy. If you have a non-fully-crossed design, see the Unbalanced Designs section.

The main function is `testMainEffectsAndInteractions` which does most of everything you need to do (as long as you have a balanced design). It has several arguments that you can read about in the documentation for that function. Usage with default arguments is straightforward.

```
mei = testMainEffectsAndInteractions(results)
```

```
mei[ mei$bfType == "10", ] #Only look at the Bayes factors in favor of there being an effect
```

```
##          param factor  levels bfType           bf           sd
## 2         pMem Factor Omnibus     10 1.494025e+11 1.792397e+10
## 4       contSD Factor Omnibus     10 2.372471e+06 2.224236e+05
## 6 pContBetween Factor Omnibus     10 8.109743e+05 9.801367e+04
##       geo.mean   geo.sd p(BF > 1) p(BF > 3) p(BF > 10)          Min
## 2 1.484085e+11 1.121047         1         1          1 1.125291e+11
## 4 2.362362e+06 1.096693         1         1          1 1.859753e+06
## 6 8.055889e+05 1.120301         1         1          1 6.356753e+05
##           2.5%       Median        97.5%          Max
## 2 1.243722e+11 1.479211e+11 181945116771 215789183771
## 4 2.061697e+06 2.366985e+06      2850923      2890949
## 6 6.743564e+05 7.934733e+05      1080374      1113563
```

The result is a data frame with many columns, the meaning of most of which are explained in the documentation for `summarizeSubsampleResults`. The first two columns give the parameter for which the test was performed (`param`) and the factor of the design for which the test was performed (`factor`). Interactions between factors are indicated by putting a colon (":") between two or more factors. For example, the interaction between letters and numbers would be indicated with `letters:numbers`. The `levels` column indicates which levels of the effect were used for the test. If `Omnibus`, that means that all levels were used and it is an omnibus test. All of the values in the `levels` column will be `Omnibus` unless you specify that pairwise comparisons should be done, which you can do by setting the `doPairwise` argument of `testMainEffectsAndInteractions` to `TRUE`.

See the `factorial_betweenItem` example for more examples of using this function and playing with its output.

Note that these hypothesis tests are sensitive to the choice of prior for the condition effect parameters. You can set the priors for a condition effect parameter as dicussed in the Priors section. Note, however, that you

cannot directly set priors on main effects or interactions, only on the condition effects. The exact relationship is complex, but using tigher priors on condition effects has the effect of making the priors on main effects and interactions more tight as well. You can use the following function to see what the parameters of the marginal priors are.

```
calculateMarginalEffectParameterPriors(results, "pMem")
```

```
##   factor   effect location    scale
## 1 Factor Factor.1        0 1.333333
## 2 Factor Factor.2        0 2.000000
## 3 Factor Factor.3        0 2.000000
```

Some caveats about these hypothesis tests: I am fairly confident that everything I am doing is mathematically and statistically principled. However, it has not been peer reviewed. In addition, although the procedures may be principled, there is a good deal of approximation involved, which could hurt the accuracy of the results. Bayes factors substantially far from 1 should be taken with a grain of salt as the approximations are likely more error-prone the more extreme the Bayes factors are. At the same time, of course, large Bayes factors do not need to be known with a great deal of precision to know the substantive result.

## Unbalanced Designs

This section is an exploration of why non-fully-crossed designs are weird to analyze. In brief, you have to make subjective choices. `testMainEffectsAndInteractions` makes one choice for you, but you can use the lower-level function `testSingleEffect` to get more control.

Consider the following non-fully-crossed (unbalanced) design, in which there are two factors, each with three levels.

```
factors = data.frame(
    letters = c('a', 'b', 'b', 'c', 'c', 'c'),
    numbers = c('1', '1', '2', '1', '2', '3'),
    mus     = c( 0,   0,   6,   0,   0,   0 )
    )
```

Let's imagine that for some parameter we get the following pattern of cell means. A "." indicates a missing cell in the design.

| / | 1 | 2 | 3 |
|---|---|---|---|
| a | 0 | . | . |
| b | 0 | 6 | . |
| c | 0 | 0 | 0 |

In this design, you can conceptually think of the following effect parameters: A grand mean or intercept, main effects of the letters factor, main effects of the numbers factor, and interaction terms.

Assume that you want to test the main effect of the letters factor. There are two ways we can think about the main effect. The first is to consider the main effect of letters without considering the interaction between letters and numbers. The second is to consider the main effect in the context of the interaction. These approaches are equivalent in a fully-crossed design that uses orthogonal contrasts (the package default for fully-crossed designs), but they are not equivalent when the design is not fully crossed, in which case the contrasts cannot be orthogonal.

