Particle Based Samplers for MCMC

Jon-Paul Cavallaro

Thursday 02 July 2020

Contents

Chapter 1

Introduction to Particle Based Sampler for MCMC

Contains implementations of particle based sampling methods for model parameter estimation. Primarily an implementation of the Particle Metropolis within Gibbs sampler outlined in the paper available at https://arxiv.org/abs/1806.10089, it also contains code for covariance estimation and time varying models.

1.1 Assumed knowledge

- Modelling knowledge
- Familiarity with particular packages? E.g. rtdists

1.2 Computational Requirements

- R Version
- Packages
- Memory
- Document computational requirements - memory 200mb
- Minutes
- Size of samples
- Memory required
- Can it be done on a grid or laptop

1.3 Background information

- What is particle metropolis
- What is Gibbs?
- Why a multivariate normal?
- The prior is 'fixed'

#PMwG sampler and Signal Detection Theory

Here we demonstrate how to use the PMwG sampler package to run a simple signal detection theory (SDT) analysis on a lexical decision task. We recognise that it is unnecessary to use the sampler package for a simple analysis such as this; however, we hope this example demonstrates the usefulness of the PMwG sampler package.

1.3.1 Signal Detection Theory analysis of lexical decision task

We assume you have an understanding of SDT and lexical decision tasks, so we'll jump straight into how you can use the PMwG package with SDT in the context of a lexical decision task.

Do we need to explain lexical decision tasks?? We desribe the procedure briefly below - Participants were asked to indicate whether a letter string was a word or a non-word.

We begin with the distributions for non-word and word stimuli. You can think of these two distributions as the 'noise' and 'signal' curves, respectively. Each distribution represents the evidence for 'word-likeness' and they are assumed to be normally distributed. The non-word distribution (or the 'noise' distribution) has a mean (μ) of 0 and a standard deviation (SD) of 1. We could estimate SD here, but we will use 1 in this example for simplicity. The mean for the word distribution is unknown at this point; however, we assign a d-prime (d') parameter to denote the difference between the mean of the non-word and the mean of the word distributions (i.e. the 'sensitivity' to word stimuli or the signal-noise difference between words and non-word). As can be seen in Figure 1.1, the word distribution mean is greater than the non-word distribution mean (in the positive direction?); however, the distributions partially overlap where non-words and words are difficult to classify.

The second parameter we denote is the criterion (C) parameter. The criterion is the point at which an individual responds non-word (to the left of C in Figure 1.2) or word (to the right of C in Figure 1.2) and it is set somewhere between the means of the two distributions. If you're biased to respond word, the criterion would move to the left. Conversely, if you're biased to respond non-word then the criterion would move to the right.

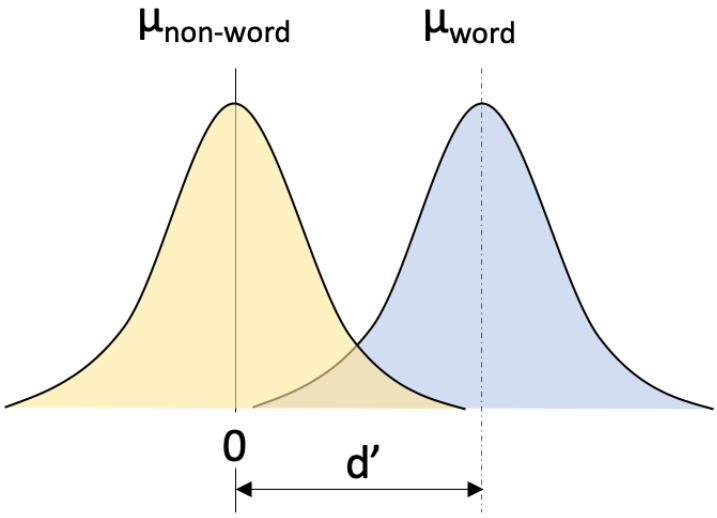

Figure 1.1: Simple SDT example of lexical decision task

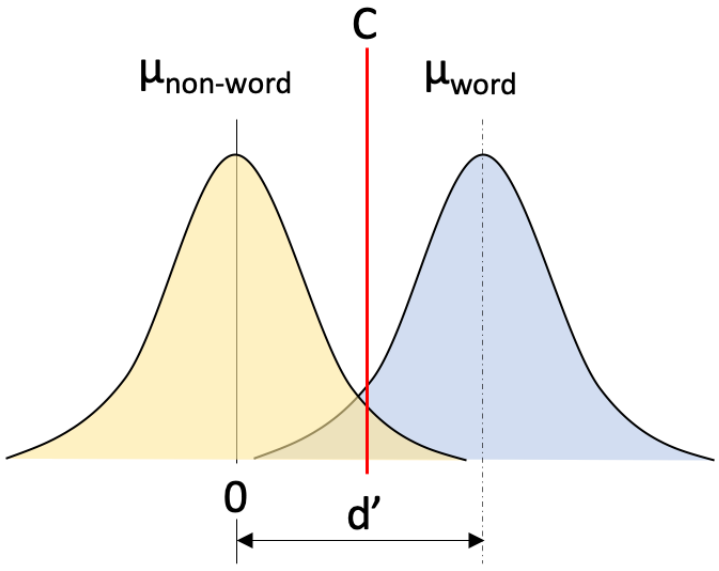

Figure 1.2: Simple SDT example of lexical decision task

Table 1.1: A fabricated dataset of 7 trials with a response and a stimuls column

resp	stim
word	word
word	word
non-word	non-word
word	word
non-word	non-word
non-word	non-word
word	non-word
non-word	word

Do we need to add something about means should be positive - to the right of NW mean of 0, otherwise the line about "given these parameters, one would expect that the word distribution would have a higher mean than the non-words, with partial overlap" does not make sense.

1.3.2 Writing a log-likelihood function

Let's write a simple log likelihood function for a fabricated data set. You can copy the code below to follow along with the example.

```
resp <- c("word", "word", "non-word", "word", "non-word", "non-word", "word", "non-wor
stim <- c("word", "word", "non-word", "word", "non-word", "non-word", "non-word", "wor
fabData <- as.data.frame(cbind(resp, stim))
```

We create our dataset by combining a response `resp` and a stimulus `stim` vector into a data frame as shown in 1.1 below.

Our log likelihood function will step through the data, line by line, and find a likelihood value for each trial, under two parameters; d-prime `d` and criterion `C`.

Remove this paragraph? Some info is covered above As mentioned above the non-word distribution has a mean of 0 and SD of 1. This then gives a reference point for where the mean of the word distribution would sit and is denoted by d'. Now we must find the location of the criterion. Setting the criterion allows us to determine which response will be made i.e. above the criterion, participant will respond word and below the criterion, participant will respond non-word.

