User Manual for Implementing VB and K-fold Cross-validation with VB for hierarchical LBA models

Viet-Hung Dao, David Gunawan, Minh-Ngoc Tran, Robert Kohn, Guy E. Hawkins, Scott D. Brown

August 12, 2021

Contents

1 Introduction

This manual has two purposes.

1. The first is to guide you on how to reproduce the VB approximation and K-fold CVVB model selection for any hierarchical LBA (HLBA) model considered in Dao et al. (2021).

2. The second is to instruct you on how to modify the Matlab code to apply to your own analysis.

To achieve the first goal please read Sections 2.1 and 3.1. For the second goal, it is necessary to to do some extra coding. The current code is written to minimize extra work. We recommend reading the whole document (you may skip section 3.3 at the first reading). Section 2.2 is important as it explains the fundamental principles and the proposed framework of the code. We recommend looking at the code while reading this document.

Before starting

- Make sure to download all the data (in the Data folder), the functions (in the Functions folder) and the main code from the website https://github.com/Henry-Dao/CVVB.
- Make sure to set the current directory in Matlab to the folder containing the files listed above.

2 Implementing the VB approximation for hierarchical LBA models

This section first shows how to implement the VB approximation for any hierarchical LBA model considered in the paper. The Matlab code is designed to be user-friendly so that you can easily choose a case study, model, and adjustment the tuning parameters of the algorithm.

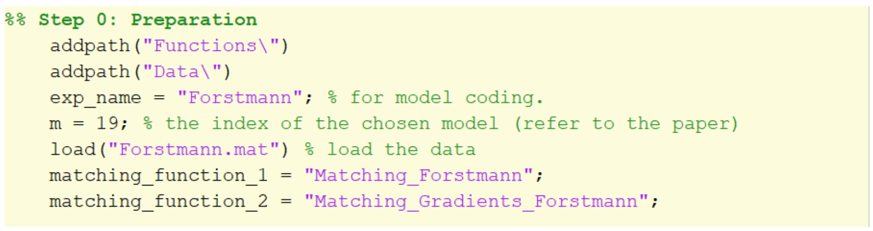
2.1 The main script file for implementing VB:

To make the VB implementation for a hierarchical LBA model simple and easy-to-use, we created a Matlab script file named `VB_main_code.m` where you can select the case study, the model you wish to estimate by VB and you can adjust the tuning parameters. The main code `VB_main_code.m`[1] consists of the following steps:

1. **Step 0: Preparation** To reproduce the results in the paper, it is only necessary to change this step. First, choose the case study and model(s). There are three available options: Forstmann (case study 1), Mnemonic (case study 2) and Lexical (case study 3). For each type of experiment, there are two corresponding functions (or two matching functions), which are explained in 2.2. For example, the code in the figure below reproduces the VB approximation for hierarchical LBA model 19[2] in the first case study:

[1]We recommend opening the script file while reading this section.

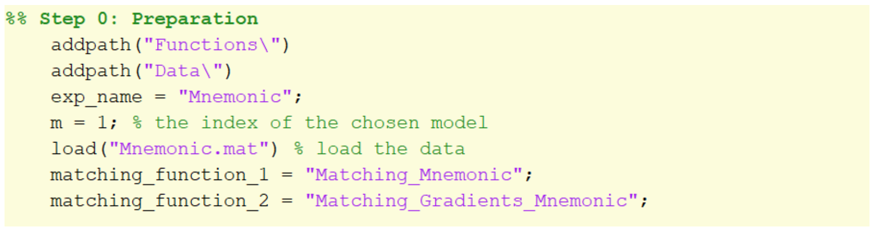
[2]for the model index, see Table 1.



```
%% Step 0: Preparation
    addpath("Functions\")
    addpath("Data\")
    exp_name = "Forstmann"; % for model coding.
    m = 19; % the index of the chosen model (refer to the paper)
    load("Forstmann.mat") % load the data
    matching_function_1 = "Matching_Forstmann";
    matching_function_2 = "Matching_Gradients_Forstmann";
```

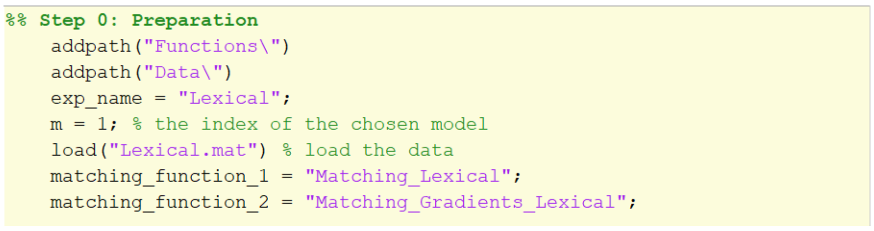
Figure 1: This code performs the VB approximation for the model having index 19 (`m = 19`) from the first case study (`exp_name = "Forstmann"`).

The experiment name, data name, and the two matching functions names must be consistent. To perform VB for a hierarchical LBA model from the other two case studies (Mnemonic and Lexical), it is necessary to change `data`, `exp_name` and the matching functions as in Figures 2 and 3.



```
%% Step 0: Preparation
    addpath("Functions\")
    addpath("Data\")
    exp_name = "Mnemonic";
    m = 1; % the index of the chosen model
    load("Mnemonic.mat") % load the data
    matching_function_1 = "Matching_Mnemonic";
    matching_function_2 = "Matching_Gradients_Mnemonic";
```

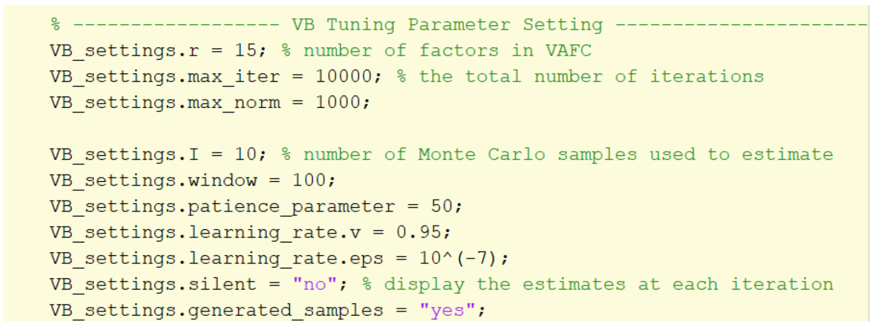
Figure 2: This code performs a VB approximation for a model having index 1 (`m = 1`) from the second case study (`exp_name = "Mnemonic"`).



```
%% Step 0: Preparation
    addpath("Functions\")
    addpath("Data\")
    exp_name = "Lexical";
    m = 1; % the index of the chosen model
    load("Lexical.mat") % load the data
    matching_function_1 = "Matching_Lexical";
    matching_function_2 = "Matching_Gradients_Lexical";
```

Figure 3: This code performs a VB approximation for a model having index 1 (`m = 1`) from the third case study (`exp_name = "Lexical"`).

You can also adjust the VB settings (number of Monte Carlo samples used to estimate the lower bound, the number of factors used in the variational distribution, etc.) as in Figure 4.

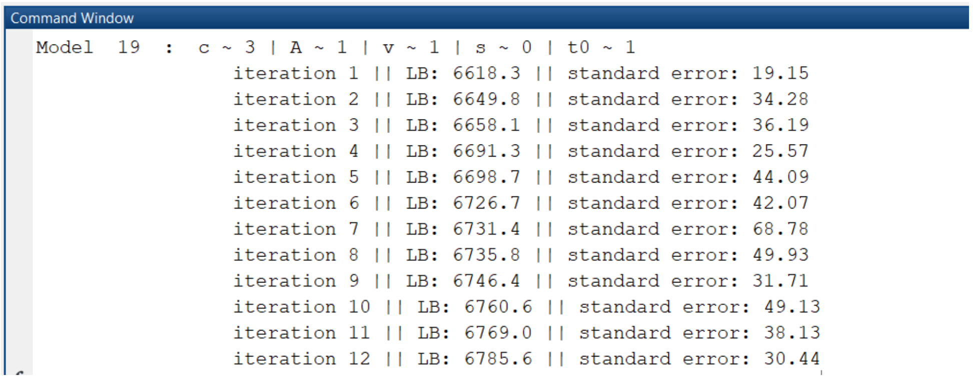


```
% ------------------ VB Tuning Parameter Setting ----------------------
VB_settings.r = 15; % number of factors in VAFC
VB_settings.max_iter = 10000; % the total number of iterations
VB_settings.max_norm = 1000;

VB_settings.I = 10; % number of Monte Carlo samples used to estimate
VB_settings.window = 100;
VB_settings.patience_parameter = 50;
VB_settings.learning_rate.v = 0.95;
VB_settings.learning_rate.eps = 10^(-7);
VB_settings.silent = "no"; % display the estimates at each iteration
VB_settings.generated_samples = "yes";
```

Figure 4: This code shows the values assigned to the tuning parameters. In particular, we use the Hybrid VAFC with 15 factors (`VB_settings.r`). The lower bound (LB) and gradient are estimated using 10 Monte Carlo samples (`VB_settings.I`). The LB estimates are smoothed over a window size of 100 iterations (`VB_settings.window`). The optimization process stops if the smoothed lower bound (SLB) estimate does not increase for 50 consecutive iterations (`VB_settings.patience_parameter`), or the number of iterations reaches its threshold of 10,000 (`VB_settings.max_iter`).

2. **Step 1: Model specification** This step can be skipped if you only want to implement the VB approximation for the hierarchical LBA models considered in the paper. If you wish to implement VB for a new hierarchical LBA model, it is necessary to define the model in the format described in Section 2.3.

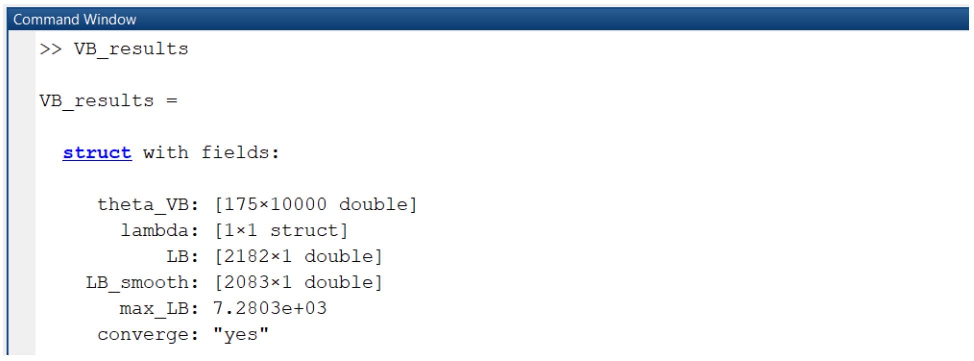
3. **Step 2: VB Algorithm** It is unnecessary to make any change in this step. After making the necessary adjustments in steps 0, you can now perform the VB approximation by running the script file. While the code is executing, the command window (figure 5) shows the lower bound estimates together with the standard errors from each iteration. If everything is correct, an increase in the lower bound estimates should be observed.



```
Model  19  :  c ~ 3 | A ~ 1 | v ~ 1 | s ~ 0 | t0 ~ 1
                 iteration 1 || LB: 6618.3 || standard error: 19.15
                 iteration 2 || LB: 6649.8 || standard error: 34.28
                 iteration 3 || LB: 6658.1 || standard error: 36.19
                 iteration 4 || LB: 6691.3 || standard error: 25.57
                 iteration 5 || LB: 6698.7 || standard error: 44.09
                 iteration 6 || LB: 6726.7 || standard error: 42.07
                 iteration 7 || LB: 6731.4 || standard error: 68.78
                 iteration 8 || LB: 6735.8 || standard error: 49.93
                 iteration 9 || LB: 6746.4 || standard error: 31.71
                 iteration 10 || LB: 6760.6 || standard error: 49.13
                 iteration 11 || LB: 6769.0 || standard error: 38.13
                 iteration 12 || LB: 6785.6 || standard error: 30.44
```

Figure 5: The estimation results shown on the command window while the script file `VB_main_code.m` is running.