We will start with the first approach, examining the main effect of letters without the interaction. I will do this by creating a design matrix and then solving for the effect parameters given the cell means.

```
X = model.matrix( ~ letters, factors,
  contrasts.arg = list(letters = "contr.treatment"))
X
```

```
##   (Intercept) lettersb lettersc
## 1           1        0        0
## 2           1        1        0
## 3           1        1        0
## 4           1        0        1
## 5           1        0        1
## 6           1        0        1
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$letters
## [1] "contr.treatment"
```

First, the design matrix, $X$, is created using treatment contrasts, which are valid when the design is not fully crossed. The solution for the vector of effect parameters is $\beta = (X^T X)^{-1} X^T \mu$, where $\mu$ are the cell means.

```
beta = solve(t(X) %*% X) %*% t(X) %*% factors$mus
zapsmall(beta)
```

```
##             [,1]
## (Intercept)    0
## lettersb       3
## lettersc       0
```

We can see that the parameter corresponding to the `b` level of letters, `lettersb`, is 3, while the `lettersc` parameter is 0. Given that treatment contrasts were used, the value for `lettersa` is 0, i.e. nothing is added to the `a` level of letters beyond the intercept. Thus, the effect parameters correspond to the marginal means of 0, 3, and 0 for `a`, `b`, and `c`, respectively. If we assume very low noise in the data, there appears to be a main effect of letters.

Now let's use the second approach and include the interaction in the design and see what happens. Let's make the design matrix, this time including both factors and their interaction.

```
X = model.matrix( ~ letters * numbers, factors,
  contrasts.arg = list(
    letters = "contr.treatment",
    numbers = "contr.treatment"
  )
)
```

This design matrix, however, has too many columns (9) for the number of cells in the design (6), making it be not full rank (one outcome of which is that $X^T X$ is uninvertable, which it needs to be). I'll take advantage of a little trick to drop excess columns.

```
q = qr(X)
kept = q$pivot[ 1:q$rank ]
X = X[ , kept ]
X
```

```
##   (Intercept) lettersb lettersc numbers2 numbers3 lettersb:numbers2
## 1           1        0        0        0        0                 0
## 2           1        1        0        0        0                 0
## 3           1        1        0        1        0                 1
## 4           1        0        1        0        0                 0
```

```
## 5           1        0        1        1        0                 0
## 6           1        0        1        0        1                 0
```

Now $X$ is full rank. You can see from the columns of $X$ that there is one interaction term, `lettersb:numbers2`. This term corresponds to the cell of the design with mean 6. We can calculate the effect parameters as before.

```
beta = solve(t(X) %*% X) %*% t(X) %*% factors$mus
zapsmall(beta)
```

```
##                   [,1]
## (Intercept)          0
## lettersb             0
## lettersc             0
## numbers2             0
## numbers3             0
## lettersb:numbers2    6
```

Now when we look at the parameter values, we can see that all of the letter main effect parameters are 0 because all of the marginal effect of cell `b2` is accounted for by the interaction. Thus, there does not appear to be a main effect of letters.

Here is a different example.

| / | 1 | 2 | 3 |
|---|---|---|---|
| a | 0 | 2 | 4 |
| b | 0 | . | . |
| c | 0 | . | . |

We can see that there is a marginal main effect of letters when nothing else is considered. However, when the main effect of numbers is included in the design, the parameters associated with the `2` and `3` levels of the numbers factor account for the apparent marginal lettters effect. This, main effects can be affected by the inclusion of other main effects.

```
factors = data.frame(
  letters = c('a', 'a', 'a', 'b', 'c'),
  numbers = c('1', '2', '3', '1', '1'),
  mus     = c( 0,   2,   4,   0,   0 )
)

X = model.matrix( ~ letters + numbers, factors,
  contrasts.arg = list(
    letters = "contr.treatment",
    numbers = "contr.treatment"
  )
)

beta = solve(t(X) %*% X) %*% t(X) %*% factors$mus
zapsmall(beta)
```

```
##             [,1]
## (Intercept)    0
## lettersb       0
## lettersc       0
## numbers2       2
## numbers3       4
```

When working with non-fully-crossed designs, you have to choose what main effects or interactions to account for. This is difficult to do with the function `testMainEffectsAndInteractions`. By default, it takes the first approach: Only estimate the main effect, but not interactions. More generally, it only includes terms up to and including the tested effect. For example, if you are testing two-factor interaction, it will estimate that interaction and the two associated main effects. If there is a third factor, it will not estimate the main effect of that third factor or any interactions including that third factor.