Here is our complete log likelihood function. We have omitted some code from the code blocks below to enhance appearance, so we encourage you to copy the log likelihood function from the following code block if you'd like to follow along with our example.

```
1 SDT_ll <- function(x, data, sample = FALSE){
2   if (!sample) {
```

```
    out <- numeric(nrow(data))
    for (i in 1:nrow(data)) {
      if (stim[i] == "word") {
      if (resp[i] == "word") {
        out[i] <- pnorm(x["C"], mean = x["d"], sd = 1,
                        log.p = TRUE, lower.tail = FALSE)
      } else {
        out[i] <- pnorm(x["C"], mean = x["d"], sd = 1,
                        log.p = TRUE, lower.tail = TRUE)
      }
    } else {
      if (resp[i] == "word") {
        out[i] <- pnorm(x["C"], mean = 0, sd = 1,
                        log.p = TRUE, lower.tail = FALSE)
      } else {
        out[i] <- pnorm(x["C"], mean = 0, sd = 1,
                        log.p = TRUE, lower.tail = TRUE)
        }
      }
  }
  sum(out)
  }
}
```

We initialise the log likelihood function with three arguments

```
SDT_ll <- function(x, data, sample = FALSE) {
```

- `x` is a named parameter vector (e.g.`pars`)
- `data` is the dataset
- `sample` = sample values from the posterior distribution (For this simple example, we do not require a `sample` argument.)

The first if statement (line 2) checks if you want to sample, this is used for posterior predictive sampling which we will cover in later chapters. and assigns NAs to your data frames response column. If you're not sampling (like us in this example), you need to create an output vector `out`. The `out` vector will contain the log likelihood value for each row/trial in your dataset. Need to explain the `(sample)` part below

```
  if (sample) {
    data$response <- NA
    } else {
      out <- numeric(nrow(data))
      }
```

From line 9, we check each row in the data set, first considering all trials with

word stimuli `if (stim[i] == "word"` (line 10), and assign a likelihood for responding word (line 12-13) or non-word (line 15-16). The word distribution has a mean of `x["d"]` (d-prime parameter) and a decision criterion parameter `x["C"]`. If the response is word, we are considering values ABOVE or to the right of C in 1.2, so we set `lower.tail =` to `FALSE`. If the response is non-word, we look for values BELOW or to the left of C in 1.2 and we set `lower.tail =` to `TRUE`. The `log.p =` argument takes the log of all likelihood values when set to `TRUE`. We do this so we can sum all likelihoods at the end of the log likelihood function. Do we need to explain p-norm??

```
8   if (!sample) {
9     for (i in 1:nrow(data)) {
10      if (stim[i] == "word") {
11        if (resp[i] == "word") {
12          out[i] <- pnorm(x["C"], mean = x["d"], sd = 1,
13                          log.p = TRUE, lower.tail = FALSE)
14    } else {
15      out[i] <- pnorm(x["C"], mean = x["d"], sd = 1,
16                      log.p = TRUE, lower.tail = TRUE)
17      }
```

From the else statement on line 18, we have the function for non-word trials i.e. `stim[i] == "non-word"`. As can be seen below, the output value `out[i]` for these trials is arrived at in a similar manner to the word trials. We set the `mean` to 0 and the standard deviation `sd` to 1. If the response is word, we are considering values ABOVE or to the right of C in 1.2, so we set `lower.tail =` to `FALSE`. If the response is non-word, we look for values BELOW or to the left of C in 1.2 and we set `lower.tail =` to `TRUE`. Again we want the log of all likelihood values so we set `log.p = TRUE`.

```
18  else {
19    if (resp[i] == "word") {
20      out[i] <- pnorm(x["C"], mean = 0, sd = 1,
21                      log.p = TRUE, lower.tail = FALSE)
22        } else {
23          out[i] <- pnorm(x["C"], mean = 0, sd = 1,
24                          log.p = TRUE, lower.tail = TRUE)
25        }
26      }
```

The final line of code on line 24 sums the `out` vector and returns a log-likelihood value for your model. The text alignment/justification for code blocks is determined by the length of the longest line in the code block

```
sum(out)
```

1.4 Testing the SDT log likelihood function

Before we run the log likelihood function, we must create a parameter vector `pars` containing the same parameter names used in our log likelihood function above i.e. we name the criterion `C` and d-prime parameter `d` and we assign arbitrary values to each parameter.

```
pars <- c(C = 0.8, d = 2)
```

We can test run our log likelihood function by passing in the parameter vector `pars` and the fabricated dataset we created above `fabData`.

```
SDT_loglike(pars, fabData)
```

```
## [1] -4.795029
```

Now, if we change the parameter values, the log-likelihood value should also change.

```
pars <- replace(pars, c(1,2), c(0.5, 1.2))
SDT_loglike(pars, fabData)
```

```
## [1] -4.532791
```

We can see the log likelihood has changed, so these values are more accurate given the data. What does this tell us? Seems insufficient to look for a change in the LL

1.5 SDT log likelihood function for Wagenmakers experiment

Now that we've covered a simple test example, let's create a log likelihood function for the Wagenmakers et al. (2008) dataset.

1.5.1 Description of Wagenmakers experiment

If you'd like to follow our example, you will need to access the Wagenmakers dataset. This can be done by installing the `rtdists` package and calling the `speed_acc` data frame. The structure of the `speed_acc` dataset will need to be modified in order to meet the requirements of the PMwG sampler. To do this, modify the dataset to match the structure illustrated in table 1.2.

Participants were asked to indicate whether a letter string was a word or a non-word. A subset of Wagenmaker et al data are shown in table 1.2, with each line representing a single trial. We have a `subject` column with a subject id (1-19), a condition column `cond` which indicates the proportion of words

Table 1.2: Subset of 10 trials from the Wagenmakers (2008) dataset.

subject	cond	stim	resp	rt	correct
1	w	lf	W	0.410	2
1	w	hf	W	0.426	2
1	w	nw	NW	0.499	2
1	w	lf	W	0.392	2
1	w	vlf	W	0.435	2
1	w	hf	W	0.376	2
1	nw	lf	W	0.861	2
1	nw	hf	W	0.563	2
1	nw	nw	NW	0.666	2
1	nw	nw	NW	1.561	2
1	nw	nw	NW	0.503	2
1	nw	nw	NW	0.445	2

to non-words presented within a block of trials. In word blocks (`cond = w`) participants completed 75% word and 25% non-word trials and for non-word (`cond = nw`) blocks 75% non-word and 25% word trials. The `stim` column lists the word's frequency i.e. is the stimulus a very low frequency word (`stim = vlf`), a low frequency word (`stim = lf`), a high frequency word (`stim = hf`) or a non-word (`stim = nw`). The third column `resp` refers to the participant's response i.e. the participant responded word (`resp = W`) or non-word (`resp = NW`). The two remaining columns list the response time (`rt`) and whether the paricipant made a correct (`correct = 2`) or incorrect (`correct = 1`) choice. For more details about the experiment please see the original paper. Is this the correct paper to reference?

Our log-likelihood function for Wagenmakers experimental data is similar to the function we wrote above, except now we require a criterion parameter for each condition and a d-prime parameter for each of the `stim` word types. This is illustrated in figure 1.3 below, where we have a non-word criterion Cnw, a word criterion Cw and three distributions for each of the `stim` types with corresponding d-prime for each distribution: dvlf, dlf, dhf.

