Modelling structured trial-by-trial variability in evidence accumulation

Steven Miletić et al.

06 September, 2025

Introduction

In this tutorial, we will demonstrate how to design dynamic evidence accumulation models and fit them to data using hierarchical Bayesian methods. We will utilize the *EMC2* package, assuming some prior knowledge of its functionalities (for reference, see the *EMC2* tutorial). Specifically, we will fit models to dataset 1 from Miletic et al. (2025), which corresponds to the 'choice, short [CS]' condition from Wagenmakers et al. (2004). In this experiment, participants were presented with a single digit and required to determine whether the digit was even or odd.

First, let's load the required packages and data:

```
rm(list = ls())
# TMP -- install the right branch
# remotes::install_github("ampl-psych/EMC2@RL",dependencies=TRUE, Ncpus=8);.rs.restartR()
# END TMP
library(EMC2)

# This is some code that contains plotting functions used later on in this tutorial
source('./plotting_utils.R')

# Load in the data
load("datasets/dataset_1.RData")

# Inspect the data
print(head(dat))
```

```
  subjects    S    R    rt
1        1 even even 0.542
2        1 even even 0.426
3        1 even even 0.501
4        1 even even 0.403
5        1  odd  odd 0.497
6        1 even even 0.518
```

Here, `subjects` refers to subject number, `S` to the presented stimulus (even/odd), `R` the response (even/odd), and `rt` the response time in seconds.

Design Specification

We begin by setting up the 'static' baseline models without structured trial-by-trial variability. For a comprehensive tutorial on specifying designs in *EMC2*, please refer to the *EMC2* tutorial [ref]. We will set up two baseline models: first using a racing diffusion model (RDM), and then using a diffusion decision model (DDM). In the RDM, we model drift rates with a mean-difference parametrization.

```
# set up contrast matrix for mean-difference parametrisation
ADmat <- matrix(c(-.5,.5), ncol=1, dimnames=list(NULL,'d'))
design_RDM <- design(model=RDM,
                  data=dat,
                  contrast=list(lM=ADmat),
                  matchfun=function(d) d$S==d$lR,
                  formula=list(B ~ 1, v ~ lM, t0 ~ 1))
```

```
 Sampled Parameters:
[1] "B"     "v"     "v_lMd" "t0"

 Design Matrices:
$B
 B
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$A
 A
 1

$s
 s
 1
```

In the DDM, we need to remember to flip the sign of the drift rate for one of the two stimulus types (in this case, even):

```
Smat <- matrix(c(-1,1), nrow = 2,dimnames=list(NULL,"dif"))
design_DDM <- design(model=DDM,
                  data=dat,
                  contrasts=list(S=Smat),
                  formula=list(Z ~ 1, v ~ S, a~1, t0 ~ 1))
```

```
 Sampled Parameters:
[1] "Z"      "v"      "v_Sdif" "a"      "t0"
```

```
 Design Matrices:
$Z
 Z
 1

$v
    S v v_Sdif
 even 1     -1
  odd 1      1

$a
 a
 1

$t0
 t0
  1

$s
 s
 1

$st0
 st0
   1

$sv
 sv
  1

$SZ
 SZ
  1
```

Trend specification

Both the descriptive trends and the formal mechanisms of dynamics work through `trend` objects in *EMC2*. In this object, you specify (1) the covariate of interest (e.g., time on task), (2) a kernel to apply to the covariate (e.g., linear, power, exponential, polynomial, or delta rule), and (3) which decision parameter is informed by the resulting covariate, and (4) the functional form of the mapping between the resulting covariate and the decision parameters, referred to as the 'base'. The `trend_help()` function gives an overview of the options:

```
trend_help()
```

```
Available kernels:
  lin_decr: Decreasing linear kernel: k = -c
  lin_incr: Increasing linear kernel: k = c
  exp_decr: Decreasing exponential kernel: k = exp(-d * c)
  exp_incr: Increasing exponential kernel: k = 1 - exp(-d * c)
  pow_decr: Decreasing power kernel: k = (1 + c)^(-d)
  pow_incr: Increasing power kernel: k = 1 - (1 + c)^(-d)
  poly2: Quadratic polynomial: k = d1 * c + d2 * c^2
  poly3: Cubic polynomial: k = d1 * c + d2 * c^2 + d3 * c^3
```

```
  poly4: Quartic polynomial: k = d1 * c + d2 * c^2 + d3 * c^3 + d4 * c^4
  delta: Standard delta rule kernel: Updates q[i] = q[i-1] + alpha * (c[i-1] - q[i-1]). Parameters: q0
  delta2: Dual kernel delta rule: Combines fast and slow learning rates and switches between them based
  deltab: Threshold learning delta rule kernel: Updates q[i] = q[i-1] + alpha * (B[i]/c[i-1] - q[i-1]).

Available base types:
  lin: Linear base: parameter + w * k
  exp_lin: Exponential linear base: exp(parameter) + exp(w) * k
  centered: Centered mapping: parameter + w*(k - 0.5)
  add: Additive base: parameter + k
  identity: Identity base: k

Trend options:
  premap: Trend is applied before parameter mapping. This means the trend parameters
          are mapped first, then used to transform cognitive model parameters before their mapping.
  pretransform: Trend is applied after parameter mapping but before transformations.
                Cognitive model parameters are mapped first, then trend is applied, followed by transfo
  posttransform: Trend is applied after both mapping and transformations.
                 Cognitive model parameters are mapped and transformed first, then trend is applied.
```

The last section, 'trend options', warrants some extra attention. In *EMC2*, the user can define parameters using model formula language, as we did above when defining the designs. *EMC2* uses these to create a design matrix by mapping the relevant factors to each design cell. In the case of the RDM, once mapped, in each design cell, only the `v`, `B`, `t0`, `s`, (and optionally `A`) parameters per accumulator remain. However, in many applications, the parameter of interest is defined as a between-accumulator difference (e.g., `v_lMd` in the RDM example above), or a between-condition difference (e.g., the effect of speed-accuracy trade-off cues on thresholds). These parameters only exist pre-mapping, and thus, the trend should be applied prior to mapping by setting `premap` to `TRUE`.

There is an important caveat here. By default, *EMC2* estimates parameters with a lower bound (e.g., non-decision time, thresholds, RDM's drift rates) on the log scale, and parameters with both a lower and upper bound on the probit scale. Parameters are *mapped* on the scale on which they are estimated, and *then* transformed. This also implies that if a trend is applied prior to mapping, the resulting parameters might later be transformed, leading to a potentially unwanted non-linear effect of the covariate on the parameter.

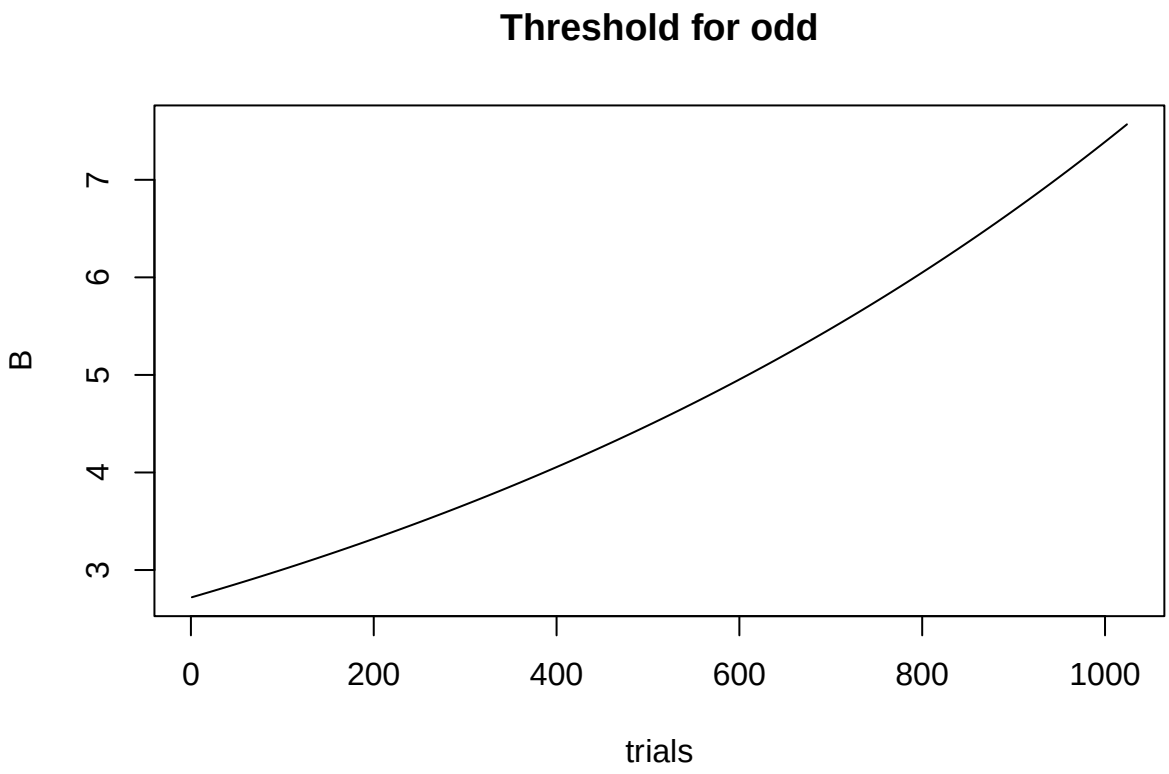
Let's clarify with an example. Let's try to impose a linear trend on thresholds in case of the RDM:

```r
# Add a 'trials' column specifying trial number
dat <- EMC2:::add_trials(dat)

# Rescale the effect of this covariate to a larger parameter range to help sampling
dat$trials2 <- dat$trials/1000
lin_trend <- make_trend(cov_names='trials2',
                        kernels = 'lin_incr',
                        par_names='B',
                        bases='lin',
                        premap=TRUE)

design_RDM_lin_B <- design(model=RDM,
                           data=dat,
                           contrast=list(lM=ADmat),
                           covariates='trials2',   # specify relevant covariate columns
                           matchfun=function(d) d$S==d$lR,
                           formula=list(B ~ 1, v ~ lM, t0 ~ 1),
                           trend=lin_trend)       # add trend
```

```
 Sampled Parameters:
[1] "B"     "v"     "v_lMd" "t0"    "B.w"

 Design Matrices:
$B
 B
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$B.w
 B.w
   1

$A
 A
 1

$s
 s
 1
```

Note that we are now sampling one extra parameter compared to before, `B.w`, which is the weight of the influence of trial number on thresholds. In $B_t = B_0 + B.w * trial$. Let's define a set of parameters and look at the resulting trial-by-trial thresholds:

```r
samplers <- make_emc(dat, design=design_RDM_lin_B)
p_vector <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'B.w'=1)

# Visualize trend
plot_trend(p_vector, samplers=samplers,
           par_name='B', subject=1,
           lR_filter='odd', main='Threshold for odd')
```