4. **Step 3: Extracting the results** When the VB estimation is finished, the VB results are stored in `VB_results`, which is a structure with the following fields:



```
Command Window
>> VB_results

VB_results =

  struct with fields:

     theta_VB: [175×10000 double]
       lambda: [1×1 struct]
           LB: [2182×1 double]
    LB_smooth: [2083×1 double]
       max_LB: 7.2803e+03
     converge: "yes"
```

Figure 6: Approximation results are stored in a structure named `VB_results`.

The most important information is the optimal variational parameter $\boldsymbol{\lambda}$ and the lower bound estimates (for convergence checking). The commands in Step 3 give the running time, the plot of the smoothed lower bound estimates and the optimal $\boldsymbol{\lambda}$.

```
%% Step 3: Extract the results
disp(['The running time is ',num2str(round(CPUtime/60,1)),' minutes'])

plot(VB_results.LB_smooth)
title('Smoothed lower bound estimates')

disp('The best lambda is ')
VB_results.lambda
```

Figure 7: Step 3 from the main script file `VB_main_code.m`.

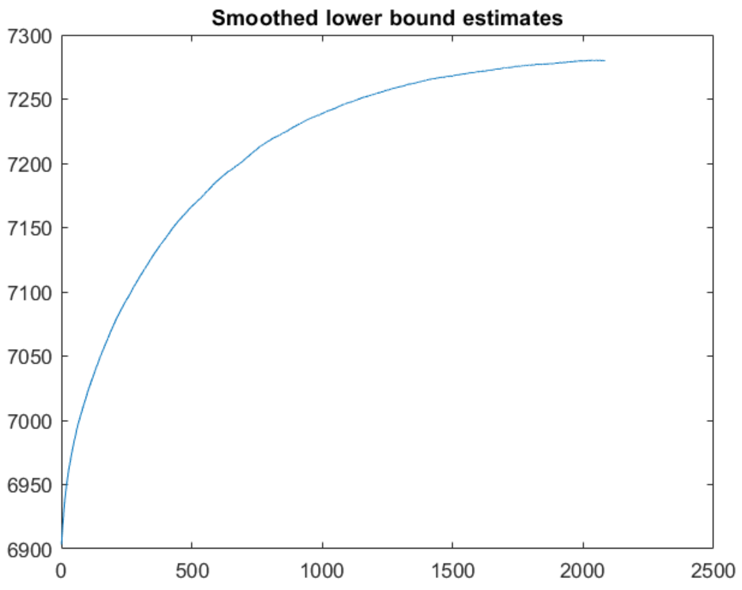


Figure 8: The smoothed lower bound estimates.

2.2 The core functions

This section explains the general framework proposed to implement VB for any hierarchical LBA model. The key part is the core functions. We first explain the motivation for introducing these functions. To implement VB, it is necessary to calculate $\log p(\boldsymbol{y}|\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|\boldsymbol{\theta})$. The conditional density of $\boldsymbol{y}$ is

$$p(\boldsymbol{y}|\boldsymbol{\theta}) = \prod_{j=1}^{J} p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j) = \prod_{j=1}^{J}\prod_{i=1}^{n_j} \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij}),$$

where $\boldsymbol{\theta} = (\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_J, \boldsymbol{\theta}_G)$ and $\boldsymbol{\alpha}_{ij}$ is the set of random effects corresponding to trial i[3] (to be explained shortly). Using this notation, evaluating $\log p(\boldsymbol{y}|\boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|\boldsymbol{\theta})$ is essentially computing $\log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij})$ and $\dfrac{\partial}{\partial \boldsymbol{\alpha_{ij}}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij})$ because

$$\log p(\boldsymbol{y}|\boldsymbol{\theta}) = \sum_{j=1}^{J}\sum_{i=1}^{n_j} \log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij}),$$

$$\nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{y}|\boldsymbol{\theta}) = \sum_{j=1}^{J}\sum_{i=1}^{n_j} \frac{\partial}{\partial \boldsymbol{\alpha_{ij}}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij});$$

i.e., what is needed is $\log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij})$ and $\dfrac{\partial}{\partial \boldsymbol{\alpha_{ij}}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij})$. In our framework, calculating $\log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij})$ and $\dfrac{\partial}{\partial \boldsymbol{\alpha_{ij}}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{\alpha}_{ij})$ for any hierarchical LBA variant is done via three core functions: the LBA density function and the two matching functions.

$$\begin{tikzcd}
\boldsymbol{\alpha}_j \arrow[r, "(1)"] & \boldsymbol{z}_j \arrow[r, "(2)"] & \boldsymbol{z}_{ij} \arrow[dr, "(3)"] & \\
& & & \boxed{\begin{array}{c}\log \mathrm{LBA}(y_{ij}|\boldsymbol{z}_{ij}) \\[1em] \dfrac{\partial}{\partial \boldsymbol{z}_{ij}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{z}_{ij})\end{array}} \arrow[dl, "(4)"] \\
\dfrac{\partial}{\partial \boldsymbol{\alpha}_j} & \dfrac{\partial}{\partial \boldsymbol{z}_j} \arrow[l, "(6)"'] & \dfrac{\partial}{\partial \boldsymbol{z}_{ij}} \arrow[l, "(5)"'] &
\end{tikzcd}$$

Figure 9: The diagram demonstrates the role and relation of the three core functions: the first matching function, the LBA density function and the second matching function.

1. The first matching function does tasks (1) to (3).
    - Task (1): Tranform the vector of random effects $\boldsymbol{\alpha}_j$ to the original (natural) form $\boldsymbol{z}_j$.

[3]Notice that $\boldsymbol{\alpha}_{ij}$ is not the ith component of $\boldsymbol{\alpha}_j$. The subscript i is used to emphasize the dependence on trial i.

- Task (2): for trial i, select the correct set of random effects $\boldsymbol{z}_{ij}$ ($\boldsymbol{z}_{ij}$ is a subvector of $\boldsymbol{z}_j$).
- Task (3): Convert the selected random effects $\boldsymbol{z}_{ij}$ into the correct format that can be used as an input for the LBA function.

2. The LBA density function (the green block) takes y_{ij} and $\boldsymbol{z}_{ij}$ as the inputs and computes $\text{LBA}(y_{ij}|\boldsymbol{z}_{ij})$ and $\dfrac{\partial}{\partial \boldsymbol{z}_{ij}} \log \text{LBA}(y_{ij}|\boldsymbol{z}_{ij})$.

3. The second matching function performs tasks (4) to (6) :

   - Task (4): Get the output $\dfrac{\partial}{\partial \boldsymbol{z}_j} \log \text{LBA}(y_{ij}|\boldsymbol{z}_{ij})$ from the LBA function.
   - Task (5): Get $\dfrac{\partial}{\partial \boldsymbol{z}_j} \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)$ by matching the derivatives $\dfrac{\partial}{\partial \boldsymbol{z}_{ij}} \log \text{LBA}(y_{ij}|\boldsymbol{z}_{ij})$ with each trial.
   - Task (6): Multiply $\dfrac{\partial}{\partial \boldsymbol{z}_j} \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)$ by the Jacobian to get $\dfrac{\partial}{\partial \boldsymbol{\alpha}_j} \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$.

The LBA density function can be used for all hierarchical LBA variants; it is only necessary to create the two matching functions that can perform tasks (1) to (6).

Remark. For computational efficiency, all the functions presented below are vectorized over trials.

2.2.1 The LBA density function

The code of this function is in the Matlab script file named `LBA_pdf.m`; [4] it is stored in folder 'Functions'. It is unnecessary to change this function, but it is useful to understand how it works.

1. **Its role:**
   The function evaluates $\log \text{LBA}(y_{ij}|\boldsymbol{z}_{ij})$, and its partial derivatives, with respect to the parameters b, c, A, v, s and τ.

2. **The inputs:** The Matlab function contains 8 input arguments: `RE`,`RT`,`b`,`A`,`v`, `s`,`tau` and `gradient`.

   - `RE` and `RT` are the observed response choices and the response time, respectively. They must be column vectors of the same length n_j.
   - Arguments `b`,`A`,`v`,`s` and `tau` represent the threshold parameters b, the upper bound A, the drift rate mean v, the drift rate standard deviation s and the non-decision time τ, respectively. Since the function allows **all the parameters to be different between two accumulators** and it is **vectorized over trials**, all the inputs must be column vectors and/or matrices

$$b = \begin{bmatrix} b_{1j}^{(1)} & b_{1j}^{(2)} \\ \vdots & \vdots \\ b_{n_j j}^{(1)} & b_{n_j j}^{(2)} \end{bmatrix}, \ A = \begin{bmatrix} A_{1j}^{(1)} & A_{1j}^{(2)} \\ \vdots & \vdots \\ A_{n_j j}^{(1)} & A_{n_j j}^{(2)} \end{bmatrix}, \ v = \begin{bmatrix} v_{1j}^{(1)} & v_{1j}^{(2)} \\ \vdots & \vdots \\ v_{n_j j}^{(1)} & v_{n_j j}^{(2)} \end{bmatrix},$$

[4] We recommend you to look at this file while reading this section.

$$s = \begin{bmatrix} s_{1j}^{(1)} & s_{1j}^{(2)} \\ \vdots & \vdots \\ s_{n_j j}^{(1)} & s_{n_j j}^{(2)} \end{bmatrix}, \tau = \begin{bmatrix} \tau_{1j}^{(1)} & \tau_{1j}^{(2)} \\ \vdots & \vdots \\ \tau_{n_j j}^{(1)} & \tau_{n_j j}^{(2)} \end{bmatrix}.$$

- The last argument `gradient` determines whether or not to compute the gradients. If `gradient = yes`, all the partial derivatives are computed.

3. **The outputs:** The results are stored in a structure with the following fields:

   - `.log` returns the log of the LBA density summed across trials.
   - `.grad_b` returns a matrix containing the partial derivates of log LBA with respect to the threshold parameters $b^{(1)}$ (column 1) and $b^{(2)}$ (column 2).

$$\texttt{.grad\_b} = \begin{bmatrix} \frac{\partial}{\partial b_{1j}^{(1)}} \log \mathrm{LBA}(y_{1j}|\theta_{1j}) & \frac{\partial}{\partial b_{1j}^{(2)}} \log \mathrm{LBA}(y_{1j}|\theta_{1j}) \\ \vdots & \vdots \\ \frac{\partial}{\partial b_{n_j j}^{(1)}} \log \mathrm{LBA}(y_{n_j j}|\theta_{n_j j}) & \frac{\partial}{\partial b_{n_j j}^{(2)}} \log \mathrm{LBA}(y_{n_j j}|\theta_{n_j j}) \end{bmatrix}$$

   - Similarly, the partial derivatives of the log LBA with respect to A, v, s, τ are stored in `.grad_A`,`.grad_v`, `.grad_s` and `.grad_tau`, respectively.