Here is our complete log likelihood function for the Wagenmakers data set.

```
SDT_ll <- function(x, data, sample=FALSE){
  if (sample){
    data$response <- NA
  } else {
    out <- numeric(nrow(data))
  }
  if (!sample){
  for (i in 1:nrow(data)) {
```

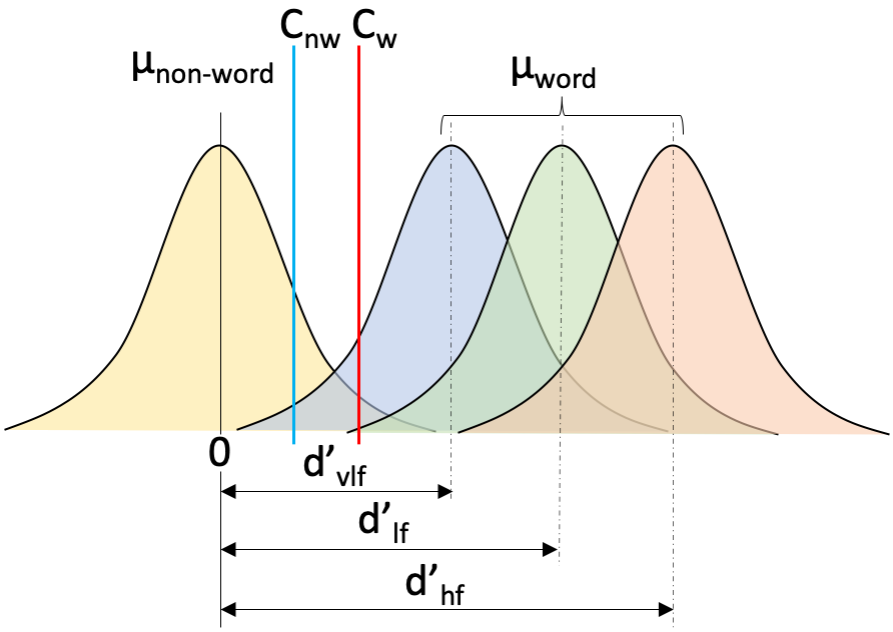

Figure 1.3: Signal detection theory example of lexical decision task

```
    if (data$cond[i] == "w") {
    if (data$stim[i] == "hf") {
      if (data$resp[i] == "W") {
        out[i] <- pnorm(x["C.w"], mean = x["HF.d"], sd = 1,
                        log.p = TRUE, lower.tail = FALSE)
      } else {
        out[i] <- pnorm(x["C.w"], mean = x["HF.d"], sd = 1,
                        log.p = TRUE, lower.tail = TRUE)
      }
    } else if (data$stim[i] == "lf"){
      if (data$resp[i] == "W"){
        out[i] <- pnorm(x["C.w"], mean = x["LF.d"], sd = 1,
                        log.p = TRUE, lower.tail = FALSE)
      } else {
        out[i] <- pnorm(x["C.w"], mean = x["LF.d"], sd = 1,
                        log.p = TRUE, lower.tail = TRUE)
        }
      } else if (data$stim[i] == "vlf") {
        if (data$resp[i] == "W") {
          out[i] <- pnorm(x["C.w"], mean = x["VLF.d"], sd = 1,
                          log.p = TRUE, lower.tail = FALSE)
        } else {
          out[i] <- pnorm(x["C.w"], mean = x["VLF.d"], sd = 1,
                          log.p = TRUE, lower.tail = TRUE)
          }
        } else {
      if (data$resp[i] == "W") {
        out[i] <- pnorm(x["C.w"], mean = 0, sd = 1,
                        log.p = TRUE, lower.tail = FALSE)
      } else {
        out[i] <- pnorm(x["C.w"], mean = 0, sd = 1,
                        log.p = TRUE, lower.tail = TRUE)
        }
      }
    } else {
      if (data$stim[i] == "hf") {
        if (data$resp[i] == "W") {
          out[i] <- pnorm(x["C.nw"], mean = x["HF.d"], sd = 1,
                          log.p = TRUE, lower.tail = FALSE)
        } else {
          out[i] <- pnorm(x["C.nw"], mean = x["HF.d"], sd = 1,
                          log.p = TRUE, lower.tail = TRUE)
        }
      }  else if (data$stim[i] == "lf") {
        if (data$resp[i] == "W") {
```

```
54            out[i] <- pnorm(x["C.nw"], mean = x["LF.d"], sd = 1,
55                            log.p = TRUE, lower.tail = FALSE)
56          } else {
57            out[i] <- pnorm(x["C.nw"], mean = x["LF.d"], sd = 1,
58                            log.p = TRUE, lower.tail = TRUE)
59          }
60        } else if (data$stim[i] == "vlf") {
61          if (data$resp[i] == "W") {
62            out[i] <- pnorm(x["C.nw"], mean = x["VLF.d"], sd = 1,
63                            log.p = TRUE, lower.tail = FALSE)
64          } else {
65            out[i] <- pnorm(x["C.nw"], mean = x["VLF.d"], sd = 1,
66                            log.p = TRUE, lower.tail = TRUE)
67          }
68        } else {
69          if (data$resp[i] == "W") {
70            out[i] <- pnorm(x["C.nw"], mean = 0, sd = 1,
71                            log.p = TRUE, lower.tail = FALSE)
72          } else {
73            out[i] <- pnorm(x["C.nw"], mean = 0, sd = 1,
74                            log.p = TRUE, lower.tail = TRUE)
75          }
76        }
77      }
78    }
79    sum(out)
80    }
81  }
```

Line 1 through to line 8 are the same as the log likelihood we wrote for the fabricated dataset above. From line 9, we calculate the log-likelihood `out[i]` for word condition trials `cond[i] == "w"` when the stimulus is a high frequency word `stim[i] == "hf"` for each response. We do this by considering the upper tail of the high frequency word distribution `lower.tail = FALSE`, from the word criterion Cw, for word responses `resp[i] == "W"` and the lower tail for non-word responses (`else` statement on line 14).

```
9   if (data$cond[i] == "w") {
10     if (data$stim[i] == "hf") {
11       if (data$resp[i] == "W") {
12         out[i] <- pnorm(x["C.w"], mean = x["HF.d"], sd = 1,
13                         log.p = TRUE, lower.tail = FALSE)
14       } else {
15         out[i] <- pnorm(x["C.w"], mean = x["HF.d"], sd = 1,
16                         log.p = TRUE, lower.tail = TRUE)
17       }
```

From line 18 and still considering trials from the word condition, we calculate the log likelihood for each response for low frequency words `stim[i] == "lf"`, very low frequency words `stim[i] == "vlf"` (line 26) and non-word stimuli (`else` statement on line 35).

```
18    else if (data$stim[i] == "lf") {
19        if (data$resp[i] == "W") {
20          out[i] <- pnorm(x["C.w"], mean = x["LF.d"], sd = 1,
21                          log.p = TRUE, lower.tail = FALSE)
22        } else {
23          out[i] <- pnorm(x["C.w"], mean = x["LF.d"], sd = 1,
24                          log.p = TRUE, lower.tail = TRUE)
25          }
26        } else if (data$stim[i] == "vlf") {
27          if (data$resp[i] == "W") {
28            out[i] <- pnorm(x["C.w"], mean = x["VLF.d"], sd = 1,
29                            log.p = TRUE, lower.tail = FALSE)
30          } else {
31            out[i] <- pnorm(x["C.w"], mean = x["VLF.d"], sd = 1,
32                            log.p = TRUE, lower.tail = TRUE)
33            }
34          } else {
35        if (data$resp[i] == "W") {
36          out[i] <- pnorm(x["C.w"], mean = 0, sd = 1,
37                          log.p = TRUE, lower.tail = FALSE)
38        } else {
39          out[i] <- pnorm(x["C.w"], mean = 0, sd = 1,
40                          log.p = TRUE, lower.tail = TRUE)
41          }
42        }
```