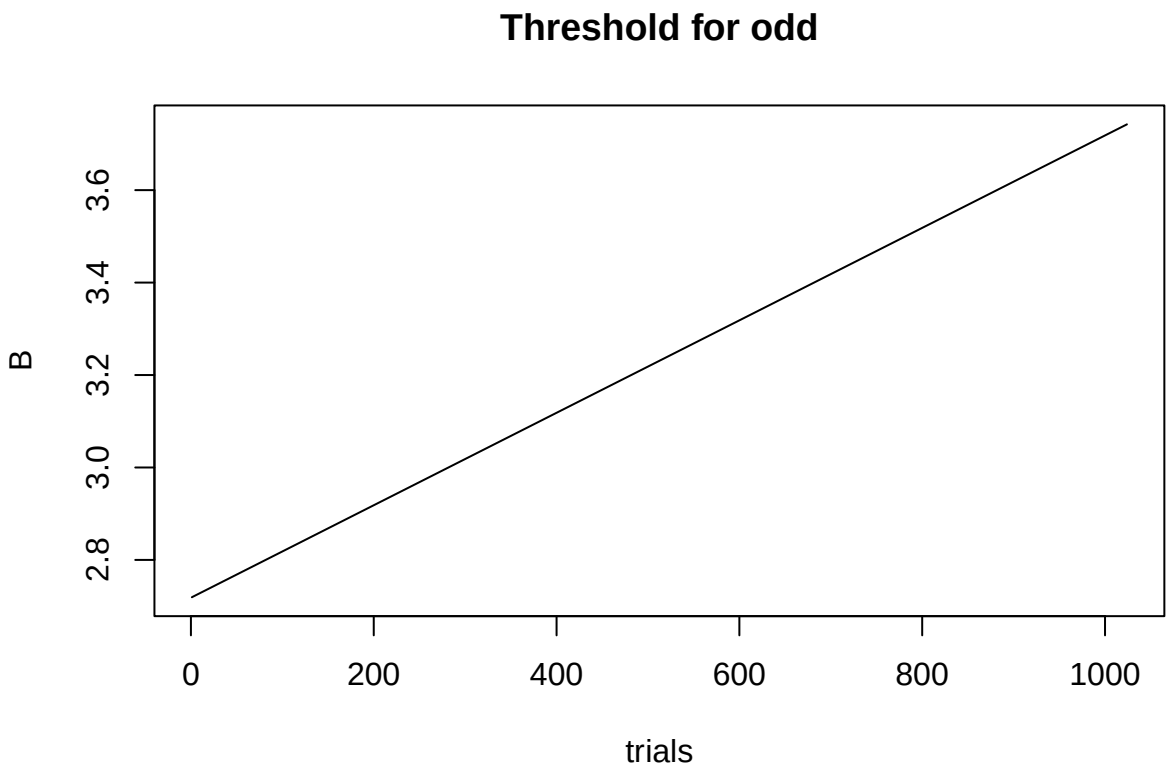
Clearly, this is not a linear increase. This happens because the threshold is sampled on the log scale, so an exponential transform is applied after mapping the parameter vector to the design cells. Since the linear trend was applied prior to mapping, this linear effect becomes non-linear.

One option to prevent this from happening is to apply the trend after mapping and transformation, as follows:

```
lin_trend2 <- make_trend(cov_names='trials2',
                         kernels = 'lin_incr',
                         par_names='B',
                         bases='lin',
                         premap=FALSE, pretransform=FALSE)
design_RDM_lin_B2 <- design(model=RDM,
                            data=dat,
                            contrast=list(lM=ADmat),
                            covariates='trials2',    # specify relevant covariate columns
                            matchfun=function(d) d$S==d$lR,
                            formula=list(B ~ 1, v ~ lM, t0 ~ 1),
                            trend=lin_trend2)        # add trend

 Sampled Parameters:
[1] "B"     "v"     "v_lMd" "t0"    "B.w"

 Design Matrices:
$B
 B
 1
```

```
$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$B.w
 B.w
   1

$A
 A
 1

$s
 s
 1
```

```
samplers <- make_emc(dat, design=design_RDM_lin_B2)
p_vector <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'B.w'=1)
# Visualize trend
plot_trend(p_vector, samplers=samplers,
           par_name='B', subject=1,
           lR_filter='odd', main='Threshold for odd')
```

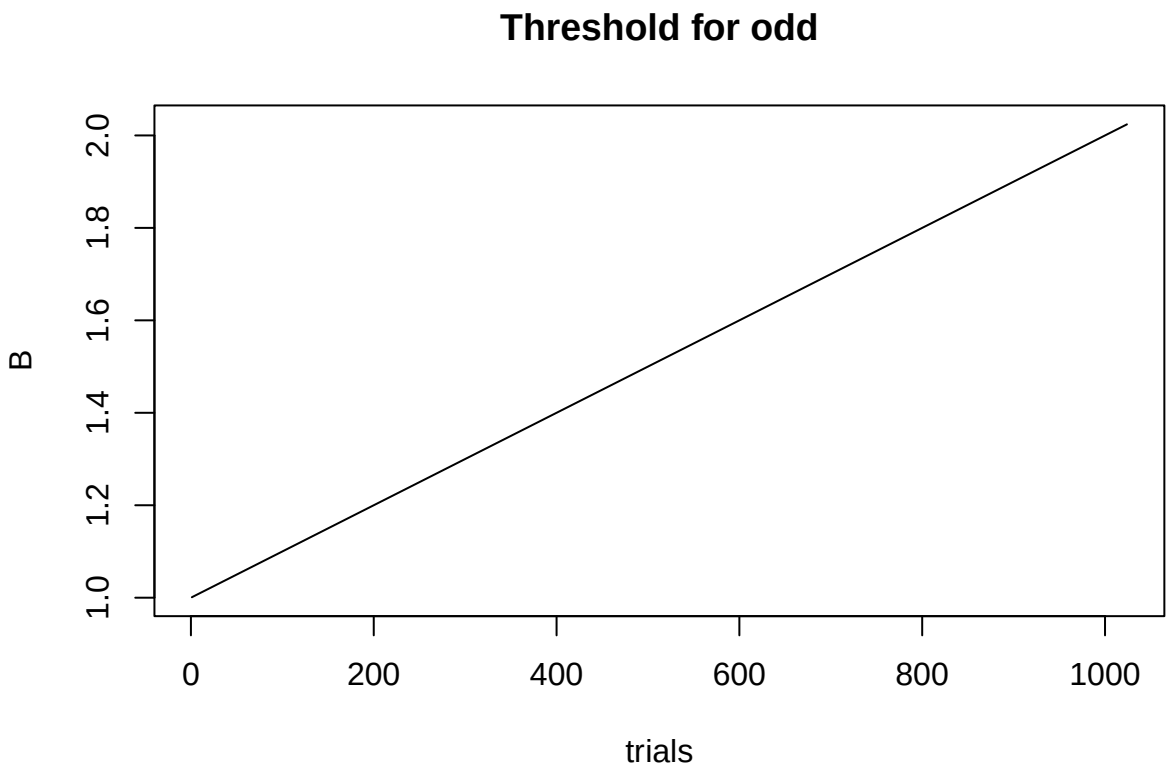


However, posttransform implies postmap, and many parameters of interest are defined only prior to mapping. So applying trends to those premap parameter types is not possible in combination with posttransform. Instead, the user can tell *EMC2* to estimate parameters on their natural scales by turning off transformations of the relevant parameters. For example:

```
lin_trend3 <- make_trend(cov_names='trials2',
                         kernels = 'lin_incr',
                         par_names='B',
                         bases='lin',
                         premap=TRUE)  # back to premap
design_RDM_lin_B3 <- design(model=RDM,
                            data=dat,
                            contrast=list(lM=ADmat),
                            covariates='trials2',
                            matchfun=function(d) d$S==d$lR,
                            # here, we tell *EMC2* to sample the threshold on the natural scale
                            transform=list(func=c('B'='identity')),
                            formula=list(B ~ 1, v ~ lM, t0 ~ 1),
                            trend=lin_trend3)       # add trend

 Sampled Parameters:
[1] "B"     "v"     "v_lMd" "t0"    "B.w"

 Design Matrices:
$B
 B
```

```
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$B.w
 B.w
   1

$A
 A
 1

$s
 s
 1
```

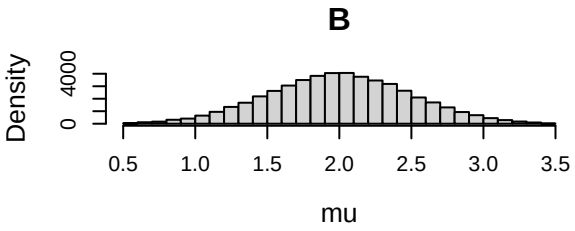
```r
samplers <- make_emc(dat, design=design_RDM_lin_B3)
p_vector <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'B.w'=1)

# visualize trend
plot_trend(p_vector, samplers=samplers,
           par_name='B', subject=1,
           lR_filter='odd', main='Threshold for odd')
```

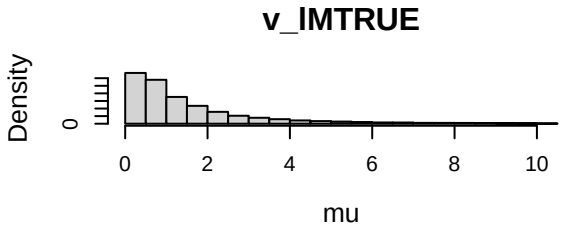


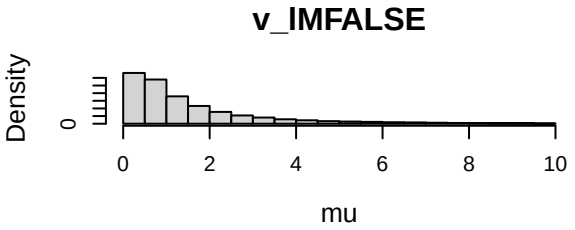
This leads to the expected effect. As a cautionary note, keep in mind that the default priors in *EMC2* are Gaussian distributions centered on 0 with unit variance. Since many cognitive model parameters cannot be negative, a N(0,1) prior is poorly chosen for those parameters that do not have support on the real line. For estimation and sampling, this usually has little influence in practice, but it may be important when estimating Bayes Factors.

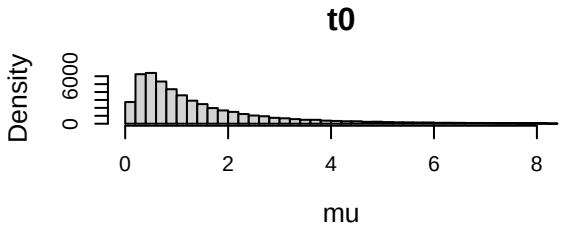
With all that in mind, we can now start sampling our first model. We set a N(2,0.5) prior on the threshold B to reduce the prior density on negative thresholds. This is a somewhat subjective choice based on earlier experience, so it reflects my (but perhaps not your) prior belief. The rest of the priors are left to their default N(0,1) – on the scale on which they are sampled.