If you want more control over what effects are estimated, you can use the lower-level function `testSingleEffect`. Here is a minimal (and not run) usage example where the main effects of letters is tested in the context of a design matrix that includes the interaction between letters and numbers.

```
testSingleEffect(results, "contSD", testedFactors = "letters", dmFactors = c("letters", "numbers"))
```

## Nested Designs

Sometimes you will have a nested design, where the levels of one factor are nested within the levels of another factor. One example of such a situation could come from a working memory experiment in which you have serial positions nested within set sizes. For example, you might have two set sizes of memoranda, such as lists of 2 or 4 colors. At the end of the presentation of a list, participants recall one of the items in the list to the best of their ability. The order of presentation of the items defines the serial position factor. For set size 2, there are serial positions 1 and 2. For set size 4, there are serial positions 1 through 4. This is a nested design because although set sizes 2 and 4 both have serial positions 1 and 2, the meaning of those serial positions is totally different within the context of the set size, due to (at least) primacy and recency effects. At set size 2, serial position 2 is the most recent item, so it would get a recency effect. At set size 4, however, serial position 2 is somewhere in the middle of the list, so recall should be hurt. Thus, this is a nested design, where serial positions must be treated as dependent on the set size. I will explain here how to analyze such a design with this package.

In the results of parameter estimation, there is a `data.frame` that can be found in `results$config$factors`. For a multi-factor design, you should have provided this `data.frame` in `config$factors` before starting parameter estimation.

The key idea is that although you provided the `factors data.frame` prior to parameter estimation, you are allowed to change it (in judicious ways) following parameter estimation.

Let's say that this is what `factors` looks like:

```
factors = data.frame(
  SS = c('2', '2', '4', '4', '4', '4'), # SS = set size
  SP = c('1', '2', '1', '2', '3', '4')  # SP = serial position
)
factors$cond = paste0(factors$SS, "_", factors$SP)

factors
```

```
##   SS SP cond
## 1  2  1  2_1
## 2  2  2  2_2
## 3  4  1  4_1
## 4  4  2  4_2
## 5  4  3  4_3
## 6  4  4  4_4
```

Imagine that we want to test a main effect of serial position, but only at one set size at a time. Let's just do set size 4 for the sake of example (it's very easy to do set size 2 from the same template). The following code block does just that (the results are not printed).

```
resCopy = results # Copy results so you don't change the original

f = resCopy$config$factors # Copy factors for brevity

f = f[ f$SS == 4, ] # Select only SS 4
f$SS = NULL # Drop the, now meaningless, SS factor

resCopy$config$factors = f #Reassign the copy

# Do a test of the effect of SP only for array size 4
testMainEffectsAndInteractions(resCopy)

# Make a parameter summary plot of the same
plotParameterSummary(resCopy)
```

Here is another example where instead of treating the design as nested, we want to treat SS2, SP2 and SS4, SP4 as both being at the end of the list. We then want to examine serial position effects by collapsing across set sizes (ignore the fact that this is probably a bad analysis strategy). We can do this by changing factor level names. We will rename SS2, SP2 to SS2, SP4 (i.e. the last set size).

```
resCopy = results

f = resCopy$config$factors

f$SP[ f$SS == 2 & f$SP == 2 ] = '4' # Rename SP 2 to SP 4 for SS 2
#DO NOT RENAME THE CELL IN THE cond COLUMN. That name is baked in.

# Although the SS factor isn't meaningless (it has 2 levels), we want to ignore it, so drop it.
f$SS = NULL

resCopy$config$factors = f

# Do tests/plots with resCopy...
```

Notes:

1. Any time you have a factor with only 1 level, you must drop that factor by setting it to `NULL`.
2. As shown in the example, you may drop rows (conditions) from factors. This is ok, it just means that you are ignoring some cells of your design for the given analysis.
3. You must retain the `cond` column and you must not change it at all (other than by dropping rows). The names of the conditions cannot be changed as they are baked in to the names of the parameters.
4. The two things that this is good for are tests of main effects and interactions and parameter summary plots, as shown in the first example.

See the `factorial_betweenItem` example for more code examples of modifying `results$config$factors`. In that case, the design is not nested, but the presence of a two-factor interaction means that simple effects (main effects of a factor within a single level of another factor) must be analyzed. The analysis of simple effects is very much like the analysis of a nested design.

# Notes

## Citing this Package and the Models

For referencing ideas in the models, please cite Hardman et al. (2017). See the references section for a full citation.