2.2.2 The two matching functions - a simple case

A simple model

The motivation for introducing the two matching functions is straightforward, but it is harder to explain their construction. We believe that the best way to proceed is to develop the ideas using a simple example. For expository purposes, consider the **hierarchical LBAmodel 3-1-1** (refer to Table 1 in the paper or Table 1 in this document). This model assumes that the separate threshold parameter c ($c = b - A$) varies across the three experimental conditions: accuracy, neutral and speed emphasis. All the other parameters, including A, v, and τ, are assumed to be the same across stimuli. The standard deviation of drift rate s is set equal to 1 and the drift rate mean v is different between the two accumulators. With this model specification, each subject has 7 random effects

$$\tilde{\boldsymbol{z}}_j = (c_j^{(a)}, c_j^{(n)}, c_j^{(s)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j)^{5};$$

the transformed random effects are

$$\boldsymbol{\alpha}_j = (\log c_j^{(a)}, \log c_j^{(n)}, \log c_j^{(s)}, \log A_j, \log v_j^{(1)}, \log v_j^{(2)}, \log \tau_j).$$

For expository purposes, suppose subject j has the following random effects:

$$b_j^{(a)} = 1.7, b_j^{(n)} = 1.1, b_j^{(s)} = 0.8, A_j = 0.6, v_j^{(1)} = 1, v_j^{(2)} = 3, \tau_j = 0.3;$$

[5] We denote $\boldsymbol{z}_j = (b_j^{(a)}, b_j^{(n)}, b_j^{(s)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j)$ and $\tilde{\boldsymbol{z}}_j = (c_j^{(a)}, c_j^{(n)}, c_j^{(s)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j)$.

and this subject performs 3 decision trials with the results

$$RT_j = \begin{bmatrix} 0.8 \\ 1.2 \\ 1.6 \end{bmatrix}, RE_j = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}, E_j = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix},$$

where RT_j and RE_j are the observed response time and response choices, respectively. The vector E_j indicates the experimental conditions in each trial (1 = 'accuracy', 2 = 'neutral' and 3 = 'speed').

The first matching function

The density for subject j is

$$p(\boldsymbol{y}_j|\boldsymbol{z}_j) = \text{LBA}(RE_{1j}, RT_{1j}|\boldsymbol{z}_{1j}) \times \text{LBA}(RE_{2j}, RT_{2j}|\boldsymbol{z}_{2j}) \times \text{LBA}(RE_{3j}, RT_{3j}|\boldsymbol{z}_{3j}).$$

As c is allowed to vary across conditions, the set of parameters $\boldsymbol{z}_{ij}$ coming into the LBA density is different across trials. In order to compute $p(\boldsymbol{y}_j|\boldsymbol{z}_j)$, for each observation, it is necessary to select the correct subset of random effects. First, consider the rule to see how it works.

- Trial 1: as $E_{1j} = 1$, the experimental condition is 'accuracy emphasis'. Therefore, the threshold parameter must be $b_j^{(a)}$ and the set of parameters corresponding to this trial is
$$\boldsymbol{z}_{1j} = (b_j^{(a)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j).$$
- Trial 2: $E_{2j} = 2$, the threshold parameter is $b_j^{(n)}$ and the set of parameters corresponding to this trial is
$$\boldsymbol{z}_{2j} = (b_j^{(n)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j).$$
- Trial 3: $E_{3j} = 3$, the threshold parameter is $b_j^{(s)}$ and the set of parameters corresponding to this trial is
$$\boldsymbol{z}_{3j} = (b_j^{(s)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j).$$

We now show how the matching function `Matching_Forstmann_model_3_1_1.m`[6] is coded in Matlab.

1. **The role of the function**
   Recall from Figure 9 that the first matching function performs the following three tasks:
   (a) Take the random effects $\boldsymbol{\alpha}_j$ and trasform to $\boldsymbol{z}_j$(task (1)).
   (b) For each trial, select the correct subset $\boldsymbol{z}_{ij}$ of $\boldsymbol{z}_j$(task (2)).

[6] We again suggest looking at the script file (stored in the folder 'Functions') while reading this section.

(c) Returns the set of $n_j \times 2$ matrices, one for each parameter, which are used as the input in `LBA_pdf.m`(task (3)).

$$b = \begin{bmatrix} b_{1j}^{(1)} & b_{1j}^{(2)} \\ \vdots & \vdots \\ b_{n_jj}^{(1)} & b_{n_jj}^{(2)} \end{bmatrix}, \; A = \begin{bmatrix} A_{1j}^{(1)} & A_{1j}^{(2)} \\ \vdots & \vdots \\ A_{n_jj}^{(1)} & A_{n_jj}^{(2)} \end{bmatrix}, \; v = \begin{bmatrix} v_{1j}^{(1)} & v_{1j}^{(2)} \\ \vdots & \vdots \\ v_{n_jj}^{(1)} & v_{n_jj}^{(2)} \end{bmatrix},$$
$$s = \begin{bmatrix} s_{1j}^{(1)} & s_{1j}^{(2)} \\ \vdots & \vdots \\ s_{n_jj}^{(1)} & s_{n_jj}^{(2)} \end{bmatrix}, \; \tau = \begin{bmatrix} \tau_{1j}^{(1)} & \tau_{1j}^{(2)} \\ \vdots & \vdots \\ \tau_{n_jj}^{(1)} & \tau_{n_jj}^{(2)} \end{bmatrix}.$$

2. **The inputs**

   - `model` is a structure containing the model specification. For this simple case, as we already know the model specification (model 3-1-1) in advance, this argument does not play any role. However, to make the function compatible with the code, you must keep this argument. The argument is important when we want to generalize the function so that it can be used for not only a single model, but a class of hierarchical LBA models. In model selection, having a matching function that can be applied for all competing models is desirable. Sections 2.3.2, 3.2 and 3.3 discuss the model specifications in detail.
   - `alpha_j` is $\boldsymbol{\alpha}_j$, which can be a column vector or a matrix (used in the VB initialization procedure).
   - `data_subject_j` is a structure containing all the data from subject j.

3. **The outputs**
   All the outputs are stored in a cell array named `z_ij`.

   - `z_ij{1}` a $n_j \times 2$ matrix containing all the threshold paramters b.
   - `z_ij{2}` a $n_j \times 2$ matrix containing all the upper bound A.
   - `z_ij{3}` a $n_j \times 2$ matrix containing all the drift rate mean v.
   - `z_ij{4}` a $n_j \times 2$ matrix containing all the drift rate standard deviation s.
   - `z_ij{5}` a $n_j \times 2$ matrix containing all the non-decision time τ.

Again, all operations are vectorized over trials to increase the computational efficiency so column vectors and matrices are unavoidable. We now consider the Matlab code, consisting of two parts:

1. **Part 1 - Transformation**
   Transform $\boldsymbol{\alpha}_j$ to the natural form $\boldsymbol{z}_j$, with

$$\boldsymbol{\alpha}_j = (\log c_j^{(a)}, \log c_j^{(n)}, \log c_j^{(s)}, \log A_j, \log v_j^{(1)}, \log v_j^{(2)}, \log \tau_j).$$

```
% ---------- Transform random effects to the natural form ---------

c_j = exp(alpha_j(1:3,:))'; % separate threshold c
A_j = exp(alpha_j(4,:))'; % upper bound A
v_j = exp(alpha_j(5:6,:))'; % drift rate means v
s_j = ones(R,2); % drift rate standard devation s = 1
tau_j = exp(alpha_j(7,:))'; % non-decision time tau
```

The natural random effects $\boldsymbol{z}_j$ are stored in a cell array named `z_j`.

2. **Part 2 - Matching**
   As the assumptions are different between parameters, different matching rules are required for different parameters. We consider each parameter separately.

   (a) First, consider the separate threshold parameter c. As this parameter **varies across experimental conditions**, the matching process is done by performing the following vector operation

$$c_j = c_j^{(a)} I_a + c_j^{(n)} I_n + c_j^{(s)} I_s,$$

   where I_a, I_n, I_s are indicator vectors of 'accuracy', 'neutral' and 'speed' condition, respectively. In our example,

$$E_j = \begin{bmatrix}1\\2\\3\end{bmatrix} \implies I_a = \begin{bmatrix}1\\0\\0\end{bmatrix}, I_n = \begin{bmatrix}0\\1\\0\end{bmatrix}, \text{ and } I_s = \begin{bmatrix}0\\0\\1\end{bmatrix},$$

   so the matched separate threshold parameter is

$$c_{\text{match}} = c_j^{(a)} \begin{bmatrix}1\\0\\0\end{bmatrix} + c_j^{(n)} \begin{bmatrix}0\\1\\0\end{bmatrix} + c_j^{(s)} \begin{bmatrix}0\\0\\1\end{bmatrix} = \begin{bmatrix}c_j^{(a)}\\c_j^{(n)}\\c_j^{(s)}\end{bmatrix} = Mc_j,$$

   with $M = [I_a \quad I_n \quad I_s]$ and $c_j = [c_j^{(n)} \quad c_j^{(n)} \quad c_j^{(n)}]^\top$. Finally, since c is the **same across accumulators**, we just need to duplicate c_{match} to obtain a $n_j \times 2$ matrix.

$$c_{ij} = {}^{7}[c_{\text{match}} \quad c_{\text{match}}] = \begin{bmatrix}c_j^{(a)} & c_j^{(a)}\\c_j^{(n)} & c_j^{(n)}\\c_j^{(s)} & c_j^{(s)}\end{bmatrix}.$$

   The following code does the computation.

```
% ----------- Seperate threshold c -----------------

I_a = (E_j == 1); I_n = (E_j == 2);   I_s = (E_j == 3);
M = [I_a I_n I_s ];
c_ij = repmat(reshape(M*c_j',n_j*R,1),1,2);
```

   (b) Next, we examine the matching rule for parameters that **do not vary according to the experimental conditions nor the accumulators**. In our example, these parameters are the upper bound A_j and the non-decision time τ_j. We duplicate the value to obtain a $n_j \times 2$ matrix.

$$A_{ij} = \begin{bmatrix}A_j & A_j\\ \vdots & \vdots\\ A_j & A_j\end{bmatrix}, \quad \tau_{ij} = \begin{bmatrix}\tau_j & \tau_j\\ \vdots & \vdots\\ \tau_j & \tau_j\end{bmatrix}.$$

[7] Again, the subscript i here indicates the parameters after matching with conditions in trial i.

```
% ----------- Upper bound A -----------------

A_ij = kron(A_j,ones(n_j,2));

% ----------- non-decision time -----------------

tau_ij = kron(tau_j,ones(n_j,2));
```

Figure 10: The Matlab code for matching A_j and τ_j.

(c) The drift rate mean is **the same across experimental conditions but different between accumulators**. This means that $v_j = [v_j^{(1)} \quad v_j^{(2)}]$ and we just need to duplicate the drift rate mean v_j over rows to obtain a $n_j \times 2$ matrix v_{ij}

$$v_{ij} = \begin{bmatrix} v_j^{(1)} & v_j^{(2)} \\ \vdots & \vdots \\ v_j^{(1)} & v_j^{(2)} \end{bmatrix}$$

```
% ----------- drift rate means -----------------

v_ij = kron(v_j,ones(n_j,1));
```

Figure 11: The Matlab code represents the matching rule for the drift rate mean v_j.

(d) Since the drift rate standard deviation is assumed to be 1, s is a $n_j \times 2$ matrix of 1s

$$s = \begin{bmatrix} 1 & 1 \\ \vdots & \vdots \\ 1 & 1 \end{bmatrix}.$$

```
% ----------- drift rate std -----------------

s_ij = ones(n_j*R,2);
```

Figure 12: Matching rule for the standard deviation s.