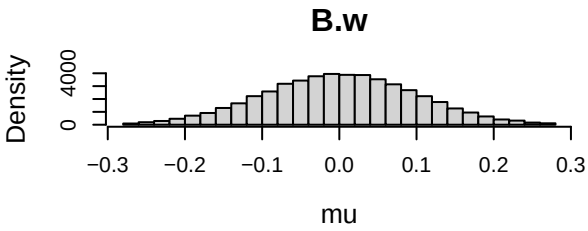
```
prior_linB <- prior(design_RDM_lin_B3, mu_mean=c(B=2, B.w=0), mu_sd=c(B=0.5,B.w=0.1))
samplers <- make_emc(dat, design=design_RDM_lin_B3, prior_list = prior_linB)
plot(samplers, prior=TRUE)
```



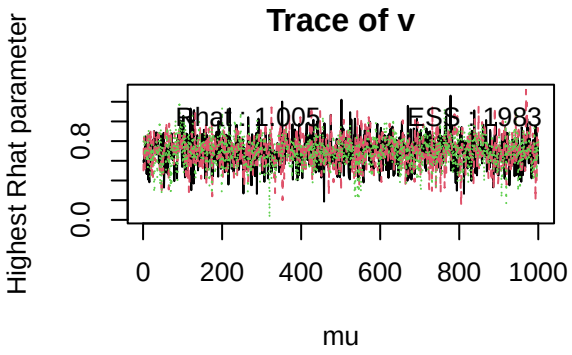


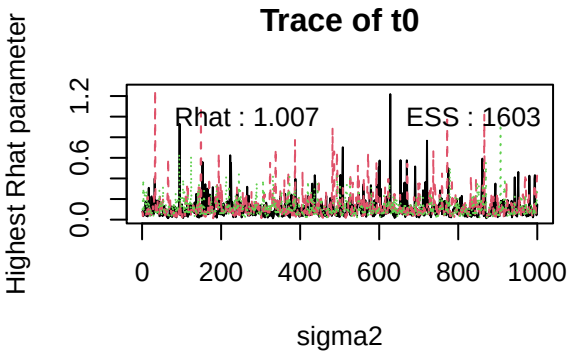


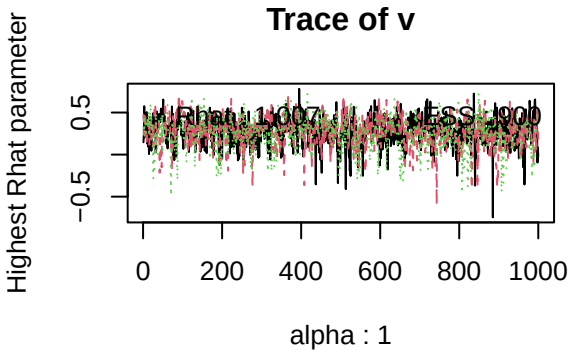




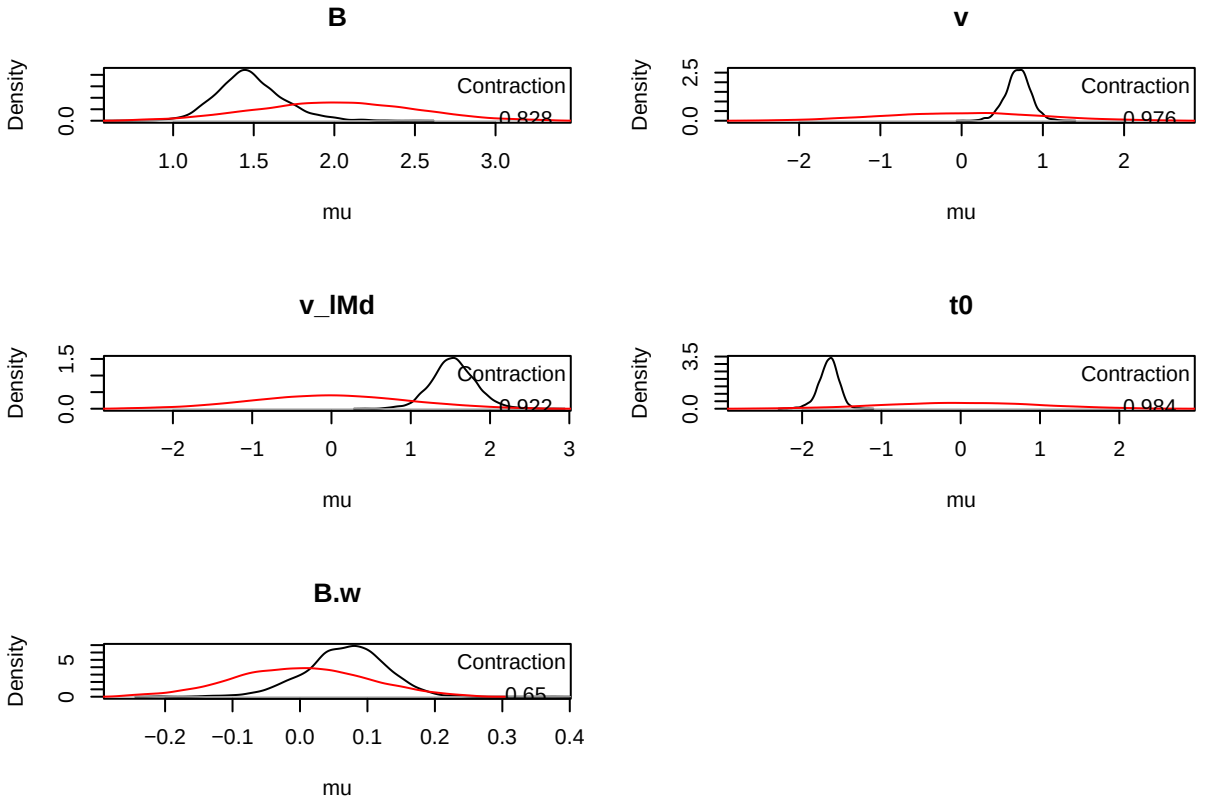

```
samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
                fileName='./samples/ds1_linB.RData')

check(samplers)

Iterations:
     preburn burn adapt sample
[1,]       0    0     0   1000
[2,]       0    0     0   1000
[3,]       0    0     0   1000

 mu
            B        v    v_lMd       t0      B.w
Rhat    1.002    1.005    1.004    1.001    1.001
ESS  2487.000 1983.000 1937.000 1841.000 2180.000

 sigma2
            B        v    v_lMd       t0      B.w
Rhat    1.005    1.004    1.003    1.007    1.003
ESS  2033.000 1526.000 1756.000 1603.000 1280.000

 alpha highest Rhat : 1
        B       v   v_lMd   t0      B.w
Rhat    1   1.007   1.006    1    1.001
ESS  1527 900.000 870.000 1983 2244.000
```








```
plot_pars(samplers)
```




```
credint(samplers)
```

```
$mu
        2.5%    50%  97.5%
B      1.103  1.462  1.933
v      0.385  0.696  0.997
v_lMd  0.994  1.537  2.100
t0    -1.950 -1.658 -1.447
B.w   -0.057  0.070  0.177
```

On a group-level, we find a small (not credible) positive `B.w` – on average, in this dataset, thresholds appear to increase by ~0.070 over the course of 1000 trials.

Other functional forms

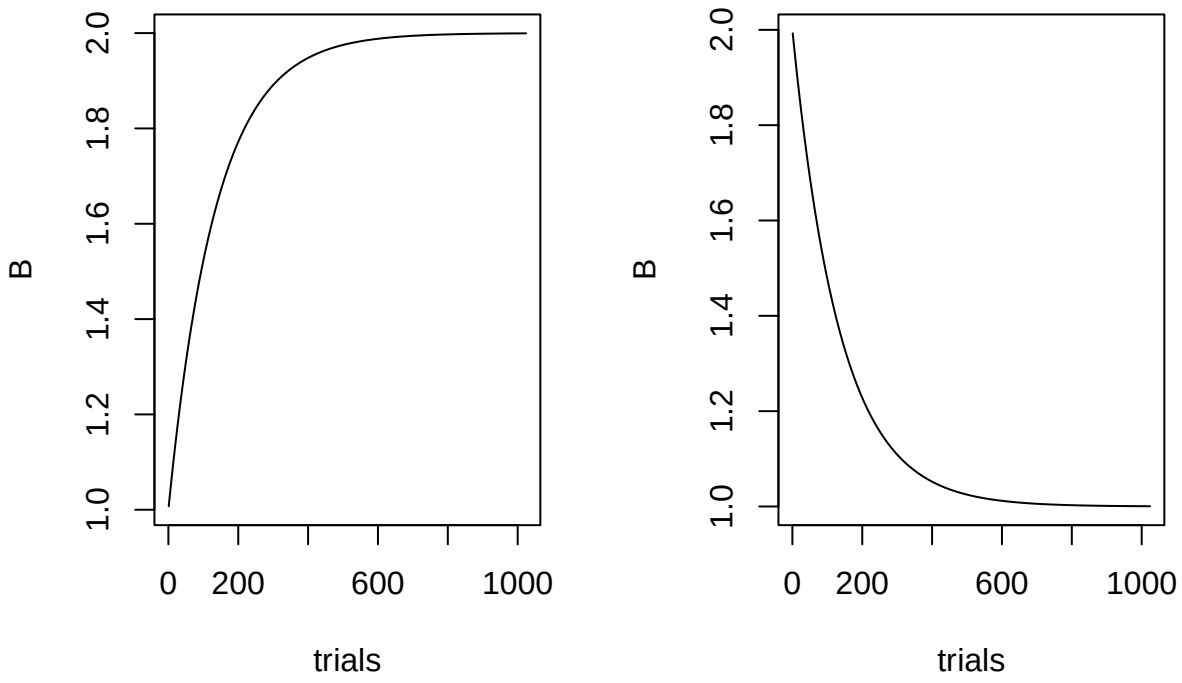
Practice effects tend to have large effects initially and then gradually decay. Asymptotic functions like exponential or power functions can be used to model such effects. *EMC2* offers increasing and decreasing kernels in both cases. To demonstrate these functional forms:

```
trend_exp_incr <- make_trend(par_names='B', cov_names = 'trials2',
                             kernels = 'exp_incr', bases = 'lin')
trend_exp_decr <- make_trend(par_names='B', cov_names = 'trials2',
                             kernels = 'exp_decr', bases = 'lin')
design_exp_incr <- design(model=RDM,
```

```
                        data=dat,
                        contrast=list(lM=ADmat),
                        covariates='trials2',
                        matchfun=function(d) d$S==d$lR,
                        transform=list(func=c('B'='identity')),
                        formula=list(B ~ 1, v ~ lM, t0 ~ 1),
                        trend=trend_exp_incr)

 Sampled Parameters:
[1] "B"      "v"      "v_lMd"  "t0"     "B.w"    "B.d_ei"

 Design Matrices:
$B
 B
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$B.w
 B.w
   1

$B.d_ei
 B.d_ei
      1

$A
 A
 1

$s
 s
 1

design_exp_decr <- design(model=RDM,
                        data=dat,
                        contrast=list(lM=ADmat),
                        covariates='trials2',
                        matchfun=function(d) d$S==d$lR,
                        transform=list(func=c('B'='identity')),
                        formula=list(B ~ 1, v ~ lM, t0 ~ 1),
                        trend=trend_exp_decr)

 Sampled Parameters:
[1] "B"      "v"      "v_lMd"  "t0"     "B.w"    "B.d_ed"
```

```
 Design Matrices:
$B
 B
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$B.w
 B.w
   1

$B.d_ed
 B.d_ed
      1

$A
 A
 1

$s
 s
 1
```

```
samplers_incr <- make_emc(dat, design_exp_incr)
samplers_decr <- make_emc(dat, design_exp_decr)

p_vector_incr <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'B.w'=1, 'B.d_ei'=2)
p_vector_decr <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'B.w'=1, 'B.d_ed'=2)

par(mfrow=c(1,2))
plot_trend(p_vector_incr, samplers=samplers_incr, par_name='B', subject=1, lR_filter='odd')
plot_trend(p_vector_decr, samplers=samplers_decr, par_name='B', subject=1, lR_filter='odd')
```



Note that the interpretation of the exponent (`B.e_ei` or `B.e_ed`) depends on the direction: In the increasing case, it corresponds to the asymptote; in the decreasing case, to the intercept.