If you use this package, please cite it:

Hardman, K.O. (2016). CatContModel: Categorical and Continuous Working Memory Models for Delayed Estimation Tasks (Version 0.7.2) [Computer software]. Retrieved from https://github.com/hardmanko/CatContModel/releases/tag/v0.7.2

Make sure to update the version number in both the name and the URL.

## Models

### Between-Item

This was the best model selected by Hardman et al. (2017). If you are going to use one categorical model from this package for color data, you should use this model. However, you could try fitting more than one model and comparing the fit of those models.

### Fully Categorical Model

If you want a fully categorical model (i.e., no continuous responding allowed), use the "betweenItem" model variant with the constant parameter value overrides functionality, as demonstrated in the following code snippet.

```
config = list(iterations=500, modelVariant="betweenItem")

#Set the probability of continuous responding to 0
pcbOverrides = setConstantParameterValue(data, "pContBetween", 0)

results = runParameterEstimation(config, data, constantValueOverrides = pcbOverrides)
```

### Within-Item

The within-item model has an issue, which is that if very few categories are active, `pContWithin` is essentially forced to be 1. Imagine the case in which there is only 1 active category at 180 degrees and that `pContWithin` is 0.5. This means that the model will predict that memory responses will be on a line essentially going from (0, 90) to (360, 270), passing through (180, 180). This line would provide terrible fit for the data, unless there was something really bizarre about the data. This forces `pContWithin` to 1 so that the line will go from (0, 0) to (360, 360) and capture the bulk of the data. With more active categories, it becomes less of an issue as there are multiple line segments, so no one part of a line segment is too far from the intercept 0, slope 1 line. The between-item model doesn't have this issue because the categorical and continuous response are treated separately. One possible solution to this is to use constant parameter values overrides to force a few categories (maybe 4) to always be active, but that does have the effect of always having a few active categories.

For estimation of `catSD` and `contSD`, the within-item model is much more fragile than the between-item model due to how those parameters are estimated. In the between-item model, `catSD` is estimated from to categorical memory responses and categorical guesses while `contSD` is estimated from continuous memory responses. For the within item model, `catSD` is estimated from categorical guessing and from a part of the

mixed categorical and continuous responses, while `contSD` is estimated from a part of the mixed categorical and continuous responses.

For more information, read the "Within-Item Model Variant" section of the Appendix of Hardman et al. (2017). In particular, notice how Equation 25 for $\sigma^W$ contains the two parameters of interest, `contSD` ($\sigma^O$) and `catSD` ($\sigma^A$). Then $\sigma^W$ gets plugged in to Equation 22, which contains data. Thus, $\sigma^W$ is straightforward to estimate because it is based on data directly, but `contSD` and `catSD` cannot be estimated directly: they could trade off perfectly without some other constraint. That other constraint comes from the estimation of `catSD` from categorical guesses.

**Important**: From the above, if there is no categorical guessing, the within-item model *cannot estimate* both `catSD` and `contSD`. In fact, it will likely be unable to estimate either of them as they will trade off horribly. If there is very little categorical guessing, like 10% of guesses, you probably can't reasonably estimate `catSD` or `contSD` either.

If there is truly 0 categorical guessing, one partial solution would be to set $\sigma^A$ to the constant value 0 (and probably also set `pCatGuess` to 0 to be sure that no categorical guessing is estimated). In this case, you cannot estimate both $\sigma^O$ and $\sigma^A$, but you can estimate their mixture, $\sigma^W$. The model doesn't diectly estimate $\sigma^W$, but you can calculate it after the fact by using the following logic.

Start from Equation 25:

$\sigma^W = [P^2 * (\sigma^O)^2 + (1 - P^2) * (\sigma^A)^2]^{1/2}$

If $\sigma^A = 0$, then the above equation becomes,

$\sigma^W = [P^2 * (\sigma^O)^2]^{1/2} = P * \sigma^O$

Note that you cannot interpret $\sigma^O$ as meaning anything in this context, because you don't know what $\sigma^A$ is (it is not actually 0). You can only interpret $\sigma^W$ as being the mixture of two unknown quantities. Given that, it may be useful to know that $\sigma^W \leq max(\sigma^O, \sigma^A)$.

If you have some small but decidedly nonzero amount of categorical guessing, it may be a better solution to set `catSD` to a constant value (based on past research, maybe).

### ZL (Zhang & Luck)

There is nothing particularly interesting about my implementation of the ZL model, but it is still a hierarchical Bayesian model with effects of experimental condition. It may be useful on its own or as a model to compare the other models to.

