Apollo: a flexible, powerful and customisable freeware package for choice model estimation and application

version 0.2.5

User manual

www.ApolloChoiceModelling.com

Stephane Hess & David Palma
Choice Modelling Centre
University of Leeds

Release date: 31 July 2021
Manual edited: 12 November 2021

Apollo is licensed under GNU GENERAL PUBLIC LICENSE v2 (GPL-2)
https://cran.r-project.org/web/licenses/GPL-2.

Apollo is provided free of charge and comes WITHOUT ANY WARRANTY of any kind. In no event will the authors or their employers be liable to any party for any damages resulting from any use of Apollo.

This manual is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License
https://creativecommons.org/licenses/by-nd/4.0/.

While the Apollo package is the result of many years of development, the core of this work was carried out under the umbrella of the European Research Council (ERC) funded consolidator grant 615596-DECISIONS. We are grateful to the many colleagues who provided suggestions and/or tested the code extensively, including Chiara Calastri, Romain Crasted dit Sourd, Andrew Daly, Jeff Dumont, Joe Molloy and Basil Schmid. We would like to especially thank Thijs Dekker for his contributions to precursors of Apollo and his guidance on the EM algorithm, Thomas Hancock for his implementation of Decision Field Theory and Annesha Enam for her contributions on MDCEV without an outside good. We are also grateful to Kay Axhausen for making the Swiss public transport route choice dataset available.

Contents

1 Introduction **8**

2 Installing *Apollo*, loading the libraries and running the code **13**

3 Data format and datasets used for examples **15**
3.1 RP-SP mode choice dataset: `apollo_modeChoiceData` . . . 16
3.2 SP route choice dataset: `apollo_swissRouteChoiceData` . . . 16
3.3 Health attitudes SP: `apollo_drugChoiceData` . . . 16
3.4 Time use data: `apollo_timeUseData` . . . 16

4 General code structure and components: illustration for MNL **18**
4.1 Initialising the code . . . 18
4.2 Reading and processing the data . . . 21
4.3 Model parameters . . . 21
4.4 Validation and preparing user inputs . . . 24
4.5 Likelihood component: the *apollo_probabilities* function . . . 24
4.5.1 Initialisation . . . 25
4.5.2 Model definition . . . 25
4.5.3 Function output . . . 27
4.6 Estimation . . . 28
4.7 Reporting and saving results . . . 32

5 Other model components **39**
5.1 Other RUM-consistent discrete choice models . . . 39
5.1.1 Nested Logit . . . 39
5.1.2 Cross-nested Logit . . . 42
5.2 Non-RUM decision rules for discrete choice . . . 45
5.2.1 Random regret minimisation (RRM) . . . 45
5.2.2 Decision field theory (DFT) . . . 47
5.3 Models for ranking, rating and continuous dependent variables . . . 50
5.3.1 Exploded Logit . . . 50
5.3.2 Ordered Logit and Ordered Probit . . . 53
5.3.3 Normally distributed continuous variables . . . 56

5.4 Discrete-continuous models . . . 57
5.4.1 Multiple Discrete Continuous Extreme Value (MDCEV) model . . . 57
5.4.2 Multiple Discrete Continuous Nested Extreme Value (MDCNEV) model . . . 61
5.5 Adding new model types . . . 63

6 Incorporating random heterogeneity 65
6.1 Continuous random coefficients . . . 65
6.1.1 Introduction . . . 65
6.1.2 Example model specification . . . 68
6.1.3 Implementation . . . 70
6.1.4 Estimation . . . 74
6.1.5 Error components . . . 74
6.2 Discrete mixtures and Latent Class . . . 76
6.3 Combining Latent Class with continuous random heterogeneity . . . 81
6.4 Multi-threading capabilities . . . 82

7 Joint estimation of multiple model components 86
7.1 Joint estimation on RP and SP data . . . 87
7.2 Joint best-worst model . . . 89
7.3 Hybrid choice model . . . 90

8 Bayesian estimation 96

9 Pre and post-estimation capabilities 102
9.1 Pre-estimation analysis of choices . . . 102
9.2 Reading in a previously saved `model` object . . . 103
9.3 Calculating model fit for given parameter values . . . 104
9.4 Likelihood ratio tests against other models . . . 105
9.5 Ben-Akiva & Swait test . . . 106
9.6 Model predictions . . . 106
9.7 Market share recovery for subgroups of data . . . 110
9.8 Comparison of model fit across subgroups of data . . . 112
9.9 Functions of model parameters and associated standard errors . . . 112
9.10 Unconditionals for random parameters . . . 114
9.10.1 Continuous random heterogeneity . . . 114
9.10.2 Latent class . . . 115
9.11 Conditionals for random coefficients . . . 115
9.11.1 Continuous random coefficients . . . 116
9.11.2 Latent class . . . 118
9.12 Summary of results for multiple models . . . 120

10 Debugging 122

11 Extensions **126**
11.1 Starting value search . . . 126
11.2 Out of sample fit (Cross validation) . . . 127
11.3 Bootstrap estimation . . . 129
11.4 Expectation-maximisation (EM) algorithm . . . 131
11.4.1 EM algorithm for LC model . . . 132
11.4.2 MMNL model with full covariance matrix for random coefficients . . . 134
11.5 Iterative coding of utilities for large choice sets . . . 138

12 Frequently asked questions **141**
12.1 General . . . 141
12.2 Installation and updating of *Apollo* . . . 141
12.3 Data . . . 142
12.4 Model specification . . . 143
12.5 Errors and failures during estimation . . . 145
12.6 Model results . . . 147

A *Apollo* versions: timeline, changes and backwards compatibility **150**

B Data dictionaries **174**

C Index of example files **179**

D Overview of functions, lists and elements **183**

E Detailed description of `model` object **204**

Bibliography **214**

Index: *Apollo* syntax **215**

List of Figures

4.1 General structure of an *Apollo* model file . . . 19
4.2 Code initialisation . . . 20
4.3 Loading data, selecting a subset and creating an additional variable . . . 21
4.4 Setting names and starting values for model parameters, and fixing some parameters to their starting values . . . 23
4.5 Using `apollo_readBeta` to load results from an earlier model as starting values . . . 24
4.6 Running `apollo_validateInputs` . . . 24
4.7 The `apollo_probabilities` function: example for MNL model . . . 36
4.8 Running `apollo_estimate` on MNL model . . . 37
4.9 On screen output obtained using `apollo_modelOutput` for MNL model . . . 38

5.1 Nested Logit implementation (extract) . . . 41
5.2 Nested Logit tree structure after estimation . . . 42
5.3 Cross-nested Logit implementation (extract) . . . 44
5.4 Cross-nested Logit structure after estimation . . . 45
5.5 Implementation of random regret MNL model . . . 46
5.6 An example of a decision-maker stopping upon reaching either an internal or external threshold . . . 47
5.7 DFT implementation . . . 51
5.8 Exploded Logit implementation . . . 54
5.9 MDCEV implementation without outside good . . . 60
5.10 MDCEV implementation with an outside good . . . 62
5.11 MDCNEV implementation and call to apollo_estimate using scaling . . . 63

6.1 Defining settings for generation of draws . . . 71
6.2 The `apollo_randCoeff` function . . . 72
6.3 The `apollo_probabilities` function for a MMNL model . . . 73
6.4 Running `apollo_estimate` for MMNL using 3 cores . . . 75
6.5 Using error components for heteroskedasticity . . . 75
6.6 Using error components for a pseudo-panel effect . . . 76
6.7 The `apollo_lcPars` function . . . 78
6.8 Implementing choice probabilities for Latent Class . . . 79

6.9 The `apollo_randCoeff` and `apollo_lcPars` functions for a Latent Class model with continuous random heterogeneity . . . 82
6.10 Implementing choice probabilities for Latent Class with continuous random heterogeneity . . . 83
6.11 Running `apollo_speedTest` . . . 85

7.1 Joint RP-SP model on mode choice data . . . 88
7.2 On screen output for RP-SP model . . . 90
7.3 Best-Worst model on drug choice data . . . 91
7.4 Hybrid choice model: draws and latent variable . . . 93
7.5 Hybrid choice model with ordered measurement model: defining probabilities . . . 94
7.6 Hybrid choice model with continuous measurement model: zero-centering indicators and defining probabilities . . . 95

8.1 Bayesian estimation in *Apollo*: model settings . . . 97
8.2 Bayesian estimation in *Apollo*: estimation process . . . 99
8.3 Bayesian estimation in *Apollo*: estimation process (parameter chains) . . . 100
8.4 Bayesian estimation in *Apollo*: output (extracts) . . . 101

9.1 Running `apollo_choiceAnalysis` (syntax and excerpt of output) . . . 103
9.2 Running `apollo_llCalc` . . . 104
9.3 Running `apollo_lrTest` . . . 105
9.4 Running `apollo_basTest` . . . 107
9.5 Running `apollo_prediction` . . . 109
9.6 Running `apollo_sharesTest` . . . 111
9.7 Running `apollo_fitsTest` . . . 113
9.8 Running `apollo_deltaMethod` . . . 114
9.9 Running `apollo_unconditionals` and `apollo_conditionals` . . . 117
9.10 Running `apollo_lcUnconditionals` and `apollo_lcConditionals` . . . 119

10.1 Example of failure during model estimation . . . 122
10.2 Debugging step 1: testing with `functionality="estimate"` . . . 123
10.3 Debugging step 2: analysing choices for respondents with zero likelihood value . . . 124
10.4 Debugging step 3: actual debugging . . . 124

11.1 Running `apollo_searchStart` . . . 128
11.2 Running `apollo_outOfSample` . . . 130
11.3 Running `apollo_bootstrap` . . . 132
11.4 EM algorithm for Latent Class: setup . . . 135
11.5 EM algorithm for Latent Class: estimation . . . 136
11.6 EM algorithm for Mixed Logit: setup . . . 138
11.7 EM algorithm for Mixed Logit: estimation . . . 139
11.8 Defining utilities for large choice sets (`Apollo_example_30.r`) . . . 140

List of Tables

B.1 Data dictionary for `apollo_modeChoiceData.csv` . . . 175
B.2 Data dictionary for `apollo_swissRouteChoiceData.csv` . . . 176
B.3 Data dictionary for `apollo_drugChoiceData.csv` . . . 177
B.4 Data dictionary for `apollo_timeUseData.csv` . . . 178

C.1 Index of example files . . . 180
C.2 Functions used by *Apollo*, with inputs and outputs . . . 181

D.1 Functions used by *Apollo*, with inputs and outputs . . . 184
D.2 Lists used by *Apollo* . . . 190
D.3 Elements in lists and functions used by *Apollo* . . . 193

E.1 Elements inside a model object estimated by *Apollo* . . . 205

Chapter 1

Introduction

Choice modelling techniques have been used across different disciplines for over four decades (see McFadden 2000 for a retrospective and Hess and Daly 2014 for recent contributions and applications across fields). For the majority of that time, the number of users of especially the most advanced models was rather small, and similarly, a small number of software packages was used by this community. In the last two decades, the pool of users of choice models has expanded dramatically, in terms of their number as well as the breadth of disciplines covered. At the same time, we have seen the development of new modelling approaches, and gains in computer performance as well as software availability have given a growing and broader group of users access to ever more advanced models.

These developments have also seen a certain fragmentation of the community in terms of software, which in part runs along discipline lines[1]. Notwithstanding the most advanced users who develop their own code for often their own models, there is first a split between the users of commercial software and those using freeware tools. Commercial packages are generally computationally more powerful but may have more limitations in terms of available model structures or the possibility for customisation. On the other hand, freeware options can have limitations in terms of performance and user friendliness but may benefit from more regular developments to accommodate new model structures.

A further key differentiation between packages is the link between user inputs and interface and the actual underlying methodology. Many existing packages, both freeware and commercial, are black box tools where the user has little or no knowledge of what goes on "under the hood". While this has made advanced models accessible to a broader group of users, a disconnect between theory and software not only increases the risk of misinterpretations and misspecifications, but can also hide relevant nuances of the modelling process and mistakenly give the impression that choice models are "easy tools" to use. On the other hand, software that relies on users to code all components from scratch arguably imposes too high a bar in terms of access.

Software also almost exclusively allows the use of only either classical estimation techniques or Bayesian techniques. This fragmentation again runs largely in parallel with discipline boundaries

[1]We intentionally do not refer to specific packages, so as not to risk any misrepresentations but also given the growing number of freeware tools, some of which we might not be aware of.

and has only served to further contribute to the lack of interaction/dialogue between the classical and Bayesian communities.

A final difference arises in terms of software environment. While commercial software usually provides a custom user interface, freeware options in general (though not exclusively) rely on existing statistical or econometric software and are made available as packages within these. The latter at times means that *freeware* packages are not really free to use (if the host software is not), while there are also cases of software being accessible only in a single operating system (e.g. Windows or Linux/MacOS, not across systems).

The above points served in large part as the motivation for the development of *Apollo*. Our aims were:

Free access: *Apollo* is a completely free package which does not rely on commercial statistical software as a host environment.

Big community: *Apollo* relies on `R` (R Core Team, 2017), which is very widely used across disciplines and works well across different operating systems.

Transparent, yet accessible: *Apollo* is neither a blackbox nor does it require expert econometric skills. The user can see as much or as little detail of the underlying methodology as desired, but the link between inputs and outputs remains.

Ease of use: *Apollo* combines easy to use `R` functions with new intuitive functions without unnecessary jargon or complexity.

Modular nature: *Apollo* uses the same code structure independently of whether the simplest Multinomial Logit model is to be estimated, or a complex structure using random coefficients and combining multiple model components.

Fully customisable: *Apollo* provides functions for many well known models but the user is able to add new structures and still make use of the overall code framework. This for example extends to coding expectation-maximisation routines.

Discrete and continuous: *Apollo* incorporates functions not just for commonly used discrete choice models but also for a family of models that looks jointly at discrete and continuous choices.

Novel structures: *Apollo* goes beyond standard choice models by incorporating the ability to estimate Decision Field Theory (DFT) models, a popular accumulator model from mathematical psychology.

Classical and Bayesian: *Apollo* does not restrict the user to either classical or Bayesian estimation but easily allows changing from one to the other.

Easy multi-threading: *Apollo* allows users to split the computational work across multiple processors without making changes to the model code.

Not limited to estimation: *Apollo* provides a number of pre and post-estimation tools, including diagnostics as well as prediction/forecasting capabilities and posterior analysis of model estimates.

While *Apollo* is easy to use, we also remain of the opinion that users of choice modelling software should understand the actual process that happens during estimation. For this reason, the user needs to explicitly include or exclude calls to specific functions that are model and dataset specific. For example, in the case of repeated choice data, the user needs to include a call to a function

that takes the product across choices for the same person (`apollo_panelProd`). Or in the case of a Mixed Logit model, the user needs to include a call to a function that averages across draws (`apollo_avgInterDraws` and/or `apollo_avgIntraDraws`). If calls to these functions are missing when needed, or if a user makes a call to a function that should not be used in the specific model, the code will fail, and provide the user with feedback about why this happened. This is in our view much better than a situation where the software permits users to make mistakes and fixes them behind the scenes.

Users of *Apollo* are asked to acknowledge the use of the software by citing the academic paper:

- Hess, S. & Palma, D. (2019), *Apollo*: a flexible, powerful and customisable freeware package for choice model estimation and application, Journal of Choice Modelling

and the manual for the version used in their work, e.g:

- Hess, S. & Palma, D. (2020), *Apollo* version 0.01, `www.ApolloChoiceModelling.com`.

Apollo is the culmination of many years of development of individual choice modelling routines, starting with code developed by Hess while at Imperial College (cf. Hess, 2005) using Ox (Doornik, 2001). This code was gradually transitioned to `R` at the University of Leeds, with substantial further developments once Palma joined the team in Leeds, bringing with him ideas developed at Pontificia Universidad Católica de Chile (cf. Palma, 2016). No code is an island, and we have been inspired especially by ALogit (ALogit, 2016) and Biogeme (Bierlaire, 2003), and *Apollo* mirrors at least some of their features.

This manual presents an overview of the capabilities of the *Apollo* package and serves as a user manual. It is accompanied online at `www.ApolloChoiceModelling.com` by numerous example files (some of which are used in this manual) and a number of free to use datasets. Users are also encouraged to visit the online help forum where numerous questions on specification have already been answered. In adddition to the detailed information in this manual, users can also obtain help on specific functions directly in R, using e.g. `?apollo_mnl` for help on the `apollo_mnl` function.

In line with our earlier point about other software, this manual does not include any comparisons with other packages, in terms of capabilities or speed. The code has been widely tested to ensure accuracy. In our view, any speed comparison offers little practical benefit. For simple models, there is a clear advantage for highly specialised code, while, for complex models, any benchmarking is impacted substantially by the specific implementation and degree of optimisation used.

In the remainder of this manual, we do not provide details on common `R` functions and syntax used in the code, or how to run `R` code, and the reader is instead referred to R Core Team (2017). For the syntax shown in this manual, it is just worth noting that in `R`, a line starting with one or more `#` characters is a comment. We tend to use a single `#` for optional lines that a user can comment in or out, and `###` for actual comments. In addition, two other points are worth raising.

- In complex models, the `R` syntax file for *Apollo* can become quite large, and a user may wish to split this into separate files, e.g. one for loading and processing the data, one for the actual model definition, etc, and then have a master file which calls the individual files (using `source`).

- For the predefined functions, the order of arguments passed to the function should be kept in the order specified in this manual[2].

Another point to raise concerns the specific naming conventions we have adopted for functions and inputs to functions. All functions within the code start with the prefix apollo_. This is then followed by the "name" of the actual function in a single word, where any new part of the name starts with a capital letter, for example `apollo_modelOutput`. The prefix `apollo_` is also used for a number of key non-function objects in the code, namely:

- the user defined settings `apollo_control`, `apollo_HB` and `apollo_draws`;
- the list of parameters `apollo_beta` and fixed parameters `apollo_fixed`; and
- the automatically generated combined inputs variable `apollo_inputs`.

The functions in *Apollo* take numerous inputs and for ease of programming, these are often combined into a *list* object. The naming convention used for these is to have the name of the function (without the `apollo_` prefix) followed by `_settings`, for example in `modelOutput_settings`. Finally, individual variables/settings do not have a prefix and again use the convention of capitalising the first letter of any new word except for the start, for example in `printDiagnostics`.

Before we proceed, a brief explanation is needed as to our choice of the name *Apollo*. Several existing packages refer to specific models in their name (e.g. ALogit, NLogit) which is not applicable in our case given the wider set of models we cover. We failed miserably in our efforts to come up with an imaginative acronym like Biogeme and so went back to Greek mythology. The obvious choice would have been Cassandra, with her gift of prophecy and the curse that nobody listened to her (a bit like choice modellers trying to sell their ideas to policy makers). Alas, the name has already been used for a large database package, so we resorted to Apollo, the Greek god of prophecy who gave this gift to Cassandra in the first place. And, as a student more versed in Greek mythology than ourselves told us, it was Apollo who slayed Python.

The remainder of this manual is organised as follows. The following chapter talks about installation before Chapter 3 introduce a number of datasets used throughout the manual. Chapter 4 provides an in-depth introduction to the code structure, using the example of a simple Multinomial Logit model. This is followed in Chapter 5 by an overview of other available model components, and a description of how the user can add his/her own models. Chapter 6 covers random heterogeneity, both discrete and continuous while Chapter 7 discusses joint estimation of multiple model components, with a focus on hybrid choice models. Bayesian estimation is covered in Chapter 8 with (mainly) post-estimation capabilities discussed in Chapter 9. Chapter 10 looks at debugging an *Apollo* model that fails in estimation, and a few extensions are discussed in Chapter 11. Finally, Chapter 12 addresses a set of frequently asked questions. A number of appendices are also included. Appendix A summarises changes across different versions of *Apollo*.

[2]Unless a user explicitly prefaces each argument with the name used in the function. For example, if a function is defined to take two inputs, namely `dependent` and `explanatory`, e.g. `model_prob(dependent,explanatory)`, and the user wants to use `choice` and `utility` as the inputs, then the function can be called as `model_prob(choice,utility)` but not as `model_prob(utility,choice)`. The latter change in order is only possible if the function is called explicitly as `model_prob(explanatory=utility,dependent=choice)`, which is the same as `model_prob(dependent=choice,explanatory=utility)`.

Appendix B contains data dictionaries, Appendix C a list of the example files and Appendix D an index of functions and variables in *Apollo*.

Chapter 2

Installing *Apollo*, loading the libraries and running the code

Apollo runs in R, with a minimum `R` version of 4.0.0. *Apollo* can be installed in two ways. If an internet connection is available, the easiest way to install it is to type the following command into the `R` console. This will also install all dependencies, i.e. other routines used by the *Apollo* package[1].

```
install.packages("apollo")
```

The second way is to install it from a file. A file containing the source code can be obtained at www.ApolloChoiceModelling.com. Then, the following command must be typed into the `R` console.

```
install.packages("C:\...\apollo_v0.2.4.tar.gz", repos = NULL, type = "source")
```

where `C:\...\apollo_v0.2.4.tar.gz` must be replaced by the correct path to the file in the user's computer, using the version that was downloaded. This will not automatically install dependencies.

The installation of the package does not need to be repeated every time `R` is started nor every time a model is to be estimated. Instead, it only needs to be done once for each new release of *Apollo* (unless R itself is updated, then the installation must be repeated).

Every time users want to estimate a model, they need to load *Apollo* into memory. This can be achieved by simply running the following line of code in R, or by including it in the source file of each model, prior to running any *Apollo* functions.

```
library(apollo)
```

Users are encouraged to check for updated versions of the package every few months. Updates, when available, can be acquired by simply re-installing the package. Installation from CRAN will

[1] For installation on macOS, users should install from binaries, rather than source.

install the latest release. Previous releases will be available from the software website, where users also have access to versions with new features that are under development prior to a full release. These versions need to be compiled locally, and users require Rtools for this purpose.