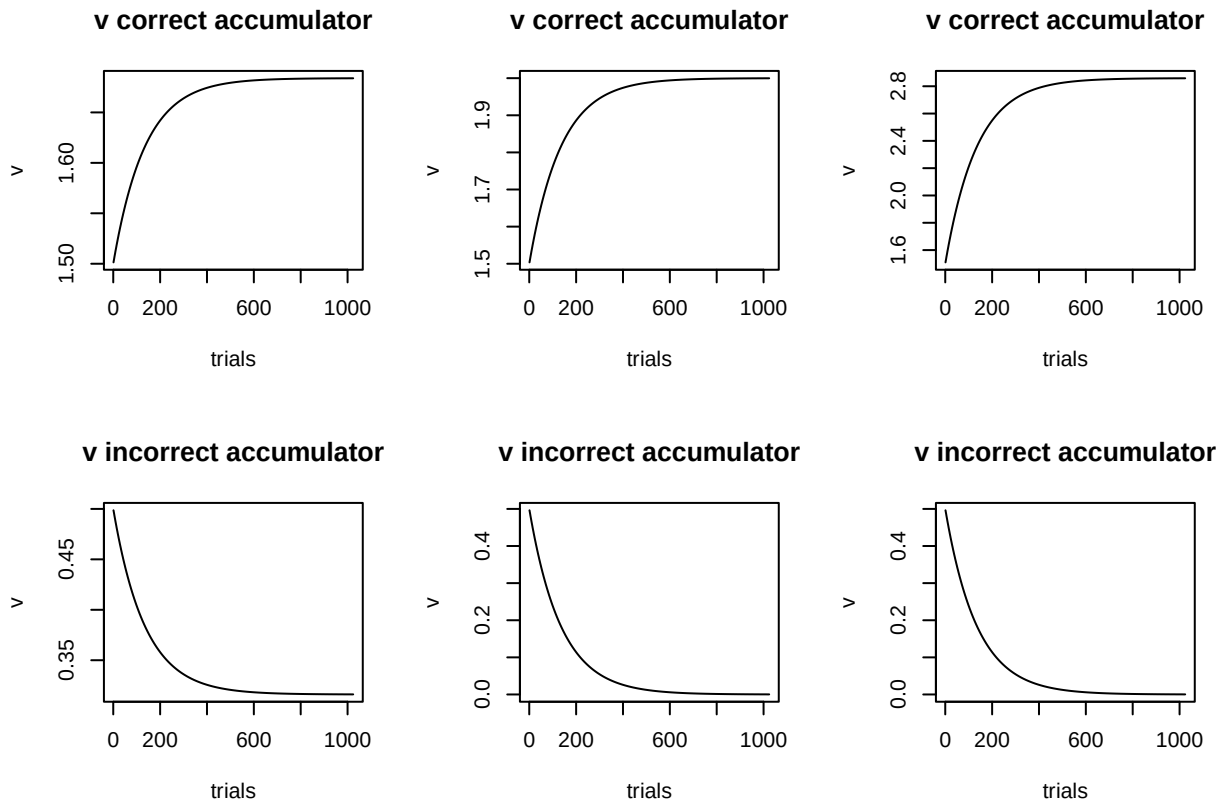
Advanced note: Enforcing a direction

Having both increasing and decreasing kernels may appear redundant, since the direction of the effect can also be flipped by the 'w' parameter in the 'lin' base. However, having both increasing and decreasing kernels makes it easy to enforce the direction of an effect. For example, we could hypothesize that 'v_lMd' increases sharply over the first few trials due to practice effects, and then stabilizes. When used in combination with the 'lin' base, the model would have the flexibility to find both increases and decreases, even though our hypothesis only allows for increases, not decreases.
We can force the sampler to find only increases by applying an exponential transform to the weight parameter 'w' of the linear base. Note that this transformation needs to be applied *prior* to the application of the trend. For this reason, this is provided as a 'pre_transform' in 'design()':

```
trend_exp_incr <- make_trend(par_names='v_lMd', cov_names = 'trials2',
                             kernels = 'exp_incr', bases = 'lin')
design_exp_incr <- design(model=RDM,
                          data=dat,
                          contrast=list(lM=ADmat),
                          covariates='trials2',
                          matchfun=function(d) d$S==d$lR,
                          pre_transform=list(func=c('v_lMd.w'='exp')),
                          transform=list(func=c('v'='identity')),
                          formula=list(B ~ 1, v ~ lM, t0 ~ 1),
                          trend=trend_exp_incr)
```

```
 Sampled Parameters:
[1] "B"          "v"          "v_lMd"      "t0"         "v_lMd.w"
[6] "v_lMd.d_ei"

 Design Matrices:
$B
 B
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$v_lMd.w
 v_lMd.w
       1

$v_lMd.d_ei
 v_lMd.d_ei
          1

$A
 A
 1

$s
 s
 1
```

```
samplers_incr <- make_emc(dat, design_exp_incr)
p_vector1 <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'v_lMd.w'=-1, 'v_lMd.d_ei'=2)
p_vector2 <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'v_lMd.w'=0, 'v_lMd.d_ei'=2)
p_vector3 <- c('B'=1, 'v'=1, 'v_lMd'=1, 't0'=0.1, 'v_lMd.w'=1, 'v_lMd.d_ei'=2)

par(mfcol=c(2,3))
plot_trend(p_vector1, samplers=samplers_incr, par_name='v', subject=1, lM_filter=TRUE, main='v correct a
plot_trend(p_vector1, samplers=samplers_incr, par_name='v', subject=1, lM_filter=FALSE, main='v incorre
plot_trend(p_vector2, samplers=samplers_incr, par_name='v', subject=1, lM_filter=TRUE, main='v correct a
plot_trend(p_vector2, samplers=samplers_incr, par_name='v', subject=1, lM_filter=FALSE, main='v incorre
plot_trend(p_vector3, samplers=samplers_incr, par_name='v', subject=1, lM_filter=TRUE, main='v correct a
plot_trend(p_vector2, samplers=samplers_incr, par_name='v', subject=1, lM_filter=FALSE, main='v incorre
```



Note that, whatever value of 'v_lMd.w' the sampler finds, it results in an increasing difference between drift rates.

So far, we only inspected time on task as defined by trial number. Other options include time on task within block (e.g., short breaks between trials could cause an increased threshold for the first few trials in each block), or the number of times a specific stimulus type has been shown. Furthermore, the parameters describing the trend could differ between blocks or even stimulus types. To test such hypotheses, we can allow the trend parameters themselves to vary with experimental factors, as defined in `formula` in `design()`. The example data does not have stimulus type recorded, and all trials were run in a single block, but for demonstrative purposes we can simulate such effects:

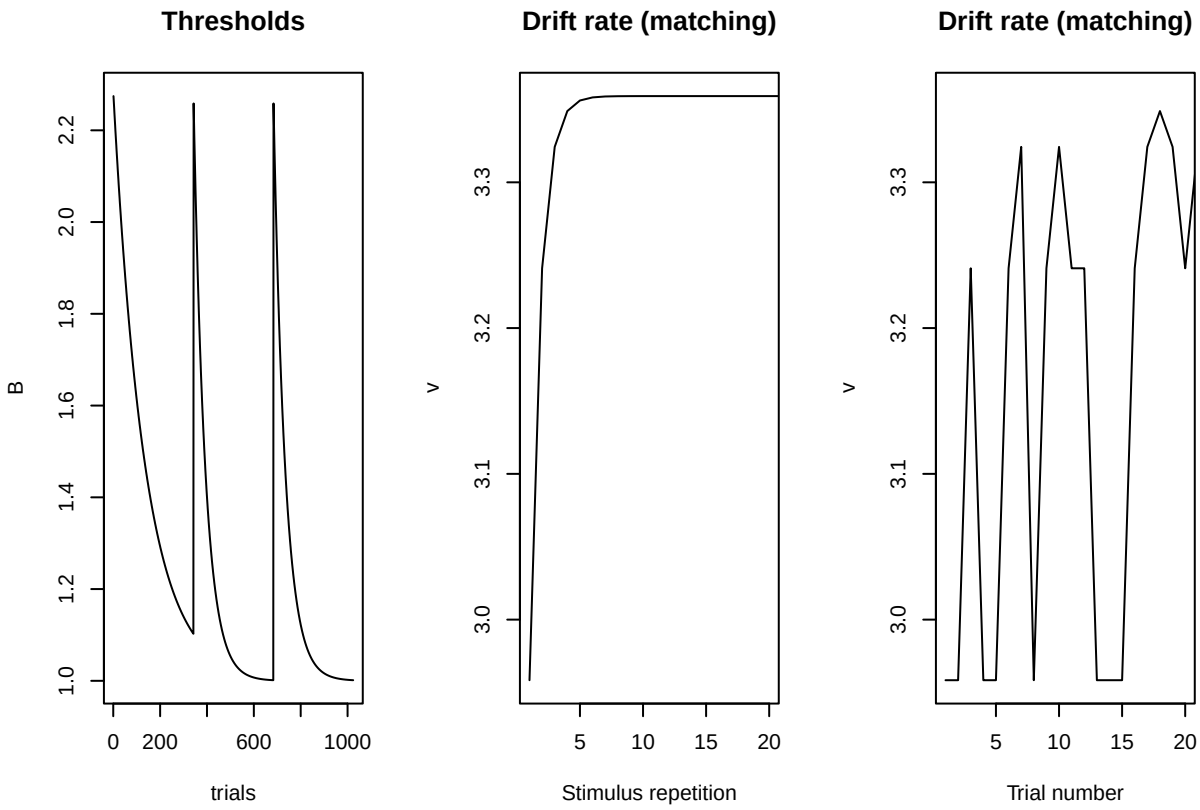
```
# simulate block number (assume 3 blocks)
dat$block <- as.factor(as.numeric(cut(dat$trials, breaks=3)))
for(subject in unique(dat$subjects)) {
  dat[dat$subjects==subject,'trial_in_block'] <- ave(seq_along(dat[dat$subjects==subject,'block']), dat
}
# simulate presented digit
dat$stimulus_number <- sample(c(1,2,3,4,6,7,8,9), size=nrow(dat), replace=TRUE)
dat$stimulus_repetition <- ave(seq_along(dat$stimulus_number), dat$stimulus_number, FUN = seq_along)

# combine two trends
trends <- make_trend(par_names=c('B', 'v_lMd'),
                     cov_names=c('trial_in_block', 'stimulus_repetition'),
                     kernels = c('exp_decr', 'exp_incr'),
                     bases = c('lin', 'lin'), )
design_multitrend <- design(model=RDM,
```

```
                        data=dat,
                        contrast=list(lM=ADmat),
                        covariates=c('trial_in_block', 'stimulus_repetition'),
                        matchfun=function(d) d$S==d$lR,
                        pre_transform=list(func=c('v_lMd.w'='exp',
                                                  'B.w'='exp')),
                        transform=list(func=c('v'='identity',
                                              'B'='identity')),
                        formula=list(B ~ 1, v ~ lM, t0 ~ 1, `B.d_ed`~block),
                        trend=trends)
```

```
 Sampled Parameters:
 [1] "B"             "v"             "v_lMd"         "t0"
 [5] "B.d_ed"        "B.d_ed_block2" "B.d_ed_block3" "B.w"
 [9] "v_lMd.w"       "v_lMd.d_ei"

 Design Matrices:
$B
 B
 1

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
  1

$B.d_ed
 block B.d_ed B.d_ed_block2 B.d_ed_block3
     1      1             0             0
     2      1             1             0
     3      1             0             1

$B.w
 B.w
   1

$v_lMd.w
 v_lMd.w
       1

$v_lMd.d_ei
 v_lMd.d_ei
          1

$A
 A
 1

$s
 s
```

```
1
```

```r
samplers_multitrend <- make_emc(dat, design_multitrend)

# thresholds
p_vector <- c('B'=1, 'v'=1, 'v_lMd'=2, 't0'=0.1,
              'B.d_ed'=2, 'B.d_ed_block2'=1, 'B.d_ed_block3'=1, 'B.w'=.25,
              'v_lMd.w'=1, 'v_lMd.d_ei'=.2)

par(mfcol=c(1,3))
plot_trend(p_vector, samplers=samplers_multitrend, par_name='B', subject=1, lR_filter='odd', main='Thres
plot_trend(p_vector, samplers=samplers_multitrend, par_name='v', subject=1, lM_filter=TRUE, main='Drift
plot_trend(p_vector, samplers=samplers_multitrend, par_name='v', subject=1, lM_filter=TRUE, main='Drift
```



Stimulus memory: Trend specification

Models with formal mechanisms of dynamics rely on delta rules. To implement delta rules, we follow the same general logic as above—the delta rule is implemented as a kernel in `make_trend`. Let's implement the stimulus memory mechanism proposed by Miletic et al. (2025). Here, the decision maker keeps track of the probability that a stimulus is even (or odd). The covariate in this case is the stimulus, which we can recode as 1 (even) and -1 (odd). With the delta rule:

$$Q_{SM,t+1} = Q_{SM,t} + \alpha_{SM} \cdot (S_t - Q_{SM,t}), \quad S \in \{-1, 1\}$$

The delta rule introduces two extra parameters: a learning rate α_{SM}, and a value for Q at the start of the experiment $Q_{SM,0}$. The latter is hard to estimate, and we tend to fix it to a constant—for this rule, a constant of 0 implies equal beliefs in both stimuli.