That covers the word condition. From line 45 through 78, we repeat the same process for the non-word conditon….

```
45  else {
46        if (data$stim[i] == "hf") {
47          if (data$resp[i] == "W") {
48            out[i] <- pnorm(x["C.nw"], mean = x["HF.d"], sd = 1,
49                            log.p = TRUE, lower.tail = FALSE)
50          } else {
51            out[i] <- pnorm(x["C.nw"], mean = x["HF.d"], sd = 1,
52                            log.p = TRUE, lower.tail = TRUE)
53          }
54        } else if (data$stim[i] == "lf") {
55          if (data$resp[i] == "W") {
56            out[i] <- pnorm(x["C.nw"], mean = x["LF.d"], sd = 1,
57                            log.p = TRUE, lower.tail = FALSE)
```

```
        } else {
          out[i] <- pnorm(x["C.nw"], mean = x["LF.d"], sd = 1,
                          log.p = TRUE, lower.tail = TRUE)
        }
      } else if (data$stim[i] == "vlf") {
        if (data$resp[i] == "W") {
          out[i] <- pnorm(x["C.nw"], mean = x["VLF.d"], sd = 1,
                          log.p = TRUE, lower.tail = FALSE)
        } else {
          out[i] <- pnorm(x["C.nw"], mean = x["VLF.d"], sd = 1,
                          log.p = TRUE, lower.tail = TRUE)
        }
      } else {
        if (data$resp[i] == "W") {
          out[i] <- pnorm(x["C.nw"], mean = 0, sd = 1,
                          log.p = TRUE, lower.tail = FALSE)
        } else {
          out[i] <- pnorm(x["C.nw"], mean = 0, sd = 1,
                          log.p = TRUE, lower.tail = TRUE)
        }
```

...and sum the values and return the log-likelihood for our model.

```
  sum(out)
```

This give us a log likelihood for all data. Let's test this...

```
pars <- log(c(C.w = 1, C.nw = 0.5, HF.d = 3, LF.d = 1.8, VLF.d = 0.7))
SDT_ll(pars, wgnmks2008, sample = FALSE)
```

```
## [1] -30801.71
```

1.5.2 Computation time of log likelihood function

You may have noticed that our log likelihood function is slow and heavy on computer time when processing the data trial by trial. We recommend you write a 'slow' log likelihood (as written above) to check it functions as it should before improving the function's efficiency. Now we'll speed up our log likelihood function. We have 16 possible values that could be assigned per line in the previous function (for the 16 cells of the design given by proportion (2) x stimuli (4) x response (2)). Rather than looping over each trial, we could calculate the log-likelihood for each cell in the design and multiply the number of instances for each subject. To do this, we add a column to the dataframe by tabling as shown in the code below

```
wgnmks2008Fast <- as.data.frame(table(wgnmks2008$subject, wgnmks2008$cond,
                                      wgnmks2008$stim, wgnmks2008$resp))
```

```
names(wgnmks2008Fast) <- c("subject", "cond", "stim", "resp", "n")
```

Now our data frame looks like this..

subject	cond	stim	resp	n
1	nw	hf	NW	1
2	nw	hf	NW	2
3	nw	hf	NW	11
4	nw	hf	NW	0
5	nw	hf	NW	6
6	nw	hf	NW	9

For our SDT log likelihood function, we add `n*` (i.e. a multiplying factor) to each of these values to calculate the model log likelihood Should we shorten this code block? Seems excessive.

```
SDT_ll_fast <- function(x, data, sample = FALSE) {
  if (!sample) {
    out <- numeric(nrow(data))
    for (i in 1:nrow(data)) {
      if (data$cond[i] == "w") {
        if (data$stim[i] == "hf") {
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = x["HF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = x["HF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        } else if (data$stim[i] == "lf") {
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = x["LF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = x["LF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        } else if (data$stim[i] == "vlf") {
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = x["VLF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = x["VLF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        } else {
          if (data$resp[i] == "W") {
```

```
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = 0,
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.w"], mean = 0,
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        }
      }else{
        if (data$stim[i] == "hf") {
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = x["HF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = x["HF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        }  else if (data$stim[i] == "lf"){
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = x["LF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = x["LF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        } else if (data$stim[i] == "vlf") {
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = x["VLF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = x["VLF.d"],
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        } else {
          if (data$resp[i] == "W") {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = 0,
                                        sd = 1, log.p = TRUE, lower.tail = FALSE)
          } else {
            out[i] <- data$n[i] * pnorm(x["C.nw"], mean = 0,
                                        sd = 1, log.p = TRUE, lower.tail = TRUE)
          }
        }
      }
    }
    sum(out)
}
```

Now we have a fast(er) SDT log likelihood function and we can compare its output with the slow log likelihood function's output to make sure it is functioning correctly.

```
pars <- log(c(C.w = 1, C.nw = 0.5, HF.d = 3, LF.d = 1.8, VLF.d = 0.7))
SDT_ll(pars, wgnmks2008, sample = FALSE)
```

```
## [1] -30801.71
```

```
SDT_ll_fast(pars, wgnmks2008Fast, sample = FALSE)
```

```
## [1] -30801.71
```

Great! Both functions produce the same log likelihood! And we can run one final check by modifying the parameter vector's values

```
pars <- log(c(C.w = 1, C.nw = 0.8, HF.d = 2.7, LF.d = 1.8, VLF.d = 1.3))
SDT_ll(pars, wgnmks2008, sample = FALSE)
```

```
## [1] -22168.95
```

```
SDT_ll_fast(pars, wgnmks2008Fast, sample = FALSE)
```

```
## [1] -22168.95
```

We recommend "speeding up" your code however you wish. We do it in the Forstmann code…. . When you're confident that your log likelihood functions correctly, you should save it as a separate script so it can be sourced when running the sampler.

1.6 PMwG Framework

Now that we have written a log likelihood function, we're ready to use the PMwG sampler package.

Let's begin by installing the PMwG samplers package. We currently recommended installing psamplers via devtools.

```
# The samplers package will be on CRAN - this step will be removed.
install_github('newcastlecl\samplers')
```

```
library(psamplers)
```

Now we require the parameter vector `pars` we specified above and a priors object called `priors`. The `priors` object is a list that contains two components:

- `theta_mu` a vector containing the prior for model parameter means
- `theta_sig` the prior covariance matrix for model parameters.

```
pars <- c("C.w","C.nw","HF.d","LF.d","VLF.d") # This is the same as the `pars` vector specified a
priors <- list(
  theta_mu = rep(0, length(pars)),
  theta_sig = diag(rep(1, length(pars)))
)
```

The `priors` object in our example is initiated with zeros. Under what conditions would this priors object differ?