Most users will run `R` from a shell such as `RStudio` (RStudio Team, 2015). A full *Apollo* model file, or any other `R` script, can also be run from the command line, without accessing `R` directly. This can be useful when running many scripts unattended, or when submitting jobs to a computer cluster. The command to do this changes depending on the operation system and the local directory structure. In Linux, the command is as follows: `R CMD BATCH model.R`. In Windows, the command is for example as follows: `"C:\Program Files\R\R-3.6.3\bin\R.exe" CMD BATCH model.R`. Note that in both cases, the working directory should be set within the model file using the `setwd` function. The output that would normally be printed to the `R` Terminal will instead be written to a file called `model.Rout`, which can be opened with any plain text editor.

Chapter 3

Data format and datasets used for examples

Apollo makes use of a format where all relevant information for a given observation is stored in the same row. Using a simple discrete choice context, this would imply that the data for all alternatives is included in the same row, rather than one row per alternative. Some choice modellers refer to this as the *wide* format, as opposed to the *long* format, which would have one row per alternative. This terminology is in fact not very helpful as, in the context of repeated measurements data, the term *wide* refers to a format where all measurements for the same person are included in one line. In the one row per observation format in a choice modelling context, there will still be multiple rows for different choices for the same person. There are good reasons for why *Apollo* is using the one row per observation format. This format is the more common format in choice modelling, uses less space, and is also more general in allowing for a mixture of different dependent variables in the same data. Analysts whose data is in the one row per alternative format need to reshape it prior to using it in *Apollo*.

This chapter presents a number of datasets used throughout the manual and in the online examples. Below, we give brief introductions to the datasets, with details on variable names provided in Appendix B. The datasets are included in the *Apollo* package itself. A user can access a given dataset, say the `apollo_drugChoiceData`, by typing `data("apollo_drugChoiceData", package="apollo")` in the console. To use any data in *Apollo*, it needs to be stored in an object called `database`, and the user would need to use `database=get(data("apollo_drugChoiceData", package="apollo"))`. While this is possible for the four datasets included with *Apollo*, the majority of applications will of course rely on users' own datasets, which will be read in from a file, typically of the comma separated volume (*csv*) format. This is also the approach we use with the examples in the manual so as to illustrate the process of reading from files to the user. The four datasets are available as *csv* files from the *Apollo* website at www.ApolloChoiceModelling.com.

3.1 RP-SP mode choice dataset: `apollo_modeChoiceData`

Our first resource is a synthetic dataset looking at mode choice for 500 travellers. For each individual, the data contains two revealed preference (RP) inter-city trips, where the possible modes were car, bus, air and rail, and where each individual has at least two of these four modes available to them. The journey options are described on the basis of access time (except for car), travel time and cost, with times in minutes, and costs in £. The data then also contains 14 stated preference (SP) tasks per person, using the same alternatives as those available on the RP journey for that person, but with an additional categorical quality of service attribute added in for air and rail, taking three levels, namely *no frills*, *wifi available*, or *food available*. For each individual, the dataset also contains information on gender, whether the journey was a business trip or not, and the individual's income.

3.2 SP route choice dataset: `apollo_swissRouteChoiceData`

Our second dataset comes from an actual SP survey of public transport route choice conducted in Switzerland (Axhausen et al., 2008). A set of 388 people were faced with 9 choices each between two public transport routes, both using train (leading to 3,492 observations in the data). The two alternatives are described on the basis of travel time, travel cost, headway (time between subsequent trains/busses) and the number of interchanges. For each individual, the dataset additionally contains information on income, car availability in the household, and whether the journey was made for commuting, shopping, business or leisure.

3.3 Health attitudes SP: `apollo_drugChoiceData`

Our third dataset is a synthetic dataset looking at drug choices for the treatment of headaches for 1,000 individuals. For each person, the data contains 10 SP tasks, each giving a choice between four alternatives, the first two being products by recognised drug companies while the final two are generic products. In each choice task, a full ranking of the four alternatives is given. The drugs are described in terms of brand (two recognised brands and three generic brands), country of origin (six countries), drug features (three types of features), risk of side effects and price. The possible levels for the attributes differ between the first two (branded) and last two (generic) alternatives. For each individual, the dataset additionally contains answers to four attitudinal questions as well as information on whether an individual is a regular user, their education and their age.

3.4 Time use data: `apollo_timeUseData`

Our fourth dataset comes from a GPS tracking survey on time use conducted in the UK (Calastri et al., 2019). A set of 447 individuals completed a digital activity log for up to 14 days, providing 2,826 days of data (first day discarded for each person). For each day, the amount of time spent in each of twelve activities is recorded, as well as some of the individual's characteristics.

The activities considered were dropping-off or picking-up, working, going to school, shopping, private business, getting petrol, social or leisure activities, vacation, doing exercise, being at home, travelling, and a last activity grouping the time allocated to other activities by the individual.

Chapter 4

General code structure and components: illustration for MNL

In this chapter, we provide an introduction to the general capabilities of the *Apollo* package by using the example of a Multinomial Logit (MNL model (McFadden, 1974), where the probability of person n choosing alternative i (out of $j = 1, \ldots, J$) in choice situation t is given by:

$$P_{i,n,t} = \frac{e^{V_{i,n,t}}}{\sum_{j=1}^{J} e^{V_{j,n,t}}}, \tag{4.1}$$

with $V_{i,n,t}$ giving the systematic component of the utility for alternative i for person n in choice situation t.

We apply this to the simple mode choice stated preference survey introduced in Section 3.1, where we use the SP part of this data, i.e. 14 choices each for 500 individuals. This example is available in the file `Apollo_example_3.r` and uses a very detailed specification of the utility function. More barebones examples are also available in `Apollo_example_1.r` and `Apollo_example_2.r`, which are models without any socio-demographics, estimated on the `RP` and `SP` data, respectively.

The structure of an *Apollo* model file varies across specifications, but a general overview is shown in Figure 4.1, and we now look at these steps in turn.

4.1 Initialising the code

The first step in every use of *Apollo* is to initialise the code. These steps are illustrated in Figure 4.2. In an optional step, we clear the memory/workspace by using `rm(list = ls())`, before loading the *Apollo* library. This is followed by calling the `apollo_initialise` function, which 'detaches' variables[1] and makes sure that output is directed to the console rather than a file. This

[1] In `R`, a user can 'attach' an object, which means that individual components in it can be called by name.

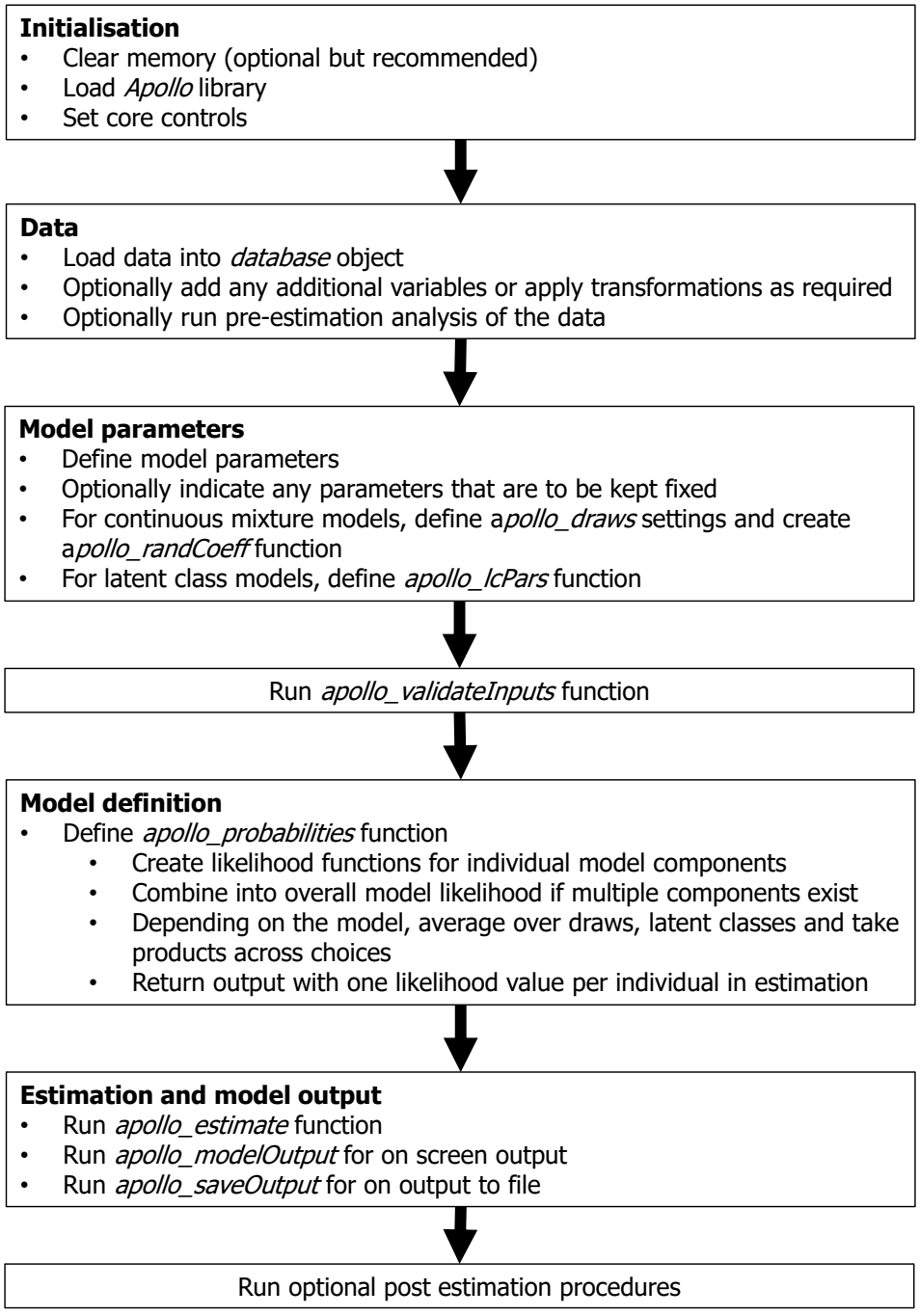

Figure 4.1: General structure of an *Apollo* model file

function is called without any arguments and does not return any output variables, i.e.:

```
apollo_initialise()
```

The user next sets a number of core controls in a list called `apollo_control`, where in our case, we only give the name of the model (where any output files will use this name too), provide a brief description of the model (for use in the output) and indicate the name (in quotes) of the column in the data which contains the identifier variable for individual decision makers. Each time, the entry on the left is an *Apollo*-defined variable whose name is not to be changed, and the user provides the value on the right, followed by a comma, except for the last element.

```
rm(list = ls())

library(apollo)

apollo_initialise()

apollo_control = list(
  modelName       = "Apollo_example_3",
  modelDescr      = "MNL model with socio-demographics on mode choice SP data",
  indivID         = "ID",
  outputDirectory = "output"
)
```

Figure 4.2: Code initialisation

Only this final setting in Figure 4.2, i.e. setting the individual ID, is a requirement without which the code will not run. For any other settings, the code will use default values when not provided by the user, as illustrated in Figure 4.6. These other settings include:

`mixing`: A boolean variable which needs to be set to TRUE when the model uses continuous random coefficients, as discussed in Section 6.1 (default is set to FALSE).

`nCores`: An integer setting the number of cores used during estimation discussed, as discussed in Section 6.4 (default is set to 1).

`workInLogs`: A boolean variable, which, when set to TRUE, means that the logs of probabilities are used when processing probabilities inside `apollo_probabilities`. This can avoid numerical issues with complex models and datasets where there are large numbers of observations per individual. This is only really useful with repeated choice, and slows down estimation (default is set to FALSE).

`seed`: An integer setting the seed used for any random number generation (default is 13).

`HB`: A boolean variable which needs to be set to TRUE for using Bayesian estimation, as discussed in Section 8 (default is FALSE).

`panelData`: A boolean variable indicating whether the data is to be treated as panel data. This is set automatically to TRUE if multiple observations are present per individual, and FALSE otherwise. If a user sets this to FALSE in the presence of multiple observations per individual, the data will be treated as cross-sectional.

`noValidation` A boolean variable, which, when set to TRUE, means that no validation checks are performed (default is FALSE).

`noDiagnostics` A boolean variable, which, when set to TRUE, means that no model diagnostics are reported. This setting is provided primarily to avoid excessively verbose output with

complex models using many components (cf. Section 7) but will be set to FALSE (default) for most models by most users.

`weights`: The name of a variable in the database containing weights for each observation, which can then be used in estimation if also using the function `apollo_weighting` (default is for weights to be missing).

`outputDirectory`: The name of a directory to be used for reading input and writing output to. This should be a subdirectory of the working directory, or an absolute path. By default, the output directory is set to the working directory.

4.2 Reading and processing the data

Figure 4.3 illustrates the process of loading the data, in this case from a *csv* file, working with only a subset of the data (in this case removing the RP observations) and creating additional variables in the data (in this case a variable with the mean income in the data). In our example, we read the data file from the working directory, which we had set to be the same as the directory with the model file. Users may need to add the path of the file depending on their local setup and file structure. If the data file contains multiple rows per individual, then these need to be next to each (i.e. contiguous) in the data file.

Three additional points need to be mentioned here:

- Firstly, the code is not limited to using *csv* files, and `R` allows the user to read in tab separated files too, for example[2].
- Secondly, some applications may combine data from multiple files. The user can either combine the data outside of `R` or do so inside `R` using appropriate merging functions, but at the point of validating the user inputs (Section 4.4), all data needs to be combined in a single `R` data.frame called `database`.
- Thirdly, any new variables created by the user, such as mean income in our case, need to be created in the database object rather than the global environment, and this needs to happen prior to validating the user inputs.

```
database = read.csv("apollo_modeChoiceData.csv",header=TRUE)
database = subset(database,database$SP==1)
database$mean_income = mean(database$income)
```

Figure 4.3: Loading data, selecting a subset and creating an additional variable

4.3 Model parameters

In this simple model, we estimate alternative specific constants (ASCs), mode specific travel time coefficients, a cost and access time coefficient and dummy coded coefficients for the service quality

[2]The reader is referred to R Core Team (2017).

attribute. In addition, we interact the constants with gender, allow for differences in the time and cost sensitivities for business travellers (generic across modes), and incorporate an income elasticity on the cost sensitivity.

With the above, the utilities for the four modes in choice situation t for individual n are given by:

$$
\begin{aligned}
U_{car,n,t} &= \delta_{car} \\
&+ (\beta_{tt,car} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,car,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,car,n,t} \\
&+ \varepsilon_{car,n,t} \\
U_{bus,n,t} &= \delta_{bus} + \delta_{bus,female-shift} \cdot z_{female,n} \\
&+ (\beta_{tt,bus} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,bus,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,bus,n,t} \\
&+ \varepsilon_{bus,n,t} \\
U_{air,n,t} &= \delta_{air} + \delta_{air,female-shift} \cdot z_{female,n} \\
&+ (\beta_{tt,air} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,air,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,air,n,t} \\
&+ \beta_{no\,frills} \cdot (x_{service,air,n,t} == 1) + \beta_{wifi} \cdot (x_{service,air,n,t} == 2) + \beta_{food} \cdot (x_{service,air,n,t} == 3) \\
&+ \varepsilon_{air,n,t} \\
U_{rail,n,t} &= \delta_{rail} + \delta_{rail,female-shift} \cdot z_{female,n} \\
&+ (\beta_{tt,rail} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,rail,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,rail,n,t} \\
&+ \beta_{no\,frills} \cdot (x_{service,rail,n,t} == 1) + \beta_{wifi} \cdot (x_{service,rail,n,t} == 2) + \beta_{food} \cdot (x_{service,rail,n,t} == 3) \\
&+ \varepsilon_{rail,n,t},
\end{aligned}
\tag{4.2}
$$

where all parameters are estimated except for δ_{car} and $\beta_{no\,frills}$, which are both fixed to a value of zero.

In the code, the user needs to define the parameters and their starting values, and also indicate whether any of the parameters are to be kept at their starting values. This process is illustrated in Figure 4.4. We first create an `R` object of the *named vector* type, called `apollo_beta`, with the name and starting value for each parameter, including any that are later on fixed to their starting values. In our case, we keep two of the parameters, namely `asc_car` and `b_no_frills`, fixed to their starting values by including their names in the character vector `apollo_fixed`, where this vector is kept empty (`apollo_fixed = c()`) if all parameters are to be estimated. Parameters included in `apollo_fixed` are kept at the value used in `apollo_beta`, which may not be zero.

```
apollo_beta=c(asc_car                = 0,
              asc_bus                = 0,
              asc_air                = 0,
              asc_rail               = 0,
              asc_bus_shift_female   = 0,
              asc_air_shift_female   = 0,
              asc_rail_shift_female  = 0,
              b_tt_car               = 0,
              b_tt_bus               = 0,
              b_tt_air               = 0,
              b_tt_rail              = 0,
              b_tt_shift_business    = 0,
              b_access               = 0,
              b_cost                 = 0,
              b_cost_shift_business  = 0,
              cost_income_elast      = 0,
              b_no_frills            = 0,
              b_wifi                 = 0,
              b_food                 = 0)

apollo_fixed = c("asc_car","b_no_frills")
```

Figure 4.4: Setting names and starting values for model parameters, and fixing some parameters to their starting values

For complex models especially, it can sometimes be beneficial to read in starting values from an earlier model, albeit that users should be mindful that this can lead to problems with convergence to the estimates of the old model. This process is made possible by the function `apollo_readBeta`, which is called as:

```
apollo_beta = apollo_readBeta(apollo_beta,
                              apollo_fixed,
                              inputModelName,
                              overwriteFixed)
```

The function returns an updated version of `apollo_beta`. The first two arguments passed to the function are already known to the reader, the remaining two are:

`inputModelName:` The name of a previously estimated model, given as a string.
`overwriteFixed:` A boolean variable indicating whether parameters that are not to be estimated should have their starting values overwritten by the input file (set to FALSE by default).

To use `apollo_readBeta`, the outputs from the input model need to have been saved in the same directory as the current model file. We illustrate the use of this function in Figure 4.5, where we read in parameters from the earlier RP model (`Apollo_example_1.r`) which did not include the socio-demographic effects or the quality of service attribute, thus meaning that only values for the 9 estimated parameters were read in, with the fixed parameter `asc_car` kept to the value from `apollo_beta` given the use of `overwriteFixed=FALSE`, where, with `overwriteFixed=TRUE`, the value from the input file would also be used for fixed parameters.

```
> apollo_beta=apollo_readBeta(apollo_beta,apollo_fixed,"Apollo_example_1",overwriteFixed=FALSE)

Out of the 19 parameters in apollo_beta, 9 were updated
  with values from the input file.
1 parameter in apollo_beta was kept fixed at its starting
  value rather than being updated from the input file.
```

Figure 4.5: Using `apollo_readBeta` to load results from an earlier model as starting values

4.4 Validation and preparing user inputs

The final step in preparing the code and data for model estimation or application is to make a call to `apollo_validateInputs` The function runs a number of checks and produces a consolidated list of model inputs. It is called as:

```
apollo_inputs=apollo_validateInputs()
```

This function takes no arguments but looks in the global environment for the various inputs required for a model. This always includes the control settings `apollo_control`, the model parameters `apollo_beta`, the vector with names of fixed parameters `apollo_fixed` and finally the data object `database`. If any of these objects are missing from the global environment, the execution of `apollo_validateInputs` fails. The function also looks for a number of optional objects, namely `apollo_HB`, which is used for Bayesian estimation (cf. Section 8), `apollo_draws` and `apollo_randCoeff`, which are used for continuous random coefficients (cf. Section 6.1), and `apollo_lcPars`, which is used for latent class (cf. Section 6.2).

Before returning the list of model inputs, `apollo_validateInputs` runs a number of validation tests on the `apollo_control` settings and the `database`. It then uses default values for any missing settings, and, in the case of panel data, adds an extra column called `apollo_sequence` which is a running index of observations for each individual in the data. Finally, the code also checks for the presence of multiple rows per individual in the data and accordingly sets `apollo_control$panelData` to TRUE or FALSE[3]. The running of `apollo_validateInputs` is illustrated in Figure 4.6. The list that is returned, `apollo_inputs`, contains the validated versions of the various objects mentioned above, e.g. `database`.

```
> apollo_inputs = apollo_validateInputs()
Several observations per individual detected based on the value of ID. Setting panelData in
  apollo_control set to TRUE.
All checks on apollo_control completed.
All checks on database completed.
```

Figure 4.6: Running `apollo_validateInputs`

4.5 Likelihood component: the *apollo_probabilities* function

The core part of the code is contained in the `apollo_probabilities` function, where we show this function for our simple MNL model in Figure 4.7. An important distinction arises between

[3] In R, elements of a list such as `apollo_control` can be referred to via `apollo_control$panelData`.

`apollo_probabilities` and other functions in *Apollo*. While the other functions we have encountered are part of the package, `apollo_probabilities` needs to be defined by the user as it is specific to the model to be estimated. The function itself is never called by the user, but is used for example by the function for model estimation `apollo_estimate` discussed below. The `apollo_probabilities` function returns probabilities, where the specific format depends on `functionality`, which takes a default value for model estimation, but other values apply for example in prediction, as discussed in Section 9.6. The value used for `functionality` depends on which function makes the call to `apollo_probabilities` and is controlled internally - the user does not need to change this, except in debugging, as explained in Chapter 10.

This function takes three inputs, namely the vector of parameters `apollo_beta`, the list of combined model inputs `apollo_inputs`, and the argument `functionality`, which takes a default value for model estimation, but other values apply for example in prediction, as discussed in Section 9.6. The value used depends on which function makes the call to `apollo_probabilities`.

In the following three subsections, we look at the individual components of the code shown in Figure 4.7.