For now, let us assume that stimulus memory influences the relative thresholds: the even threshold increases, and the odd threshold decreases (by the same amount). To achieve this, we need to parameterize thresholds with the same mean-difference parameterization as we used for drift rates. That is,

$$B_{odd} = B_{mean} + B_{lRd}$$
$$B_{even} = B_{mean} - B_{lRd}$$

The B_{lRd} term is the parameter influenced by the updated covariate—however, since we do not necessarily expect an across-trial *mean* difference between thresholds, we set B_{lRd} as a constant to 0.

Using a linear base, we then allow the threshold difference to vary on a trial-by-trial basis with

$$B_{lRd,t} = B_{lRd} + w_{SM} \cdot Q_{SM,t}$$

Note the extra parameter here: w_{SM}.

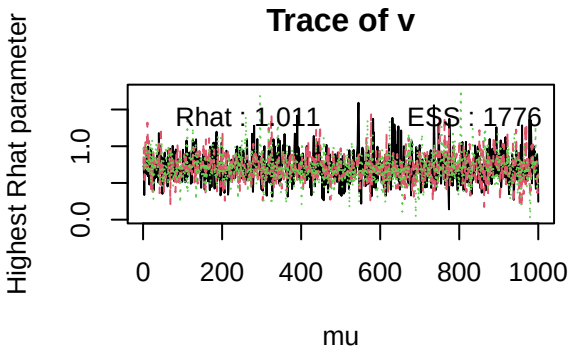
```
dat$Stim1 <- ifelse(dat$S=='even', 1, -1)
SM_trend <- make_trend(cov_names=c('Stim1'),
                       kernels = 'delta',
                       par_names='B_lRd',
                       bases='lin',
                       premap=TRUE,
                       filter_lR=TRUE)  # see advanced notes for an explanation of what this does.
design_RDM_SM <- design(model=RDM,
                        data=dat,
                        contrast=list(lM=ADmat, lR=ADmat),
                        covariates='Stim1',   # specify relevant covariate columns
                        matchfun=function(d) d$S==d$lR,
                        formula=list(B ~ lR, v ~ lM, t0 ~ 1),
                        trend=SM_trend,
                        constants=c('B_lRd'=0, 'B_lRd.q0'=0))
```

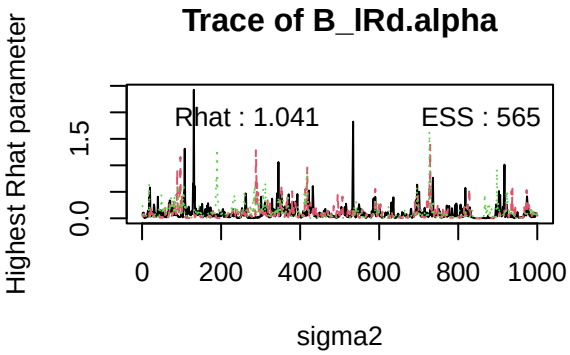
```
 Sampled Parameters:
[1] "B"           "v"           "v_lMd"       "t0"          "B_lRd.w"
[6] "B_lRd.alpha"

 Design Matrices:
$B
   lR B B_lRd
 even 1  -0.5
  odd 1   0.5

$v
    lM v v_lMd
  TRUE 1   0.5
 FALSE 1  -0.5

$t0
 t0
```

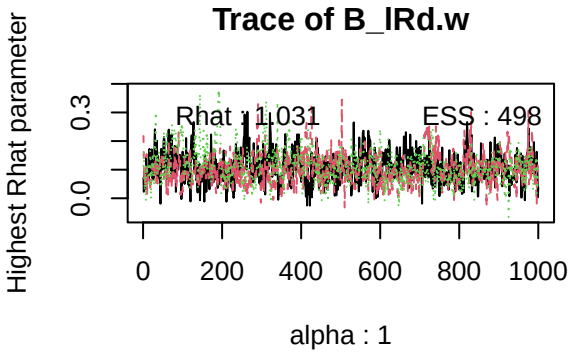
```
  1

$B_lRd.w
 B_lRd.w
       1

$B_lRd.q0
 B_lRd.q0
        1

$B_lRd.alpha
 B_lRd.alpha
           1

$A
 A
 1

$s
 s
 1
```

We can now apply a similar prior to B as above, and fit the model. Note data compression is turned off automatically, since the delta rule needs to be applied to the covariate on each trial individually.

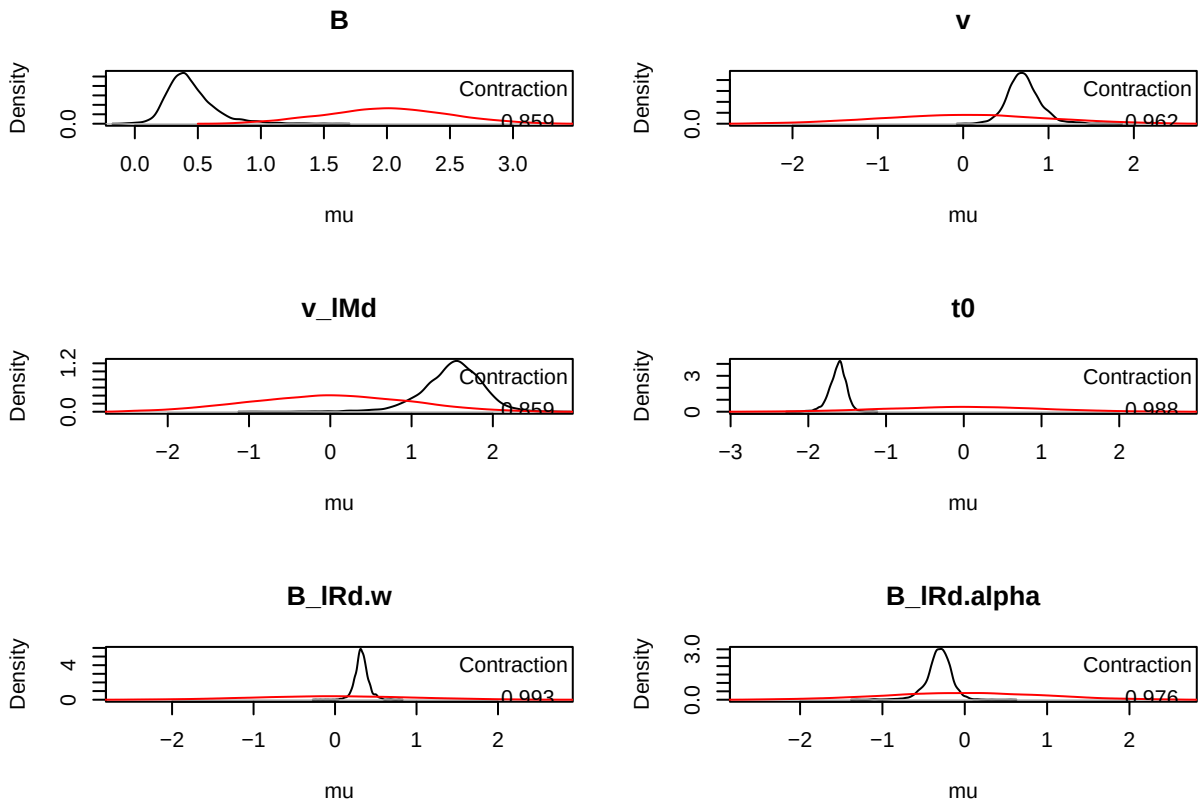
```
prior_SMB <- prior(design_RDM_SM, mu_mean=c(B=2), mu_sd=c(B=0.5))
samplers <- make_emc(dat, design=design_RDM_SM, prior_list = prior_SMB, compress=FALSE)

samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
                fileName='./samples/ds1_SMB.RData')

check(samplers)

Iterations:
     preburn burn adapt sample
[1,]       0    0     0   1000
[2,]       0    0     0   1000
[3,]       0    0     0   1000

 mu
            B        v    v_lMd       t0  B_lRd.w B_lRd.alpha
Rhat    1.003    1.011    1.002    1.003    1.008        1.01
ESS  1098.000 1776.000 1585.000 1309.000 1489.000      862.00

 sigma2
           B        v    v_lMd       t0 B_lRd.w B_lRd.alpha
Rhat   1.007    1.007    1.004    1.001   1.023       1.041
ESS  863.000 1494.000 1360.000 1312.000 835.000     565.000

 alpha highest Rhat : 1
            B       v  v_lMd       t0 B_lRd.w B_lRd.alpha
Rhat    1.005   1.012   1.01    1.007   1.031       1.028
ESS  1751.000 675.000 695.00 1883.000 498.000     363.000
```








```
plot_pars(samplers)
```



We find excellent convergence properties. Note that learning rates are by default sampled on the probit scale, so if we want to know the mean learning rate, we need to map and transform the parameters first:

```
credint(samplers, map=TRUE)
```

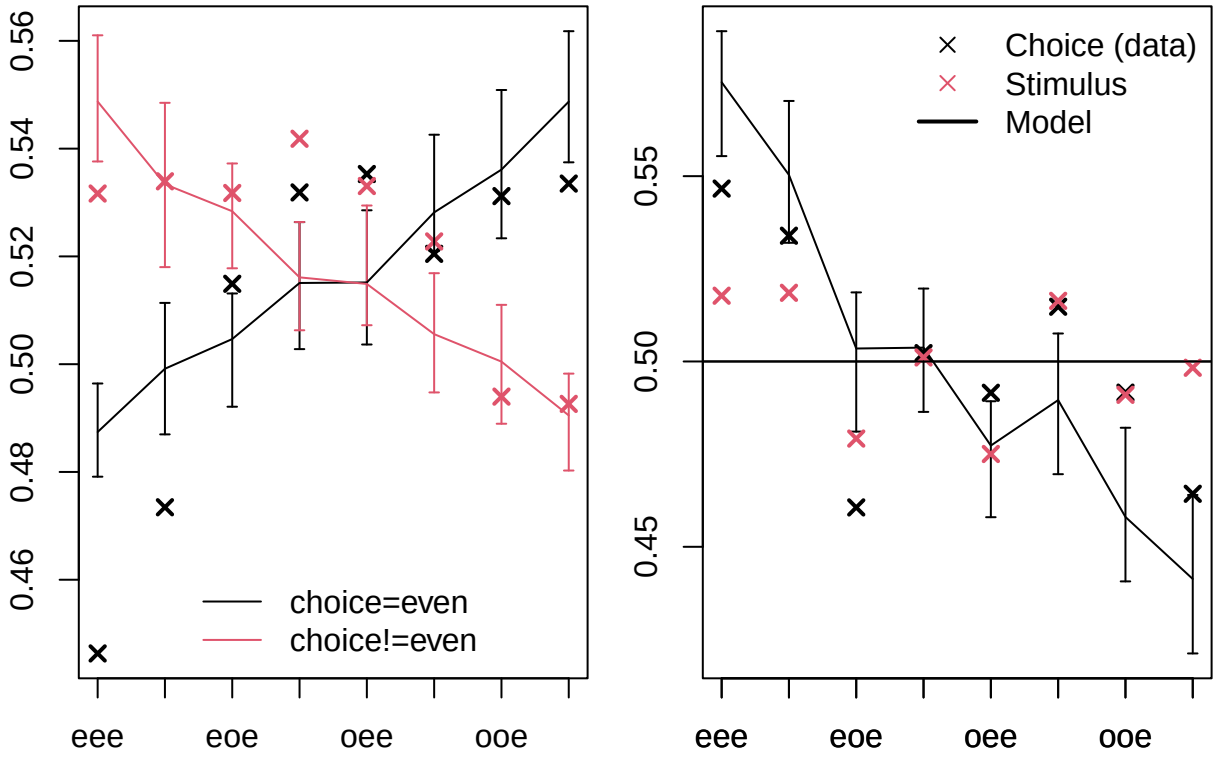
```
$mu
              2.5%   50% 97.5%
B_lReven     1.169 1.503 2.489
B_lRodd      1.169 1.503 2.489
v_lMTRUE     3.432 4.325 5.222
v_lMFALSE    0.498 0.937 2.160
t0           0.153 0.199 0.239
B_lRd.w      0.160 0.324 0.508
B_lRd.alpha  0.258 0.380 0.482
```

So we find a learning rate of ~0.368, and the threshold of the odd accumulator is 0.333 lower than the threshold for the even accumulator if the participant has a Q_{SM}-value of 1.