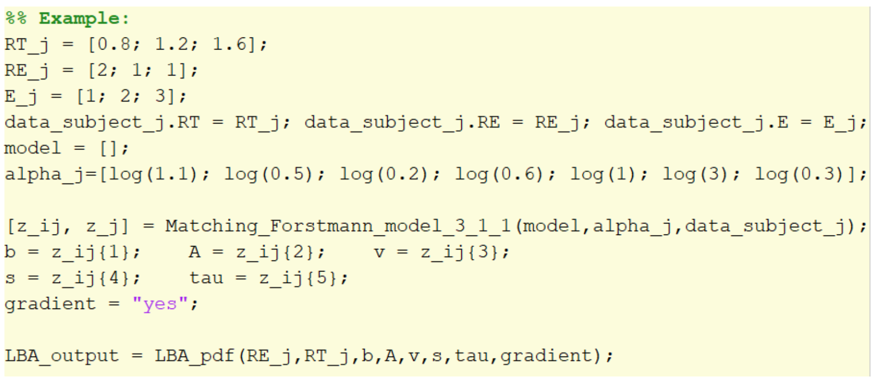
(e) Finally, convert c_{ij} to b_{ij} and store the results $\boldsymbol{z}_{ij}$ and $\boldsymbol{z}_j$ in cell arrays `z_ij` and `z_j`, respectively.

```
% ----------- threshold b -----------------

b_ij = c_ij + A_ij; % threshold b_ij = c_ij + A_ij.
b_j = c_j + A_j; % threshold b_j = c_j + A_j.

% ---------- Store the results in cell arrays ---------

z_ij = cell(5,1);
z_ij{1} = b_ij;   z_ij{2} = A_ij;   z_ij{3} = v_ij;
z_ij{4} = s_ij;   z_ij{5} = tau_ij;

z_j = cell(5,1);
z_j{1} = b_j;   z_j{2} = A_j;   z_j{3} = v_j;
z_j{4} = s_j;   z_j{5} = tau_j;
```

We now consider an example showing how the first matching function and the LBA density function work.

Example:

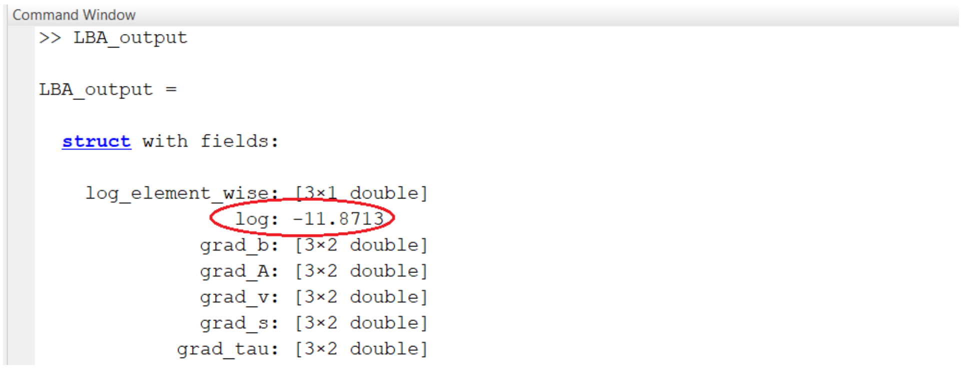
Figure 13 shows the Matlab code to compute $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ for the given data using the first matching function and the LBA density function.



```
%% Example:
RT_j = [0.8; 1.2; 1.6];
RE_j = [2; 1; 1];
E_j = [1; 2; 3];
data_subject_j.RT = RT_j; data_subject_j.RE = RE_j; data_subject_j.E = E_j;
model = [];
alpha_j=[log(1.1); log(0.5); log(0.2); log(0.6); log(1); log(3); log(0.3)];

[z_ij, z_j] = Matching_Forstmann_model_3_1_1(model,alpha_j,data_subject_j);
b = z_ij{1};    A = z_ij{2};    v = z_ij{3};
s = z_ij{4};    tau = z_ij{5};
gradient = "yes";

LBA_output = LBA_pdf(RE_j,RT_j,b,A,v,s,tau,gradient);
```

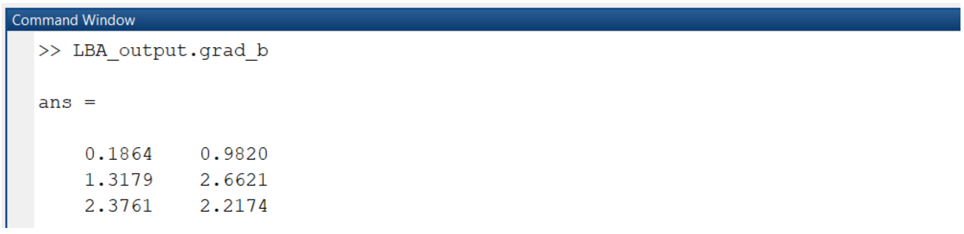
Figure 13: Matlab code to compute $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$.

We obtain $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j) = -11.8713$ from the output `LBA_output.log`.



```
Command Window
>> LBA_output

LBA_output =

  struct with fields:

    log_element_wise: [3×1 double]
                 log: -11.8713
              grad_b: [3×2 double]
              grad_A: [3×2 double]
              grad_v: [3×2 double]
              grad_s: [3×2 double]
            grad_tau: [3×2 double]
```

The partial derivaties with respect to the threshold parameters are stored in `.grad_b`



```
Command Window
>> LBA_output.grad_b

ans =

    0.1864    0.9820
    1.3179    2.6621
    2.3761    2.2174
```

The second matching function

The previous section shows that the function `LBA_pdf.m` returns a set of $n_j \times 2$ matrices of partial derivates, one for each parameter. However, the gradient vector

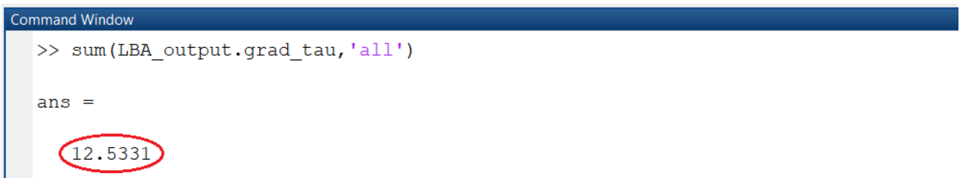
$$\nabla_{\boldsymbol{\alpha}_j} \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j) = \left(\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(c;a)}}, \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(c;n)}}, \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(c;s)}}, \dots, \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(\tau)}} \right),$$

is required with $\alpha_j^{(c;a)}$ denoting $\log c_j^{(a)}$, $\alpha_j^{(c;n)}$ denoting $\log c_j^{(n)}$, etc.

We first show how to obtain
$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(\tau)}}$$
from LBA_output.grad_tau. Notice that in the current model (model 3-1-1), τ **does not depend on the conditions nor the accumulators**, so $\tau_{ij}^{(1)} = \tau_{ij}^{(2)} = \tau_j$, for $i = 1, \dots, n_j$. Therefore, the partial derivative with respect to τ_j is

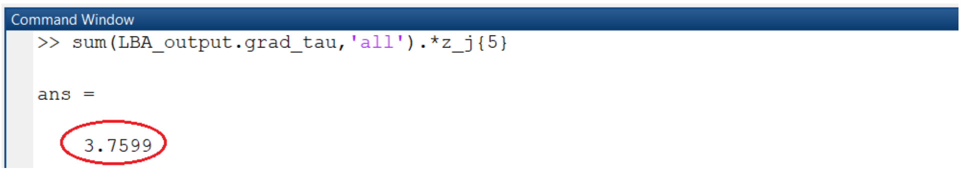
$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \tau_j} = \sum_{k=1}^{2}\sum_{i=1}^{n_j} \frac{\partial}{\partial \tau_{ij}^{(k)}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{z}_{ij}).$$

The derivative is obtained by **summming all elements** of matrix LBA_output.grad_tau:


```
Command Window
>> sum(LBA_output.grad_tau,'all')

ans =

   12.5331
```

Use the chain rule to obtain

$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(\tau)}} = \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \tau_j} \times \frac{\partial \tau_j}{\partial \alpha_j^{(\tau)}} = \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \tau_j} \times e^{\alpha_j^{(\tau)}} = \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \tau_j} \times \tau_j.$$


```
Command Window
>> sum(LBA_output.grad_tau,'all').*z_j{5}

ans =

    3.7599
```

Notice that $\tau_j = e^{\alpha_j^{(\tau)}}$ and the value of τ_j is stored in z_j{5}. Hence,

$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(\tau)}} = 3.7599$$

.

Similarly to the non-decision time τ_j, **the upper bound A_j does not depend on the conditions nor the accumulators**, so the partial derivative with respect to A_j is

$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial A_j} = \sum_{k=1}^{2}\sum_{i=1}^{n_j} \frac{\partial}{\partial A_{ij}^{(k)}} \log \mathrm{LBA}(y_{ij}|\boldsymbol{z}_{ij}).$$

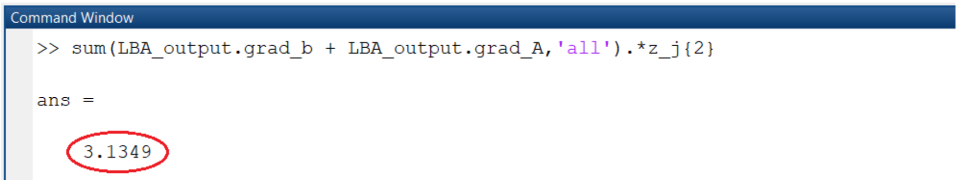
As $\alpha_j^{(A)}$ appears in both A_j and b_j via the following transformations

$$\begin{cases} \alpha_j^{(c)} = \log(b_j - A_j) \\ \alpha_j^{(A)} = \log A_j \end{cases} \Longrightarrow \begin{cases} b_j = e^{\alpha_j^{(c)}} + e^{\alpha_j^{(A)}} \\ A_j = e^{\alpha_j^{(A)}} \end{cases},$$

the partial derivative with respect to $\alpha_j^{(A)}$ is

$$\begin{aligned}\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(A)}} &= \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial b_j} \times \frac{\partial b_j}{\partial \alpha_j^{(A)}} + \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial A_j} \times \frac{\partial A_j}{\partial \alpha_j^{(A)}} \\ &= \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial b_j} \times e^{\alpha_j^{(A)}} + \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial A_j} \times e^{\alpha_j^{(A)}} \\ &= \left(\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial b_j} + \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial A_j}\right) \times A_j\end{aligned}$$

We obtain
$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)}{\partial \alpha_j^{(A)}} = 3.1349$$
by using the command


```
Command Window
>> sum(LBA_output.grad_b + LBA_output.grad_A,'all').*z_j{2}

ans =

    3.1349
```