4.5.1 Initialisation

Any use of the `apollo_probabilities` function begins with a call to `apollo_attach` which enables the user to then call individual elements within for example the database by name, e.g. using `female` instead of `database$female`. This function is called as:

```
apollo_attach(apollo_beta,
              apollo_inputs)
```

The function does not return an object as output and the user does not need to change the arguments for this function. The call to this function is immediately followed by a command instructing `R` to run the function `apollo_detach` once the code exits `apollo_probabilities`. This ensures that this call is made even if there is an error that leads to a failure (and hence hard exit) from `apollo_probabilities`. This call is made as:

```
on.exit(apollo_detach(apollo_beta,
                      apollo_inputs))
```

With the database now having been attached, all elements within it can be referred to it by name inside `apollo_probabilities` and referring to `database` is no longer permitted.

We next initialise a list (a flexible `R` object) called `P` which will contain the probabilities for the model, where this is a requirement for any type of model used with the code.

4.5.2 Model definition

With $\varepsilon_{car,n,t}$, $\varepsilon_{bus,n,t}$, $\varepsilon_{air,n,t}$ and $\varepsilon_{rail,n,t}$ in Equation 4.2 being distributed identically and independently (*iid*) across individuals and choice scenarios following a type I extreme value distri-

bution, we obtain an MNL mode (cf. Luce, 1959; McFadden, 1974) , with the probability for alternative i in choice task t for person n given by:

$$P_{i,n,t}(\boldsymbol{\beta}) = \frac{z_{avail,i,n,t} \cdot e^{V_{i,n,t}}}{\sum_{j=1}^{J} z_{avail,j,n,t} \cdot e^{V_{j,n,t}}}, \tag{4.3}$$

where $\boldsymbol{\beta}$ is a vector combining all model parameters, $V_{j,n,t}$ refers to the part of the utility functions in Equation 4.2 that excludes the error term $\varepsilon_{j,n,t}$, and where $z_{avail,j,n,t}$ takes a value of 1 if alternative j is available in choice set t for person n, and 0 otherwise.

In the central part of the `apollo_probabilities` function, the user defines the actual model, where in our example, this is a simple MNL model. No limits on flexibility are imposed on the user with the *Apollo* package. A number of prewritten functions for common models are made available in the package, going beyond MNL, as discussed in Section 5. Additionally, the user can define his/her own models, as discussed in Section 5.5. Finally, this part of the code can contain either a single model, as shown here, or multiple individual model components, as discussed in Section 7.

The `apollo_mnl` function is called via:

```
P[["model"]] = apollo_mnl(mnl_settings,
                          functionality)
```

The function returns probabilities for the model, where depending on `functionality`, this is for the chosen alternative only or for all alternatives. The output of the function is saved in a component of the list `P` where for single component models such as here, this element is called `P[["model"]]`. The function takes as its core input a list called `mnl_settings` which has four compulsory inputs and two optional inputs. We will now look at these in turn.

`alternatives:` A named vector containing the names of the alternatives as defined by the user, and for each alternative, giving the value used in the dependent variable in the data. In our case, these simply go from 1 to 4.

`avail:` A list containing one element per alternative, using the same names as in `alternatives`. For each alternative, we define the availability either through a vector of values of the same length as the number of observations (i.e. a column from the data) or by a scalar of 1 if an alternative is always available. A user can also set `avail=1` (or omit its use) which implies that all of the alternatives are available for every choice observation in the data.

`choiceVar:` A vector of length equal to the number of observations, containing the chosen alternative for each observation. In our example, this column is simply called `choice`.

`V:` A list object containing one utility for each alternative, using the same names as in `alternatives`, where any linear or non-linear specification is possible. The contents of `V` are complicated and are thus generally defined prior to calling the function, as in Figure 4.7. In our case, we pre-compute the interactions with socio-demographic variables in the lines preceding the definition of the actual utilities, creating for example the new parameter

`b_tt_car_value`. This helps keep the code organised, makes it easier to add additional interactions and also avoids unnecessary calculations. The latter point can be understood by noting that in our example, the impact of income and purpose on the cost coefficient is calculated just once and then used in each of the four utilities, rather than being calculated four times.

`rows:` This is an optional argument which is missing by default. It allows the user to specify a vector called `rows` of the same length as the number of rows in the data. This vector needs to use logical statements to identify which rows in the data are to be used for this model. For any observations in the data where the entry in rows is set to FALSE, the probability for the model will be set to 1. This means that, for this model, this observation does not contribute to the calculation of the likelihood and hence estimation of the model parameters. It is useful for example in the case of hybrid choice models, a point we return to in Section 7. When omitted from the call to `apollo_mnl`, all rows are used, as in our example in Figure 4.7.

`componentName:` This is an optional argument of the character type which allows the user to specify a name of the given model component. This is then used in various places in diagnostic tests and model outputs. If omitted, a default is used by *Apollo*, where this varies across model types, using for example `MNL` in the case of `apollo_mnl`.

In the code example, we actually create the utilities `V` outside `mnl_settings` first just for ease of coding, but they can similarly be created directly inside the list. What matters if using the former approach is that they are then copied into a component called `V` inside `mnl_settings`.

4.5.3 Function output

The final component of the `apollo_probabilities` function prepares the output of the function. This performs further processing of the `P` list, which needs to include an element called `model`, where, in our example, this is the only element in `P`. The specific functions to be called in this part of the code depend on the data and model, where once again, the actual inputs to these functions are not to be changed by the user.

In our specific example, the only additional manipulation of the raw probabilities produced by `apollo_mnl` is a call to `apollo_panelProd` which multiplies the probabilities across individual choice observations for the same individual, thus recognising the repeated choice nature of our data. This function is only to be used in the presence of multiple observations per individual. When estimating a model, the code computes the probability for the chosen alternative, say $j^*_{n,t}$ in choice task t for person n, i.e. $P_{j^*_{n,t}}$, using Equation 4.3. The contribution by person n to the likelihood function, with a given value for the vector of model parameters $\boldsymbol{\beta}$, is then given by:

$$L_n(\boldsymbol{\beta}) = \prod_{t=1}^{T_n} P_{j^*_{n,t}}, \tag{4.4}$$

where T_n is the number of separate choice situations for person n. This function is called as:

```
P = apollo_panelProd(P,
                     apollo_inputs,
                     functionality)
```

All arguments of this function have been described already. When called in model prediction (cf. Section 9.6), the multiplication across choices is omitted, i.e. the function returns an unmodified version of `P`, with one row per observation.

Independent of the model specification, the function `apollo_probabilities` always ends with the same two commands. First is `apollo_prepareProb` which prepares the output of the function depending on `functionality`, e.g. with different output for estimation and prediction. This is called as:

```
P = apollo_prepareProb(P,
                       apollo_inputs,
                       functionality)
```

This is followed by

```
return(P)
```

which ensures that `P` is returned as the output of `apollo_probabilities`.

We earlier mentioned the possible use of weights by including the setting `weights` in `apollo_control`. If the user wants to use weights, then they must also call the function `apollo_weighting` prior to `apollo_prepareProb`. This is called as:

```
P = apollo_weighting(P,
                     apollo_inputs,
                     functionality)
```

Note that when weights are used, they are applied during estimation, prediction and any other application of the model (calculation of the model null log-likelihood, conditionals, unconditionals, etc.). If the user wants, for example, to estimate without weights but predict with them, then we recommend setting the values of all weights to 1 during estimation, and to their real values during forecasting.

4.6 Estimation

Now that we have defined our model, we can perform model estimation by calling the function `apollo_estimate` and saving the output from it in an object called `model`. This function uses

the `maxLik` package (Henningsen and Toomet, 2011) for classical estimation, where Bayesian estimation is discussed in Section 8. In its simplest form, this function is called via:

```
model = apollo_estimate(apollo_beta,
                        apollo_fixed,
                        apollo_probabilities,
                        apollo_inputs)
```

where we have already covered all four arguments. The function may also be called with an additional argument, namely `estimate_settings`, i.e.:

```
model = apollo_estimate(apollo_beta,
                        apollo_fixed,
                        apollo_probabilities,
                        apollo_inputs,
                        estimate_settings)
```

The additional input `estimate_settings` is a list which contains a number of settings for estimation. None of these settings is compulsory and default settings will be used for any omitted settings, or indeed all settings when calling `apollo_estimate` without the `estimate_settings` argument. The possible settings to include in this list are:

`estimationRoutine`: A character object which can take the values BFGS (for the Broyden 1970 - Fletcher 1970 - Goldfarb 1970 - Shanno 1970 algorithm), BHHH (for the Berndt-Hall-Hall-Hausman algorithm, Berndt et al. 1974 or NR (for the Newton-Raphson algorithm), where the specific syntax is for example `estimationRoutine="BFGS"` (default is set to BFGS).

`maxIterations`: An integer setting a maximum on the number of iterations (default is set to 200).

`writeIter`: A boolean variable, which, when set to TRUE, means that the values of parameters at each iteration are saved into a file in the working directory, saved as `modelName_iterations.csv` where `modelName` is as defined in `apollo_control`. This allows the user to monitor progress during estimation, which is useful especially for complex models. The use of this feature is only possible when using BFGS (default is set to TRUE).

`hessianRoutine`: A character variable indicating what routine to use for calculating the final Hessian. Possible values are `numDeriv` for the numDeriv package (Gilbert and Varadhan, 2016), `maxLik` for the same package as used in estimation, and `none` for no covariance matrix calculation. We have generally found that `numDeriv` is more reliable than `maxLik` for computing the covariance matrix, but its use may lead to issues with complex mixture models. If numDeriv fails, the code reverts to using maxLik (default is set to `numDeriv`).

`printLevel`: A numeric variable which can take levels from 0 to 3 and controls the level of detail printed out during estimation, with higher levels meaning more detail (default is set to 3).

`constraints`: A character vector defining constraints on the parameters to be applied during estimation. This is only possible with `BFGS`, and several limitations apply. Only linear constraints are supported. Only $>$, $\geq$, and $=$ constraints are allowed, and they cannot be mixed, e.g. $\beta_1 \geq 0$ and $\beta_2 = \beta_1$ cannot be applied at the same time. Fixed parameters (those whose value do not change throughout estimation) cannot be included in constraints. When writing constraints, all parameter names must be on the left side of the expression and only numeric values on the right. So, for example, if we want the following two constraints: $\beta_1 \geq 1$ and $\beta_2 \geq \beta_1$, then we wound make `contraints = c("b_1>=0", "-b_1 + b_2>=0")`. Constraints can also be defined using matrix notation using the `maxLik` (Henningsen and Toomet, 2011) coding approach.

`scaling`: A named vector of scalings to be applied to individual parameters during estimation. This can help estimation if the scale of individual parameters at convergence is very different. In classical estimation, the user can specify scales for individual model parameters. For example, if the unscaled specification involves a component $\beta_k\, x_k$ in the utility function, and if the user wishes to apply a scale of s_{β_k}, the starting value will be automatically adjusted to $\beta_k^* = \frac{1}{s_{\beta_k}} x_k$, the utility component will be adjusted to $s_{\beta_k}\beta_k^*\, x_k$ and the maximum likelihood estimation will optimise the value of β_k^*. The final model estimates will be translated to the original scale, i.e. returning estimates for β_k[4]. The aim of this process is to have the parameters that are actually used in model estimation, i.e. β_k^*, to be of a similar scale. An example of this is given for the MDCNEV model in Section 5.4.2. For Bayesian estimation, the scales are applied to the posterior parameter chains. Parameters included in `apollo_fixed` should not be included in `scaling`.

`numDeriv_settings`: A list of optional settings to be passed on to `numDeriv` when this is used for the Hessian (cf. Gilbert and Varadhan, 2016).

`bootstrapSE`: A numeric variable indicating the number of bootstrap samples to calculate standard errors. The default is 0, meaning no bootstrap standard errors will be calculated. The number must be zero or a positive integer.

`bootstrapSeed`: A numeric variable indicating the seed for the bootstrap sampling. The default is 24. If changed, it must be a positive integer. This value is only used if `bootstrapSE` $>$ 0. In general, there is no need to change this value. If the user wants to add new repetitions to a completed or interrupted estimation with bootstrap standard errors, the draws used will automatically be different.

`silent`: A boolean variable, which, when set to TRUE, means that no information is printed to the screen during estimation (default is set to FALSE).

Figure 4.8 illustrates what happens when running `apollo_estimate` on our simple MNL model. The code first checks the model specification used inside `apollo_probabilities` and reports some basic diagnostics. The validation and diagnostic steps are skipped if the user has set a value of TRUE for `noValidation` or `noDiagnostics`, respectively, in `apollo_control`. For MNL, the

[4]When using scaling in Bayesian estimation in *Apollo*, not all estimates are returned to their original scale after estimation. Indeed, the scaling is applied to the parameter chains directly, and producing scaled values for the underlying Normals is not convenient. We thus report the scaled outputs only for the non-random parameters, the random parameters after transformation to the actual distributions used, and the posterior means.

checks include for example ensuring that no unavailable alternatives are chosen. The code also checks if any parameters are included in `apollo_beta` that do not influence the calculation of the model likelihood, or if the probabilities of the model are zero at the starting values. This is followed by the main estimation process and finally the calculation of the Hessian. Prior to that step, which can take a long time in complex models (and may fail), the code also prints out the final estimates.

We can see from Figure 4.8 that the estimation uses minimisation of the negative of the log-likelihood, hence the positive values, which is of course equivalent to maximisation of the log-likelihood itself.

Model estimation creates an object of the `model` type. The contents of this depend on the specific type of model used, as well as on the estimation routine. Possible contents of this are shown in Table D.2 in Appendix D, and described in more detail in Appendix E.

Model estimation is the most likely step during which failures are encountered when working with the *Apollo* package. These could be either caused by errors in using the `R` syntax, resulting in generic `R` error messages, or errors made in the use of the various *Apollo* functions, leading to more specific error or warning messages. Not all warnings will be terminal and the code will continue to run and report warnings after completion. It is in this case entirely possible that the estimation has reached an acceptable solution with the warning messages for example being a result of the estimation process trying parameter values that lead to numeric issues in some iterations.

If a user runs the entire script contained in a model file including any post-estimation processes in one go, then errors during estimation will cause further problems in the steps that follow, but the reporting of those problems will likely become less intuitive further down the line. The user should in that case return to the first error message obtained and identify the cause of this and remedy it in the code. In general, running the code section by section is advisable to avoid this issue as far as possible.

The outcomes of model estimation are saved in a list called `model`, which contains amongst other things the estimates (`model$estimates`) and the classical and robust covariance matrices (`model$varcov` and `model$robvarcov`). The robust covariance matrix is computed using the 'sandwich' estimator (cf. Huber, 1967), which is defined as:

$$S = (-H)^{-1} B (-H)^{-1} \tag{4.5}$$

where H is the Hessian matrix, i.e. the matrix of second derivatives of the log likelihood function with respect to the model parameters to be estimated, and B is the Berndt-Hall-Hall-Hausman (BHHH) matrix (Berndt et al., 1974), defined as the matrix which has in cell jk the value

$$B_{j,k} = \sum_n L_{j,n} L_{kn} \tag{4.6}$$

where $L_{j,n}$ is the first derivative with respect to model parameter j of the contribution to the log-likelihood function from observation n.

In purely cross-sectional estimation, the sandwich estimator corrects the standard errors for general mis-specification of the model. In a panel specification, it can additionally corrects the

standard errors to accommodate the panel nature. It has long been recognised (see e.g. Louviere and Woodworth, 1983) that individual choices for the same respondent are not independent, and that the information content of N respondents each giving T responses is usually considerably less than that of NT respondents each giving a single response. As a result, the estimates of accuracy given by cross-sectional modelling are incorrect, i.e. such models are likely to produce biased standard errors, with the expectation being a downwards bias. When there is no explicit retention of information between observations, H will be identical whether the data is considered as a panel or not. However, although the first derivative components used in calculating B are the same for the panel and the non-panel cases, $B_j = \sum_{n,t} L_{j,n,t} = \sum_n (\sum_t L_{j,n,t})$, the calculation procedure for S is different for the panel case and the non-panel cases, because the specification of an "observation" is different. Specifically, if we treat the data as being panel observations, we make the calculation:

$$B_{j,k} = \sum_n L_{j,n} L_{k,n} = \sum_n \left(\sum_t L_{j,n,t} \right) \left(\sum_t L_{k,n,t} \right) \tag{4.7}$$

whereas, if we treat panel data as being independent, the calculation is:

$$B_{j,k} = \sum_n \sum_t L_{j,n,t} L_{k,n,t} \tag{4.8}$$

which is clearly different and will typically give a matrix with larger components on the diagonal, so that inverting the matrix will (incorrectly) indicate smaller estimation errors.

4.7 Reporting and saving results

Now that after completing model estimation, the user can output the results to the console (screen) and/or a set of different output files. Two separate functions are used for this, namely `apollo_modelOutput` for output to the screen, and `apollo_saveOutput` for output to files. These two commands do not return an object as output, i.e. are called without an object to assign the output to. In their default versions, these functions are called with only the `model` object as input, i.e.:

```
apollo_modelOutput(model)
```

and

```
apollo_saveOutput(model)
```

In addition, it is possible to call both functions with an additional argument that is a list of settings, i.e.

```
apollo_modelOutput(model,
                   modelOutput_settings)
```

and

```
apollo_saveOutput(model,
                  saveOutput_settings)
```

The two lists `modelOutput_settings` and `saveOutput_settings` have a number of arguments that are all optional, namely:

- `printFixed`: If set to TRUE, the code will print fixed parameters as well as estimated parameters (default is TRUE for `apollo_modelOutput` and `apollo_saveOutput`).
- `printClassical`: If set to TRUE, the code will output classical standard errors as well as robust standard errors, computed using the sandwich estimator. This setting then also affects the reporting of t-ratios, p-values and covariance/correlation matrices. If the computation of classical standard errors fails for some parameters, the user is alerted to this even if classical standard errors are not reported (default is TRUE for `apollo_modelOutput` and `apollo_saveOutput`).
- `printPVal`: If set to 0, p-values are not reported. If set to 1, then one-sided p-values are reported. If set to 2, then two-sided p-values are reported (default value is 0 for both `apollo_modelOutput` and `apollo_saveOutput`).
- `printT1`: if set to TRUE, t-ratios against 1 are reported in addition to t-ratios against 0, where this is useful for Nested Logit models and for multipliers (default is FALSE for `apollo_modelOutput` and `apollo_saveOutput`).
- `printDataReport`: If set to TRUE, a summary of each model's dependant variable is reported (default is TRUE for `apollo_modelOutput` and `apollo_saveOutput`).
- `printModelStructure`: If set to TRUE, a summary of the model structure is reported. Only some models report their model structure: NL, CNL and MDCNEV (default is TRUE for `apollo_modelOutput` and `apollo_saveOutput`).
- `printCovar`: If set to TRUE, the covariance matrix of parameters is reported (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`).
- `printCorr`: if set to TRUE, the correlation matrix of parameters is reported (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`).
- `printOutliers`: If set to TRUE, the 20 worst outliers in terms of lowest average probabilities for the chosen alternative are reported (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`). Alternatively, a scalar can be provide to use instead of 20 to change the number of outliers reported.
- `printChange`: If set to TRUE, the changes from the starting values are reported for the estimated parameters (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`).
- `printFunctions`: If set to TRUE, a copy of `apollo_probabilities`, `apollo_randomCoeff`, `apollo_lcPars`, `apollo_control`, and `estimate_settings$scaling`, are printed, as well as a list of the methods attempted to calculate the Hessian matrix (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`)

The main outputs controlled by the above settings will determine what `apollo_saveOutput` writes

into the main output file, which will be called modelName_output.txt where modelName is as defined in apollo_control. When saving outputs to files, saveOutput_settings list has five additional possible settings, namely:

saveEst: If set to TRUE, the code will save a *csv* file with the parameter estimates, standard errors and t-ratios, saved as modelName_estimates.csv (default is TRUE).
saveCov: If set to TRUE, a *csv* file will be produced with the covariance matrix, saved as modelName_robcovar.csv, where, if printClassical==TRUE, a separate file will be produced with the classical covariance matrix, saved as modelName_covar.csv (default is TRUE).
saveCorr: If set to TRUE, a *csv* file will be produced with the correlation matrix, saved as modelName_robcorr.csv, where, if printClassical==TRUE, a separate file will be produced with the classical correlation matrix, saved as modelName_corr.csv (default is TRUE).
saveModelObject: If set to TRUE, an output file of the *rds* (an R format) will be produced containing the model object, saved as modelName_model.rds (default is TRUE).
writeF12: If set to TRUE, the code will produce an F12 file, which is an output format used by the ALogit software (ALogit, 2016)[5], saved as modelName.f12 (default is FALSE).

It is worth noting that if a user has previously run apollo_saveOutput from the same model, the old output files will be renamed rather than overwritten.

An example of the on screen output is shown in Figure 4.9. For apollo_saveOutput, a text file containing output using the above settings will be produced, using a filename corresponding to apollo_control$modelName. The default settings imply a more verbose output for the log file as opposed to the on screen output, in addition to files with estimates, covariance matrices etc, unless instructed not to. The majority of the output in Figure 4.9 is self-explanatory. The output reports starting and final log-likelihood, and where appropriate (model dependent), also the log-likelihood at zero values for all parameters. A number of goodness-of-fit statistics are included. For discrete choice models, *Apollo* reports the ρ^2 measure, given as:

$$\rho^2 = 1 - \frac{LL\left(\hat{\theta}\right)}{LL\left(0\right)} \tag{4.9}$$

as well as the adjusted ρ^2, with:

$$\bar{\rho^2} = 1 - \frac{LL\left(\hat{\theta}\right) - K}{LL\left(0\right)}, \tag{4.10}$$

where K is the number of estimated parameters. In addition, for all models, *Apollo* reports the Akaike Information Criterion (AIC), given by:

$$AIC = -2LL\left(\hat{\theta}\right) + 2K, \tag{4.11}$$

[5]This is file containing all key model outputs. It is also produced by Biogeme and ALogit provides a shell to compare the results across models using these files, which can come from different estimation packages. See www.alogit.com

with K being number of estimated parameters, and the Bayesian Information Criterion (BIC), given by:

$$BIC = -2LL\left(\hat{\theta}\right) + K\ln\left(N\right), \tag{4.12}$$

where N is the number of observations in the data.