We can now check whether the SM mechanism can explain stimulus-history effects by plotting the RTs (by choice) and the probability of choosing left as a function of the previous three presented stimuli. First, generate posterior predictives:

```
pp <- predict(samplers, n_cores=10)
save(pp, file='./ds1_SMB_pps.RData')
```

```
load('./ds1_SMB_pps.RData')
par(mar=c(3,2,2,1))
plot_history_effects(dat, pp, n_hist=3)
```



The left panel of this plot shows the mean RT as a function of the three previous trials' stimuli (ooo = odd, odd, odd; ooe = odd, odd, even; etc.). The data show a cross-over pattern, such that if the previous three stimuli were odd, 'odd' responses are faster than 'even' responses; and vice versa. The model tends to capture this cross-over pattern fairly well, with some misfits mostly due to an apparent asymmetry in the cross-over in the data.

The right panel shows the probability of choosing even (black) and the probability of the stimulus being even (red) as a function of local trial history. The black line demonstrates that participants are more likely (~.55) to choose 'even' if the previous three trials were 'even'. The red crosses are important as a quality check. If the stimuli are generated truly randomly, the red crosses should be approximately 0.5 in all cases. However, some experiments use pseudorandomization, which can lead to negative correlations in local stimulus histories. This can confound any stimulus history effects present in the human data. True stimulus history effects are those that are larger than the ones present in the data, e.g., visible as 'slopes' in data that are steeper than the stimulus history contingencies in the experiment design.

Advanced note: filter_lR

Under the hood, EMC2 works with data-augmented design matrices (`dadms`). A `dadm` contains one row per accumulator per trial – so for race models applied to two-alternative forced choice tasks, this will typically result in two rows per trial. We can inspect the `dadm` for the first subject in the samplers object to see that the covariate `Stim1` is indeed the same across accumulators in each trial:

```
head(samplers[[1]]$data[[1]])
```

```
  subjects    S    R   rt trials trials2 Stim1   lR    lM winner
1        1 even even 0.54      1   0.001     1 even  TRUE   TRUE
2        1 even even 0.54      1   0.001     1  odd FALSE  FALSE
3        1 even even 0.42      2   0.002     1 even  TRUE   TRUE
4        1 even even 0.42      2   0.002     1  odd FALSE  FALSE
5        1 even even 0.50      3   0.003     1 even  TRUE   TRUE
6        1 even even 0.50      3   0.003     1  odd FALSE  FALSE
  Stim1_lRfiltered
1                1
2               NA
3                1
4               NA
5                1
6               NA
```

The delta rule is applied to the covariate as it is specified in the `dadm`. Since, in the examples in this tutorial, the covariate is the same for both accumulators, blindly applying a delta rule to the covariate in the `dadm` would lead to *two* updates every trial: One for the first accumulator, and then another one for the second accumulator. The Q-values would then also differ between accumulators. The correct behavior is to update only once every trial (i.e., for the first accumulator), and then feed forward the updated Q-value to every second (and third, fourth, ...) accumulator. This can be done by setting the covariate for every second, third, fourth, etc accumulator to NA, since NA-values are ignored when updating. `filter_lR=TRUE` does this: it creates a new column `covariate_lRfiltered`, and applies the delta rule to that column, feeding forward updated Q-values when encounting an NA-value.
In many applications when using dynamic EAMs, this is likely the intended behavior. However, there are two exceptions: 1) The DDM only has one accumulator per trial, so no filtering has to be done (or can be done); and 2) when using RL-EAMs, applied to instrumental learning tasks, the covariate can differ between the two accumulators (feedback can be given to one specific option, for example). In such cases, more advanced handling of the covariate is required, and `filter_lR` should not be TRUE.

Stimulus memory: DDM and drift rate effects

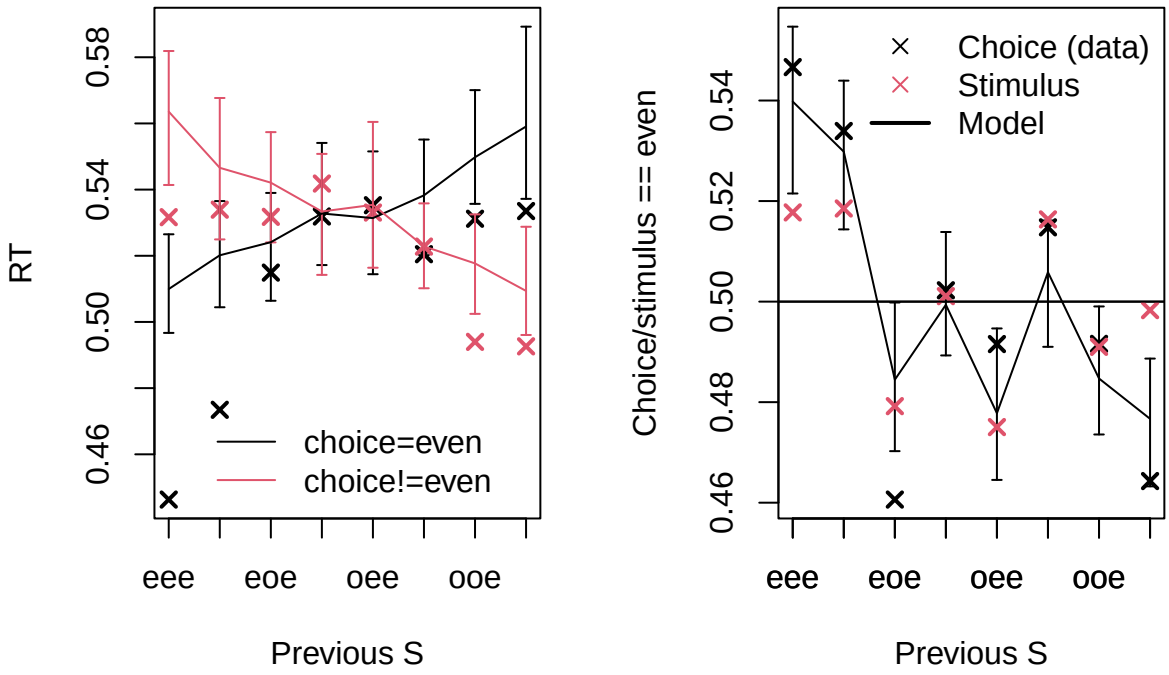
The example above replicates Miletic et al. (2025) by assuming an RDM and assuming that stimulus memory causes a threshold (start point) bias. *EMC2* is flexibly designed to implement your own hypotheses. For example, we can propose a DDM instead and hypothesize that drift rates rather than start points are biased. This can be implemented as follows:

```
trend_SM_DDM <- make_trend(cov_names='Stim1',
                           kernels = 'delta',
                           par_names='v',
                           bases='lin',
                           premap=TRUE)
Smat <- matrix(c(-1,1), nrow = 2,dimnames=list(NULL,"dif"))
design_DDM <- design(model=DDM,
                     data=dat,
                     covariates=c('Stim1'),
                     contrasts=list(S=Smat),
                     formula=list(v ~ S, a~1, t0 ~ 1),
                     constants=c('v.q0'=0, 'v'=0),
                     trend=trend_SM_DDM)

# since v is estimated on the natural scale (in case of the DDM),
```

```
# we do not need to change the transformations, and we can use the default priors.
samplers <- make_emc(dat, design=design_DDM, compress=FALSE)
samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
                fileName='./samples/ds1_SMv_DDM.RData')
```

However, this does not appear to fit as well qualitatively and loses model comparisons:

```
pp_ddm <- predict(samplers, n_cores=10)
save(pp_ddm, file='./samples/ds1_SMv_DDM_pps.RData')

plot_history_effects(dat, pp_ddm)
```




```
compare(sList=list(rdm=get(load('./samples/ds1_SMB.RData')),
                   ddm=get(load('./samples/ds1_SMv_DDM.RData'))))
```

```
       MD wMD   DIC wDIC  BPIC wBPIC EffectiveN meanD Dmean  minD
rdm -6906   1 -7026    1 -6955     1         71 -7097 -7123 -7168
ddm -5267   0 -5344    0 -5272     0         73 -5417 -5434 -5490
```

Accuracy memory: Trend specification

Accuracy memory follows the same logic as above, with a different covariate (errors) and a different parameter (urgency, v). Errors are the inverse of accuracy; they are 0 when the response matches the stimulus, and 1

otherwise. Unlike the presented stimulus, accuracy/error is not a static property of the experimental design but depends on the observed behavior. This also means that, when simulating behavior, accuracy needs to be re-calculated based on the newly simulated data.