The next step is to load your log likelihood function/script.

```
source(file = "yourLogLikelihoodFile.R")
```

Once you've setup your parameters, priors and written a log likelihood function, the next step is to initialise the `sampler` object.

```
sampler <- pmwgs(
  data = wgnmks2008Fast,
  pars = pars,
  prior = priors,
  ll_func = SDT_ll_fast
)
```

The `pmwgs` function takes a set of arguments (listed below) and returns a list containing the required components for performing the particle metropolis within Gibbs steps.

- `data` =a data frame (e.g.`wgnmks2008Fast`) with a column for participants called `subject`
- `pars` = the model parameters to be used (e.g.`pars`)
- `prior` = the priors to be used (e.g.`priors`)
- `ll_func` = name of log likelihood function you've sourced above (e.g.`SDT_ll_fast`)

```
sampler <- pmwgs(
  data = wgnmks2008Fast,
  pars = pars,
  prior = priors,
  ll_func = SDT_ll_fast
)
```

1.6.1 Model start points

You have the option to set model start points. We use 0 for the mean (mu) and a variance of 0.01. If you chose not to specify start points, the sampler will randomly sample points from the prior distribution.

```
start_points <- list(
  mu = rep(0, length.out = length(pars)),
  sig2 = diag(rep(.01, length(pars)))
)
```

The `start_points` object contains two vectors:

- `mu` a vector of start points for the mu of each model parameter
- `sig2` vector containing the start points of the covariance matrix of covariance between model parameters.

1.6.2 Running the sampler

Okay - now we are ready to run the sampler.

```
sampler <- init(sampler, theta_mu = start_points$mu,
                theta_sig = start_points$sig2)
```

Here we are using the `init` function to generate initial start points for the random effects and storing them in the `sampler` object. First we pass the `sampler` object from above that includes our data, parameters, priors and log likelihood function. If we decided to specify our own start points (as above), we would include the `theta_mu` and `theta_sig` arguments.

Now we can run the sampler using the `run_stage` function. The `run_stage` function takes four arguments:

- `x` the `sampler` object including parameters
- `stage =` the sampling stage (e.g. `"burn"`, `"adapt"` or `"sample"`)
- `iter =` is the number of iterations for the sampling stage
- `particles =` is the number of particles generated on each iteration
- `display_progress =` shows progress bar for current stage
- `n_cores =` numner of processor cores to run stage on
- `epsilon =` must be >0<1 We will write something here

It is optional to include the `iter =` and `particles =` arguments. If these are not included, `iter` and `particles` default to 1000. The number of iterations you choose for your burn in stage is similar to choices made when running deMCMC, however, this varies depending on the time the model takes to reach the 'real' posterior space.

First we run our burn-in stage by setting `stage =` to `"burn"`.

```
burned <- run_stage(sampler, stage = "burn", iter = 1000, particles = 20, display_progr
```

Now we run our adaptation stage by setting `stage = "adapt"`. This function creates an efficient proposal distribution. The sampler will attempt to create the proposal distribution after 20 unique particles have been accepted for each

subject. The sampler will then test whether the distribution was able to be created and if it was created, the sampler will move to the next stage otherwise the sampler will continue to sample. The number of iterations needs to be great enough to generate enough unique samples, but not too large as to make..... Check this with Scott/Guy

```
adapted <- run_stage(burned, stage = "adapt", iter = 1000, particles = 20, n_cores = 8)
```

At the start of the `sampled` stage, the sampler object will create a 'proposal' distribution for each subject's random effects using a conditional multi-variate normal. This proposal distribution is then used to efficiently generate new particles for each subject which means we can reduce the number of particles on each iteration whilst still achieving acceptance rates.

```
sampled <- run_stage(adapted, stage = "sample", iter = 1000, particles = 20, n_cores = 8)
```

1.7 Check sampling process

It is a good idea to check your samples by producing some simple plots as shown below. The first plot gives an indication of the chains for the group level parameters. In this example, you will see the chains take only several iterations before arriving at the posterior, however, this may not always be the case. Each parameter chain (lines on plot 1.4) should be stationary i.e. the chain should not trend up or down, and once the chain reaches the posterior, the chain should remain relatively 'thin'. If the chain is wide or continually jump between large values (e.g. move between -3 and 3) then there is likely an error in your log likelihood function.

Check with Reilly sentence below

As you can see in 1.4, the chains are clearly separate and are stable (not trending in any direction), as well as being relatively thin (largest variance ~1).

The second plot below (1.5) shows the likelihoods across iterations for each subject. Again we see that the likelihood values jump up after only a few iterations and then remain stable, with only slight movement.

1.8 Simulating posterior data

Now we'll cover the sample operation within the fast log likelihood function. We will use this on the full data set. The sample operation can be carried out in several ways (using rbinom etc). Please not that we do NOT recommend using this approach below and this should serve as an example only.

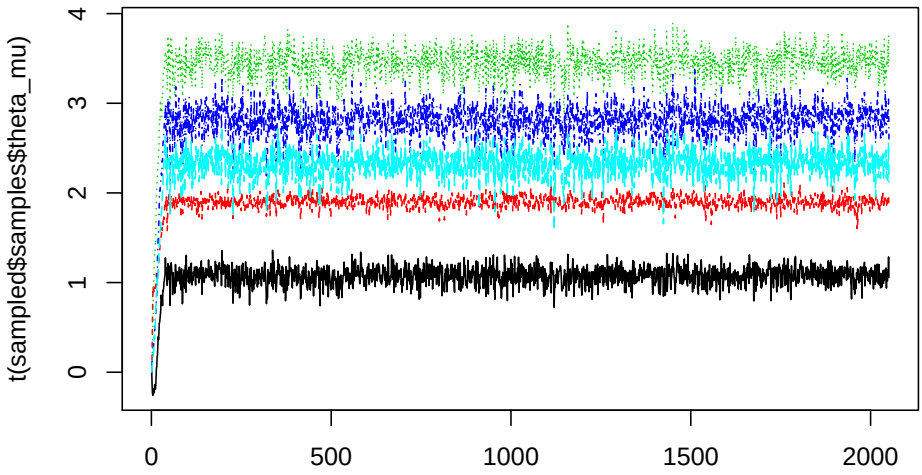

Figure 1.4: Posterior samples of parameters

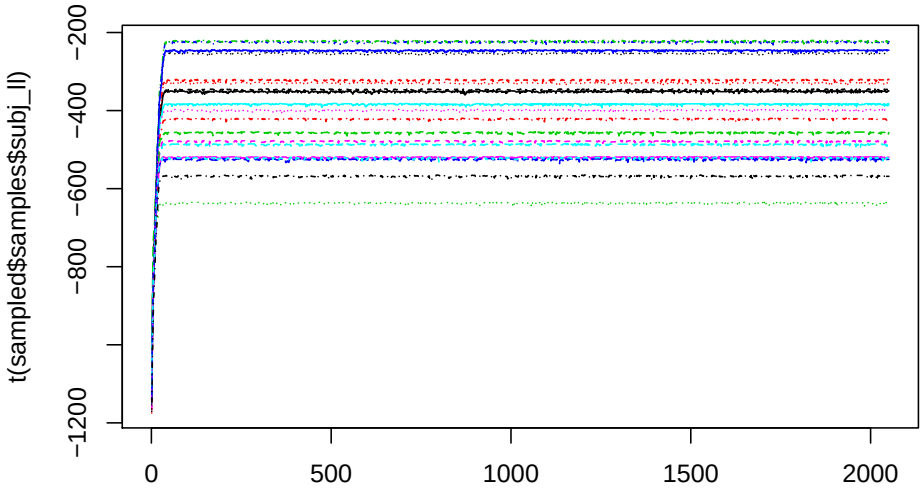

Figure 1.5: Posterior samples of subject log likelihoods

The `sample` process is similar to what we've covered above. We begin by assigning `NA`s to the response column to prepare it for simulated response data. We then consider a subset of the data, beginning with word condition and high-frequency `hf` word stimuli trials.

```
  else{
    data$resp <- NA
   for (i in 1:nrow(data)){
      if (data$cond[i] == "w"){
        if (data$stim[i] == "hf"){
```