In addition, *Apollo* reports the number of estimated parameters, the estimation time (divided into subcomponents) and iterations taken, as well as the eigenvalue of the Hessian that is closest to zero. Small values can indicate convergence issues. A special warning message is displayed if some of the eigenvalues are positive.

`apollo_attach`

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Create alternative specific constants and coefficients using interactions with socio-demographics
  asc_bus_value    = asc_bus  + asc_bus_shift_female * female
  asc_air_value    = asc_air  + asc_air_shift_female * female
  asc_rail_value   = asc_rail + asc_rail_shift_female * female
  b_tt_car_value   = b_tt_car + b_tt_shift_business * business
  b_tt_bus_value   = b_tt_bus + b_tt_shift_business * business
  b_tt_air_value   = b_tt_air + b_tt_shift_business * business
  b_tt_rail_value  = b_tt_rail + b_tt_shift_business * business
  b_cost_value     = ( b_cost +  b_cost_shift_business * business ) * ( income / mean_income ) ^
      ↪ cost_income_elast

  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V = list()
  V[['car']]  = asc_car        + b_tt_car_value  * time_car                              + b_cost_value *
      ↪ cost_car
  V[['bus']]  = asc_bus_value  + b_tt_bus_value  * time_bus  + b_access * access_bus  + b_cost_value *
      ↪ cost_bus
  V[['air']]  = asc_air_value  + b_tt_air_value  * time_air  + b_access * access_air  + b_cost_value *
      ↪ cost_air  + b_no_frills * ( service_air == 1 )  + b_wifi * ( service_air == 2 )  + b_food * (
      ↪ service_air == 3 )
  V[['rail']] = asc_rail_value + b_tt_rail_value * time_rail + b_access * access_rail + b_cost_value *
      ↪ cost_rail  + b_no_frills * ( service_rail == 1 ) + b_wifi * ( service_rail == 2 ) + b_food * (
      ↪ service_rail == 3 )

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives = c(car=1, bus=2, air=3, rail=4),
    avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar    = choice,
    V            = V
  )

  ### Compute probabilities using MNL model
  P[["model"]] = apollo_mnl(mnl_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 4.7: The `apollo_probabilities` function: example for MNL model

```
model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

Testing likelihood function...

Overview of choices for MNL model component :
                                    car     bus     air     rail
Times available                 5446.00 6314.00 5264.00 6118.00
Times chosen                    1946.00  358.00 1522.00 3174.00
Percentage chosen overall         27.80    5.11   21.74   45.34
Percentage chosen when available  35.73    5.67   28.91   51.88

Pre-processing likelihood function...

Testing influence of parameters..................
Starting main estimation
Initial function value: -6359.334
Initial gradient value:
              asc_bus                asc_air               asc_rail
           -503.34620              12.28335             854.39711
 asc_bus_shift_female  asc_air_shift_female asc_rail_shift_female
           -230.62587              26.58006             432.55875
            b_tt_car               b_tt_bus               b_tt_air
       -124003.22221         -185262.48861            2945.11909
           b_tt_rail    b_tt_shift_business               b_access
        127028.10185          -159805.02716            5306.38628
               b_cost b_cost_shift_business      cost_income_elast
          9748.82869           22058.72104            -309.78035
               b_wifi                b_food
           630.32976             233.20089
initial  value 6359.334287
iter   2 value 5983.261941
...
iter  25 value 4830.944739
final  value 4830.944739
converged
Estimated parameters:
                         Estimate
asc_car                  0.000000
asc_bus                  0.286717
asc_air                 -0.903270
asc_rail                -2.092578
asc_bus_shift_female     0.340222
asc_air_shift_female     0.268170
asc_rail_shift_female    0.189612
b_tt_car                -0.013107
b_tt_bus                -0.021266
b_tt_air                -0.016578
b_tt_rail               -0.007051
b_tt_shift_business     -0.006234
b_access                -0.021152
b_cost                  -0.076187
b_cost_shift_business    0.033379
cost_income_elast       -0.613842
b_no_frills              0.000000
b_wifi                   1.026703
b_food                   0.422063

Computing covariance matrix using analytical gradient.
 0%....25%....50%....75%....100%
Negative definite Hessian with maximum eigenvalue: -2.818775
Computing score matrix...
Calculating LL(0)...
Calculating LL of each model component...
```

Figure 4.8: Running `apollo_estimate` on MNL model

```
apollo_modelOutput(model)

Model run using Apollo for R, version 0.2.4 on Darwin by stephane.hess
www.ApolloChoiceModelling.com

Model name                            : Apollo_example_3
Model description                     : MNL model with socio-demographics on mode choice SP data
Model run at                          : 2021-02-04 15:55:08
Estimation method                     : bfgs
Model diagnosis                       : successful convergence
Number of individuals                 : 500
Number of rows in database            : 7000
Number of modelled outcomes           : 7000

Number of cores used                  :  1
Model without mixing

LL(start)                             : -6359.334
LL(0)                                 : -8196.021
LL(final)                             : -4830.945
Rho-square (0)                        :  0.4106
Adj.Rho-square (0)                    :  0.4085
AIC                                   :  9695.89
BIC                                   :  9812.4


Estimated parameters                  :  17
Time taken (hh:mm:ss)                 :  00:00:10.49
     pre-estimation                   :  00:00:2.73
     estimation                       :  00:00:2.63
     post-estimation                  :  00:00:5.13
Iterations                            :  27
Min abs eigenvalue of Hessian         :  2.818775

Estimates:
                          Estimate        s.e.   t.rat.(0)    Rob.s.e. Rob.t.rat.(0)
asc_car                   0.000000          NA          NA          NA          NA
asc_bus                   0.286717    0.582949      0.4918    0.549508      0.5218
asc_air                  -0.903270    0.373537     -2.4182    0.361569     -2.4982
asc_rail                 -2.092578    0.353359     -5.9220    0.351090     -5.9602
asc_bus_shift_female      0.340222    0.132783      2.5622    0.145285      2.3418
asc_air_shift_female      0.268170    0.091506      2.9306    0.095283      2.8145
asc_rail_shift_female     0.189612    0.073759      2.5707    0.078168      2.4257
b_tt_car                 -0.013107  7.3440e-04    -17.8471  7.6775e-04    -17.0718
b_tt_bus                 -0.021266    0.001598    -13.3111    0.001518    -14.0096
b_tt_air                 -0.016578    0.002774     -5.9771    0.002671     -6.2074
b_tt_rail                -0.007051    0.001811     -3.8930    0.001765     -3.9960
b_tt_shift_business      -0.006234  6.0073e-04    -10.3773  5.9078e-04    -10.5522
b_access                 -0.021152    0.002865     -7.3841    0.002706     -7.8155
b_cost                   -0.076187    0.002097    -36.3300    0.002095    -36.3619
b_cost_shift_business     0.033379    0.002739     12.1844    0.002572     12.9783
cost_income_elast        -0.613842    0.030094    -20.3976    0.030606    -20.0566
b_no_frills               0.000000          NA          NA          NA          NA
b_wifi                    1.026703    0.056142     18.2876    0.057815     17.7586
b_food                    0.422063    0.055027      7.6702    0.056464      7.4749
```

Figure 4.9: On screen output obtained using `apollo_modelOutput` for MNL model

Chapter 5

Other model components

In Chapter 4, we gave a detailed overview of the approach to specifying and estimating models in the *Apollo* package. In this section, we look at the use of other model components, thus replacing the part of the code discussed in Section 4.5.2. We discuss ready-to-use functions for a number of commonly used models. We cover structures belonging the family of random utility models as well as those that do not. We then look at models that are not for discrete choice, looking at ordered choice data as well as discrete-continuous data. We finally explain how a user can add his/her own model functions.

5.1 Other RUM-consistent discrete choice models

5.1.1 Nested Logit

For the Nested Logit (NL) model (Daly and Zachary, 1978; McFadden, 1978; Williams, 1977), we adopt the efficient implementation of Daly (1987) but adapt it to the more commonly used version which divides the utilities by the nesting parameter in the within nest probabilities (see the discussions in Train 2009, chapter 4, and Koppelman and Wen 1998). Let us assume we have a nesting structure with three levels and that alternative i falls into nest o_m on the lowest level of nesting, which itself is a member of nest m on upper level of nesting, with m being in the root nest. Using λ for nesting parameters, we would then have that $0 < \lambda_{o_m} \leq \lambda_m \leq \lambda_r \leq 1$. The probability[1] of person n choosing alternative i in choice situation t is then given by:

$$P_{i,n,t} = P_{m,n,t}\, P_{(o_m|m),n,t}\, P_{(i|o_m),n,t} \tag{5.1}$$

[1] For the sake of simplicity of notation, we assume here that all alternatives are available in every choice situation for every person. In the code, we of course allow for departures from this assumption.

where

$$P_{(i|o_m),n,t} = \frac{e^{(\frac{V_{i,n,t}}{\lambda_{o_m}})}}{\sum_{j \in o_m} e^{(\frac{V_{j,n,t}}{\lambda_{o_m}})}} \tag{5.2}$$

$$P_{(o_m|m),n,t} = \frac{e^{(\frac{\lambda_{o_m}}{\lambda_m} I_{o_m,n,t})}}{\sum_{l_m=1}^{M_m} e^{(\frac{\lambda_{l_m}}{\lambda_m} I_{l_m,n,t})}} \tag{5.3}$$

$$P_{m,n,t} = \frac{e^{(\frac{\lambda_m}{\lambda_r} I_{m,n,t})}}{\sum_{m=1}^{M} e^{(\frac{\lambda_l}{\lambda_r} I_{l,n,t})}} \tag{5.4}$$

with

$$I_{m,n,t} = \log \sum_{l_m=1}^{M_m} e^{(\frac{\lambda_{l_m}}{\lambda_m} I_{l_m,n,t})} \tag{5.5}$$

$$I_{o_m,n,t} = \log \sum_{j \in o_m} e^{(\frac{V_{j,n,t}}{\lambda_{o_m}})}, \tag{5.6}$$

where $0 < \lambda_{o_m} \leq \lambda_m \leq \lambda_r \leq 1$. For normalisation, we set $\lambda_r = 1$, i.e. using normalisation at the top.

In the efficient implementation of Daly (1987), we then work in logs, where we define a set of elementary alternatives $\boldsymbol{E}$ and a tree function $\boldsymbol{t}$. The tree function gives us a set of composite nodes $\boldsymbol{C} = (\boldsymbol{t}(j), \boldsymbol{t}(\boldsymbol{t}(j)), \ldots \forall j \in \boldsymbol{E})$. For each elementary alternative, there is a single path up to the root r, where, for alternative i, this is given by: $\boldsymbol{A}(i, r, \boldsymbol{t}) = (i, \boldsymbol{t}(i), \boldsymbol{t}(\boldsymbol{t}(i)) \ldots r)$. We then have that:

$$log(P_{i,n,t}) = \sum_{a \in \boldsymbol{A}(j,r,\boldsymbol{t})} \frac{1}{\lambda_{\boldsymbol{t}(a)}} \left(V_{a,n,t} - \widetilde{V_{\boldsymbol{t}(a),n,t}} \right). \tag{5.7}$$

where $\lambda_{\boldsymbol{t}(a)}$ is the nesting parameter for the nest that contains a, and for any non-elementary elements a, we have:

$$\widetilde{V_{a,n,t}} = \lambda_a \, log \sum_{l \in a} exp(\frac{V_{l,n,t}}{\lambda_a}) \tag{5.8}$$

where $l \in a$ gives all the elements contained in a, which can be a mixture of nests and elementary alternatives. For normalisation, we set $\lambda_r = 1$, and for consistency with utility maximisation, we then have that $0 < \lambda_a \leq 1$, $\forall a$ and $\lambda_a \leq \lambda_{\boldsymbol{t}(a)}$, i.e. the λ terms in a given chain $\boldsymbol{A}(j, r, \boldsymbol{t})$ decrease as we go from the root down the tree.

In the actual user syntax, we adopt an approach inspired by ALogit[2] (ALogit, 2016) where a user needs to specify a chain going from the root to each of the elementary alternatives. To illustrate this, we look at an example on the data described in Section 3.1, where we implement a three-level NL (`Apollo_example_5.r`). A simpler two-level model is available in `Apollo_example_4.r`.

[2] www.alogit.com

```
### Specify nests for NL model
nlNests      = list(root=1, PT=lambda_PT, fastPT=lambda_fastPT)

### Specify tree structure for NL model
nlStructure= list()
nlStructure[["root"]]    = c("car","PT")
nlStructure[["PT"]]      = c("bus","fastPT")
nlStructure[["fastPT"]] = c("air","rail")

### Define settings for NL model
nl_settings <- list(
  alternatives = c(car=1, bus=2, air=3, rail=4),
  avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
  choiceVar    = choice,
  V            = V,
  nlNests      = nlNests,
  nlStructure  = nlStructure
)

### Compute probabilities using NL model
P[["model"]] = apollo_nl(nl_settings, functionality)
```

Figure 5.1: Nested Logit implementation (extract)

Note that *Apollo* does not impose any constraints on the nesting parameters, and it is the user's role to ensure that the starting values and final estimates are consistent with theory.

In the first level of the tree, alternatives are divided into public transport (PT) alternatives and car, while the PT alternatives are then further split into a nest containing rail and air (fastPT), where bus is on its own. To estimate this model, we specify two additional parameters compared to the MNL model in Section 4.5.2 in the `apollo_beta` vector, say `lambda_PT` and `lambda_fastPT`.

Just like `apollo_mnl`, the `apollo_nl` function is called as follows:

```
P[["model"]] = apollo_nl(nl_settings,
                         functionality)
```

The list `nl_settings` contains all the same elements as for MNL, i.e. the compulsory inputs `alternatives`, `avail`, `choiceVar`, `V` and the optional inputs `rows` and `componentName`. For further details on these, the reader is referred back to Section 4.5.2. For Nested Logit, the list `nl_settings` needs to contain two additional arguments, namely:

`nlNests`: A named vector containing the names of the nests and the associated structural parameters λ. For each λ, we give the name of the associated parameter. This list needs to include the `root`, which is the only nest for which the choice of name is not free for the user to determine.

`nlStructure`: A list containing one element per nest, where each element is a vector with the names of the contents of that nest, which can itself be a mix of nests and elementary alternatives.

In our example, as illustrated in Figure 5.1 (where we do not show the definition of alternatives, availabilities, choices and utilities, as these remain the same as in the MNL model in Section 4.5.2), we have three nests, where this includes the root. The order of elements is of no importance as they are identified by the nest names, yet, for consistency, using the same order as in the model structure which follows is advisable. For each nest, we give the nesting parameter, using the

```
Nest: root (1)
|--------Alternative: car
'-Nest: PT (0.6952)
   |-----Alternative: bus
   '-Nest: fastPT (0.5861)
      |-Alternative: air
      '-Alternative: rail
```

Figure 5.2: Nested Logit tree structure after estimation

parameter names previously defined in `apollo_beta`. The final step in the definition of the NL model is a call to `apollo_nl` with the appropriate inputs. Extensive checks are performed by this function, notably ensuring that for each alternative, there is exactly one chain from the root to the bottom of the tree.

After estimation, the model reports estimates for all parameters, as for any model, but in addition prints out (except if `apollo_control$noDiagnostics==FALSE`) the resulting tree structure with the estimated nesting parameters in brackets (and this is repeated with `apollo_modelOutput` and `apollo_saveOutput` if `printDiagnostics` is set to TRUE in their respective settings). In the case of our example, shown in Figure 5.2, we see that, as intended, there is a direct link from the root to the car alternative, while all other alternatives are nested in a public transport nest, with a further layer of nesting for rail and air within that nest. The nesting parameters also follow the required decreasing trend when going from the root down the tree.

5.1.2 Cross-nested Logit

For our implementation of the Cross-nested Logit (CNL) model (Vovsha, 1997), we follow the "Generalised Nested Logit" (GNL) model of Wen and Koppelman (2001), with all nesting parameters freely estimated, and the constraint on the allocation parameters (showing the membership of alternative j in nest m) that $0 \leq \alpha_{j,m} \leq 1$, $\forall j, m$ and $\sum_j \alpha_{j,m} = 1$, $\forall m$. Only two-level versions of CNL are available through the `apollo_cnl` function, i.e. one layer of nests below the root, with the membership of a non-root nest being made up entirely of elementary choice alternatives.

In our implementation, each alternative needs to fall into at least one nest on the second level of the tree, where this can be a single alternative nest. We then have M nests, S_1 to S_M, where $\alpha_{j,m}$ represents allocation of alternative j to nest S_m. We have that $0 \leq \alpha_{j,m} \leq 1 \ \forall \, j, m$ and $\sum_{m=1}^{M} \alpha_{j,m} = 1 \ \forall \, j$. The probabilities are then given by a sum over nests:

$$P_{i,n,t} = \sum_{m=1}^{M} P_{S_m,n,t} \, P_{(i|S_m),n,t} \tag{5.9}$$

where

$$P_{S_m,n,t} = \frac{\left(\sum_{j\in S_m}\left(\alpha_{j,m}e^{V_{j,n,t}}\right)^{\frac{1}{\lambda_m}}\right)^{\lambda_m}}{\sum_{l=1}^{M}\left(\sum_{j\in S_l}\left(\alpha_{j,l}e^{V_{j,n,t}}\right)^{\frac{1}{\lambda_l}}\right)^{\lambda_l}} \tag{5.10}$$

$$P_{(i|S_m),n,t} = \frac{\left(\alpha_{i,m}e^{V_{i,n,t}}\right)^{\frac{1}{\lambda_m}}}{\sum_{j\in S_m}\left(\alpha_{j,m}e^{V_{j,n,t}}\right)^{\frac{1}{\lambda_m}}} \tag{5.11}$$

Just like `apollo_mnl` and `apollo_nl`, the `apollo_nl` function is called as follows:

```
P[["model"]] = apollo_cnl(cnl_settings,
                          functionality)
```

The list `cnl_settings` contains all the same elements as for MNL, i.e. the compulsory inputs `alternatives`, `avail`, `choiceVar`, `V` and the optional inputs `rows` and `componentName`. For further details on these, the reader is referred back to Section 4.5.2. For Cross-nested Logit, the list `cnl_settings` needs to contain two additional arguments, namely:

`cnlNests`: A named vector containing the names of the nests and the associated structural parameters λ. For each λ, we give the name of the associated parameter. Unlike in `apollo_nl`, the `root` is not included for `apollo_cnl` as only two-level structures are used.

`cnlStructure`: A matrix showing the allocation of alternatives to nests, with one row per nest and one column per alternative, using the same ordering as in `alternatives` and `cnlNests`.

For our implementation example on the simple mode choice data, we define a structure where air is nested together with rail (fast PT), and bus is nested together with rail (ground-based PT), but there is no joint nest membership for bus and air. Finally, car is nested on its own. This means that the only alternative for which allocation parameters need to be estimated is the rail alternative, where we have that $\alpha_{rail,fastPT} + \alpha_{rail,groundPT} = 1$, with both $\alpha_{rail,fastPT}$ and $\alpha_{rail,groundPT}$ being constrained to be between 0 and 1. Imposing constraints directly on the estimation routine is inefficient and can affect the standard error calculations. We instead recommend the use of a logistic transform, where, with alternative j having an estimated allocation parameter for M different nests, we have that, for nest m:

$$\alpha_{j,m} = \frac{e^{(\alpha_{0,j,m})}}{\sum_{l=1}^{M} e^{(\alpha_{0,j,l})}}, \tag{5.12}$$

where a normalisation is required, for example fixing $\alpha_{0,j,1} = 0$.

Like in the NL model, we define a vector of names for the nests, `cnlNests`, which defines the names of the nests and the associated structural parameters λ, using the parameter names previously defined in `apollo_beta`.

In our example (`Apollo_example_6.r`), as illustrated in Figure 5.3 (where we do not show the definition of alternatives, availabilities, choices and utilities, as these remain the same as in the MNL and NL models), we have three nests, one for air and rail (fastPT), one for bus and

rail (groundPT), and one for car, which is nested on its own. The nesting parameter for the car nest is set to 1 given this is a single alternative nest. Our CNL implementation is limited to a two-level structure, and all elementary alternatives need to belong to at least one nest below the root, even if these are single alternative nests. This means that all nests defined by the user are automatically positioned below the root and the root is thus not included in the definition of the nest or tree structure given by the user. Note that, as for Nested Logit, *Apollo* does not impose any constraints on the nesting parameters, and it is the user's role to ensure that the starting values and final estimates are consistent with theory. However, *Apollo* will not estimate a model where, at the starting values, the allocation parameters for a single alternative sum to a value different from 1.

```
### Specify nests for CNL model
cnlNests = list(fastPT=lambda_fastPT, groundPT=lambda_groundPT, car=1)

## Specify nest allocation parameters for alternatives included in multiple nests
alpha_rail_fastPT   = exp(alpha0_rail_fastPT)/(exp(alpha0_rail_fastPT) + exp(alpha0_rail_groundPT))
alpha_rail_groundPT = 1 - alpha_rail_fastPT

## Specify tree structure, showing membership in nests (one row per nest, one column per alternative)
cnlStructure      = matrix(0, nrow=length(cnlNests), ncol=length(V))
cnlStructure[1,]  = c( 0,  0, 1, alpha_rail_fastPT  ) # fastPT
cnlStructure[2,]  = c( 0,  1, 0, alpha_rail_groundPT) # groundPT
cnlStructure[3,]  = c( 1,  0, 0, 0                  ) # car

### Define settings for CNL model
cnl_settings <- list(
  alternatives = c(car=1, bus=2, air=3, rail=4),
  avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
  choiceVar    = choice,
  V            = V,
  cnlNests     = cnlNests,
  cnlStructure = cnlStructure
)

### Compute probabilities using CNL model
P[["model"]] = apollo_cnl(cnl_settings, functionality)
```

Figure 5.3: Cross-nested Logit implementation (extract)

For the allocation or nest membership parameters, car and bus both fall into one nest exactly, so they have $\alpha_{j,m} = 1$ for the specific nest m they fall into and the remaining ones are set to zero. For rail, we define $\alpha_{rail,fastPT} = \frac{e^{(\alpha_{0,rail,fastPT})}}{e^{(\alpha_{0,rail,fastPT})}+e^{(\alpha_{0,rail,groundPT})}}$, and obviously $\alpha_{rail,groundPT} = 1 - \alpha_{rail,fastPT}$, while we use the normalisation that $\alpha_{0,rail,groundPT} = 0$, by including the parameters `alpha0_rail_fastP` in `apollo_fixed`. The crucial part of the definition of a CNL model is again the actual model structure, which in our code is again called `cnlStructure`, where this is now made up of a matrix with one row per nest, and one alternative per column, where the entry in a given cell corresponds to the appropriate allocation parameter. The order of rows and columns needs to be consisted with the order in `cnlNests` and `alternatives`, respectively.