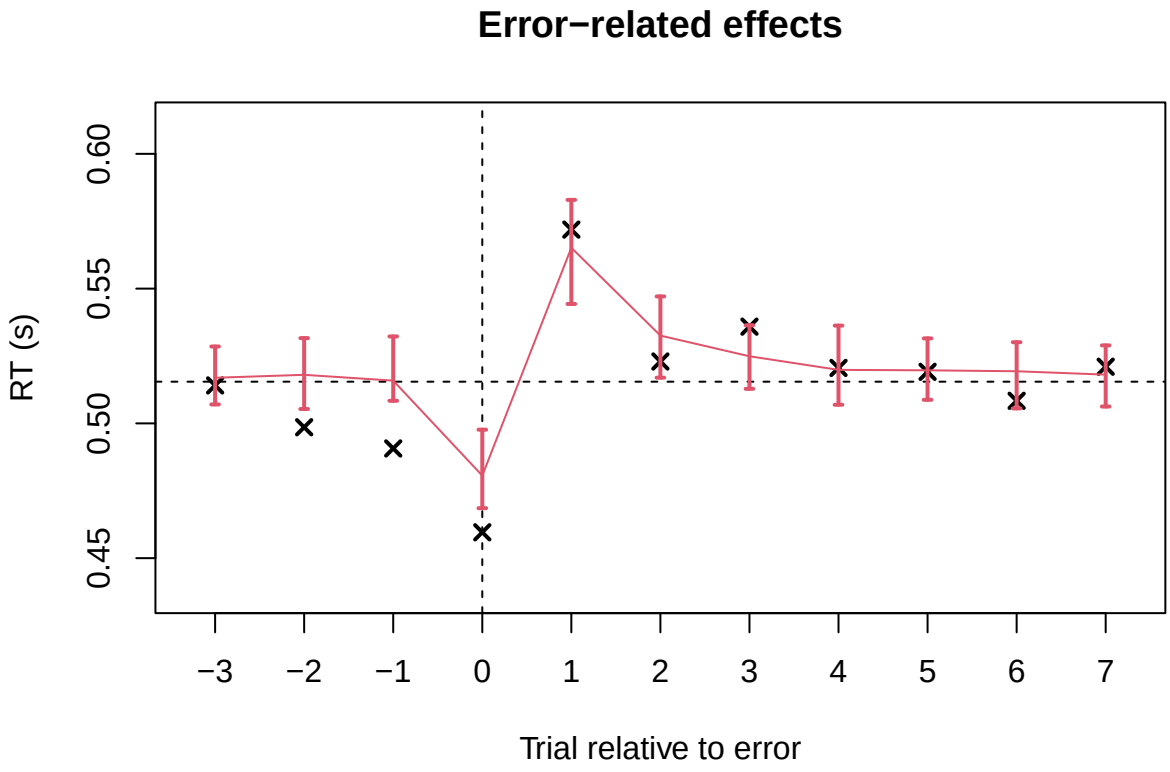
For this reason, it is useful to have *EMC2* calculate accuracy with a function rather than specifying it in the dataframe (in which case it would be treated as a static experimental factor and assumed fixed across simulations). The same function is applied when generating posterior predictives, so the accuracy is correctly determined in that phase.

```
AM_trend <- make_trend(cov_names='error',
                       kernels = 'delta',
                       par_names='v',
                       bases='lin',
                       premap=TRUE,
                       filter_lR=TRUE)  # see advanced notes for an explanation of what this does.
design_RDM_AM <- design(model=RDM,
                        data=dat,
                        contrast=list(lM=ADmat, lR=ADmat),
                        functions=list(error=function(x) x$S!=x$R),
                        covariates='error',
                        matchfun=function(d) d$S==d$lR,
                        formula=list(B ~ 1, v ~ lM, t0 ~ 1, s~lM),
                        trend=AM_trend,
                        constants=c('v.q0'=0, 's'=1))
# note that 1) we allow within-trial noise s to vary with accumulator
# match (correct/incorrect), which will allow us to capture the
# relative speed of errors.

# note that 2) in this case, we are not estimating v on the natural scale.
# This is because the RDM has no known analytic likelihood for negative drift rates,
# the AM mechanism can push drift rates on individual trials to negative values when
# applied on the natural scale. In this specific case, this can lead to difficulties
# sampling. Hence, the effect of errors on urgency as we implement it here,
# is not strictly linear -- it is linear on the log-scale, and then exponentiated.
# The cognitive interpretation is similar.

samplers <- make_emc(dat, design=design_RDM_AM, compress=FALSE)
samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
                fileName='./samples/ds1_AM.RData')
credint(samplers)
```

Accuracy memory can help explain error-related effects, like post-error slowing. To visualize, we can generate posterior predictives and then use a convenience function to plot mean RT as a function of trial number relative to an error. Note that, since the covariate in this case depends on observed behavior, we need to simulate data and determine the covariate one trial at a time. *EMC2* detects whether the covariate is `rt`, `R`, or the output of one of the functions provided to design. If it is, it automatically simulates data trial-by-trial. As vectorisation is not possible in this case, this is slower.

```
samplers <- get(load('./samples/ds1_AM.RData'))

pp <- predict(samplers, n_cores=10, conditional_on_data=FALSE)
save(pp, file='./samples/ds1_AM_pps.RData')

# needed for plotting
pp$accuracy <- 1-pp$error
```

```
dat$accuracy <- dat$S==dat$R

# calculate post-error slowing statistics
data_PES <- getErrorEffects(dat)
pp_PES <- getErrorEffects(pp, mc.cores=10)

# and plot with convenience function
par(mfrow=c(1,1))
plotPES(data_PES=data_PES$average, pp_PES=pp_PES$average,
        mean_rt=mean(aggregate(rt~subjects,dat,mean)[,2]),
        main='Error-related effects')
```



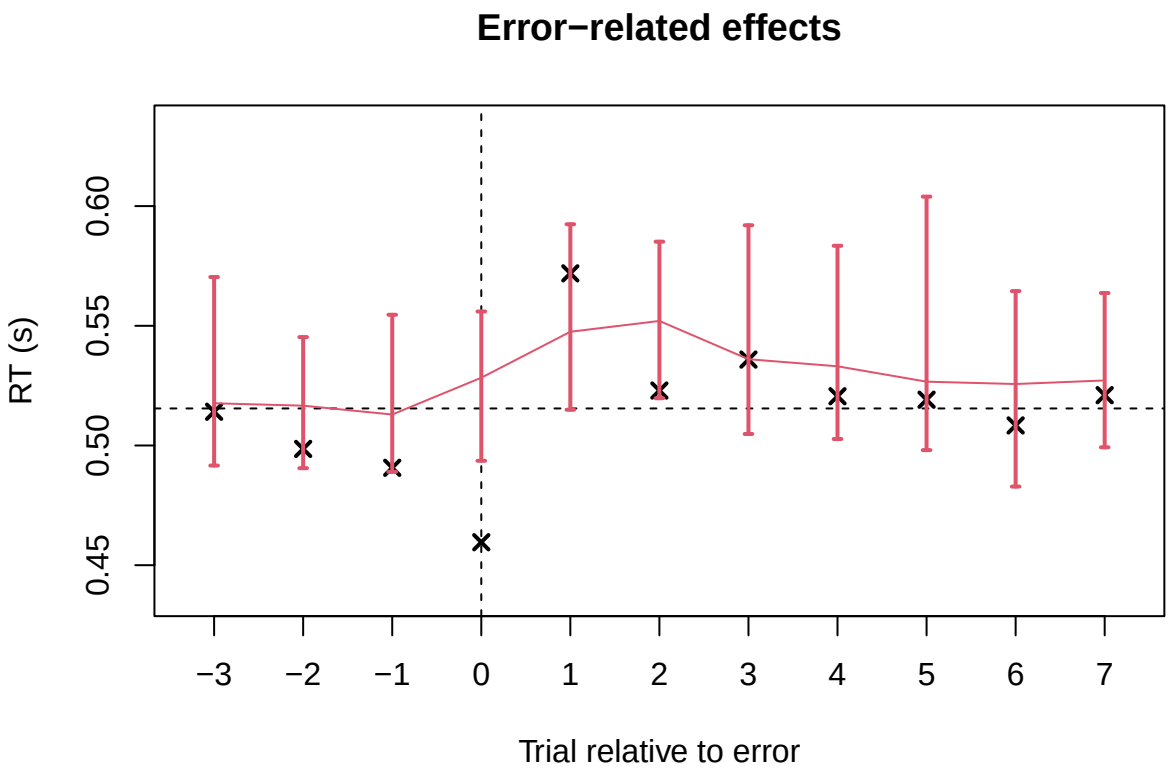
Like in the case of stimulus memory, the accuracy memory mechanism can be adjusted to other EAMs like the DDM, and it can affect other parameters. The DDM does not have an urgency mechanism, but we could, for example, allow AM to affect thresholds. This trend can be specified as post-transform since the thresholds are the same across all design cells. So in this case, we can sample thresholds on the log scale, then transform to the natural scale, and then apply the trend.

```
trend_AM_DDM <- make_trend(cov_names='error',
                           kernels = 'delta',
                           par_names='a',
                           bases='lin',
                           premap=FALSE, pretransform=FALSE)
Smat <- matrix(c(-1,1), nrow = 2,dimnames=list(NULL,"dif"))
design_DDM <- design(model=DDM,
                     data=dat,
```

```
                functions=list(error=function(x) x$S!=x$R),
                covariates='error',   # specify relevant covariate columns
                contrasts=list(S=Smat),
                formula=list(v ~ S, a~1, t0 ~ 1),
                constants=c('a.q0'=0),
                trend=trend_AM_DDM)

samplers <- make_emc(dat, design=design_DDM, compress=FALSE)
samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
               fileName='./samples/ds1_AMa_DDM.RData')
```

However, this again does not appear to fit as well qualitatively, and loses model comparisons:

```
samplers <- get(load('./samples/ds1_AMa_DDM.RData'))
pp_ddm <- predict(samplers, n_cores=10, conditional_on_data=FALSE)
save(pp_ddm, file='./samples/ds1_AMa_DDM_pps.RData')


# needed for plotting
pp_ddm$accuracy <- 1-pp_ddm$error
dat$accuracy <- dat$S==dat$R

# calculate post-error slowing statistics
data_PES <- getErrorEffects(dat)
pp_ddm_PES <- getErrorEffects(pp_ddm, mc.cores=10)

# and plot with convenience function
par(mfrow=c(1,1))
plotPES(data_PES=data_PES$average, pp_PES=pp_ddm_PES$average,
        mean_rt=mean(aggregate(rt~subjects,dat,mean)[,2]),
        main='Error-related effects')
```

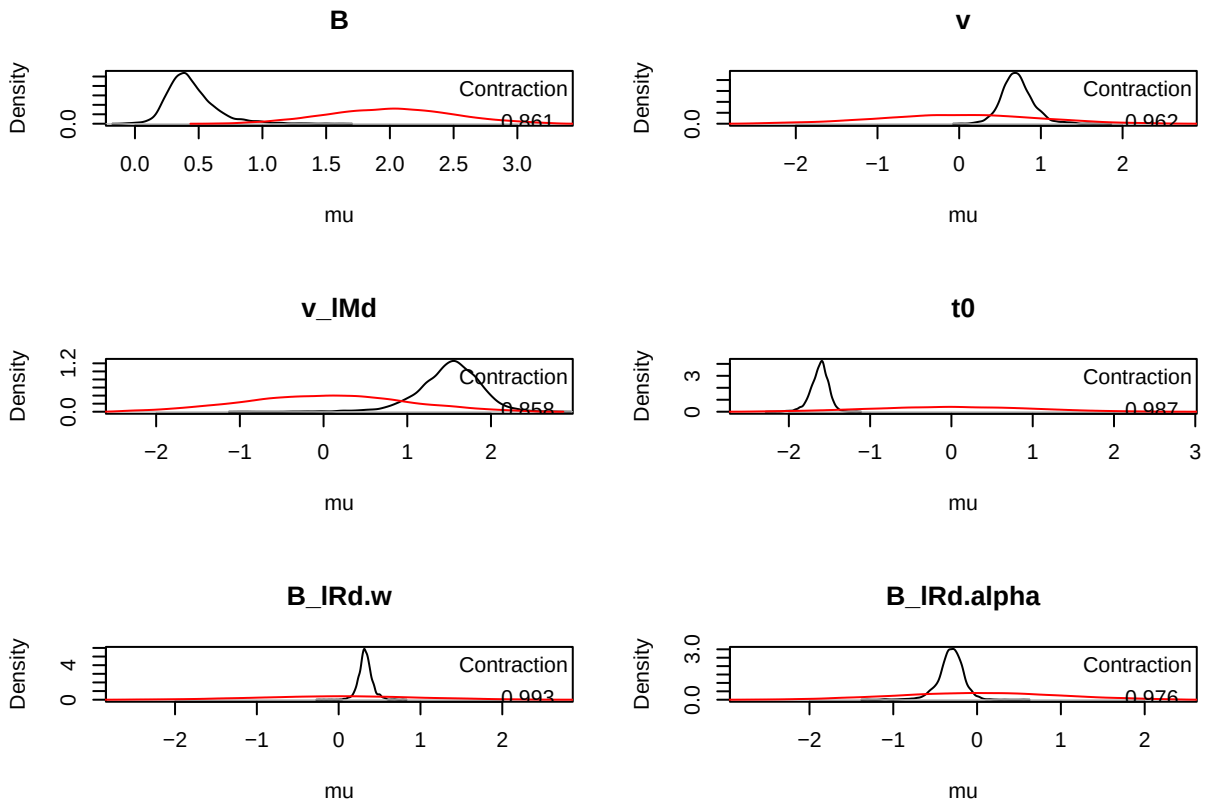



```
# and loses model comparisons again
compare(sList=list(rdm=get(load('./samples/ds1_AM.RData')),
                   ddm=get(load('./samples/ds1_AMa_DDM.RData'))))
```

```
       MD wMD   DIC wDIC  BPIC wBPIC EffectiveN meanD Dmean  minD
rdm -6904   1 -7044    1 -6971     1         73 -7118 -7138 -7191
ddm -5314   0 -5410    0 -5335     0         75 -5485 -5516 -5560
```