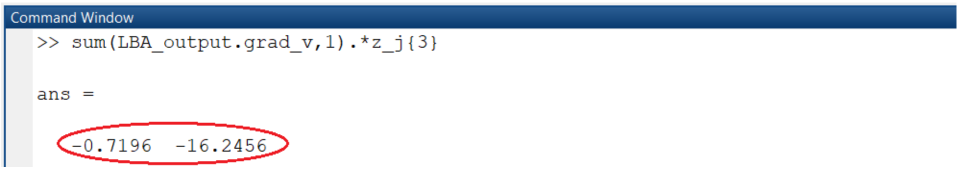
The drift rate means v are common across trials but different between accumulators, so $v_{ij}^{(1)} = v_j^{(1)}$ and $v_{ij}^{(2)} = v_j^{(2)}, i = 1 \dots, n_j$. The partial derivatives with respect to $v_j^{(1)}$ and $v_j^{(2)}$ are

$$\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial v_j^{(1)}} = \sum_{i=1}^{n_j} \frac{\partial}{\partial v_{ij}^{(1)}} \log \text{LBA}(y_{ij}|\boldsymbol{z}_{ij}); \quad \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial v_j^{(2)}} = \sum_{i=1}^{n_j} \frac{\partial}{\partial v_{ij}^{(2)}} \log \text{LBA}(y_{ij}|\boldsymbol{z}_{ij}).$$

Use the chain rule and that $v_j^{(1)} = e^{\alpha_j^{(v;1)}}$, $v_j^{(2)} = e^{\alpha_j^{(v;2)}}$, to obtain

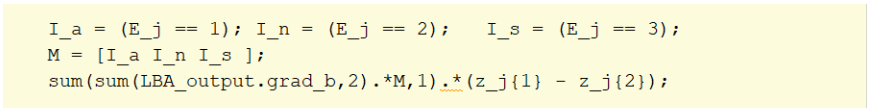
$$\begin{aligned}\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \alpha_j^{(v;1)}} &= \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial v_j^{(1)}} \times \frac{\partial v_j^{(1)}}{\alpha_j^{(v;1)}} = \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial v_j^{(1)}} \times e^{\alpha_j^{(v;1)}}, \\ \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \alpha_j^{(v;2)}} &= \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial v_j^{(2)}} \times \frac{\partial v_j^{(2)}}{\alpha_j^{(v;2)}} = \frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial v_j^{(2)}} \times e^{\alpha_j^{(v;2)}}.\end{aligned}$$

Since the partial derivatives with respect to $v_j^{(1)}$ and $v_j^{(2)}$ are stored in 2 separate columns in `LBA_output.grad_v`, we **sum over rows** to get the derivatives.


```
Command Window
>> sum(LBA_output.grad_v,1).*z_j{3}

ans =

   -0.7196  -16.2456
```

We obtain $\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \alpha_j^{(v;1)}} = -0.7196$ and $\frac{\partial \log p(\boldsymbol{y}_j|\boldsymbol{z}_j)}{\partial \alpha_j^{(v;2)}} = -16.2456.$

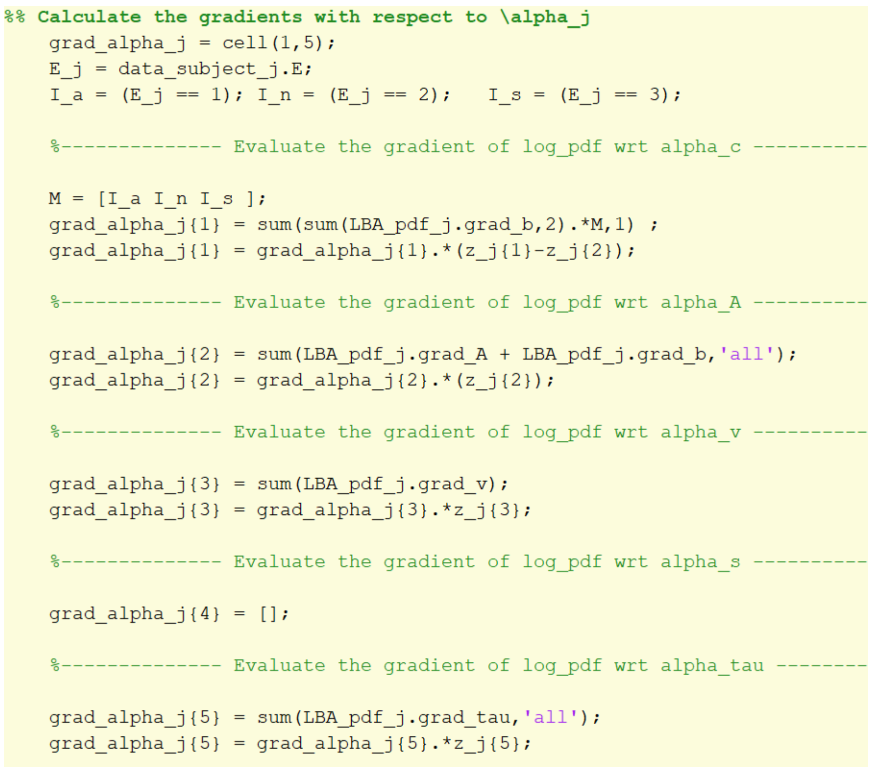
Finally, **the threshold b_j depends on three conditions but does not change across accumulators**, $b_{ij}^{(1)} = b_{ij}^{(2)}, i = 1, \dots, n_j$. As the parameter is common between accumulators, we first sum over columns of matrix `LBA_output.grad_b`. Then, we select the values according to the conditions indicated by E_j. The code below does this.



```
I_a = (E_j == 1); I_n = (E_j == 2);   I_s = (E_j == 3);
M = [I_a I_n I_s ];
sum(sum(LBA_output.grad_b,2).*M,1).*(z_j{1} - z_j{2});
```

Figure 14 shows the code for the second matching function; the code is stored in the script file named `Matching_Gradients_Forstmann_model_3_1_1.m`.

1. **The inputs**
   - `model` is a structure containing the model specification. As explained, this argument does not play any role here (it is important only when we generalize the funcion), but it must be present.
   - `LBA_pdf` cotains all the results from `LBA_pdf.m`
   - `z_j` is $\boldsymbol{z}_j$ obtained from the first matching function `Matching_Forstmann_model_3_1_1.m`
   - `data_subject_j` is a structure containing all the data from subject j.
2. **The outputs** The function gives the gradient vector $\nabla_{\boldsymbol{\alpha}_j} \log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ which is stored in a cell array named `grad_alpha_j`.
   - `grad_alpha_j{1}` contains the partial derivatives of $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ with respect to $\alpha_j^{(c)}$.
   - `grad_alpha_j{2}` contains the partial derivatives of $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ with respect to $\alpha_j^{(A)}$.
   - `grad_alpha_j{3}` contains the partial derivatives of $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ with respect to $\alpha_j^{(v)}$.
   - `grad_alpha_j{4}` contains the partial derivatives of $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ with respect to $\alpha_j^{(s)}$.
   - `grad_alpha_j{5}` contains the partial derivatives of $\log p(\boldsymbol{y}_j|\boldsymbol{\alpha}_j)$ with respect to $\alpha_j^{(\tau)}$.



```
%% Calculate the gradients with respect to \alpha_j
    grad_alpha_j = cell(1,5);
    E_j = data_subject_j.E;
    I_a = (E_j == 1); I_n = (E_j == 2);   I_s = (E_j == 3);

    %------------- Evaluate the gradient of log_pdf wrt alpha_c ----------

    M = [I_a I_n I_s ];
    grad_alpha_j{1} = sum(sum(LBA_pdf_j.grad_b,2).*M,1) ;
    grad_alpha_j{1} = grad_alpha_j{1}.*(z_j{1}-z_j{2});

    %------------- Evaluate the gradient of log_pdf wrt alpha_A ----------

    grad_alpha_j{2} = sum(LBA_pdf_j.grad_A + LBA_pdf_j.grad_b,'all');
    grad_alpha_j{2} = grad_alpha_j{2}.*(z_j{2});

    %------------- Evaluate the gradient of log_pdf wrt alpha_v ----------

    grad_alpha_j{3} = sum(LBA_pdf_j.grad_v);
    grad_alpha_j{3} = grad_alpha_j{3}.*z_j{3};

    %------------- Evaluate the gradient of log_pdf wrt alpha_s ----------

    grad_alpha_j{4} = [];

    %------------- Evaluate the gradient of log_pdf wrt alpha_tau --------

    grad_alpha_j{5} = sum(LBA_pdf_j.grad_tau,'all');
    grad_alpha_j{5} = grad_alpha_j{5}.*z_j{5};
```

Figure 14: Matlab code for the second matching function `Matching_Gradients_Forstmann_model_3_1_1.m`.

2.3 Implementing VB for your hierarchical LBA model

So far, we have explained the important roles played by the core functions. The construction of the two matching functions is demonstrated in detail for a simple case (model 3-1-1), but it should be straightforward to create the matching functions corresponding to a new hierarchical LBA model on your data. There are a few more things you need to do in order to make your functions, the hierarchical LBA models and data compatible with our code.

2.3.1 Updating `VB_main_code.m`

Suppose you have created two matching functions used for your hierarchical LBA model and data (congratulation !), and that your functions are `Your_Matching_Function_1.m` and `Your_Matching_Function_2.m`. Notice that the matching rules might change between experiments but the input and output of these functions must not change. You need to load your data and update your matching functions in the script file `VB_main_code.m` as Figure 15 shows. [8].

[8]Remember to place `Your_data.mat` and your two matching functions in folders `Data` and `Functions`, respectively.

```
%% Step 0: Preparation
    addpath("Functions\")
    addpath("Data\")
    exp_name = "Your_experiment";
    m = 1; % the index of the chosen model
    load("Your_data.mat") % load the data
    matching_function_1 = "Your_Matching_Function_1";
    matching_function_2 = "Your_Matching_Function_2";
```

Figure 15: The adjustments required in the main script file when applied to a new hierarchical LBA model. Note that the variable `exp_name` must be set as `exp_name = "Your_experiment"` to let Matlab know that a new hierarchical LBA model is being implemented.

2.3.2 Model specification

It is important to specify the model when a new hierarchical LBA model is implemented. Recall that all hierarchical LBA models considered here have the following dependence structure:

$$\begin{aligned} p(\boldsymbol{y}, \boldsymbol{\alpha}_{1:J}, \boldsymbol{\mu}_{\boldsymbol{\alpha}}, \boldsymbol{\Sigma}_{\boldsymbol{\alpha}}, a_1, \ldots, a_{D_\alpha}) =& N(\boldsymbol{\mu}_{\boldsymbol{\alpha}} | \boldsymbol{\mu}_{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}, \boldsymbol{\Sigma}_{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}) \mathrm{IW}(\boldsymbol{\Sigma}_{\boldsymbol{\alpha}} | v_\alpha, a_1, \ldots, a_{D_\alpha}) \times \\ & \prod_{d=1}^{D_\alpha} \mathrm{IG}(a_d | \mathcal{A}_d^{-2}, 1) \prod_{j=1}^{J} N(\boldsymbol{\alpha}_j | \boldsymbol{\mu}_{\boldsymbol{\alpha}}, \boldsymbol{\Sigma}_{\boldsymbol{\alpha}}) \prod_{i=1}^{n_j} \mathrm{LBA}(y_{ij} | \boldsymbol{\alpha}_j). \end{aligned}$$

Model variants differ only in the dimension of the parameter space and the constraints on the parameters. Hence, a convenient way to specify a model is to define a structure named `model` with three fields: `dim`, `constraints` and `prior_par`.

- `dim` is the vector indicating the number of random effects allowed for the 5 parameters in the following order $c, A, \boldsymbol{v}, s$ and τ; e.g., in model 3-1-1, since each subject has 7 random effects $\tilde{\boldsymbol{z}}_j = (c_j^{(a)}, c_j^{(n)}, c_j^{(s)}, A_j, v_j^{(1)}, v_j^{(2)}, \tau_j)$, the corresponding dimension index vector is `dim` $= [3\ 1\ 2\ 0\ 1]$.