We then take the criterion for the word condition i.e. `C.w`. To simulate a response given our parameters we use `rnorm` to pick a random value from a normal distribution with `mean = HF.d` (i.e. high frequency word stimulus) and a SD of 1 and we test that value against the word criterion `C.w`. If the value is larger than `C.w`, the simulated response will be word otherwise, the simulated response will be non-word.

```
data$resp[i] <- ifelse(test = (rnorm(1, mean = x["HF.d"],
                                     sd = 1)) > x["C.w"], "word", "non-word")
```

We repeat this process for each condition and stimulus combination as shown in the code block below.

```
{ else if (data$stim[i] == "lf") {
  data$resp[i] <- ifelse(test = (rnorm(1, mean = x["LF.d"],
                                       sd = 1)) > x["C.w"], "word", "non-word")
  } else if (data$stim[i] == "vlf") {
    data$resp[i] <- ifelse(test = (rnorm(1, mean = x["VLF.d"],
                                         sd = 1)) > x["C.w"], "word", "non-word")
    } else {
      data$resp[i] <- ifelse(test = (rnorm(1, mean = 0,
                                           sd = 1)) > x["C.w"], "word", "non-word")
      }
} else {
  if (data$stim[i] == "hf") {
    data$resp[i] <- ifelse(test = (rnorm(1, mean = x["HF.d"],
                                         sd = 1)) > x["C.nw"], "word", "non-word")
    } else if (data$stim[i] == "lf") {
          data$resp[i] <- ifelse(test = (rnorm(1, mean = x["LF.d"],
                                               sd = 1)) > x["C.nw"], "word", "non-word")
        } else if (data$stim[i] == "vlf") {
          data$resp[i] <- ifelse(test = (rnorm(1, mean = x["VLF.d"],
                                               sd = 1)) > x["C.nw"], "word", "non-word")
        } else {
          data$resp[i] <- ifelse(test = (rnorm(1, mean = 0,
                                               sd = 1)) > x["C.nw"], "word", "non-word")
          }
```

```
        }
```

Now we can run our simulation. Below is some code to achieve this.

```
n.posterior <- 20 # Number of samples from posterior distribution for each parameter.
pp.data <- list()
S <- unique(wgnmks2008$subject)
data = split(x = wgnmks2008, f = wgnmks2008$subject)
for (s in S) {
  cat(s," ")
  iterations = round(seq(from = 1051, to = sampled$samples$idx, length.out = n.posteri
  for (i in 1:length(iterations)) {
    x <- sampled$samples$alpha[, s, iterations[i]]
    names(x) <- pars
    tmp <- SDT_ll_fast(x = x, data = wgnmks2008[wgnmks2008$subject == s,], sample = TRU
    if (i == 1) {
      pp.data[[s]] <- cbind(i,tmp)
    } else {
      pp.data[[s]] <- rbind(pp.data[[s]], cbind(i, tmp))
    }
  }

}
```

And now we can plot samples against the data.

Figure 1.6 shows 20 posterior draws (dots) plotted against the data (bars). The posterior draws are for each individual subject - shown here is the average proportion of word responses. Evidently, the model appears to fit the data well.

#PMwG sampler and sequential sampling models

In this chapter we'll demonstrate how to use the PMwG sampler with a sequential sampling model; the Linear Ballistic Accumulator (LBA). Please ensure the psamplers package is installed. We currently recommended installing psamplers via devtools.

```
# The samplers package will be on CRAN - this step will be removed.
install_github('newcastlecl\pmwg')
```

```
library(pmwg)
```

1.9 Description of Forstmann experiment

Forstmann et al looked at neural correlates of decision making under time pressure, with an aim to identify areas of the brain associated with speed-accuracy tradeoff. Imaging (fMRI) and behavioural data was collected; however, we

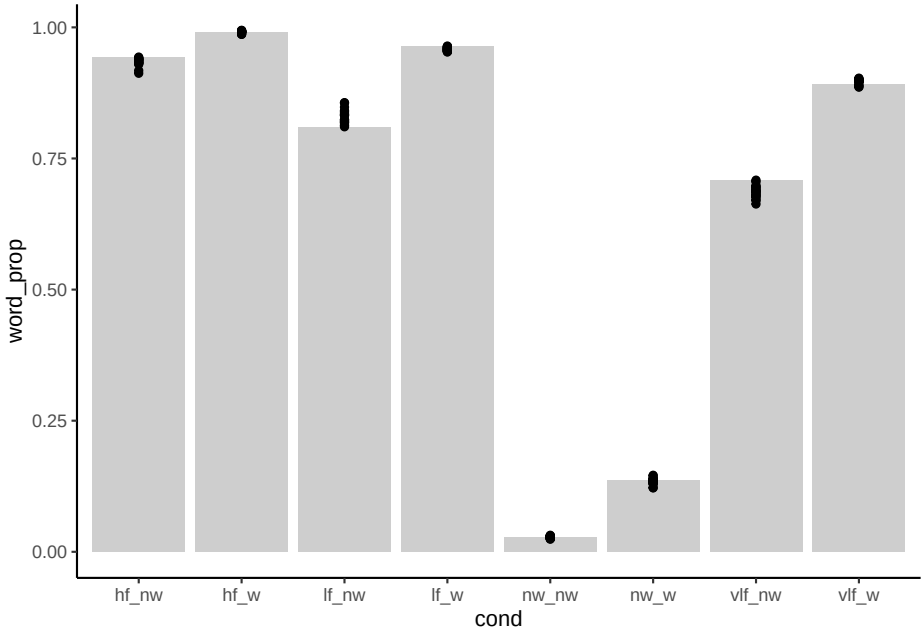

Figure 1.6: Appropriate figure caption HERE

will analyse behavioural data from the decision-making task only. In terms of modelling the data, Forstmann expected to find differences in thresholds (direction?) for each of the three speed-emphasis conditions. We have included the Forstmann et als data in the psamplers package as a data frame object named `forstmann`. The sampler requires a data frame with a `subject` column. The subject column data type can be a factor or numeric.