## Stimuli

These models were developed with colors in mind, but there is nothing about the models that restricts them to color stimuli. The models can be used with any stimuli that exist on a circular or linear space, although this does not mean that the models will necessarily be appropriate for those data.

Always attempt at least some diagnostics to verify that the model is working well. One of the best diagnostics is to see whether the data produced by the fitted model looks like the real data. See the Posterior Predictive Distribution section for information on how to do this.

### Linear Data

To use this package with linear data (i.e. data the do not wrap around/are not circular), you will need to modify the main parameter estimation configuration in a few ways. Imagine that you have linear data and that the possible range of responses goes from -100 to 100 (in whatever the units of the data are). You would create a configuration list as follows:

```
config = list(iterations=500, modelVariant="betweenItem",
  dataType = "linear", responseRange = c(-100, 100))

results = runParameterEstimation(config, data)
```

Note that two new, important values are provided: `dataType` and `responseRange`. `dataType` is either "linear" or "circular" (defaults to "circular"). `responseRange` is a length 2 vector of numeric values giving the range of possible responses.

Note that you do not need to provide `responseRange`. If you do not provide it, `responseRange` will be set to the range of the observed data. This is done across all participants, not per participant. If different participants were given tasks that differed in terms of the possible range of responses, you won't be able to use this package without modification. In many cases, the observed range of the data and the possible range of responses are very similar, but if you have data in which the observed and possible ranges differ substantially, you should provide the possible range.

You do not need to provide the possible range of the studied values. The possible range of studied values is irrelevant because studied values are treated as having known true values, which means that they have no distributional form.

Once the model is configured to work with linear data, all of the analyses proceed as usual. All of the parameters have the same basic meaning regardless of whether circular or linear data are used. Some plotting functions might behave a little differently, but they should work.

The choice of `modelVariant` (e.g. betweenItem) is independent of the `dataType`: All model variants work will either data type.

Note that the default priors assume that the data are circular in degrees, which is sort of like saying that they have a range of 360 degrees. If your linear data have a much smaller range, like 30 units, or a much larger range, like 5000 units, you should consider whether you need to adjust the hierarchical priors on `contSD`, `catSD`, and `catSelectivity`. In particular, I would mostly consider adjusting the prior mean and variance on the means of those three parameters. Likewise, you should probably change the condition effect prior scales for those parameters. Finally, you should probably adjust the Metropolis-Hastings tuning values for those three parameters. See the code example below:

```
# Imagine that the data are on the interval [0, 20]. The continuous imprecision
# standard deviations will likely be small-ish.
# Use continuous SD with mean 5 and variance 225 (15^2).
# Set condition effect scale to 2 to reflect that you are expecting small-ish effects.
priorOverrides = list(contSD.mu = 5, contSD.var = 225, contSD_cond.scale = 2)

# If the contSD values are small, they don't need to jump as far in the MH steps.
# Use small-ish MH tuning values.
mhTuningOverrides = list(contSD = 1, contSD_cond = 0.5)

# You should probably also do this for catSD and catSelectivity, too
# (except that there are usually no condition effects for catSD and catSelectivity).

# Use the prior and MH tuning overrides.
results = runParameterEstimation(config, data,
    priorOverrides = priorOverrides,
    mhTuningOverrides = mhTuningOverrides)
```

Currently, the `catMu` parameters are not allowed outside of the range of the data. This may change at some point once I make a hard decision about how to allow it to happen. Details: There is a calculation in the likelihood function in which the density of an observed response given a `catMu` parameter is calculated. The density function is a truncated normal. If the `catMu` parameter is far outside of the possible range of

responses, the proportion of the normal distribution within the response range (which is the truncation scale factor) is approximately 0. To calculate the truncated density, a normal density is divided by the truncation scale factor. Division by 0 is bad, so I have to choose how to prevent division by 0. There are no good solutions (at least that I know of), only less bad solutions. The current less bad solution is to prevent the `catMu` parameters from being outside of the response range.

### catSelectivity

`catSelectivity` is really hard to estimate. Don't expect to be able to do anything fancy with it, like giving it a condition effect as discussed in the Constraining Parameter Estimation section.

## References


Hardman, K. O., Vergauwe, E., & Ricker, T. J. (2017). Categorical working memory representations are used in delayed estimation of continuous colors. *Journal of Experimental Psychology: Human Perception and Performance*, *43*(1), 30–54. http://doi.org/10.1037/xhp0000290

Zhang, W., & Luck, S. J. (2008). Discrete fixed-resolution representations in visual working memory. *Nature*, *453*(8), 233–235. http://doi.org/10.1038/nature06860