- `constraints` is a string array of length 5 representing the constraints on each parameter. You can choose your notation freely as long as it is consistent across the core functions. For example, for model 3-1-1, we set `constraints = ["3" "1" "2" "0" "1"]`.

- `prior_par` is a structure capturing all the prior hyperparameters: $\boldsymbol{\mu}_{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}$ (`prior_par.mu`), $\boldsymbol{\Sigma}_{\boldsymbol{\mu}_{\boldsymbol{\alpha}}}$ (`prior_par.cov`), v_α (`prior_par.v_a`) and $\mathcal{A}_d$ (`prior_par.A_d`).

```
%% Step 1: Model specification
if exp_name == "Your_experiment"
    model{m}.dim = [3 1 2 0 1];
    D_alpha = sum(model{m}.dim);
    model{m}.constraints = ["3" "1" "2" "0" "1"];
    model{m}.prior_par.mu = zeros(D_alpha,1);
    model{m}.prior_par.cov = eye(D_alpha);
    model{m}.prior_par.v_a = 2;
    model{m}.prior_par.A_d = ones(D_alpha,1);
else
    Model_specification
end
```

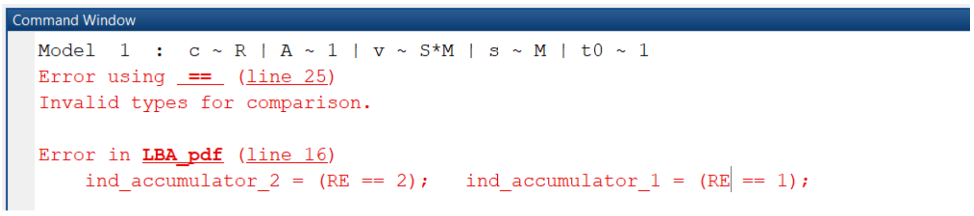
Figure 16: A hierarchical LBA model is described by a structure `model`, with fields `dim`, `constraints` and `prior_par`. The model structure is defined in step 1 in `VB_main_code.m`.

2.3.3 Data Format

To use the code, it is necessary to convert the data into the required format. To use a different data format, you will need to make the necessary changes to the code; we do not recommend this if you do not fully understand how the code works. The data structure must be a cell array named `data`, each subject corresponds to a cell which is a structure having the following compulsory fields.

1. `RT` a column vector containing observed response time.

2. `RE` a column vector containing observed response choices. Notice that all reponse choices must be binary variables taking the values 1 or 2. It means that if the recorded response choice is 'error' and 'correct', it is necessary to convert it to 1('error') and 2('correct') or 1('correct') and 2('error'). The order of the random effects depends on how this binary variable is set. For example, if the drift rate mean v varies across accumulators, the order of the drift rate means would be $(v^{(e)}, v^{(c)})$ if 'error' = 1 and 'correct' = 2, or would be $(v^{(c)}, v^{(e)})$ if 'correct' = 1 and 'error' = 2. You may get this error if the response choices are not converted to the numerical values 1 or 2.



Command Window

```
Model  1  :  c ~ R | A ~ 1 | v ~ S*M | s ~ M | t0 ~ 1
Error using  ==   (line 25)
Invalid types for comparison.

Error in LBA_pdf (line 16)
    ind_accumulator_2 = (RE == 2);   ind_accumulator_1 = (RE == 1);
```

Figure 17: The error is due to the observed responses being non numerical ('old' and 'new'). A simple solution is to convert all responses to the numerical values 1 and 2.

Depending on the experiments, you might also need to create other fields for extra attributes such as stimuli, experimental conditions, etc.

3 Implementing K-fold-CVVB model selection

We first show how to perform K-fold CVVB model selection for the case studies considered in our paper. The crucial part of this section is showing how to generalize the matching functions so that they can be used for a range of competing models, not just a single one. To demonstrate, we show how the matching functions are generalized in the first two case studies.

3.1 The main script file for implementing CVVB

Similarly to the previous section, we create an unique script file named `CVVB_main_code.m` containing all the main parts. To reproduce the CVVB model selection results given in the paper, you need to select the case study in the preparation step (step 0) (similarly to `VB_main_code.m` in section 2.1).

```
%% Step 0: Preparation
    addpath("Functions/") % to load all the required functions
    addpath("Data/") % to load the data
    load("Forstmann.mat")
    exp_name = "Forstmann"; % for model coding.
    matching_function_1 = "Matching_Forstmann";
    matching_function_2 = "Matching_Gradients_Forstmann";
    K = 5; % number of folds in CV
```

While the script is running, the command window (see Figure 18) shows the estimation results.

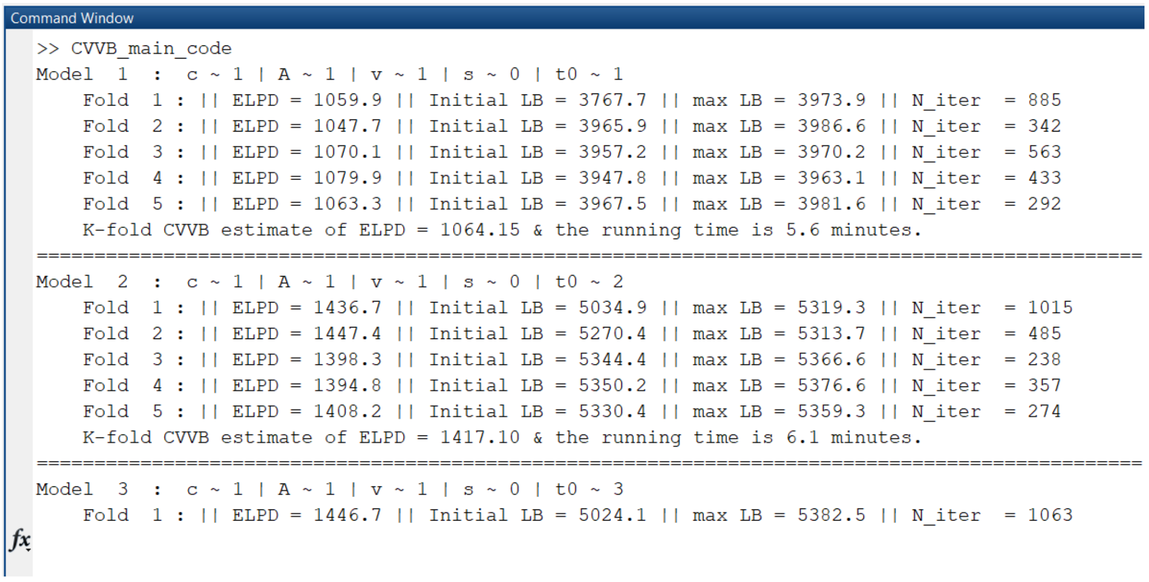
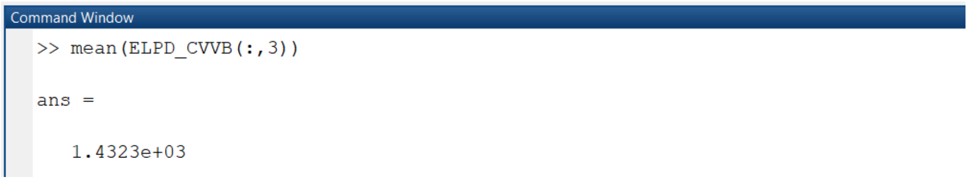

```
>> CVVB_main_code
Model  1  :  c ~ 1 | A ~ 1 | v ~ 1 | s ~ 0 | t0 ~ 1
    Fold  1 : || ELPD = 1059.9 || Initial LB = 3767.7 || max LB = 3973.9 || N_iter  = 885
    Fold  2 : || ELPD = 1047.7 || Initial LB = 3965.9 || max LB = 3986.6 || N_iter  = 342
    Fold  3 : || ELPD = 1070.1 || Initial LB = 3957.2 || max LB = 3970.2 || N_iter  = 563
    Fold  4 : || ELPD = 1079.9 || Initial LB = 3947.8 || max LB = 3963.1 || N_iter  = 433
    Fold  5 : || ELPD = 1063.3 || Initial LB = 3967.5 || max LB = 3981.6 || N_iter  = 292
    K-fold CVVB estimate of ELPD = 1064.15 & the running time is 5.6 minutes.
=================================================================================================
Model  2  :  c ~ 1 | A ~ 1 | v ~ 1 | s ~ 0 | t0 ~ 2
    Fold  1 : || ELPD = 1436.7 || Initial LB = 5034.9 || max LB = 5319.3 || N_iter  = 1015
    Fold  2 : || ELPD = 1447.4 || Initial LB = 5270.4 || max LB = 5313.7 || N_iter  = 485
    Fold  3 : || ELPD = 1398.3 || Initial LB = 5344.4 || max LB = 5366.6 || N_iter  = 238
    Fold  4 : || ELPD = 1394.8 || Initial LB = 5350.2 || max LB = 5376.6 || N_iter  = 357
    Fold  5 : || ELPD = 1408.2 || Initial LB = 5330.4 || max LB = 5359.3 || N_iter  = 274
    K-fold CVVB estimate of ELPD = 1417.10 & the running time is 6.1 minutes.
=================================================================================================
Model  3  :  c ~ 1 | A ~ 1 | v ~ 1 | s ~ 0 | t0 ~ 3
    Fold  1 : || ELPD = 1446.7 || Initial LB = 5024.1 || max LB = 5382.5 || N_iter  = 1063
fx
```

Figure 18: The results shown in the command window while `CVVB_main_code.m` is running. Models with indexes 1 and 2 are completed with the K-fold CVVB estimates 1,064.15 and 1,417.10, respectively. Matlab is running CVVB for model 3 with fold 2 left out.

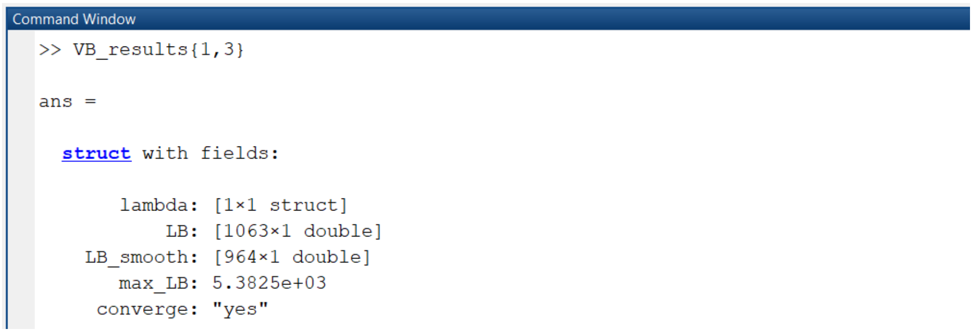
The K-fold CVVB estimate of ELPD

Models are ranked based on the predictive performance which is commonly measured by a quantity called expected log predictive density or ELPD (Dao et al., 2021, page 9). The ELPD estimates are stored in a a matrix named `ELPD_CVVB`. The ELPD estimate for a model with index `m` and with the fold `k` left out is stored in row `k` and column `m`. To get $\widehat{\text{ELPD}}_{\text{K-CVVB}}$ for model `m`, we take the average of the elements in column `m` of `ELPD_CVVB`. For instance, $\widehat{\text{ELPD}}_{\text{K-CVVB}}$ for model `m = 3` is


```
>> mean(ELPD_CVVB(:,3))

ans =

   1.4323e+03
```

VB results

The VB results are stored in a cell array named `VB_results`. Cell `VB_results{k,m}` contains the results for the model having index `m` on the data when fold `k` is left out. If you want to examine the VB results for model 3 when fold 1 is left out, type `VB_results{1,3}` to get



```
>> VB_results{1,3}

ans =

  struct with fields:

       lambda: [1×1 struct]
           LB: [1063×1 double]
    LB_smooth: [964×1 double]
       max_LB: 5.3825e+03
     converge: "yes"
```

From this structure, you can plot the lower bound estimates or get the optimal λ (refer to Section 2.1).