Table 1.3 shows the first ten trials from the Forstmann dataset. Participants `(n = 19)` were asked to indicate whether a cloud of dots in a random-dot kinematogram (RDK) moved to the left or the right of the screen. The IV was a within-subject, speed-accuracy manipulation where, before each trial began, pariticipants were instructed to make their choice accurately `(condition = 1)`, with urgency`(condition = 3)`or were presented with a neutral message `(condition = 2)`. Choice and response time data was collected. Choices were coded as correct `(correct = 2)` or incorrect `(correct = 1)` and response times `(rt)` were recorded in seconds. For more information about the design of the experiment please see the original paper.

Table 1.3: First 10 trials in Forstmann dataset. The 'forstmann' dataset is an object/data frame

	subject	condition	stim	resp	rt
1115	1	1	2	2	0.4319
1116	1	3	2	2	0.5015
1117	1	3	1	1	0.3104
1118	1	1	2	2	0.4809
1119	1	1	1	1	0.3415
1120	1	2	1	1	0.3465
1121	1	2	1	1	0.3572
1122	1	2	2	2	0.4042
1123	1	2	1	1	0.3866
1124	1	1	2	2	0.3683

1.10 Setting up the sampler

There are a number of preliminary steps we need to complete before running the sampler. Let's begin by defining the Linear Ballistic Accumulator (LBA) (Brown and Heathcote, 2008) model parameters.

- `b` threshold parameter (the evidence required to make a response)
- `v` is drift rate or average speed of evidence accumulation
- `A` is the model's start point
- `t0` is non-decision time.
- Do we need to mention `sv` here?

Now we know the LBA model parameters, we can create a vector of model parameter names, which we'll use in our log likelihood function. You can name this object as you wish; however, in our example, we will call it `pars`. The parameters you list in the `pars` vector must match the names and number of parameters you include in your log likelihood function.

```
pars <- c("b1", "b2", "b3", "A", "v1", "v2", "t0")
```

For the Forstmann dataset, we need three threshold parameters (`b1`, `b2`, and `b3` i.e. one for each condition) because we assume that the participant responds to each condition with a different level of caution. We include two drift rate parameters: `v1` for the incorrect accumulator and `v2` for the correct accumulator, a start point parameter `A` and a non-decision time `t0` parameter. We've made a decision to set the `sv` to 1 to satisfy the scaling properties of the model, as such we haven't included the `sv` parameter in the `pars` vector - it is found in the LBA's likelihood function (see below).

Next we create a `priors` object; a list that contains two components Do we need to explain what priors are and why we do this?

- `theta_mu` a vector containing the prior for model parameter means
- `theta_sig` the prior covariance matrix for model parameters.

```
priors <- list(theta_mu = rep(0, length(pars)),
  theta_sig = diag(rep(1, length(pars)))
)
```

The `priors` object in our example is initiated with zeros. Under what conditions would this priors object differ?

1.11 Writing the LBA Log-Likelihood Function

Now we have our parameter vector and priors object, we can write our log likelihood functions. Just as we did with the SDT example, we'll write a slow and a fast log likelihood function. The runtime difference is caused by calling the `dLBA` function line-by-line for the slow log likelihood and calling the `dLBA` function once for all the data in the fast log likelihood function. When writing a new log likelihood function, we suggest starting with a slow, line-by-line function for easier debugging.

The `lba_loglike` function takes three arguments:

- `x` is a named parameter vector (e.g. `pars`)
- `data` is your data set (e.g.`forstmann`)
- `sample = FALSE` calculates a density function or `TRUE` generates a posterior, predictive sample that matches the shape of data.

The log-likelihood function requires three arguments: - x: a vector of named parameter values. - data: a data set (which must include a "subject" column) - sample: to generate a posterior predictive sample that matches the shape of data (when set to TRUE)

The log likelihood function shown below includes functions from the `rtdists` package for generating data and estimating density.

Or you can write a log likelihood function as we have done with the LBA log likelihood function below. If you'd like to run through this example, it is best to copy the `lba_loglike` function from the code block below rather than copying from the following separate code chunks, as some curly braces have been removed from code chunks.

```
lba_loglike <- function(x, data, sample = FALSE) {
  x <- exp(x)
  if (any(data$rt < x["t0"])) {
    return(-1e10)
  }

  bs <- x["A"] + x[c("b1", "b2", "b3")][data$condition]
```

```
8
9   if (sample) {
10    out <- rtdists::rLBA(n = nrow(data),
11                         A = x["A"],
12                         b = bs,
13                         t0 = x["t0"],
14                         mean_v = x[c("v1", "v2")],
15                         sd_v = c(1, 1),
16                         distribution = "norm",
17                         silent = TRUE)
18  } else {
19    out <- rtdists::dLBA(rt = data$rt,
20                         response = data$correct,
21                         A = x["A"],
22                         b = bs,
23                         t0 = x["t0"],
24                         mean_v = x[c("v1", "v2")],
25                         sd_v = c(1, 1),
26                         distribution = "norm",
27                         silent = TRUE)
28    bad <- (out < 1e-10) | (!is.finite(out))
29    out[bad] <- 1e-10
30    out <- sum(log(out))
31  }
32  out
33 }
```

The first line in the `lba_loglike` function (Line 2 below) takes the exponent of the parameter values to move all parameter values to the real line. The purpose of this is to….. Line 3 and 4 then checks RTs are faster than the non-decision time parameter, and zero those RTs that are faster than non-decision time.

```
1 lba_loglike <- function(x, data, sample = FALSE) {
2   x <- exp(x)
3   if (any(data$rt < x["t0"])) {
4     return(-1e10)
5   }
```

Next (Line 7) we create a vector containing threshold parameters for each row in the data set that takes into account the condition and adds the start point value. We add the start point parameter `A` value to each threshold parameter so that threshold is greater than the start point value.

```
7   bs <- x["A"] + x[c("b1", "b2", "b3")][data$condition]
```

The `if else` statement below (Line 9-32) does one of two things: `if (sample)` calculates the posterior predictive data and the `else` statement calculates the

density function. Toward the end of the `else` statment (line 28) we take all implausible likelihood values, assign them to the `bad` object and set them to zero. The final line in the `else` statement takes the log of all likelihood values, sums them and then assigns the model's log likelihood value to the `out` variable.

```
9   if (sample) {
10    out <- rtdists::rLBA(n = nrow(data),
11                         A = x["A"],
12                         b = bs,
13                         t0 = x["t0"],
14                         mean_v = x[c("v1", "v2")],
15                         sd_v = c(1, 1),
16                         distribution = "norm",
17                         silent = TRUE)
18  } else {
19    out <- rtdists::dLBA(rt = data$rt,
20                         response = data$correct,
21                         A = x["A"],
22                         b = bs,
23                         t0 = x["t0"],
24                         mean_v = x[c("v1", "v2")],
25                         sd_v = c(1, 1),
26                         distribution = "norm",
27                         silent = TRUE)
28    bad <- (out < 1e-10) | (!is.finite(out))
29    out[bad] <- 1e-10
30    out <- sum(log(out))
31  }
32  out
```

The next step is to include your log likelihood function. This must be called before you create the sampler object in the following step. You can load your log likelihood function from an external script:

```
source(file = "yourLogLikelihoodFile.R")
```

Once you've setup your parameters, priors and your log likelihood function, the next step is to initialise the `sampler` object.

```
sampler <- pmwgs(
  data = forstmann,
  pars = pars,
  prior = priors,
  ll_func = lba_loglike
)
```

The `pmwgs` function takes a set of arguments (listed below) and returns a list containing the required components for performing the particle metropolis within

Gibbs steps.