In the model output, the code reports the resulting tree structure with the estimated allocation and nesting parameters. In the case of our example, shown in Figure 5.4, we see that, as intended, car, bus and air all belong to one nest only, while the estimation has shown that the split for rail is almost 50-50, with the λ parameter being smaller in the fastPT nest. In reporting the allocation parameters, the code uses the final values used inside `cnlStructure`, i.e. after the logistic transform in our example.

```
Structure for CNL model component :
             car (alpha) bus (alpha) air (alpha) rail (alpha)      lambda
fastPT nest        0.000       0.000       1.000        0.4929      0.4012
groundPT nest      0.000       1.000       0.000        0.5071      0.5285
car nest           1.000       0.000       0.000        0.0000      1.0000
```

Figure 5.4: Cross-nested Logit structure after estimation

5.2 Non-RUM decision rules for discrete choice

In this section, we present the use of two alternatives to RUM in *Apollo*, namely random regret minimisation (RRM) and Decision Field Theory (DFT).

5.2.1 Random regret minimisation (RRM)

The fundamental assumption in regret theory is that what matters is not only the realised outcome but also on what could have been obtained by selecting a different course of action. This means that the model incorporates anticipated feelings of regret that would be experienced once ex-post decision outcomes are revealed to be "unfavourable". The value of an alternative can thus only be assigned following a cross-wise evaluation of alternatives, and this is the cause for substantial increases in computational complexity with large choice sets.

In the example here, we use the standard RRM model of Chorus (2010). For more complex specifications of the model, see van Cranenburgh et al. (2015), with Apollo implementations available at https://www.advancedrrmmodels.com/.

We define the deterministic regret for alternative i ($i = 1, \ldots, I$) for respondent n in choice task t as:

$$R_{i,n,t} = \sum_{k=1}^{K} \sum_{j \neq i} \ln\left(1 + e^{\beta_k\left(x_{j,n,t,k} - x_{i,n,t,k}\right)}\right) \tag{5.13}$$

where β_k is the coefficient associated with attribute x_k, with $k = 1, \ldots, K$. The regret is informed by all the pairwise comparisons, where regret for alternative i increases whenever an alternative $j \neq i$ performs better than i on a given attribute. When using extreme value error terms, RRM models are in fact Logit models, albeit not RUM-consistent models. With the assumption of type I extreme value errors, the probability of respondent n choosing alternative i in choice task t, is now simply given by a MNL formula as:

$$P_{i,n,t} = \frac{e^{-R_{i,n,t}}}{\sum_{j=1}^{J} e^{-R_{j,n,t}}}. \tag{5.14}$$

In RRM, we minimise the regret rather than maximising the utility, and this is achieved by maximising the negative regret in Equation 5.14.

Given the above point, any of the Logit family models in *Apollo* can be used also for regret minimisation, by simply replacing the utilities (i.e. `V`) by the negative of regret. The labour

intensive part comes in specifying the regret functions for the alternatives, i.e. implementing Equation 5.13.

An example of an RRM implementation is given in Figure 5.5, where we apply a MNL (i.e. non-nested) version of RRM to the mode choice data from Section 3.1. We use a simpler implementation than in Section 4.5.2, with no socio-demographics. This example is available in `Apollo_example_7.r`.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Prepare regret components for categorical variables
  access_car  = 0
  RFrills_car = b_no_frills
  RFrills_bus = b_no_frills
  RFrills_air  = b_no_frills * ( service_air  == 1 ) + b_wifi * ( service_air  == 2 ) + b_food * (
      ↪ service_air  == 3 )
  RFrills_rail = b_no_frills * ( service_rail == 1 ) + b_wifi * ( service_rail == 2 ) + b_food * (
      ↪ service_rail == 3 )

  ### List of regret functions: these must use the same names as in mnl_settings, order is irrelevant
  R = list()
  R[['car']]  = asc_car  +
    log(1+exp(b_tt_bus*time_bus   - b_tt_car*time_car)) +
    log(1+exp(b_tt_air*time_air   - b_tt_car*time_car)) +
    log(1+exp(b_tt_rail*time_rail - b_tt_car*time_car)) +
    log(1+exp(b_cost*(cost_bus  - cost_car))) +
    log(1+exp(b_cost*(cost_air  - cost_car))) +
    log(1+exp(b_cost*(cost_rail - cost_car))) +
    log(1+exp(b_access*(access_bus - access_car))) +
    log(1+exp(b_access*(access_air - access_car))) +
    log(1+exp(b_access*(access_rail- access_car))) +
    log(1+exp(RFrills_bus  - RFrills_car))  +
    log(1+exp(RFrills_air  - RFrills_car))  +
    log(1+exp(RFrills_rail - RFrills_car))

    ...

  ### Define settings for RRM model, which is MNL with negative regret as utility
  mnl_settings <- list(
    alternatives = c(car=1, bus=2, air=3, rail=4),
    avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar    = choice,
    V            = lapply(R, "*", -1)
  )

  ### Compute probabilities using MNL model
  P[['model']] = apollo_mnl(mnl_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 5.5: Implementation of random regret MNL model

In a RUM model, only the attributes applying to a given alternative are used in the utility for that alternative, and the absence of access time for car or service quality for car and bus is of no importance. In a RRM model, we need to create these attributes given that the differences across the alternatives are used for all attributes. We next define the regret function for each alternative, where we here only show the regret for the car alternative, with a corresponding formulation applying for other modes, each time using Equation 5.13. For the alternative specific

constants (ASCs), we adopt the convention of entering them directly into the regret function rather than using Equation 5.13. In the `mnl_settings` list, we now define `V` to be the negative of `R` by multiplying each element in `R` by `-1`. We finally make the call to `apollo_mnl`.

5.2.2 Decision field theory (DFT)

Decision field theory (DFT) originates in mathematical psychology (Busemeyer and Townsend, 1992, 1993) and is very different to both RUM and RRM. The key assumption under a DFT model is that the preferences for alternatives update over time. The decision-maker considers the alternatives until they reach an internal threshold (similar to the concept of satisficing, where one of the options is deemed 'good enough') or some external threshold (i.e. some time constraint, where a decision-maker stops deliberating on the alternatives as a result of running out of time to make the decision).

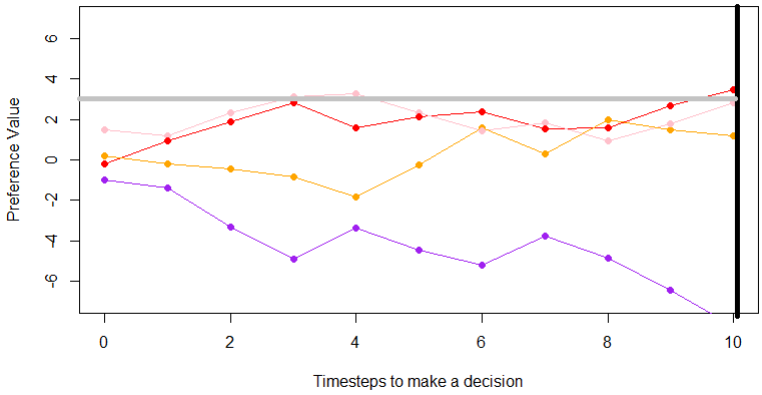

Figure 5.6: An example of a decision-maker stopping upon reaching either an internal or external threshold

An example of a decision process under DFT is given in Figure 5.6. In this particular example, the decision-maker chooses different alternatives if they make their choice after reaching an internal threshold (which is represented by the horizontal line) on the 4th preference updating timestep or if they conclude after 10 steps upon reaching a time threshold. Mathematically, DFT has been operationalised differently depending on whether internal or external thresholds are used. A full specification of DFT with internal thresholds is given by Busemeyer and Townsend (1993), while we focus here on DFT with external thresholds (c.f. Roe et al. (2001) for the first adaptation of DFT with external thresholds for multiple alternatives). For DFT with an external threshold, the preference values update stochastically as a result of the assumption that a decision-maker considers just one attribute of an alternative at each timestep. Consequently, the preference values for each alternative update iteratively:

$$P_t = S \cdot P_{t-1} + V_t, \tag{5.15}$$

where P_t is a column vector containing the preference values of each alternative i at time t. S is a feedback matrix with memory and sensitivity parameters (detailed in Equation 5.16) and V_t

is a valence vector (Equation 5.17), which varies depending on which attribute is attended to at time t. The feedback matrix used in *Apollo* is based on the definition by Hotaling et al. (2010):

$$S = I - \phi_2 \times \exp(-\phi_1 \times D^2), \tag{5.16}$$

where I is the identity matrix of size n and n is the number of alternatives. The feedback parameter has two free parameters. The first, ϕ_1, is a sensitivity parameter, which allows for competition between alternatives that are more similar (in terms of attribute values). The second, ϕ_2, is a memory parameter, which captures whether attributes considered at the start of the deliberation process or attributes considered at the end are more important. Finally, D is some measure of distance between the alternatives. In our code, we use the Euclidean distance for simplicity. Next, the valence vector can be described as:

$$V_t = C \cdot M \cdot W_t + \varepsilon_t, \tag{5.17}$$

where C is a contrast matrix used to rescale the attribute values such that they sum to zero, M is a matrix containing the attribute values for all of the alternatives, $W_t = [0..1..0]'$ with entry $k = 1$ if and only if attribute k is the attribute being attended to by the decision-maker at timestep t, and ε_t is an error term.

The implementation of DFT in *Apollo* allows for two different ways of accounting for the relative importance of attributes. A user may define attribute importance weights w_k, for each attribute, that are to be estimated and which correspond to the likelihood of a decision-maker attending to that attribute k. These however have the limitation that they must sum to one, which consequently requires the user to have a priori knowledge on the directionality of attributes (Hancock et al., 2018). Alternatively, the analyst may use 'attribute scaling coefficients'. These have many benefits (see Hancock et al. 2019 for a detailed explanation of these), including, most importantly, avoiding the limitation of having to sum to one. By instead assuming that each attribute is attended to with the same likelihood (all weights, $w_k = 1/n$), the relative importance can instead enter as a set of scaling coefficients, s_k, which are applied to the attributes before they are entered (through M in Equation 5.17) into the calculation of the valence vector at each timestep.

The error term ε is drawn from independent and identically distributed normal draws with mean 0 and a standard deviation which is an estimated parameter. Consequently, the preference values P_t converge to a multivariate normal distribution (Roe et al., 2001). To calculate the probabilities of alternatives under DFT we thus simply require the expectation and covariance of P_t (ξ_t and Ω_t, respectively). Hence, the probability of choosing alternative j from a set of J alternatives at time t is:

$$P_{DFT}\left[\max_{i \in J} P_t[i] = P_t[j]\right] = \int_{X>0} exp\left[-(X-\Gamma)'\Lambda^{-1}(X-\Gamma)/2\right] / (2\pi|\Lambda|^{0.5}) dX, \tag{5.18}$$

with X the set of differences between the preference value for the chosen alternatives and each other alternative, $X = [P_t[j] - P_t[1], ..., P_t[j] - P_t[J]]'$. Additionally, we require transformations of the expectation and covariance, $\Gamma = L\xi_t$, $\Lambda = L\Omega_t L'$, with L a matrix comprised of a column

vector of 1s and a negative identity matrix of size $J-1$ where J is the number of alternatives. The column vector of 1s is placed in the ith column where i is the chosen alternative.

An implementation of DFT is given in `apollo_dft`, which is called as:

```
P[["model"]] = apollo_dft(dft_settings,
                          functionality)
```

where `dft_settings` contains the following elements:

`alternatives`: A named vector containing the names of the alternatives, as for other discrete choice models.

`avail`: A list containing availabilities, as for other discrete choice models.

`choiceVars`: A variable indicting the column in the database which identifies the alternative chosen in a given choice situation, as for other discrete choice models.

`attrValues`: A list with attribute values for alternatives, where this list contains one list per alternative, using the names from `alternatives`. Each alternative-specific list then contains the attribute values for that alternative, with one entry per attribute, where these are all column vectors with one entry per observation. DFT requires all alternatives to have each of the specified attributes, so by default will set attribute values of zero for any attributes not provided for a given alternative. Note that attributes specified here that are not included in either `attrWeights` or `attrScalings` will be ignored.

`altStart`: A list containing a starting preference value for each alternative, using the same names as `alternatives`. As with other models, these are generally defined in `apollo_beta`, but could involve interactions with socio-demographics or be randomly distributed across individuals and/or observations.

`attrWeights`: A list of weights, with one for each attribute. These should sum to one and will be adjusted accordingly if they do not. As mentioned above, any attributes included in `attrValues` but missing from `attrWeights` will be ignored. Conversely, any attribute missing in `attrValues` but included in `attrWeights` will be created in `attrValues` but set to zero. Note that `attrWeights` should be set to 1 if `attrScalings` is provided.

`attrScalings`: A list of scaling parameters that are applied to attribute values before they are passed into M in Equation 5.17. These do not need to sum to 1 across the set of attributes. As mentioned above, any attributes included in `attrValues` but missing from `attrScalings` will be ignored. Conversely, any attribute missing in `attrValues` but included in `attrScalings` will be created in `attrValues` but set to zero. Note that `attrScalings` should be set to 1 if `attrWeights` is provided.

`procPars`: A list containing the four DFT 'process parameters'. The first of these is `error_sd`, which corresponds to the standard deviation of the error term in Equation 5.17. The second, `timesteps`, is the number of preference updating timesteps (t in Equations 5.15 and 5.17). [apollo_dft will automatically adjust the number of timesteps such that there is at least one timestep. The final process parameters are the sensitivity and process parameters, `phi1` and `phi2`, from Equation 5.16. All of these parameters can be entered as single values to be used across the dataset, or can take choice-set dependent values.

`rows`: The optional rows argument already described for the earlier models.
`componentName`: The optional argument giving a name to the model component described already for earlier models.

An example of a DFT implementation is given in Figure 5.7, where we apply a DFT model with scale parameters to the mode choice data from Section 3.1. We use an identical implementation to that of the MNL model in Section 4.5.2, with the same socio-demographics parameters. This is a key advantage of using scaling parameters (with the weights instead being fixed) in a DFT model, as it allows us to make equivalent adjustments to the parameters. This example is available in `Apollo_example_9.r`, where a simpler DFT model without covariates applied to the Swiss route choice data is available in `Apollo_example_8.r`.

Values for alternatives without a given attribute (wifi, food and access time for car, for example) are set to zero (and would be automatically set to zero if not initially provided). Additionally, DFT weights are automatically rescaled to sum to one, therefore attribute specific scalings (such as the one for the travel time coefficient in this example) are more efficiently employed through the use of attribute scaling parameters. Consequently, `attrWeights` is set to 1 in `dft_settings`.

Note that DFT process parameters can often cause identification or estimation issues (c.f. Hancock et al. 2019). Consequently, care is required, particularly when estimating DFT models on datasets where the process parameters are unlikely to have an impact, as poor initial starting values for the parameters can result in convergence to poor local optima. Here, we adjust the process parameters to aid estimation. We use exponentials to restrict the number of deliberation timesteps to be greater than 1 and the sensitivity parameter to be positive, and a logistic transform to ensure the memory parameter falls between 0 and 1. Additionally, with this data, we fix `error_sd` by including it in `apollo_fixed`. Finally, it is preferable to use non-zero starting values for all parameters.

5.3 Models for ranking, rating and continuous dependent variables

Especially when developing hybrid choice models (cf. Section 7.3, some of the dependent variables in the model will not be of the discrete choice type. We now look at how to model such dependent variables in *Apollo*.

5.3.1 Exploded Logit

Datasets may include the full ranking for alternatives, in which case an Exploded Logit model can be used. In particular, with J different alternatives for individual n in choice situation t, we may observe the ranking $R_{nt} = \langle R_{nt,1}, \ldots, R_{nt,J} \rangle$, where $R_{nt,1}$ is the index for the alternative which is ranked the highest, i.e. the choice in a simple discrete choice setting. Note that this is different from the convention where $R_{nt,j}$ is the rank of alternative j; here, $R_{nt,j}$ refers to the specific alternative ranked in j^{th} place.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Create alternative specific constants and coefficients using interactions with socio-demographics
  asc_bus_value   = asc_bus   + asc_bus_shift_female  * female
  asc_air_value   = asc_air   + asc_air_shift_female  * female
  asc_rail_value  = asc_rail  + asc_rail_shift_female * female
  b_tt_car_value  = b_tt_car  + b_tt_shift_business * business
  b_tt_bus_value  = b_tt_bus  + b_tt_shift_business * business
  b_tt_air_value  = b_tt_air  + b_tt_shift_business * business
  b_tt_rail_value = b_tt_rail + b_tt_shift_business * business
  b_cost_value    = ( b_cost  + b_cost_shift_business * business ) * ( income / mean_income ) ^
      ↪ cost_income_elast

  ### List of attribute values
  attrValues = list()
  attrValues[['car']]  = list(time=time_car , access=0           , cost=cost_car , wifi=0
      ↪                       , food=0                    )
  attrValues[['bus']]  = list(time=time_bus , access=access_bus , cost=cost_bus , wifi=0
      ↪                       , food=0                    )
  attrValues[['air']]  = list(time=time_air , access=access_air , cost=cost_air , wifi=1*(service_air
      ↪ == 2), food=1*(service_air == 3))
  attrValues[['rail']] = list(time=time_rail, access=access_rail, cost=cost_rail, wifi=1*(service_rail
      ↪ == 2), food=1*(service_rail == 3))

  ### List of initial preference values
  altStart = list()
  altStart[['car']]  = asc_car
  altStart[['bus']]  = asc_bus_value
  altStart[['air']]  = asc_air_value
  altStart[['rail']] = asc_rail_value

  ### List of attribute scaling factors
  attrScalings = list(time   = list(car = b_tt_car_value, bus = b_tt_bus_value, air = b_tt_air_value,
      ↪ rail = b_tt_rail_value),
                      access = b_acc,
                      cost   = b_cost_value,
                      wifi   = b_wifi,
                      food   = b_food)

  ### List of process parameters
  procPars  = list(
    error_sd  = p_error_sd,
    timesteps = 1+exp(p_timesteps),
    phi1      = exp(p_phi1),
    phi2      = exp(p_phi2)/(1+exp(p_phi2))
  )

  ### Define settings for DFT model component
  dft_settings <- list(
    alternatives = c(car=1, bus=2, air=3, rail=4),
    avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar    = choice,
    attrValues   = attrValues,
    altStart     = altStart,
    attrWeights  = 1,           ### Using scaling factors, so attrWeights must be set to 1.
    attrScalings = attrScalings,
    procPars     = procPars
  )

  ### Compute choice probabilities using DFT model
  P[['model']] = apollo_dft(dft_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 5.7: DFT implementation

We then have that the probability of the observed ranking is given by:

$$P_{nt} = \prod_{i=1}^{J-1} \frac{e^{\mu_i V_{R_{nt,i}}}}{\sum_{j=i}^{J} e^{\mu_i V_{R_{nt,j}}}}, \tag{5.19}$$

where this is given by a product of Logit probabilities for all but the last ranking (which is just a single alternative), where the denominator gradually omits alternatives, and where we allow for differences in scale across the stages, with an appropriate normalisation, e.g. $\mu_1 = 1$.

In *Apollo*, the Exploded Logit model is implemented in the function `apollo_el`, which is called called as follows:

```
P[["model"]] = apollo_el(el_settings,
                         functionality)
```

where the contents of `el_settings` are a little different from the earlier MNL, NL and CNL models. In particular, we have:

`alternatives:` A named vector containing the names of the alternatives, as for MNL, NL and CNL.

`avail:` A list containing availabilities, as for MNL, NL and CNL.

`choiceVars:` A list containing the names of the variables indicating the column in the database which identify the choices at each stage in the ranking, except for the final (worst) alternative. If not all alternatives are available for all individuals, then some of the later rankings will not apply for these individuals, and the user should put a value of -1 in the data for those entries. For example, if a given person only has two out of the four alternatives available, then the third and fourth ranking should be given as `-1` in the data for that individual.

`V:` A list of utilities, as for MNL, NL and CNL.

`scales:` An optional argument given by a list, with one entry per stage in the ranking, giving the scale parameter to be used in that stage.

`rows:` The optional `rows` argument already described for the earlier models.

`componentName:` The optional argument giving a name to the model component described already for earlier models.

An example using the Exploded Logit model is given in `Apollo_example_10.r`, using the drug choice data from Section 3.3. We use a dummy coded specification for the three categorical variables, along with a continuous specification for risk and cost.

The utility for alternative j in choice situation t for individual n is given by:

$$\begin{aligned} V_{j,n,t} = & \sum_{s=1}^{5} \beta_{brand_s} \cdot \left(x_{brand_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{6} \beta_{country_s} \cdot \left(x_{country_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{3} \beta_{characteristic_s} \cdot \left(x_{characteristic_{j,n,t}} == s\right) \\ & + \beta_{side\_effects} \cdot x_{side\_effects_{j,n,t}} \\ & + \beta_{price} \cdot x_{price_{j,n,t}} \end{aligned} \tag{5.20}$$

For the first three rows in Equation 5.20, one of the β parameters in each row is constrained to zero (dummy coded), and not all levels apply for each alternative, as described in Appendix B.