3.2 Generalization - Case Study 1 (Forstmann)

We explained how to construct two matching functions for a specific model (model 3-1-1). The next goal is to extend the functions so that they can be used not only for this simple model but for all 27 of its variants.

3.2.1 Model specification

As mentioned in section 2.3, a model is defined by a structure with fields: `dim`, `constraints` and `prior_par`. To capture all the competing models, we extend a structure to a cell array where each cell is a structure capturing information for a candidate model. The specification of all the competing models is stored in a cell array named `model`. Details on model specification for this case study and other case studies are in the script file named `Model_specification.m`.

Model index m	Model $(c - \boldsymbol{v} - \tau)$	Model index m	Model $(c - \boldsymbol{v} - \tau)$	Model index m	Model $(c - \boldsymbol{v} - \tau)$
1	1-1-1	10	2-3-3	19	3-1-1
2	1-1-2	11	2-3-2	20	3-1-2
3	1-1-3	12	2-3-1	21	3-1-3
4	1-2-3	13	2-2-1	22	3-2-3
5	1-2-2	14	2-2-2	23	3-2-2
6	1-2-1	15	2-2-3	24	3-2-1
7	1-3-1	16	2-1-3	25	3-3-1
8	1-3-2	17	2-1-2	26	3-3-2
9	1-3-3	18	2-1-1	27	3-3-3

Table 1: The 27 hierarchical LBA models considered in the first case study.

3.2.2 The matching functions

We examine the generalized matching rules for this case study, and suggest looking at the script files `Matching_Forstmann.m` and
`Matching_Gradients_Forstmann.m` while reading this section.

1. **Transform $\boldsymbol{\alpha}_j$ to $\boldsymbol{z}_j$:** Notice that the length of $\boldsymbol{\alpha}_j$ varies according to the models and is unknown in advance, so the components of $\boldsymbol{\alpha}_j$ must be specified based on the number of random effects allowed for each component given by `model.dim`. For instance, if the current model has `dim`= [3, 1, 4, 0, 1], it means that the first three components of $\boldsymbol{\alpha}_j$ belong to the threshold $\alpha_j^{(c)}$. The fourth component of $\boldsymbol{\alpha}_j$ is $\alpha_j^{(A)}$. The drift rate mean $\alpha_j^{(v)}$ has 4 components, starting from the fifth component of $\boldsymbol{\alpha}_j$, etc.

```
% ---------- Trasform random effects to the natural form ---------
c_j = exp(alpha_j(1:model.dim(1),:))';
A_j = exp(alpha_j(model.dim(1)+1:sum(model.dim(1:2)),:))';
v_j = exp(alpha_j(sum(model.dim(1:2))+1:sum(model.dim(1:3)),:))';
s_j = [exp(alpha_j(sum(model.dim(1:3))+1:sum(model.dim(1:4)),:))' ones(R,1)];
tau_j = exp(alpha_j(sum(model.dim(1:4))+1:sum(model.dim),:))';
```

2. **Random effects that are common between accumulators but may vary across the experimental conditions:** c_j (`z_j{1}`) and τ_j (`z_j{5}`) are two parameters of this type. We first consider the separate threshold c_j. Based on the separate threshold c_j, all the 27 competing models can be classified into three possible categories:

   (a) c_j is allowed to be different between 3 conditions (`model.constraints(1) == "3"`).

   (b) c_j is common in 'accuracy' and 'neutral' conditions but different under the 'speed' condition (`model.constraints(1) == "2"`).

   (c) c_j is the same across 3 conditions (`model.constraints(1) == "1"`).

   Similarly to the idea presented in section 2.2.2, the selection of the correct separate threshold parameter c_j can be done by a matrix operation. Using the same notation introduced in Section 2.2.2, we have $M = [I_a \quad I_n \quad I_s]$ when `model.constraints(1) == "3"` and $M = [(I_a + I_n) \quad I_s]$ for `model.constraints(1) == "2"` (Simply combine the 'accuracy' and 'neutral' conditions). The Matlab code below does this.
   .

```
% ----------- Separate threshold c -----------------

if (model.constraints(1)== "3")
    M = [I_a I_n I_s ];
    c_ij = repmat(reshape(M*c_j',n_j*R,1),1,2);
elseif (model.constraints(1)== "2")
    M = [(I_a + I_n) I_s ];
    c_ij = repmat(reshape(M*c_j',n_j*R,1),1,2);
else
    c_ij = kron(c_j,ones(n_j,2));
end
```

   The model assumptions on the non-decision time τ_j are the same as in the code above; we only need to replace `c_j`, `c_ij` and `model.constraints(1)` with `tau_j`, `tau_ij` and `model.constraints(5)`, respectively.

3. **For random effects that depend on both the experimental conditions and the accumulators:** The drift rate mean v_j (`z_j{3}`) is of this type. We examine each possible case.

- If v_j changes according to 3 conditions (`model.constraints(3) == 3`), then

  $$v_j = (v_j^{(a;1)}, v_j^{(a;2)}, v_j^{(n;1)}, v_j^{(n;2)}, v_j^{(s;1)}, v_j^{(s;2)})$$

  The selected pairs of drift rate means are

  $$v_{ij} = [v_j^{(a;1)} \quad v_j^{(a;2)}] \otimes I_a + [v_j^{(n;1)} \quad v_j^{(n;2)}] \otimes I_n + [v_j^{(s;1)} \quad v_j^{(s;2)}] \otimes I_s.$$

- If v_j varies across 'speed 'and other conditions (`model.constraints(3) == 2`), then

  $$v_j = (v_j^{(a+n;1)}, v_j^{(a+n;2)}, v_j^{(s;1)}, v_j^{(s;2)})$$

  The selected pairs of drift rate means are

  $$v_{ij} = [v_j^{(a+n;1)} \quad v_j^{(a+n;2)}] \otimes (I_a + I_n) + [v_j^{(s;1)} \quad v_j^{(s;2)}] \otimes I_s.$$

- The third case occurs when v_j is common across all conditions

  $$v_{ij} = v_j \otimes \mathbf{1}_{n_j \times 2},$$

  where $\mathbf{1}_{n_j \times 2}$ is an $n_j \times 2$ matrix of ones.

```
% ----------- drift rate means -----------------

if (model.constraints(3)== "3")
    v_ij = kron(v_j(:,1:2),I_a) + kron(v_j(:,3:4),I_n) + ...
        kron(v_j(:,5:6),I_s);
elseif (model.constraints(3)== "2")
    v_ij = kron(v_j(:,1:2),(I_a + I_n)) + kron(v_j(:,3:4),I_s);
else
    v_ij = kron(v_j,ones(n_j,1));
end
```

Similarly to the first matching function, the second generalized matching function is constructed as in `Matching_Gradients_Forstmann.m` .

3.3 Generalization - Case study 2 (Mnemonic)[9]

We now examine the second case study which compares 16 hierarchical LBA models. First, we explain the model specification. Let E represent the emphasis conditions (accuracy (a) or speed (s)), S denotes the stimulus (new (n) or old (o)) and R stands for the accumulator factor (new (n) or old (o)). The parameter M takes one value for accumulators whose response matches with the stimulus and a different value for accumulators whose response fails to match the stimulus. Denote m for matched and mm for mismatched.

[9]Readers might skip this section at a first reading.

3.3.1 Model specification

Consider the least constrained LBA model, where speed vs. accuracy emphasis (E) is allowed to affect all parameters except the standard deviation of drift rate s. The parameter c is allowed to vary with the response accumulator (R), and the mean drift rate (v) is allowed to vary with the stimulus (S). We denote the least constrained LBA model as

$$c \sim E * R \quad \& \quad A \sim E \quad \& \quad v \sim E * S * M \quad \& \quad s \sim M \quad \& \quad \tau \sim E$$

In the hierarchical version of this model, each subject has **17 random effects** (we set the standard deviation of the drift rate of the matched accumulator $s^{(m)}$ to 1 for model identifiability):

$$c^{(s,n)}, c^{(s,o)}, c^{(a,n)}, c^{(a,o)}, A^{(s)}, A^{(a)}, v^{(s,n,m)}, v^{(s,n,mm)}, v^{(s,o,m)}, v^{(s,o,mm)},$$
$$v^{(a,n,m)}, v^{(a,n,mm)}, v^{(a,o,m)}, v^{(a,o,mm)}, s^{(mm)}, \tau^{(s)}, \tau^{(a)}.$$

To simplify notation, and without lack of clarity, s represents both the drift rate standard deviation and the speed emphasis to simplify notation. Table 2 gives all the competing models.

Model index	Model specification				
`m`	c	A	v	s	τ
Model 1	R	1	$S * M$	M	1
Model 2	R	1	$S * M$	M	E
Model 3	R	1	$E * S * M$	M	1
Model 4	R	1	$E * S * M$	M	E
Model 5	R	E	$S * M$	M	1
Model 6	R	E	$S * M$	M	E
Model 7	R	E	$E * S * M$	M	1
Model 8	R	E	$E * S * M$	M	E
Model 9	$E * R$	1	$S * M$	M	1
Model 10	$E * R$	1	$S * M$	M	E
Model 11	$E * R$	1	$E * S * M$	M	1
Model 12	$E * R$	1	$E * S * M$	M	E
Model 13	$E * R$	E	$S * M$	M	1
Model 14	$E * R$	E	$S * M$	M	E
Model 15	$E * R$	E	$E * S * M$	M	1
Model 16	$E * R$	E	$E * S * M$	M	E

Table 2: The 16 competing hierarchical LBA models considered in the second case study. 1 means the corresponding parameter is the same across all factors.