This is interesting, because if you look at the estimated parameters, there appear to be reasonably large effects:

```
plot_pars(samplers)
```




```
credint(samplers)
```

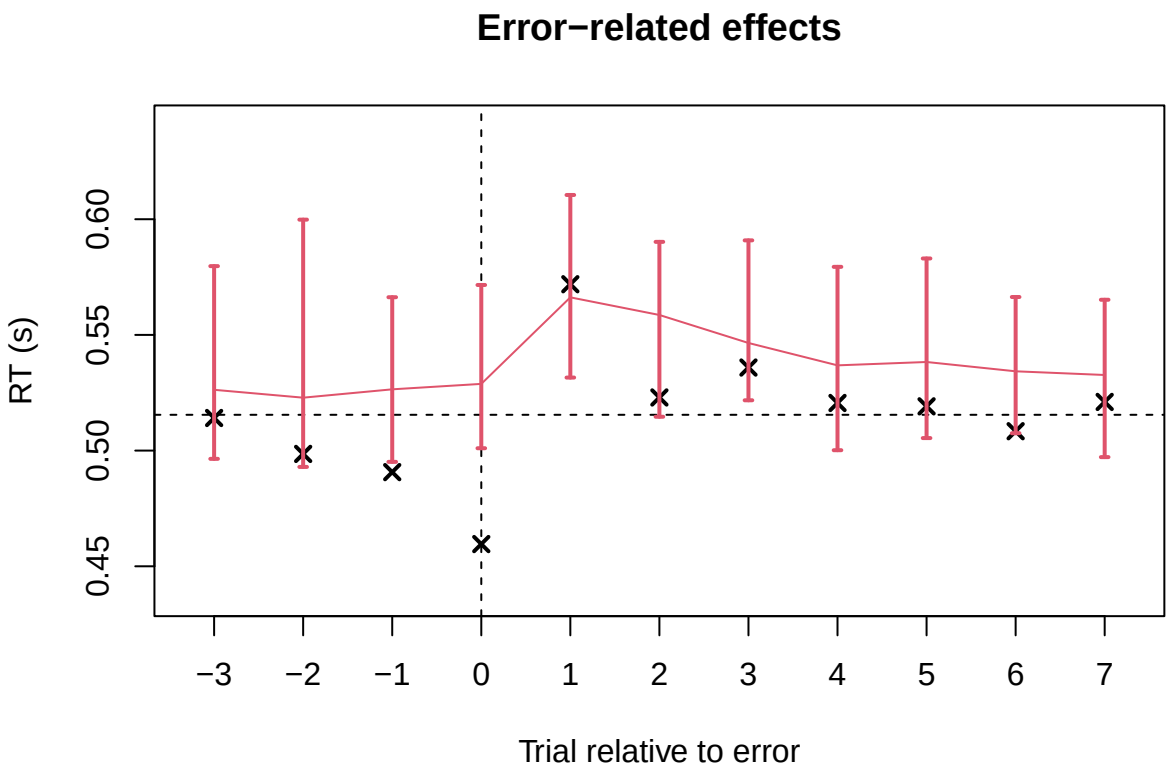
```
$mu
               2.5%    50%  97.5%
B             0.156  0.407  0.912
v             0.360  0.698  1.132
v_lMd         0.690  1.530  2.158
t0           -1.877 -1.613 -1.429
B_lRd.w       0.160  0.324  0.508
B_lRd.alpha -0.650 -0.305 -0.045
```

So the parameters appear to improve the model fit somehow, but just not explain post-error slowing. If we look at the parameter values of `a.w` and `a.alpha`, we find that there is quite a low learning rate (`pnorm(-1.4)=0.081`; compared to a learning rate of ~0.363 in the RDM example above), and errors tend to increase thresholds (as `a.w > 0`). We can put a few other pieces of information together to understand what is happening: `a.q0=0`, so in the model, participants start out with a belief they make no errors. Obviously, they will, and the low learning rate will lead the Q_{AM} values to gradually increase over trials. This, in turn, gradually increases the thresholds. Recall in our earlier RDM example, where we estimated a linear trend on thresholds, we also found some evidence for gradually increasing thresholds over time. So we can hypothesize that what is going on here is that there is evidence for gradually increasing thresholds – due to some mechanism unrelated to error processing – which are now erroneously captured by the AM mechanism.

If so, what would happen if we impose a minimum value on the learning rate? This way, the mechanism can only capture fast changes. We can do this by setting a lower bound:

```
trend_AM_DDM2 <- make_trend(cov_names='error',
                        kernels = 'delta',
```

```
                        par_names='a',
                        bases='lin',
                        premap=FALSE, pretransform=FALSE)
design_DDM2 <- design(model=DDM,
                  data=dat,
                  functions=list(error=function(x) x$S!=x$R),
                  covariates='error',   # specify relevant covariate columns
                  contrasts=list(S=Smat),
                  formula=list(v ~ S, a~1, t0 ~ 1),
                  transform=list(lower=c('a.alpha'=0.1)),
                  constants=c('a.q0'=0),
                  trend=trend_AM_DDM2)

samplers <- make_emc(dat, design=design_DDM2, compress=FALSE)
samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
                fileName='./samples/ds1_AMa2_DDM.RData')
pp_ddm <- predict(samplers, n_cores=10, conditional_on_data=FALSE)
save(pp_ddm, file='./samples/ds1_AMa2_DDM_pps.RData')

# needed for plotting
pp_ddm$accuracy <- 1-pp_ddm$error
dat$accuracy <- dat$S==dat$R

# calculate post-error slowing statistics
data_PES <- getErrorEffects(dat)
pp_ddm_PES <- getErrorEffects(pp_ddm, mc.cores=10)

# and plot with convenience function
par(mfrow=c(1,1))
plotPES(data_PES=data_PES$average, pp_PES=pp_ddm_PES$average,
        mean_rt=mean(aggregate(rt~subjects,dat,mean)[,2]),
        main='Error-related effects')
```



This seems to have helped, but perhaps ignoring the slow increase in thresholds is not the best approach. We can still model it by combining a delta rule and a linear trend:

*** MULTIPLE TRENDS ON ONE PARAMETER CANNOT BE DONE YET ***

Fluency memory: Trend specification

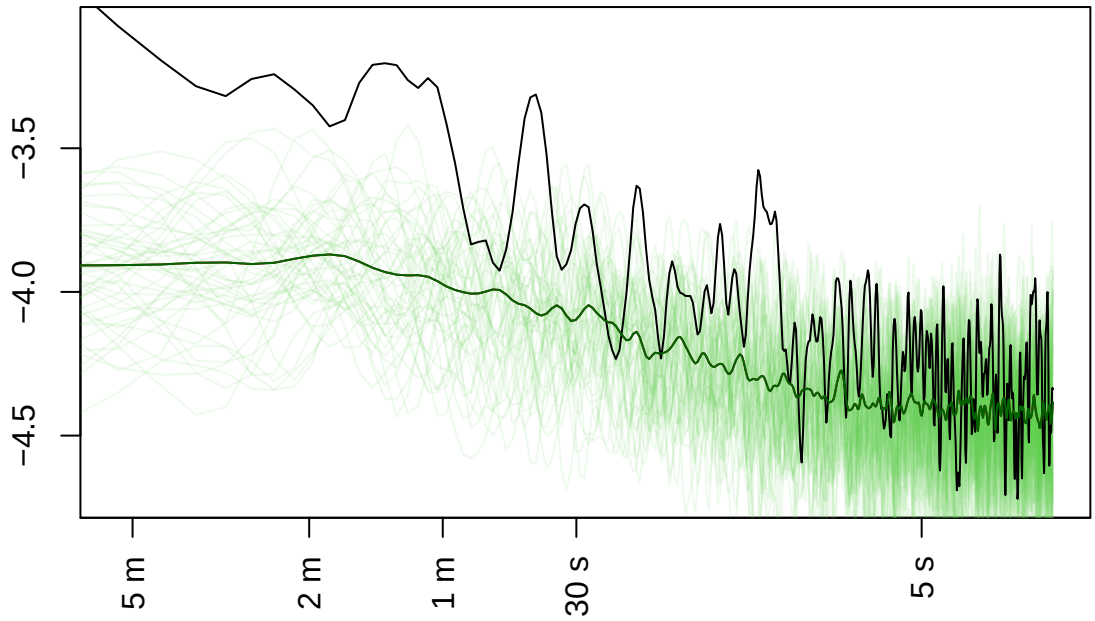
Fluency memory is a formalization of the idea that participants adjust their behavior according to perceived difficulty. To estimate perceived difficulty, we rely on a few simplifying assumptions: participants can quantify (1) their thresholds, and (2) their response time. In race models, the ratio of threshold over response time constitutes a measure of mean speed, which can be used as a proxy for difficulty. After all, we expect processing efficiency to be lower for difficult trials than for easy trials. Formally, $Q_{FM,t+1} = Q_{FM,t} + \alpha_{FM}(b_t/rt_t - Q_{FM,t})$. Contrary to stimulus and accuracy memory, fluency memory is a very specific effect, and its implementation is not very flexible. It can only work for race models; it must influence thresholds; and the thresholds should be estimated on the natural scale. *EMC2* implements it as a separate kernel `deltab` with covariate `rt`:

```
trend_FM=make_trend(kernel='deltab',
                    base='lin',
                    cov_names ='rt',
                    par_names='B', premap = TRUE, filter_lR=TRUE)

design_FM <- design(model=RDM,
                    data=dat,
                    contrast=list(lM=ADmat),
                    matchfun=function(d) d$S==d$lR,
```

```
                formula=list(B ~ 1, v ~ lM, t0 ~ 1, s~lM),
                transform=list(func=c(B='identity'),          # should also update prior in this case
                               lower=c(B.alpha=0.05)),
                constants=c('B.q0'=3, 's'=log(1)),            # Why 3? When fitting a static RDM, the
                trend=trend_FM)
prior_FM <- prior(design_FM, mu_mean=c('B'=2), mu_sd=c('B'=0.5))
samplers <- make_emc(dat, design=design_FM, compress=FALSE, prior_list=prior_FM)

samplers <- fit(samplers, cores_per_chain=6, cores_for_chains=3,
                fileName='samples/ds1_FM.RData')
check(samplers)
pp <- predict(samplers, n_cores=10, conditional_on_data=FALSE)
save(pp, file='./samples/ds1_FM_pps.RData')
```

Fluency memory provides an explanation for the slower fluctuations in the observed response times. One useful way to visualize these is by creating a power spectrum of the RT data. We provide a convenience function that does this for the data and the posterior predictives of the estimated model:

```
# convenience function to plot the spectrum
plotSpectrum(dat=dat, pp=pp, trial_duration=1.265)  # mean trial duration of this task was 1.265 s, whi
```



which demonstrates that fluency can account for the fluctuations that are moderate to fast, but not the slowest fluctuations.