The implementation of the model is shown in Figure 5.8. Special care is required for the qualitative attributes. These are coded as text in the data, and one parameter needs to be associated with each level, where we impose an appropriate normalisation in `apollo_fixed` to set the parameter for one level to zero for each attribute. The levels that are included in the utility functions differ across alternatives, as reflected in the design of the survey (cf. Table B.3). The key different from an MNL model arises in the inclusion of `choiceVars` instead of `choiceVar` in `el_settings` where this differs from giving a single preferred alternative for each observation and instead giving one column for each stage in the ranking except for the final stage. We also provide scale parameters for these three stages in `el_settings$scales`, where the scale for the first stage is normalised to 1.

5.3.2 Ordered Logit and Ordered Probit

For ordinal dependent variables, the function `apollo_ol` provides an implementation of the Ordered Logit model, while `apollo_op` provides an implementation of the Ordered Probit model. These models are used where, with $Y_{n,t}$ being the observed value for the dependent variable for the tth observation for individual n, $Y_{n,t}$ can take S different possible values, going from $s = 1, \ldots, S$.

Ordered Logit

In an Ordered Logit (OL) model, the probability of observing value s is given by:

$$P_{Y_{n,t}=s} = \frac{e^{\tau_s - V_{n,t}}}{1 + e^{\tau_s - V_{n,t}}} - \frac{e^{\tau_{s-1} - V_{n,t}}}{1 + e^{\tau_{s-1} - V_{n,t}}} \tag{5.21}$$

The likelihood of the observed value $Y_{n,t}$ is then given by:

$$L_{Y_{n,t}} = \sum_{s=1}^{S} \delta_{(Y_{n,t}=s)} \left[\frac{e^{\tau_s - V_{n,t}}}{1 + e^{\tau_s - V_{n,t}}} - \frac{e^{\tau_{s-1} - V_{n,t}}}{1 + e^{\tau_{s-1} - V_{n,t}}} \right], \tag{5.22}$$

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### List of utilities: these must use the same names as in el_settings, order is irrelevant
  V = list()
  V[['alt1']] = ( b_brand_Artemis*(brand_1=="Artemis") + b_brand_Novum*(brand_1=="Novum")
                + b_country_CH*(country_1=="Switzerland") + b_country_DK*(country_1=="Denmark") +
                    ↪ b_country_USA*(country_1=="USA")
                + b_char_standard*(char_1=="standard") + b_char_fast*(char_1=="fast acting") +
                    ↪ b_char_double*(char_1=="double strength")
                + b_risk*side_effects_1
                + b_price*price_1)
  V[['alt2']] = ( b_brand_Artemis*(brand_2=="Artemis") + b_brand_Novum*(brand_2=="Novum")
                + b_country_CH*(country_2=="Switzerland") + b_country_DK*(country_2=="Denmark") +
                    ↪ b_country_USA*(country_2=="USA")
                + b_char_standard*(char_2=="standard") + b_char_fast*(char_2=="fast acting") +
                    ↪ b_char_double*(char_2=="double strength")
                + b_risk*side_effects_2
                + b_price*price_2)
  V[['alt3']] = ( b_brand_BestValue*(brand_3=="BestValue") + b_brand_Supermarket*(brand_3=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_3=="PainAway")
                + b_country_USA*(country_3=="USA") + b_country_IND*(country_3=="India") + b_country_RUS
                    ↪ *(country_3=="Russia") + b_country_BRA*(country_3=="Brazil")
                + b_char_standard*(char_3=="standard") + b_char_fast*(char_3=="fast acting")
                + b_risk*side_effects_3
                + b_price*price_3 )
  V[['alt4']] = ( b_brand_BestValue*(brand_4=="BestValue") + b_brand_Supermarket*(brand_4=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_4=="PainAway")
                + b_country_USA*(country_4=="USA") + b_country_IND*(country_4=="India") + b_country_RUS
                    ↪ *(country_4=="Russia") + b_country_BRA*(country_4=="Brazil")
                + b_char_standard*(char_4=="standard") + b_char_fast*(char_4=="fast acting")
                + b_risk*side_effects_4
                + b_price*price_4 )

  ### Define settings for exploded logit
  el_settings = list(
    alternatives = c(alt1=1, alt2=2, alt3=3, alt4=4),
    avail        = list(alt1=1, alt2=1, alt3=1, alt4=1),
    choiceVars   = list(best, second_pref, third_pref),
    V            = V,
    scales       = list(1,scale_2,scale_3)
  )

  ### Compute exploded logit probabilities
  P[["model"]]=apollo_el(el_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 5.8: Exploded Logit implementation

where, for normalisation, we set $\tau_S = +\infty$ and $\tau_0 = -\infty$, such that the probability of $Y_{n,t} = 1$ is given by $\frac{e^{\tau_1 - V_{n,t}}}{1+e^{\tau_1 - V_{n,t}}}$ while the probability of $Y_{n,t} = S$ is given by $1 - \frac{e^{\tau_{S-1} - V_{n,t}}}{1+e^{\tau_{S-1} - V_{n,t}}}$. In our notation, $V_{n,t}$ is the utility used inside the Ordered Logit model, which will be a function of characteristics of the decision maker and the scenario that the dependent variable relates to.

For an example using `apollo_ol`, see the section on hybrid choice models (Section 7.3). In

Apollo, the `apollo_ol` function is called as follows:

```
P[["model"]] = apollo_ol(ol_settings,
                         functionality)
```

where the contents of `ol_settings` are a little different from the earlier MNL, NL and CNL models. In particular, we have:

`outcomeOrdered:` A variable indicating the column in the database which identifies the level selected for the ordinal variable in each observation.

`V:` A numeric vector containing the explanatory variable used in the Ordered Logit model, i.e. the utility in Equation 5.21.

`tau:` A list containing the thresholds that are used in the model. The thresholds can be either scalars (i.e. the same across all observations), vectors (one value per observation) or matrices or cubes if allowing for random heterogeneity in the thresholds. The list should have one fewer element than the number of possible values for the dependent variable `Y`. Extreme thresholds at $-\inf$ and $+\inf$ are added automatically by *Apollo*.

`coding:` An optional argument of numeric or character vector type which is only required as an input if the dependent variable does not use an incremental coding from 1 to a value equal to the number of possible values for the dependent variable `Y`. This can be used both if the dependent variable is numeric in the data but not monotonic or with unequal increment, or if the dependent variable is given in string format.

`rows:` The optional `rows` argument already described for the earlier models.

`componentName:` The optional argument giving a name to the model component described already for earlier models.

Ordered Probit

In an Ordered Probit (OP) model, the probability of observing value s is given by:

$$\begin{aligned} P_{Y_{n,t}=s} &= P(\tau_{s-1} < V_{n,t} + \varepsilon < \tau_s) \\ &= P(\varepsilon < \tau_s - V_{n,t}) - P(\varepsilon < \tau_{s-1} - V_{n,t}) \\ &= \Phi(\tau_s - V_{n,t}) - \Phi(\tau_{s-1} - V_{n,t}) \end{aligned}$$

The likelihood of the observed value $Y_{n,t}$ is then given by:

$$L_{Y_{n,t}} = \sum_{s=1}^{S} \delta_{(Y_{n,t}=s)} \left[\Phi(\tau_s - V_{n,t}) - \Phi(\tau_{s-1} - V_{n,t})\right], \tag{5.23}$$

where, for normalisation, we set $\tau_S = +\infty$ and $\tau_0 = -\infty$, such that the probability of $Y_{n,t} = 1$ is given by $\Phi(\tau_1 - V_{n,t})$ while the probability of $Y_{n,t} = S$ is given by $1 - \Phi(\tau_{s-1} - V_{n,t})$. In our

notation, $V_{n,t}$ is the utility used inside the Ordered Probit model, which will be a function of characteristics of the decision maker and the scenario that the dependent variable relates to.

While this manual does not include an example with an Ordered Probit, the section on hybrid choice models (Section 7.3) does include an example with an Ordered Logit. As both models are analogous, just changing the function calls in that example from `apollo_ol` to `apollo_op` will be enough to use an Ordered Probit instead of the original Ordered Logit. In *Apollo*, the `apollo_op` function is called as follows:

```
P[["model"]] = apollo_op(op_settings,
                         functionality)
```

where the contents of `op_settings` are the same as in `ol_setting`. In particular, we have:

`outcomeOrdered`: A variable indicating the column in the database which identifies the level selected for the ordinal variable in each observation.

`V`: A numeric vector containing the explanatory variable used in the Ordered Logit model, i.e. the utility in Equation 5.21.

`tau`: A list containing the thresholds that are used in the model. The thresholds can be either scalars (i.e. the same across all observations), vectors (one value per observation) or matrices or cubes if allowing for random heterogeneity in the thresholds. The list should have one fewer element than the number of possible values for the dependent variable `Y`. Extreme thresholds at $-\text{inf}$ and $+\text{inf}$ are added automatically by *Apollo*.

`coding`: An optional argument of numeric or character vector type which is only required as an input if the dependent variable does not use an incremental coding from 1 to a value equal to the number of possible values for the dependent variable `Y`. This can be used both if the dependent variable is numeric in the data but not monotonic or with unequal increment, or if the dependent variable is given in string format.

`rows`: The optional `rows` argument already described for the earlier models.

`componentName`: The optional argument giving a name to the model component described already for earlier models.

5.3.3 Normally distributed continuous variables

For continuous dependent variables (or ordinal dependent variables that are treated as continuous) the function `apollo_normalDensity` is available, which is an implementation of the Normal probability density function. This implies that the probability of observing the specific value for the dependent variable Y in situation t for person n is given by:

$$P\left(Y_{n,t}\right) = \frac{\phi\left(\frac{Y_{n,t}-X_{n,t}-\mu}{\sigma}\right)}{\sigma}, \tag{5.24}$$

where $X_{n,t}$ is the explanatory variable used, μ and σ are the estimated means and standard deviations, and ϕ is the standard Normal density function.

For an example using `apollo_normalDensity`, see the section on hybrid choice models (Section 7.3). The `apollo_normalDensity` is called as follows:

```
P[["model"]] = apollo_normalDensity(normalDensity_settings,
                                    functionality)
```

where the contents of `normalDensity_settings` now include:

`outcomeNormal`: A variable indicating the column in the database which contains the value for the dependent variable in each observation.
`xNormal`: A numeric vector containing the explanatory variable used in Equation 5.24.
`mu`: The parameter used as the mean for the Normal density.
`sigma`: The parameter used as the standard deviation for the Normal density.
`rows`: The optional `rows` argument already described for the earlier models.
`componentName`: The optional argument giving a name to the model component described already for earlier models.

5.4 Discrete-continuous models

While choice modelling is generally best known for the study of the choice between mutually exclusive alternatives, a large body of research has also looked at the joint choice of multiple alternatives and the *consumption* of different quantities of each of these. Especially the family of Multiple Discrete Continuous Extreme Value models has received extensive interest in recent years, and two of these models are implemented in *Apollo*.

5.4.1 Multiple Discrete Continuous Extreme Value (MDCEV) model

The MDCEV model (Bhat, 2008) is a representation of a multiple discrete-continuous decisions process. Such a process consist of choosing one or more elements from a set of alternatives, and then choosing a non-negative amount of each of the chosen elements. Examples of such a process are consumption (what products or services to buy and how much of each), and time use (what activities to engage with and for how long). More formally, the MDCEV model is a stochastic implementation of the classical consumer maximization processes, where consumers allocate resources (e.g. their income) in a way that maximizes their utility. This problem can be formulated as follows:

$$\begin{aligned} \underset{x_k \forall k}{\text{Max}} \quad & \sum_{k=1}^{K} \frac{\gamma_k}{\alpha_k} \psi_k \left(\left(\frac{x_k}{\gamma_k} + 1 \right)^{\alpha_k} - 1 \right) \\ \text{subject to} \quad & \sum_{k=1}^{K} x_k p_k = B \end{aligned} \tag{5.25}$$

$$\psi_k = \exp(V_k + \varepsilon_k), \tag{5.26}$$

where K is the number of alternatives, x_k is the amount consumed of product k, and p_k is the unit price or cost of alternative k, and B is the budget available to the individual for consumption. The term ε_k is an independent and identically distributed random disturbance following a *Gumbel(0, σ)* distribution. Finally, α_k and γ_k are parameters determining satiation, while V_k determines each alternative's base utility (i.e. its marginal utility at zero consumption).

The probability of an observed vector of consumptions is then given by:

$$\begin{aligned} &P({x_1}^*, {x_2}^*, \cdots, {x_M}^*, 0, \cdots, 0) \\ =&\frac{1}{p_1}\frac{1}{\sigma^{M-1}}\left(\prod_{m=1}^{M} f_m\right)\left(\sum_{m=1}^{M}\frac{p_m}{f_m}\right)\left(\frac{\prod_{m=1}^{M} e^{V_i/\sigma}}{\left(\sum_{k=1}^{K} e^{W_k/\sigma}\right)^M}\right)(M-1)!, \end{aligned} \tag{5.27}$$

where $f_i = \frac{1-\alpha_i}{x_i^*+\gamma_i}$ and $W_k = V_k + (\alpha_k - 1)\log(\frac{x_k^*}{\gamma_k}+1) - \log(p_k)$, and x_k^* is the observed (optimum) consumption of product k.

A revised formulation as shown in Equation 5.28 is obtained when an outside good is included among the alternatives. An outside good is a product that is consumed by all individuals in the sample. The outside good usually represents an aggregate measure of the consumption of all products that are not of interest for the study. For example, if a study focuses on use of leisure time, the outside good might be all activities that are not leisure (such as sleeping, work, travelling, etc.), while the inside goods (i.e. all alternatives that are not the outside good) could deal with leisure in a more detailed way (e.g. going to the park, hiking, going to the cinema, meeting friends, etc.).

$$\begin{aligned} &P({x_1}^*, {x_2}^*, \cdots, {x_M}^*, 0, \cdots, 0) \\ =&\frac{1}{\sigma^{M-1}}\left(\prod_{m=1}^{M} f_m\right)\left(\sum_{m=1}^{M}\frac{p_m}{f_m}\right)\left(\frac{\prod_{m=1}^{M} e^{V_i/\sigma}}{\left(\sum_{k=1}^{K} e^{W_k/\sigma}\right)^M}\right)(M-1)!, \end{aligned} \tag{5.28}$$

The function `apollo_mdcev` calculates the loglikelihood of an MDCEV model, using equation 5.27 if no outside good is provided and uses equation 5.28 if an outside good is provided. An example of a function call, as well as a definition of its arguments follow. The function is called as:

```
P[["model"]] = apollo_mdcev(mdcev_settings,
                            functionality)
```

The list `mdcev_settings` contains the following objects:

`alternatives`: Character vector containing the name of all alternatives. If one of these alternatives is called `outside`, it will automatically be used as the outside good.

`avail`: List of availabilities, using the names from `alternatives`. Each element can be scalar (0 or 1) or a vector detailing availability for each observation.

`continuousChoice`: List of continuous consumption, using the names from `alternatives`. Each element must be a vector of length N (number of observations) indicating the amount consumed.

`V:` A list of length K (i.e. number of alternatives), containing the deterministic part of the base utility of each alternative. The outside good should have `V=0` if included.

`alpha:` List containing the α parameter for each alternative, using the names from `alternatives`.

`gamma:` List containing the γ parameter for each alternative, using the names from `alternatives`, excluding any outside good.

`sigma:` Scalar representing the scale parameter of the error term. If there is no price variation across products, this should be fixed to 1.

`cost:` List containing the cost or price of each alternative, using the names from `alternatives`. Each element can be a scalar if the price does not change across observations, or a vector of length equal to the number of observations in the data, detailing the price for each observation.

`budget:` Vector with the amount of the resource (e.g. money or time) available for each observation. It must be equal to the total consumption of that observation.

`minConsumption:` Optional argument, which, if provided, should be a list with as many elements as alternatives. Each element can be either a scalar or a vector defining the minimum consumption of an alternative if it is consumed.

`outside:` Optional argument with the name of an outside good. This is not needed if one of the alternatives is already called `outside`.

`rows:` The optional `rows` argument already described for the earlier models.

`componentName:` The optional argument giving a name to the model component described already for earlier models.

As discussed at length by Bhat (2008), different profiles exist for normalisation of a MDCEV model, either using a generic α and alternative-specific γ parameters, an α parameter only for the outside good (and set to zero for others) along with alternative-specific γ parameters, or alternative specific α terms with $\gamma = 1$ for all goods. In our examples below, we use a generic α and alternative-specific γ parameters. Other profiles can be implemented by simply changing which parameters are generic and which are alternative specific, and making some α terms equal to zero, as appropriate.

We include two examples of the MDCEV model on the time use data described in Section 3.4. The first example, `Apollo_example_11.r` does not include an outside good. We illustrate this in Figure 5.9.

We begin by defining the names of the alternatives, availabilities and continuous consumptions, where we turn minutes into hours. We then create the list of utilities, where, in our example, these include alternative specific constants only, where we fix δ_{home} to zero for identification. This is followed by the definition of a generic α parameter, which is constrained to be below 1 by using a logistic transform, with $\alpha = \frac{1}{1+e^{-\alpha_{base}}}$, and the set of γ parameters, where these are alternative-specific in our case. We finally define the costs, turn the budget into hours, and make the call to `apollo_mdcev`. In this example, we also fix `sig` to its starting value of 1 in `apollo_fixed`.

The second example, `Apollo_example_12.r`, groups together some alternatives to create an outside good. It also incorporates socio-demographics in the utility function, though not in the α and γ terms, which is however also possible in *Apollo*. We illustrate this example in Figure 5.10.

To create the new activities, we sum some of the activities up after reading in the data, where

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define individual alternatives
  alternatives = c("dropOff",
                   ...
                   "other")

  ### Define availabilities
  avail = list(dropOff  = 1,
               ...
               other    = 1)

  ### Define continuous consumption for individual alternatives
  continuousChoice = list(dropOff  = t_a01/60,
                          ...
                          other    = t_a12/60)

  ### Define utilities for individual alternatives
  V = list()
  V[["dropOff" ]] = delta_dropOff
  ...
  V[["other"   ]] = delta_other

  ### Define alpha parameters
  alpha = list(dropOff  = 1 /(1 + exp(-alpha_base)),
               ...
               other    = 1 /(1 + exp(-alpha_base)))

  ### Define gamma parameters
  gamma = list(dropOff      = gamma_dropOff,
               ...
               other        = gamma_other)

  ### Define costs for individual alternatives
  cost = list(dropOff      = 1,
              ...
              other        = 1)

  ### Define budget
  budget = budget/60

  ### Define settings for MDCEV model
  mdcev_settings <- list(alternatives      = alternatives,
                         avail             = avail,
                         continuousChoice  = continuousChoice,
                         V                 = V,
                         alpha             = alpha,
                         gamma             = gamma,
                         sigma             = sig,
                         cost              = cost,
                         budget            = budget)

  ### Compute probabilities using MDCEV model
  P[["model"]] = apollo_mdcev(mdcev_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 5.9: MDCEV implementation without outside good

we create an outside good by combining time spent travelling with time spent at home. We also create a generic leisure activity. The remainder of the specification is no different in principle from that in Figure 5.10 with the exception of there being an alternative called `outside`, and with using more detailed utility functions.

5.4.2 Multiple Discrete Continuous Nested Extreme Value (MDCNEV) model

The MDCNEV is an extension to the MDCEV model, proposed by Pinjari and Bhat (2010a). It incorporates correlation between alternatives, in a similar way to the Nested Logit (NL), where correlation can be introduced by nesting, i.e. grouping alternatives that are correlated among them. The implementation of MDCNEV in *Apollo* allows for only a single level of nesting and is also only valid for models with an outside good, i.e. a product that is consumed in every observation. The likelihood function of the model is as follows.

$$
\begin{aligned}
&P(x_1{}^*, x_2{}^*, \ldots, x_M{}^*, 0, \ldots, 0) \\
&= |J| \frac{\prod_{i \in \text{chosen alts}} e^{\frac{V_i}{\theta_i}}}{\prod_{s=1}^{S_M} \left(\sum_{i \in s^{th}_{nest}} e^{\frac{V_i}{\theta_s}} \right)^{q_s}} \\
&\cdot \sum_{r_1=1}^{q_1} \cdots \sum_{r_s=1}^{q_s} \cdots \sum_{r_{S_M}=1}^{q_{S_M}} \left\{ \prod_{s=1}^{S_M} \left[\frac{\left(\sum_{i \in s^{th}_{nest}} e^{\frac{V_i}{\theta_s}} \right)^{\theta_s}}{\sum_{s=1}^{S_k} \left\{ \left(\sum_{i \in s^{th}_{nest}} e^{\frac{V_i}{\theta_s}} \right)^{\theta_s} \right\}} \right]^{q_s - r_s + 1} \left(\prod_{s=1}^{S_M} sum(X_r s) \right) \left(\sum_{s=1}^{S_M} (q_s - r_s + 1) - 1 \right)! \right\},
\end{aligned}
\tag{5.29}
$$

For a detailed explanation of the values in the equation, see Pinjari and Bhat (2010a).

The `apollo_mdcnev` function is called as:

```
P[["model"]] = apollo_mdcev(mdcnev_settings,
                            functionality)
```

Aside from the previously defined contents in `mdcnev_settings`, we now have two additional inputs, namely:

mdcnevNests: A named vector containing the names of the nests and the associated structural parameters *theta*. For each *theta*, we give the name of the associated parameter. Unlike in `apollo_nl`, the `root` is not included for `apollo_cnl` as only two-level structures are used.

mdcnevStructure: A matrix showing the allocation of alternatives to nests, with one row per nest and one column per alternative, using the same ordering as in `alternatives` and `mdcnevStructure`. Element (i, j) should take value 1 if alternative j belongs to nest i, and zero otherwise.