3.3.2 The matching functions

To understand the matching rules, let us consider the following trails:

- **Trial 1**: S = new, E = speed, C = true (or M = matched), R = new and RT = t. The density function is

$$\text{LBA}(RE = \text{new}, RT = t | c^{(s,n)}, c^{(s,o)}, A^{(s)}, v^{(s,n,m)}, v^{(s,n,mm)}, 1, s^{(mm)}, \tau^{(s)}) = f(t)(1-F(t)),$$

with

$$f(t|c^{(s,n)}, A^{(s)}, v^{(s,n,m)}, 1, \tau^{(s)}) \text{ and } F(t|c^{(s,o)}, A^{(s)}, v^{(s,n,mm)}, s^{(mm)}, \tau^{(s)})$$

- **Trial 2**: S = old, E = speed, C = true (or M = matched), R = old and RT = t. The density function is

  $$\text{LBA}(RE = \text{old}, RT = t|c^{(s,n)}, c^{(s,o)}, A^{(s)}, v^{(s,o,m)}, v^{(s,o,mm)}, 1, s^{(mm)}, \tau^{(s)}) = f(t)(1-F(t)),$$

  with

  $$f(t|c^{(s,o)}, A^{(s)}, v^{(s,o,m)}, 1, \tau^{(s)}) \text{ and } F(t|c^{(s,n)}, A^{(s)}, v^{(s,o,mm)}, s^{(mm)}, \tau^{(s)})$$

- **Trial 3**: S = new, E = accuracy, C = False (or M = mismatched), R = old and RT = t. The density function is

  $$\text{LBA}(RE = \text{old}, RT = t|c^{(a,n)}, c^{(a,o)}, A^{(a)}, v^{(a,n,m)}, v^{(a,n,mm)}, 1, s^{(mm)}, \tau^{(a)}) = f(t)(1-F(t))$$

  with

  $$f(t|c^{(a,o)}, A^{(a)}, v^{(a,n,mm)}, s^{(mm)}, \tau^{(a)}) \text{ and } F(t|c^{(a,n)}, A^{(a)}, v^{(a,n,m)}, 1, \tau^{(a)})$$

- **Trial 4**: S = old, E = accuracy, C = False (or M = mismatched), R = new and RT = t. The density function is

  $$\text{LBA}(RE = \text{new}, RT = t|c^{(a,n)}, c^{(a,o)}, A^{(a)}, v^{(a,o,m)}, v^{(a,o,mm)}, 1, s^{(mm)}, \tau^{(a)}) = f(t)(1-F(t))$$

  with

  $$f(t|c^{(a,n)}, A^{(a)}, v^{(a,o,mm)}, s^{(mm)}, \tau^{(a)}) \text{ and } F(t|c^{(a,o)}, A^{(a)}, v^{(a,o,m)}, 1, \tau^{(a)})$$

The first generalized matching function is called `Matching_Mnemonic.m`. It looks very much like `Matching_Forstmann.m`, so most parts of the code are familiar. Therefore, we focus on explaining the parts that are new or different.

1. **Random effects that are common between accumulators but may vary across the experimental conditions:** A_j (`z_j{2}`) and τ_j (`z_j{5}`) are two parameters of this type. We show how to match the upper bound A_j; the non-decision τ_j is done similarly. All competing models can be classified into three possible categories:

   (a) If A_j changes between the 'speed' and 'accuracy' conditions (`model.constraints(1) == "E"`), then $A_j = (A_j^{(s)}, A_j^{(a)})$. The selected pairs of drift rate means are

   $$A_{ij} = A_j^{(s)} \otimes I_s + A_j^{(a)} \otimes I_a.$$

   (b) A_j does not vary according to experimental conditions (`model.constraints(1) == "1"`). Then,

   $$A_{ij} = A_j \otimes \mathbf{1}_{n_j \times 2}.$$

```
    % ----------- The start point parameter A -----------------

if (model.constraints(2) == "E")
    A_ij = repmat(kron(A_j(:,1),I_s) + kron(A_j(:,2),I_a),1,2);
elseif (model.constraints(2) == "1")
    A_ij = kron(A_j,ones(n_j,2));
end
```

The matching rule for the non-decision time τ_j is similar as shown below.

```
% ----------- The non-decision time parameter tau -----------------
if (model.constraints(5) == "E")
    tau_ij = repmat(kron(tau_j(:,1),I_s) + kron(tau_j(:,2),I_a),1,2);
elseif (model.constraints(5) == "1")
    tau_ij = kron(tau_j,ones(n_j,2));
end
```

2. **Random effects that are different between accumulators and might vary across the experiment conditions:** The separate threshold c_j (z_j{1}) is of this type. We examine each possible case.

   - If c_j changes between the 'speed' and 'accuracy' conditions (`model.constraints(1) == "E*R"`), then
$$c_j = (c_j^{(s;o)}, c_j^{(s;n)}, c_j^{(a;o)}, c_j^{(a;n)})$$
     The selected pairs of drift rate means are
$$c_{ij} = [c_j^{(s;o)} \quad c_j^{(s;n)}] \otimes I_s + [c_j^{(a;o)} \quad c_j^{(a;n)}] \otimes I_a.$$

   - If the c_j only vary across accumulators but not the experimental conditions (`model.constraints(1) == "R"`), then
$$c_j = (c_j^{(o)}, c_j^{(n)}).$$
     The selected pairs of drift rate means are
$$c_{ij} = [c_j^{(o)} \quad c_j^{(n)}] \otimes \mathbf{1}_{n_j \times 2},$$
     where $\mathbf{1}_{n_j\times 2}$ is an $n_j \times 2$ matrix of ones.

```
    % ----------- The threshold parameter c -----------------

if (model.constraints(1) == "E*R") %c_j=c^(s,o),c^(s,n),c^(a,o),c^(a,n)
    c_ij = kron(c_j(:,1:2),I_s) + kron(c_j(:,3:4),I_a);
elseif (model.constraints(1) == "R") % c_j =  c^(o), c^(n)
    c_ij = kron(c_j,ones(n_j,1));
end
```

3. **Random effects that depend on stimuli and response (matched vs mismatched):** The standard deviation s_j (z_j{4}) is of this type. $s_j = (s_j^{(m)}, s_j^{(mm)}) = (1, s_j^{(mm)})$ (recall that $s_j^{(m)} = 1$ for model identification). Notice that the 'match' and 'mismatch' are determined based on comparing the stimuli S and the response R. For example

- The ‘match’ standard deviation when the responses match with the stimuli, that is $S = R =$‘old’ or $S = R =$ ‘new’.
- The ‘mismatch’ standard deviation when the responses do not match with the stimuli, that is $S =$‘old ’and $R =$ ‘new ’or vice versa.

```
% ----------- The drift rate std s -----------------

s_ij_old = kron(s_j, I_old) + repmat(I_new,R,1);
s_ij_new = kron(s_j, I_new) + repmat(I_old,R,1);
s_ij = [s_ij_old s_ij_new];
```

4. **Random effects that are different between accumulators and might vary across the experimental conditions:** The separate threshold v_j (`z_j{3}`) is of this type. We examine each possible case.

   - If v_j does not change with the ‘speed ’ and ‘accuracy’ conditions (`model.constraints(3) == "S*M"`), then

   $$v_j = (v_j^{(o;m)}, v_j^{(o;mm)}, v_j^{(n;m)}, v_j^{(n;mm)}).$$

   The selected pairs of drift rate means are

   $$v^{(o)} = v_j^{(o;m)} \otimes I_{\text{old}} + v_j^{(n;mm)} \otimes I_{\text{new}};$$
   $$v^{(n)} = v_j^{(o;mm)} \otimes I_{\text{old}} + v_j^{(n;m)} \otimes I_{\text{new}}.$$
   $$v_{ij} = [v^{(o)} \quad v^{(n)}].$$

   - If v_j changes between the ‘ speed ’and ‘accuracy’ conditions (`model.constraints(3) == "E*S*M"`), then

   $$v_j = \begin{bmatrix} v_j^{(s;o;m)}, & v_j^{(s;o;mm)}, & v_j^{(s;n;m)}, & v_j^{(s;n;mm)}, \\ v_j^{(a;o;m)}, & v_j^{(a;o;mm)}, & v_j^{(a;n;m)}, & v_j^{(a;n;mm)} \end{bmatrix}.$$

   The selected pairs of drift rate means are

$$v^{(o)} = v_j^{(s;o;m)} \otimes I_{\text{speed;old}} + v_j^{(s;n;mm)} \otimes I_{\text{speed;new}} + v_j^{(a;o;m)} \otimes I_{\text{acc.;old}} + v_j^{(a;n;mm)} \otimes I_{\text{acc.;new}};$$
$$v^{(n)} = v_j^{(s;o;mm)} \otimes I_{\text{speed;old}} + v_j^{(s;n;m)} \otimes I_{\text{speed;new}} + v_j^{(a;o;mm)} \otimes I_{\text{acc.;old}} + v_j^{(a;n;m)} \otimes I_{\text{acc.;new}}.$$
$$v_{ij} = [v^{(o)} \quad v^{(n)}].$$

```
% ----------- The drift rate mean v -----------------

if (model.constraints(3) == "S*M") %v=v^(o,m),v^(o,mm),v^(n,m),v^(n,mm)
    v_ij_old = kron(v_j(:,1),I_old) + kron(v_j(:,4),I_new);
    v_ij_new = kron(v_j(:,2),I_old) + kron(v_j(:,3),I_new);

elseif (model.constraints(3) == "E*S*M")
%       v = [v^(s,o,m), v^(s,o,mm), v^(s,n,m), v^(s,n,mm),...
%            v^(a,o,m), v^(a,n,mm), v^(a,o,m), v^(a,o,mm)]
    I_so = I_s.*I_old; I_sn = I_s.*I_new;
    I_ao = I_a.*I_old; I_an = I_a.*I_new;

    v_ij_old = kron(v_j(:,1),I_so) + kron(v_j(:,5),I_ao) +...
        kron(v_j(:,4),I_sn) + kron(v_j(:,8),I_an);
    v_ij_new = kron(v_j(:,2),I_so) + kron(v_j(:,6),I_ao) +...
        kron(v_j(:,3),I_sn) + kron(v_j(:,7),I_an);
end
v_ij = [v_ij_old v_ij_new];
```

The second generalized matching function can be constructed similarly to the first matching function; see `Matching_Gradients_Mnemonic.m` for the details.

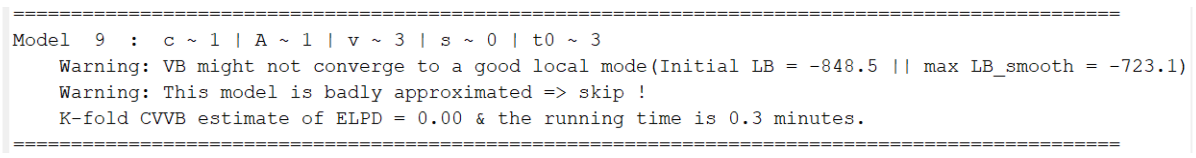
3.4 Practical recommendations

1. A well-known issue with fixed-form VB is that often there are local maxima. Converging to a poor local maximum often leads to a poor approximation. This problem often occurs when the initial value is poor. We reduce the possibility of getting to a poor local mode by carefully considering the choice of initial value.

    (a) We suggest running MCMC for a small numer of iterations (100 to 200 iterations), throwing away the first half, taking the average of the other half and then using it as the initial value.

    (b) In the CVVB procedure, we can also use the optimal $\boldsymbol{\lambda}$ obtained from the first run (fold 1 left out) as the initial value for other runs (other folds are left out).

2. It is not unusual that a small proportion of models are poorly approximated. Our automatic CVVB procedure issues a warning message when the model output is suspicious. As the computational cost is cheap, we recommend that you recheck the results by rerunning the CVVB for these models .



```
===========================================================================================
Model  9  :  c ~ 1 | A ~ 1 | v ~ 3 | s ~ 0 | t0 ~ 3
    Warning: VB might not converge to a good local mode(Initial LB = -848.5 || max LB_smooth = -723.1)
    Warning: This model is badly approximated => skip !
    K-fold CVVB estimate of ELPD = 0.00 & the running time is 0.3 minutes.
===========================================================================================
```

Figure 19: A warning message will be given for suspicious model output.

3. Comparing model ranks based on $\widehat{\text{ELPD}}_{\text{K-CVVB}}$ between independent runs is also helpful and can increase confidence about the quality of the classification.

References

Viet-Hung Dao, David Gunawan, Minh-Ngoc Tran, Robert Kohn, Guy E Hawkins, and Scott D Brown. Efficient selection between hierarchical cognitive models: Cross-validation with variational Bayes. *arXiv preprint arXiv:2102.06814*, 2021.