- `data =` your data - a data frame (e.g.`forstmann`) with a column for participants called `subject`
- `pars =` the model parameters to be used (e.g.`pars`)
- `prior =` the priors to be used (e.g.`priors`)
- `ll_func =` name of log likelihood function you've sourced above (e.g.`lba_loglike`)

1.11.1 Model start points

You have the option to set model start points. We have specified sensible start points for the Forstmann dataset. If you chose not to specify start points, the sampler will randomly sample points from the prior distribution.

```
start_points <- list(
  mu = c(.2, .2, .2, .4, .3, 1.3, -2),
  sig2 = diag(rep(.01, length(pars)))
)
```

The `start_points` object contains two vectors:

- `mu` a vector of start points for the mu of each model parameter
- `sig2` vector containing the start points of the covariance matrix of covariance between model parameters.

1.11.2 Running the sampler

Okay - now we are ready to run the sampler.

```
sampler <- init(sampler, theta_mu = start_points$mu,
                theta_sig = start_points$sig2)
```

Here we are using the `init` function to generate initial start points for the random effects and storing them in the `sampler` object. First we pass the `sampler` object from above that includes our data, parameters, priors and log likelihood function. If we decided to specify our own start points (as above), we would include the `theta_mu` and `theta_sig` arguments.

Now we can run the sampler using the `run_stage` function. The `run_stage` function takes four arguments:

- `x` the `sampler` object including parameters
- `stage =` the sampling stage (e.g. `"burn"`, `"adapt"` or `"sample"`)
- `iter =` is the number of iterations for the sampling stage
- `particles =` is the number of particles generated on each iteration

It is optional to include the `iter =` and `particles =` arguments. If these are not included, `iter` and `particles` default to 1000. The number of iterations you choose for your burn in stage is similar to choices made when running deMCMC, however, this varies depending on the time the model takes to reach the 'real' posterior space.

First we run our burn in stage by setting `stage =` to `"burn"`. Here we have set iterations to be 500, which may take some time.

```
burned <- run_stage(sampler, stage = "burn", iter = 500, particles = 1000)
```

Now we run our adaptation stage by setting `stage = "adapt"` Because we have not specified number of iterations or particles, the sampler will use the default value of 1000 for each of these arguments. N.B. The sampler will quit adaptation stage after 20 unique values have been accepted for each subject. This means adaptation may not use all 1000 iterations.

```
adapted <- run_stage(burned, stage = "adapt")
```

At the start of the `sampled` stage, the sampler object will create a 'proposal' distribution for each subject's random effects using a conditional multi-variate normal. This proposal distribution is then used to efficiently generate new particles for each subject which means we can reduce the number of particles on each iteration whilst still achieving acceptance rates.

```
sampled <- run_stage(adapted, stage = "sample", iter = 100, particles = 100)
```

Chapter 2

Example 2 - Single threshold parameter

In this second example, we will use a single `b` threshold parameter (the evidence required to make a response). Altering the code/model is simple. First, update your parameter object (e.g.`pars`) by including a single `b` threshold parameter:

```
pars <- c("b", "A", "v1", "v2", "t0")
```

The `priors` object will update based on the length of your parameter object. Remove this?

```
priors <- list(theta_mu = rep(0, length(pars)),
  theta_sig = diag(rep(1, length(pars)))
)
```

The next step is to modify your log likelihood function by updating the threshold parameter vector (`bs`) so that the name and number of threshold parameters (`b`) matches the name and number of threshold parameters you've specified in your `pars` vector. In this case, we require a single `b` parameter.

```
  bs <- x["A"] + x["b"][data$condition]
```

The subsequent lines of code in the log likelihood function remain unchanged.

As mentioned in example 1, you may set model start points. It is important to provide the same number of values for `mu` as the number of parameters you've set in your `pars` vector.

```
start_points <- list(
  mu = c(.2,.4, .3, 1.3, -2), # We have set five parameter start points here,
                              # matching the number of parameters in the pars vector.
  sig2 = diag(rep(.01, length(pars)))
```

```
)
```

And now you're ready to run the sampler as outlined in example 1.

Chapter 3

Example 3 - Wagenmakers (2008) Experiment 2

See iSquared paper eperiment 2

Chapter 4

Common Problems (Better name required) Troubleshooting page

4.1 How to write a log likelihood function

- What key elements are required in a log likelihood function to be used in the sampler
- Show comparison times for slow LL and fast LL
- Check list for commmon errors - brief list check this, check that etc.

4.1.1 Writing your log likelihood function: Tips, errors and check list

1. The parameter specified does not exist

The parameter name is not specified to be estimated i.e. it is not in the parameter names argument or it is misspelled. Make sure `pars` vector contains the same parameter names you have included in your log likelihood function and it is the same length. Do not rely on the log likelihood function to throw an error in this case. (e.g.`x['b']`)

2. All non-continuous data frame variables must be a `factor`.

Data frame variables should be `factors` unless the variable is a continuous variable e.g. response time. If you pass `character` variables to `if` statements and/or `for` loops in your log likelihood function, errors will not occur, however, your log likelihood estimate will be incorrect. For example, avoid using character

strings like `data$condition == "easy"`. If you must use a character string, be sure to convert the string to a factor with `as.factor`.

3. Spelling errors or mismatched column name references

Correctly reference data frame column names in your log likelihood function e.g. `data$RT` != `data$rt`.

4. When initialising a vector of parameter values - values are not filling in properly

E.g. When a vector for b for all the values across the data set to be used, but there are NAs filling it somewhere.

5. Make sure operations are done on the right scale.

6. Data frame variables are scaled appropriately for the model

Check your variables are correctly scaled and in the correct units. For example, with the LBA, response times must be in seconds rather than milliseconds.

7. The log likelihood is printed/outputted at the end of function

Make sure your log likelihood function prints an estimate at the end of the function and the estimate is correctly obtained e.g. sum the log likelihood estimates for all trials/rows.

8. Sampling error occurs

When sampling, the generated columns are not outputted

9. When executing functions row by row (i.e. trial-wise), index MUST be included

If writing a trial-wise/row-wise function (e.g. `if` statement, `for` loop), index `i` must be specified.

```
if (data$condition == "easy")      # Incorrect reference when iterating over variable
if (data$condition[i] == "easy")   # Include i index
```

10. Changing parameter values changes the log likelihood estimate

A simple check to run on your log likelihood function is to modify your parameter values and observe the change to log likelihood estimate. Then check if changing parameter values which rely on conditions actually change the log likelihood estimate.

11. Make sure you have the latest version of the PMwG Samplers package

checkVersion("psamplers")

Chapter 5

Non RT/Choice example

Bibliography

Brown, S. and Heathcote, A. (2008). The simplest complete model of choice response time: Linear ballistic accumulation. *Cognitive psychology*, 57(3):153–178.

Wagenmakers, E.-J., Ratcliff, R., Gomez, P., and McKoon, G. (2008). A diffusion model account of criterion shifts in the lexical decision task. *Journal of memory and language*, 58(1):140–159.