The example `Apollo_example_13.r` is a nested version of the model with an outside good used in Figure 5.10, i.e. `Apollo_example_12.r`. The model uses two nests, one for *mandatory* activities (work, school, private) and one for *optional* activities (all others, including the outside good). In Figure 5.11, we only show the part of the code that differs from the standard MDCEV model.

```
database = read.csv("apollo_timeUseData.csv",header=TRUE)

### Create consumption variables for combined activities
database$t_outside = rowSums(database[,c("t_a01", "t_a06", "t_a10", "t_a11", "t_a12")]) # outside good:
    ↪ time spent at home and travelling
database$t_leisure = rowSums(database[,c("t_a07", "t_a08", "t_a09")])

...

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define individual alternatives
  alternatives  = c("outside",
                    ...
                    "leisure")

  ### Define availabilities
  avail = list(outside  = 1,
               ...
               leisure  = 1)

  ### Define continuous consumption for individual alternatives
  continuousChoice = list(outside  = t_outside/60,
                          ...
                          leisure  = t_leisure/60)

  ### Define utilities for individual alternatives
  V = list()
  V[["outside"]]  = 0
  ...
  V[["leisure"]]  = delta_leisure  + delta_leisure_wknd*weekend

  ### Define alpha parameters
  alpha = list(outside  = 1 /(1 + exp(-alpha_base)),
               ...
               leisure  = 1 /(1 + exp(-alpha_base)))

  ### Define gamma parameters
  gamma = list(work     = gamma_work,
               ...
               leisure  = gamma_leisure)

  ### Define costs for individual alternatives
  cost = list(outside  = 1,
              ...
              leisure  = 1)

  ### Define budget
  budget = budget/60

  ### Define settings for MDCEV model
  mdcev_settings <- list(alternatives      = alternatives,
                         avail             = avail,
                         continuousChoice  = continuousChoice,
                         V                 = V,
                         alpha             = alpha,
                         gamma             = gamma,
                         sigma             = sig,
                         cost              = cost,
                         budget            = budget)

  ### Compute probabilities using MDCEV model
  P[["model"]] = apollo_mdcev(mdcev_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 5.10: MDCEV implementation with an outside good

We define two nests, and assign the appropriate θ parameter to each, where in our example, `theta_optional` is further fixed to 1 via `apollo_fixed` as its estimate was not significantly different from 1. We then describe the allocation of alternatives to nests using a matrix of ones and zeros, with one row per nest and one column per alternative, where each alternative falls into exactly one nest. Finally, we make the call to `apollo_mdcnev`. For this model, we use scaling of some of the parameters in model estimation given the earlier findings of very diverse scales for the individual parameters in the corresponding simple MDCEV model, i.e. `Apollo_example_12.r`.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ...

  ### Define nesting structure
  mdcnevNests = list(mandatory = theta_mandatory,
                     optional  = theta_optional)

  mdcnevStructure  = matrix(0, nrow=length(mdcnevNests), ncol=length(V))
  ###                      outside work school shopping private leisure
  mdcnevStructure[1,] = c(      0,   1,     1,        0,       1,       0) # mandatory
  mdcnevStructure[2,] = c(      1,   0,     0,        1,       0,       1) # optional

  ...

  ### Define settings for MDCNEV model
  mdcnev_settings <- list(...
                          mdcnevNests       = mdcnevNests,
                          mdcnevStructure   = mdcnevStructure)

  ### Compute probabilities using MDCNEV model
  P[["model"]] = apollo_mdcnev(mdcnev_settings, functionality)

  ...
}
```

Figure 5.11: MDCNEV implementation and call to apollo_estimate using scaling

5.5 Adding new model types

As already mentioned, users of *Apollo* are not restricted to those models for which functions are available in the code. Any model that yields a probability for an outcome can be used in the code and parameters for the model can be estimated using either classical or Bayesian estimation. The advantage of the predefined functions is of course that they run a large number of checks to avoid issues with mis-specification and produce output for different user needs. The level of these checks and output flexibility that a user implements for new models will vary as a function of the user's needs.

The user has the option of either creating new functions in `R` that are defined outside `apollo_probabilities` much in the same way as for example `apollo_mnl`, or to simply code the probabilities for a model inside `apollo_probabilities`. An example of the latter approach is shown in Section 11.4.

Clearly, coding models as separate functions is preferable in terms of reusability as well as code organisation. Users who are interested in coding their own functions should inspect the code for some of the implemented functions for guidance, for example using `apollo_ol` as a simple start. For user defined models to be compatible with *Apollo*, a number of simple basic requirements need to be fulfilled. In particular, the function needs to take `functionality` as an argument to

be able to produce different output depending on the value passed to it for `functionality`. Not all possible values discussed for `functionality` in this manual need to be implemented for new models, but essential capabilities include the ability to deal with the following four settings for `functionality`:

estimate: Return the probabilities for each row in the data, using a vector, matrix or cube (array) depending on the presence of random coefficients (cf. Section 6.1).
validate: Return TRUE if all tests are passed, or TRUE if no tests are implemented.
zero_LL: Return the log-likelihood of the model component with all parameters at zero, set to NA if not applicable for given model.
output: Same as `functionality="estimate"`.

If the models are to be compatible with random coefficients, they furthermore need to be able to produce probabilities as three-dimensional arrays, as discussed in Section 6.1. Additionally, if the models are components of an overall model from which predictions are to be made (cf. Section 9.6), then output from the function is also needed with `functionality==prediction`, even if returning `NA` for that model component.

An example of a new model implemented in a fully compatible way with *Apollo* is presented in the compressed folder `emdcev.R` which is available in the examples page of the *Apollo* website (www.ApolloChoiceModelling.com. The compressed folder contains five files:

`apollo_emdcev.R` : This file contains the definition of the new function, and extension to the MDCEV model. The function receives two arguments: `emdcev_settings` and `functionality`. While the first argument is a list grouping all relevant inputs for the model, the second argument determines the behaviour of the function, e.g. `"estimation"`, `"prediction"`, etc.
`apollo_odStgoTrips.csv` : Revealed preference dataset containing the number of trips performed in a day by a household. It also includes some sociodemographics as well as the macro-area-location of the household. This data comes from the 2012 Santiago origin-destination survey (www.sectra.cl).
`apollo_odStgoTripsDictionary.txt` : Text file defining each variable contained in `apollo_odStgoTrips.csv`.
`apollo_timeUseData.csv` : Time use database (see 3.4).
`timeUse.r` : Example model script using the extended MDCEV on the time use data (`apollo_timeUseData.csv`).
`trips.R` : Example model script using the extended MDCEV on the trips data (`apollo_odStgoTrips.csv`).

Chapter 6

Incorporating random heterogeneity

In this chapter, we describe how to use the *Apollo* package to incorporate random coefficients. We look first at continuous random heterogeneity before looking at Discrete Mixtures (DM) and Latent Class (LC) models, and also a combination of the two. Finally, we discuss multi-core estimation, which is beneficial for models with random heterogeneity. In this section, we use the simple binary public transport route choice SP data described in Section 3.2.

6.1 Continuous random coefficients

6.1.1 Introduction

The *Apollo* package allows for a very general use of continuous random coefficients. The code works for models allowing for intra-individual mixing (i.e. heterogeneity across observations for the same individual), inter-individual mixing (i.e. heterogeneity across individual people), as well as a mixture of the two. For background, we provide a brief recap of the discussions in Hess and Train (2011) on this topic.

In cross-sectional data, we would have a sample of N individuals, indexed as $n = 1, \ldots, N$, where each individual is observed to face only one choice situation. Let β_n be a vector of the true, but unobserved taste coefficients for consumer n. We assume that $\beta_n \, \forall n$ is *iid* over consumers with density $g\left(\beta \mid \Omega\right)$, where Ω is a vector of parameters of this distribution, such as the mean and variance. Let j_{n^*} be the alternative chosen by consumer n, such that $P_n\left(j_{n^*} \mid \beta\right)$ gives the probability of the observed choice for consumer n, conditional on β. The Mixed Logit probability of consumer n's chosen alternative is

$$P_n\left(j_{n^*} \mid \Omega\right) = \int_{\beta} P_n\left(j_n^* \mid \beta\right) g\left(\beta \mid \Omega\right) d\beta. \tag{6.1}$$

The log-likelihood function is then given by:

$$\mathrm{LL}\left(\Omega\right) = \sum_{n=1}^{N} \ln\left(\int_{\beta} P_n\left(j_n^* \mid \beta\right) g\left(\beta \mid \Omega\right) \mathrm{d}\beta\right), \tag{6.2}$$

Since the integrals do not take a closed form, they are approximated by simulation. The simulated log-likelihood is:

$$\mathrm{SLL}\left(\Omega\right) = \sum_{n=1}^{N} \ln\left(\frac{1}{R}\sum_{r=1}^{R} P_n\left(j_n^* \mid \beta_{r,n}\right)\right). \tag{6.3}$$

where $\beta_{r,n}$ gives the rth draw (out of R) from $g(\beta \mid \Omega)$ for individual n. Different draws are used for the N consumers, for a total of NR draws.

When we have multiple observations per individual, we typically make the assumption that sensitivities vary across people, but stay constant across individuals. We would then have that the likelihood of the sequence of choices for person n is given by:

$$P_n\left(\Omega\right) = \int_{\beta} \prod_{t=1}^{T_n} P_{n,t}\left(j_{n,t}^* \mid \beta\right) g\left(\beta \mid \Omega\right) d\beta, \tag{6.4}$$

where $j_{n,t}^*$ is the alternative chosen by individual n in choice situation t. Note that, since the same sensitivities apply to all choices by a given consumer, the integration over the density of β applies to all the consumer's choices combined, rather than each one separately.

The log-likelihood function for the observed choices is then:

$$\mathrm{LL}\left(\Omega\right) = \sum_{n=1}^{N} \ln\left(\int_{\beta}\left[\prod_{t=1}^{T_n}\left(P_{n,t}\left(j_{n,t}^* \mid \beta\right)\right)\right] g\left(\beta \mid \Omega\right) \mathrm{d}\beta\right). \tag{6.5}$$

The simulated LL (SLL) is:

$$\mathrm{SLL}\left(\Omega\right) = \sum_{n=1}^{N} \ln\left(\frac{1}{R}\sum_{r=1}^{R}\left[\prod_{t=1}^{T_n}\left(P_{n,t}\left(j_{n,t}^* \mid \beta_{r,n}\right)\right)\right]\right). \tag{6.6}$$

Note that in this formulation, the product over choice situations is calculated for each draw; the product is averaged over draws; and *then* the log of the average is taken. The SLL is the sum over consumers of the log of the average (across draws) of products. The calculation of the contribution to the SLL function for consumer n involves the computation of RT_n Mixed Logit probabilities.

Instead of utilising the panel nature of the data, the model could be estimated *as if* each choice were from a different consumer. That is, the panel data could be *treated as if* they were cross-sectional. The objective function is similar to Equation 6.2 except that the multiple choice situations by each consumer are represented as being for different individuals:

$$\mathrm{LL}\left(\Omega\right) = \sum_{n=1}^{N}\sum_{t=1}^{T_n} \ln\left(\int_{\beta} P_{n,t}\left(j_{n,t}^* \mid \beta\right) g\left(\beta \mid \Omega\right) \mathrm{d}\beta\right), \tag{6.7}$$

where the integration across the distribution of taste coefficients is applied to each choice, rather than to each consumer's sequence of choices. This function is simulated as:

$$\mathrm{SLL}\left(\Omega\right) = \sum_{n=1}^{N}\sum_{t=1}^{T_n} \ln\left(\frac{1}{R}\sum_{r=1}^{R} P_{n,t}\left(j_{n,t}^* \mid \beta_{r,t,n}\right)\right). \tag{6.8}$$

where $\beta_{r,t,n}$ is the rth draw from $g(\beta \mid \Omega)$ for choice situation t for individual n. Different draws are used for the T_n choice situations for consumer n, as well as for the N consumers. Consumer n's contribution to the SLL function utilises RT_n draws of β rather than R draws as in Equation 6.6, but involves the computation of the same number of Logit probabilities as before, namely, RT_n. The difference is that the averaging across draws is performed before taking the product across choice situations.

We now generalise the specification on panel data to include intra-personal taste heterogeneity in addition to inter-personal heterogeneity. Let $\beta_{n,t} = \alpha_n + \gamma_{n,t}$ where α_n is distributed across consumers but not over choice situations for a given consumer, and $\gamma_{n,t}$ is distributed over choice situations as well as consumers. That is, α_n captures inter-personal variation in tastes while $\gamma_{n,t}$ captures intra-personal variation. Their densities are denoted as $f(\alpha)$ and $h(\gamma)$, respectively,[1] where their dependence on underlying parameters, contained collectively in Ω, is suppressed for convenience.

The LL function is given by:

$$LL\left(\Omega\right) = \sum_{n=1}^{N} \ln \left[\int_{\alpha} \left(\prod_{t=1}^{T_n} \left(\int_{\gamma} P_{n,t}\left(j_{n,t}^{*} \mid \alpha, \gamma\right) h\left(\gamma\right) \mathrm{d}\gamma \right) \right) f\left(\alpha\right) \mathrm{d}\alpha \right]. \tag{6.9}$$

The two levels of integration create two levels of simulation, which can be specified as:

$$SLL = \sum_{n=1}^{N} \ln \left[\frac{1}{R} \sum_{r=1}^{R} \left(\prod_{t=1}^{T_n} \frac{1}{K} \sum_{k=1}^{K} \left(P_{n,t}\left(j_{n,t}^{*} \mid \alpha_{r,n}, \gamma_{k,t,n}\right)\right)\right)\right]. \tag{6.10}$$

This simulation uses R draws of α for consumer n, along with $K\,T_n$ draws of γ. Note that, in this specification, the same draws of γ are used for all draws of α. That is, $\gamma_{k,t,n}$ does not have an additional subscript for r. The total number of evaluations of a Logit probability for consumer n is equal to $R\,K\,T_n$, compared to $R\,T_n$ when there is only inter-personal variation.

The *Apollo* package allows the user to incorporate continuous random heterogeneity for all types of models. In a model using `apollo_mnl` inside `apollo_probabilities`, we would thus obtain a Mixed Multinomial Logit (MMNL) model, while, with a CNL core, i.e. `apollo_cnl`, we would have a Mixed CNL model. Users can similarly specify and estimate mixed MDCEV models (an example is included in `Apollo_example_17.r`, and clearly also hybrid choice structures, as described in Section 7). It is straightforward to combine continuous random heterogeneity with deterministic heterogeneity for individual parameters, as shown in our example. There are very few limits imposed on what parameters can incorporate continuous random heterogeneity, opening up the use of error components for correlation across alternatives and heteroskedasticity (cf. Section 6.1.5). The parameters (in the models made available with *Apollo*) for which random heterogeneity is not allowed are:

- the allocation parameters α in a CNL model
- the σ parameter in a MDCEV model
- the θ parameters in a MDCNEV model

[1] The mean of β_n is captured in α_n such that the mean of $\gamma_{n,t}$ is zero.

A very flexible implementation is used that minimises the changes in the code that are required to introduce random coefficients or to change between the different layers of integration. In particular, the package works with arrays in three dimensions. For a model without continuous random coefficients, the likelihood for a model (prior to multiplying across observations for the same individual) is contained in a column vector of length O, where O is the number of observations in the data. If we introduce continuous random heterogeneity across individual people, with typically multiple observations per person, the likelihood is given by a OxR_1 matrix, with one row per observation, and one column per draw from the random coefficients, where we use R_1 draws per random coefficient and per individual. Here, the same draws would be reused across the T_n rows for a given individual n, meaning that we would have N sets of draws, where N is the number of individuals. In the presence of additional heterogeneity across observations for the same respondent, the likelihood becomes a cube of dimensions OxR_1xR_2, where in this third dimension, different draws are used across different observations for the same individual. As described by Hess and Train (2011), a given inter-individual draw is then associated with multiple intra-individual draws. If only intra-individual heterogeneity is used, the cube collapses to an array of dimensions Ox1xR_2, i.e. a matrix but with columns going into the third dimension rather than second dimension. Depending on the type of heterogeneity (inter and inter) present in the model, different operations are required in terms of averaging across draws and multiplying across choices, and we discuss these in detail in our example below.

A number of guidelines are appropriate at this stage:

- If a user has panel data, i.e. multiple observations for at least some of the individuals, inter-individual draws are used for variation across individuals, and intra-individual draws for variation across observations for the same individual.
- If a user has cross-sectional data, i.e. only one observation per individual, only inter-individual draws should be used.
- If a user has panel data, but uses `panelData=FALSE` in `apollo_control`, the data will be treated as cross-sectional, and only inter-individual draws should be used.

6.1.2 Example model specification

In what follows, we show the specification of a MMNL model with various levels of heterogeneity on the route choice data described in Section 3.2. We specify the utility of alternative j for individual n in choice situation t in willingness to pay space as:

$$V_{n,j,t} = \delta_j + \beta_{TC,n}\left(\beta_{VTT,n,t}TT_{n,j,t} + TC_{n,j,t} + \beta_{VHW,n}HW_{n,j,t} + \beta_{VCH}CH_{n,j,t}\right), \qquad (6.11)$$

where $TT_{n,j,t}$, $TC_{n,j,t}$, $HW_{n,j,t}$ and $CH_{n,j,t}$ refer to the travel time, travel cost, headway and interchanges attributes, respectively, for alternative j in choice situation t for individual n. The treatment in terms of deterministic and random heterogeneity differs across the various parameters, as we will now explain in turn:

- Alternative specific constants (ASC) are included to capture any left-right bias in the survey, where we set $\delta_2 = 0$ for normalisation. No random or deterministic heterogeneity is incorporated for δ_1.

- The travel cost coefficient $\beta_{TC,n}$ multiplies the entire remainder of the utility function, meaning that our model produces direct estimates of willingness-to-pay (WTP) measures through working in WTP space (Train and Weeks, 2005). We use a negative log-uniform distribution (cf. Hess et al., 2017) for this coefficient, capturing inter-individual heterogeneity only, with

$$\beta_{TC,n} = -exp\left(a_{\log(\beta_{TC})} + b_{\log(\beta_{TC})} \cdot \xi_{tc,n}\right), \tag{6.12}$$

  where $\xi_{tc,n}$ follows a uniform distribution across individuals (but is constant across choices for the same individual), and $a_{\log(\beta_{TC})}$ and $b_{\log(\beta_{TC})}$ are the offset and range, respectively, for the Uniform distribution used for the log of β_{TC}.
- The value of travel time parameter $\beta_{VTT,nt}$ gives a direct estimate of the monetary valuation of travel time (VTT). We use a very flexible distribution for this coefficient. We begin with a lognormal distribution at the inter-individual level, but add additional heterogeneity across choices for the same individual, a semi non-parametric term to allow for deviation from the lognormal distribution at the individual level (Fosgerau and Mabit, 2013), and a deterministic multiplier to allow for differences between business and non-business travellers. We have:

$$\begin{aligned}\beta_{VTT,n,t} = exp[&\mu_{\log(\beta_{VTT})} \\ &+ \sigma_{\log(\beta_{VTT}),inter} \cdot \xi_{tt,n} \\ &+ \sigma_{\log(\beta_{VTT}),inter,2} \cdot \xi^2_{tt,n} \\ &+ \sigma_{\log(\beta_{VTT}),intra} \cdot \xi_{tt,nt}] \\ &\cdot (\gamma_{VTT,business} \cdot x_{business,n} + (1 - x_{business,n}))\,,\end{aligned} \tag{6.13}$$

  where $\mu_{\log(\beta_{VTT})}$ is the estimated mean for the log of β_{VTT}, $\sigma_{\log(\beta_{VTT}),inter}$ and $\sigma_{\log(\beta_{VTT}),inter,2}$ are the standard deviation and first additional Fosgerau and Mabit (2013) polynomial term at the inter-individual level, multiplying the inter-individual level standard normally distributed $\xi_{tt,n}$ error term and its square, respectively, and $\sigma_{\log(\beta_{VTT}),intra}$ captures additional intra-individual heterogeneity by multiplying a standard normally distributed error term which also varies observations for the same individual, $\xi_{tt,nt}$. Finally $\gamma_{VTT,business}$ is a multiplier for business travellers, for whom $x_{business,n} = 1$. The subscript t on $\beta_{VTT,n,t}$ reflects the fact that β_{VTT} is distributed across individuals and across choices.
- The value of headway parameter $\beta_{VHW,n}$ again follows a lognormal distribution only at the inter-individual level, but with correlation with the inter-individual heterogeneity in the value of travel time parameter $\beta_{VTT,nt}$, such that:

$$\beta_{VHW,n} = exp\left(\mu_{\log(\beta_{VHW})} + \sigma_{\log(\beta_{VHW})} \cdot \xi_{hw,n} + \sigma_{\log(\beta_{VHW},\beta_{VTT})} \cdot \xi_{tt,n}\right), \tag{6.14}$$

  where $\xi_{hw,n}$ again follows a standard Normal distribution across individuals, and where the reuse of $\xi_{tt,n}$ from the $\beta_{VTT,nt}$ definition allows us to capture correlation betwen $\beta_{VHW,n}$ and $\beta_{VTT,nt}$ through the estimate $\sigma_{\log(\beta_{VHW},\beta_{VTT})}$. If the parameter $\sigma_{\log(\beta_{VHW},\beta_{VTT})}$ is positive , we get positive correlation between the distribution of $\beta_{VTT,n,t}$ and $\beta_{VHW,n}$,

meaning that people who are more sensitive to travel time are also more sensitive to headway, and vice versa, while a negative estimate would imply negative correlation, meaning that people who are more sensitive to travel time are less sensitive to headway, and vice versa. The actual correlation is complicated in this case because of the semi-non parametric term and the lognormals, but can be calculated empirically from the draws produced by `apollo_unconditionals`.

- The value of interchanges parameter β_{VCH} is estimated without any heterogeneity, hence the lack of subscript.

We use this highly complex specification with a view to illustrating both the flexibility of the code and the ease of implementation of complex models.

6.1.3 Implementation

We explain the implementation of the model from Section 6.1.2 in four simple steps. We do not revisit obvious steps such as the definition of parameters to estimate. The model is implemented in `Apollo_example_16.r`, with simpler Mixed Logit models also available in `Apollo_example_14.r` and `Apollo_example_15.r`.

Settings

The first step is to set `mixing=TRUE` in `apollo_control`. This setting is a requirement for using continuous random heterogeneity. A user may additionally set `nCores` to a value larger than 1 in `apollo_control`, a point we return to in Section 6.4. We would thus for example have:

```
apollo_control = list(modelName = "Apollo_example_16",
                      ...
                      mixing = TRUE,
                      nCores = 3)
```

Draws

The second step concerns the generation of draws for random distributions. In our case, we need to produce uniformly distributed inter-individual draws for $\xi_{tc,n}$, normally distributed inter-individual draws for $\xi_{tt,n}$ and $\xi_{hw,n}$, and normally distributed intra-individual draws for $\xi_{tt,n,t}$. Draws are generated by *Apollo* whenever `mixing==TRUE` in `apollo_control`, using the settings defined in a list called `apollo_draws`. This process happens during `apollo_validateInputs`. The setup used for this is illustrated in Figure 6.1.

In `apollo_draws`, the user needs to create settings for the type of draws, both for inter (`interDrawsType`) and intra-individual (`intraDrawsType`) draws. Seven pre-defined types of draws are available in *Apollo*, namely:

pmc for pseudo-Monte Carlo draws;
halton for Halton draws Halton (1960), not recommended for more than five random coefficients, (cf. Bhat, 2003);

```
apollo_draws = list(
  interDrawsType = "halton",
  interNDraws    = 100,
  interUnifDraws = c("draws_tc_inter"),
  interNormDraws = c("draws_hw_inter","draws_tt_inter"),
  intraDrawsType = "mlhs",
  intraNDraws    = 100,
  intraUnifDraws = c(),
  intraNormDraws = c("draws_tt_intra")
)
```

Figure 6.1: Defining settings for generation of draws

mlhs for MLHS draws (Hess et al., 2006);
sobol for Sobol draws (Sobol', 1967);
sobolOwen for Sobol draws with Owen scrambling (Owen, 1995);
sobolFaureTezuka for Sobol draws with Faure-Tezuka scrambling (Faure and Tezuka, 2000); and
sobolOwenFaureTezuka for Sobol draws with both Owen and Faure-Tezuka scrambling

While the type of draws used can differ between the inter and intra-individual sets of draws, multiple sets of draws within either category (i.e. inter or intra) will come from the same type. In our case, we use Halton draws for the inter-individual draws and MLHS draws for the intra-individual draws. The use of Halton draws is possible here given the low number of random coefficients, but Halton draws are not advised for more than 5 random coefficients given colinearity issues (cf. Bhat, 2003). When using the same type of draws for both inter and intra-individual draws, different parts of the sequence (e.g. primes for Halton) are used for the two types.

The user needs to next specify how many draws are to be used per individual for inter-individual draws, and per observation for intra-individual draws. This is set via `interNDraws` and `intraNDraws`, respectively. The number can differ between these two dimensions of integration. We use 100 inter-individual draws per parameter and per individual, and 100 intra-individual draws per parameter and per choice situation. If only inter-individual draws are to be used, then a setting of `intraNDraws = 0` is used, with a corresponding approach for intra-individual draws only. Alternatively, these settings can be omitted by the user.

Finally, the user needs to define the actual random disturbances or sets of draws, by giving each set of draws a name which can be used later in the model specification, and by determining whether the draws are Normally or Uniformly distributed, by including their names in `interNormDraws` and `interUnifDraws`, respectively, in the case of inter-individual draws, and `intraNormDraws` and `intraUnifDraws`, respectively, in the case of intra-individual draws. These two distributions (standard Normal and Uniform between 0 and 1) are used as the base for any other distributions later in the code. All the draws in our example follow standard Normal distributions, except $\xi_{tc,n}$, which comes from a Uniform distribution between 0 and 1. A user can either specify empty vectors for any settings that are not in use, such as `intraUnifDraws = c()` in our case, or omit these settings entirely.

Some users may want additional flexibility to combine different types of draws or to generate their own draws. This is possible in *Apollo* by giving the name of a user generated object in

`apollo_draws$interDrawsType` and/or `apollo_draws$intraDrawsType` instead of providing one of the seven specific types of draws listed above. Using the example from Figure 6.1, let us assume the user wants to provide his/her own draws for inter-individual mixing, but continue to use the *Apollo* generated MLHS draws for intra-individual mixing. In that case, the user needs to replace `halton` by for example `ownInterDraws`, where this is a list, with one element per random set of draws. Each entry in the list needs to have a name, where this same set of names is then used across `interUnifDraws` and `interNormDraws` to instruct the code to either leave the draws untransformed or apply an inverse Normal CDF. The draws provided in the list `ownInterDraws` should thus be uniformly distributed. The user also still needs to specify `interNDraws` and `intraNDraws`. Each element of the list of draws provided by the user (`ownInterDraws` in our example) should be a matrix containing the user-generated draws. In the case of inter-individual draws, each matrix must have one row per individual in the database and `interNDraws` columns, while, for intra-individual draws, the matrix must have one row per observation in the database, and `intraNDraws` columns.

Random coefficients

The third step concerns the actual definition of those coefficients in the model that follow a random distribution. For this, the code includes an additional function defined outside the `apollo_probabilities` function, namely `apollo_randCoeff`. Just as with `apollo_probabilities`, this is a function that the user does not call but which the user defines.

```
apollo_randCoeff = function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["b_tc"]] = -exp( mu_log_b_tc
                            + sigma_log_b_tc_inter      * draws_tc_inter )

  randcoeff[["v_tt"]] =  ( exp( mu_log_v_tt
                            + sigma_log_v_tt_inter      * draws_tt_inter
                            + sigma_log_v_tt_inter_2    * draws_tt_inter ^ 2
                            + sigma_log_v_tt_intra      * draws_tt_intra   )
                          * ( gamma_vtt_business    * business + ( 1 - business ) ) )

  randcoeff[["v_hw"]] =  exp( mu_log_v_hw
                            + sigma_log_v_hw_inter      * draws_hw_inter
                            + sigma_log_v_hw_v_tt_inter * draws_tt_inter )

  return(randcoeff)
}
```

Figure 6.2: The `apollo_randCoeff` function

This function takes `apollo_beta` and `apollo_inputs` as inputs and generates a new list which contains the random coefficients, incorporating any deterministic effects too. This step is shown in Figure 6.2, where the correspondence with Equations 6.12 to 6.14 should be clear. The contents of `apollo_randCoeff` will vary across model specifications, only the first line (`randCoeff = list()`) and final line (`return(randCoeff)`) are to remain as in the example.

Model definition

The final step consists of adapting the `apollo_probabilities` function to work with random coefficients. This step is in essence the easiest as the writing of the utility functions and probabilities is equivalent to the approach used in the models without random heterogeneity. This is

possible thanks to having defined the actual random coefficients in the `apollo_randCoeff` function which means that the user can now simply use the elements contained in the `randCoeff` list.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Function initialisation: do not change the following three commands
  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V = list()
  V[['alt1']] = asc_1 + b_tc*(v_tt*tt1 + tc1 + v_hw*hw1 + v_ch*ch1)
  V[['alt2']] = asc_2 + b_tc*(v_tt*tt2 + tc2 + v_hw*hw2 + v_ch*ch2)

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives  = c(alt1=1, alt2=2),
    avail         = list(alt1=1, alt2=1),
    choiceVar     = choice,
    V             = V
  )

  ### Compute probabilities using MNL model
  P[['model']] = apollo_mnl(mnl_settings, functionality)

  ### Average across intra-individual draws
  P = apollo_avgIntraDraws(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Average across inter-individual draws
  P = apollo_avgInterDraws(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 6.3: The `apollo_probabilities` function for a MMNL model

We illustrate this in Figure 6.3. As we can see, we still define the model as MNL, as this is the model structure conditional on the random coefficients. The only distinction with the earlier MNL example is that we make calls to two additional functions towards the end of `apollo_probabilities`. In the MNL example in Figure 4.7, we made a call to `apollo_panelProd`, which takes the product across choices for the same individual, before preparing the probabilities for output using `apollo_prepareProb`. In our MMNL model, the probabilities are however not now given by a vector with one value per choice task, but a cube with one colum per inter-individual draw in the second dimension, and one colum per intra-individual draw in the third dimension. The actual log-likelihood function for our model is thus given by:

$$L\left(\mathbf{\Omega}\right) = \prod_{n=1}^{N} \int_{\xi_{tc,n}} \int_{\xi_{tt,n}} \int_{\xi_{hw,n}} \prod_{t=1}^{T_n} \int_{\xi_{tt,n,t}} P_{j^*_{n,t}} \mathrm{d}\xi_{tt,n,t} \mathrm{d}\xi_{hw,n} \mathrm{d}, \xi_{tt,n} \mathrm{d}\xi_{tc,n}, \tag{6.15}$$

The two layers of integration need to be approximated using numerical simulation, where different functions are used for simulation at the inter-individual and intra-individual level. These

two functions, `apollo_avgIntraDraws()` and `apollo_avgInterDraws` are called as:

```
P = apollo_avgIntraDraws(P,
                         apollo_inputs,
                         functionality)
```

and

```
P = apollo_avgInterDraws(P,
                         apollo_inputs,
                         functionality)
```

In our example, we first average across intra-individual draws, using `apollo_avgIntraDraws`. We then take the product over choices, using `apollo_panelProd`, before averaging across the inter-individual draws using `apollo_avgInterDraws` to obtain a column vector once again, with one row per individual. We finally call `apollo_prepareProb`.

While the example here is for a MMNL model, i.e. a mixture of a MNL kernel, it is similarly possible to use for example a Mixed Nested Logit model, and in *Apollo*, this is straightforward by replacing `apollo_mnl` with `apollo_nl` and defining appropriate additional arguments.

6.1.4 Estimation

The estimation of a continuous Mixed Logit model uses the same routine `apollo_estimate` as our other models, and the code automatically finds the draws and random coefficients in `apollo_inputs`. This is illustrated in Figure 6.4, where we use 3 cores in estimation, and where the use of a MNL kernel inside the MNL model is made clear by the output.

6.1.5 Error components

The main focus when using random parameters in choice models is to introduce heterogeneity in sensitivities to individual attributes. However, random parameters can similarly be used as "error components", with a view to introducing for example heteroskedasticity or correlation across alternatives. An introduction to the use of error components is given in Train (2009, chapter 6.3). With error components, special care is required in relation to identification and normalisation, an issue ignored by many and discussed in detail by Walker et al. (2007).

The use of error components is straightforward in *Apollo*, and simply involves the use of random parameters that do not multiply an attribute[2]. We show two examples of code here, one of them for introducing heteroskedasticity, and the other for capturing the so called pseudo panel effect.

Let us assume we wanted to extend the example in Section 6.1.2 by allowing for heteroskedasticity in the utility for the first alternative, where this is at the level of an individual rather

[2]This is the *traditional* use of error components, although there are also cases where error components multiply attributes, such as in Hess et al. (2007b)

```
> model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

Testing likelihood function...

Overview of choices for MNL model component :
                                   alt1    alt2
Times available                 3492.00 3492.00
Times chosen                    1734.00 1758.00
Percentage chosen overall         49.66   50.34
Percentage chosen when available  49.66   50.34

Pre-processing likelihood function...
Preparing workers for multithreading...

Testing influence of parameters.............
Starting main estimation
Initial function value: -2406.92
Initial gradient value:
                     asc_1                mu_log_b_tc       sigma_log_b_tc_inter                mu_log_v_tt
             -12.81823643              -15.52002504               -7.78667663              11.50106846
     sigma_log_v_tt_inter     sigma_log_v_tt_inter_2       sigma_log_v_tt_intra                mu_log_v_hw
             -0.10508620              11.27132782               -0.04135154              33.79959253
     sigma_log_v_hw_inter sigma_log_v_hw_v_tt_inter                      v_ch        gamma_vtt_business
             -0.06491518               0.12617193              44.46988351               5.22909841
initial  value 2406.919551
iter   2 value 2383.868879
...
```

Figure 6.4: Running `apollo_estimate` for MMNL using 3 cores

than an observation. This involves adding a $N(0, \sigma_{hsk})$ term to the first utility. We show only those parts of the affected code in Figure 6.5, where any other lines remain as in Figures 6.1 to 6.3, and where we do not show the `apollo_beta` section which will now include the definition of the new parameter `sigma_hsk`. The user needs to create a new set of draws and a random component that creates the $N(0, \sigma_{hsk})$ term, called `ec` in our example, which is then added to the utility of the first alternative.

```
apollo_draws = list(
  ...
  interNormDraws = c("draws_hw_inter","draws_tt_inter","draws_hsk"),
  ...
)

apollo_randCoeff = function(apollo_beta, apollo_inputs){
  ...
  randcoeff[["ec"]] = sigma_hsk * draws_hsk
  ...
}

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){
  ...
  V[['alt1']] = asc_1 + b_tc*(v_tt*tt1 + tc1 + v_hw*hw1 + v_ch*ch1) + ec
  ...
}
```

Figure 6.5: Using error components for heteroskedasticity

Another popular use of error components aims to capture an individual-specific effect that creates correlation across choices for the same respondent, the so called pseudo-panel effect. It is clearly impossible for identification reasons to add the same error components to all J alternatives. As for example reported by Yáñez et al. (2011), it has become common practice to instead include the same error component in $J-1$ of the utility functions. Unfortunately, such a specification now introduces correlation across those $J-1$ alternatives, as well as heteroskedasticity. Adding

an error component to just a single alternative is no better, while randomly varying the omitted error component across respondents (cf. Yáñez et al., 2011) is also problematic, as it creates random variations in the correlation and/or heteroskedasticity structure across individuals. An alternative is to use an approach first put forward by Hess et al. (2008) which consists of adding *iid* (across respondents but not observations) error components to *all* of the alternatives, where this model is identified in panel data given the independent nature of the error components. We illustrate this process in Figure 6.6, again based on the example from Section 6.1.2. We now require two separate sets of draws, and create two independently but identically distributed $N(0, \sigma_{panel})$ terms in `apollo_randCoeff`. These use different draws for each error component, but the same parameter `sigma_panel` for the standard deviation. One term is then included in each utility function.

```
apollo_draws = list(
  ...
  interNormDraws = c("draws_hw_inter","draws_tt_inter","draws_panel_1","draws_panel_2"),
  ...
)

apollo_randCoeff = function(apollo_beta, apollo_inputs){
  ...
  randcoeff[["ec1"]] = sigma_panel * draws_panel_1
  randcoeff[["ec2"]] = sigma_panel * draws_panel_2
  ...
}

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){
  ...
  V[['alt1']] = asc_1 + b_tc*(v_tt*tt1 + tc1 + v_hw*hw1 + v_ch*ch1) + ec1
  V[['alt2']] = asc_2 + b_tc*(v_tt*tt2 + tc2 + v_hw*hw2 + v_ch*ch2) + ec2
  ...
}
```

Figure 6.6: Using error components for a pseudo-panel effect

6.2 Discrete mixtures and Latent Class

Apollo offers the same degree of flexibility with Latent Class and Discrete Mixture models as with continuous mixture models. Unlike with continuous mixtures, the α parameters in CNL, the σ parameters in MDCEV, and the θ parameters in MDCNEV can also vary across classes.

In a Latent Class model, heterogeneity is accommodated by making use of separate classes with different values for the vector β in each class. With S classes, we have S instances of β, say β_1 to β_S, with the possibility of some elements staying fixed across some classes. Individual n belongs to class s with probability $\pi_{n,s}$, where $0 \leq \pi_{n,s} \leq 1 \; \forall s$ and $\sum_{s=1}^{S} \pi_{n,s} = 1$.

Let $P_{i,n,t}(\beta_s)$ give the probability of respondent n choosing alternative i in choice situation t, conditional on n falling into class s, where $P_{i,n,t}$ is typically specified as a MNL model, but this is not a requirement in theory or in *Apollo*. Indeed, there is now a substantial body of work using different model structures (often based on different decision rules) in different classes (cf. Hess et al., 2012).

The unconditional (on s) choice probability is then given by:

$$P_{i,n,t}\left(\beta_1,\ldots,\beta_S\right) = \sum_{s=1}^{S} \pi_{n,s}\, P_{i,n,t}\left(\beta_s\right) \tag{6.16}$$

In the presence of repeated choice data, it is natural to perform the mixing at the level of individual people, and we then have that the probability of the sequence of choices/observations for person n is given by:

$$L_n\left(\beta\right) = \sum_{s=1}^{S} \pi_{n,s} \prod_{t=1}^{T_n} P_{j^*_{n,t}}\left(\beta_s\right) \tag{6.17}$$

where $\beta = \langle \beta_1,\ldots,\beta_S \rangle$, and where $j^*_{n,t}$ gives the alternative chosen by person n in choice situation t.

In the most basic version, the class allocation probabilities $\pi_{n,s}$ are constant across respondents, i.e. $\pi_{n,s} = \pi_s \,\forall n$. The real flexibility of the model arises when linking class allocation to socio-demographics, where we use a class allocation model, with typically an underlying Logit structure, such that:

$$\pi_{n,s} = \frac{e^{\delta_s + g(\gamma_s, z_n)}}{\sum_{l=1}^{S} e^{\delta_l + g(\gamma_l, z_n)}}, \tag{6.18}$$

where δ_s is an offset, and γ_s is a vector of parameters capturing the influence of the vector of individual characteristics z_n on the class allocation probabilities. For normalisation, δ_s is fixed to 0 for one of the S classes, as is γ_s. In a model with constant class allocation probabilities across individuals, we would only estimate the vector of constants δ.

To illustrate the implementation of Latent Class models in *Apollo*, we provide an example on the Swiss route choice data also used for the Mixed Logit model in Section 6.1.2. We develop a model with two classes, where all four marginal utility parameters (time, cost, headway and interchanges) vary across the classes, but where the ASCs are kept fixed across classes. For the class allocation model, we use two socio-demographic characteristics, namely whether an individual was on a commute journey or not, and whether they had a car available to them. This example is implemented in `Apollo_example_20.r`, where a simpler Latent Class model without covariates is available in `Apollo_example_18.r`.

The development of a Latent Class model in *Apollo* consists of two steps, which we now look at it turn.

Defining Latent Class parameters

We first implement a function called `apollo_lcPars`, which performs a role analogous to `apollo_randCoeff` for continuous mixtures. This is thus another function that is not called by the user but which is developed by the user for the specific model that is to be used. Like

`apollo_randCoeff`, this function takes `apollo_beta` and `apollo_inputs` as inputs and generates a new list which contains the parameters that vary across classes as well as the class allocation probabilities. The contents of `apollo_lcPars` will vary across model specifications, only the first line (`lcPars = list()`) and final line (`return(lpars)`) are to remain as in the example.

The implementation for our example is shown in Figure 6.7. We create a list called `lcPars` which contains the values for the different parameters across classes, as well as the class allocation probabilities. As can be seen from Figure 6.7, we first produce one element in the list for each of the four marginal utility coefficients. Each one of these elements is a list in itself, and contains the values for the coefficients across the two classes. If more classes are to be used, more entries are added into each one of these lists, where the possibility exists of keeping the values constant for some parameters across some or all of the classes (in which case the number of values still needs to be the same as the number of classes, but some of them are repeated). Note that for parameters that are kept constant across all (i.e. not just some) of the classes, such as the ASCs in our example, there is no need (though also no harm) to include them in `lcPars`.

```
apollo_lcPars=function(apollo_beta, apollo_inputs){
  lcpars = list()
  lcpars[["beta_tt"]] = list(beta_tt_a, beta_tt_b)
  lcpars[["beta_tc"]] = list(beta_tc_a, beta_tc_b)
  lcpars[["beta_hw"]] = list(beta_hw_a, beta_hw_b)
  lcpars[["beta_ch"]] = list(beta_ch_a, beta_ch_b)

  V=list()
  V[["class_a"]] = delta_a + gamma_commute_a*commute + gamma_car_av_a*car_availability
  V[["class_b"]] = delta_b + gamma_commute_b*commute + gamma_car_av_b*car_availability

  mnl_settings = list(
    alternatives = c(class_a=1, class_b=2),
    avail        = 1,
    choiceVar    = NA,
    V            = V
  )
  lcpars[["pi_values"]] = apollo_mnl(mnl_settings, functionality="raw")

  lcpars[["pi_values"]] = apollo_firstRow(lcpars[["pi_values"]], apollo_inputs)

  return(lcpars)
}
```

Figure 6.7: The `apollo_lcPars` function

We next calculate the class allocation probabilities, i.e. $\pi_{n,s}, \forall n, s$. We use a MNL model for the class allocation probabilities, and thus produce utility functions for the two classes. The utility for the second class could in our case simply be set to 0 as the parameters are all normalised to 0 in `apollo_fixed`. We then use the `apollo_mnl` function to calculate the probabilities, with two alternatives which are always both available (hence `avail=1`). Two points need noting here. First, unlike in other applications of the in-built functions, we now explicitly use the functionality `raw` when calling `apollo_mnl` - this ensures that the probabilities are returned for all alternatives, or in this case all classes. When using `raw`, we also do not need to define the chosen alternative, and thus set `choiceVar=NA`.

At this point, we obtain a value for the probability for the two classes for each row in the data. However, for the calculation in Equation 6.17, we require the class allocation probabilities at the individual rather than observation level, i.e. $\pi_{n,s}$. This is achieved by running the additional function `apollo_firstRow` on the part of `lcPars` containing the class allocation probabilities. This is a general function that can be applied to probabilities, data elements, etc. In particular,

by calling:

```
x = apollo_firstrow(x,
                    apollo_inputs)
```

we replace `x` by a version where only the first entry for each individual is retained. The object `x` can be a vector, matrix or cube. In our example, calling `apollo_firstrow(lcPars[["pi_values"]],apollo_inputs)` retains the first row in each element of `lcPars[["pi_values"]]` for each individual.

Model definition

We next turn to the calculation of the actual Latent Class choice probabilities, a process that is illustrated in Figure 6.8. As can be seen, we first create a generic version of `mnl_settings` that contains those settings which will be constant across classes, namely the alternatives, availabilities and choice variable.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define settings for MNL model component that are generic across classes
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2),
    avail        = list(alt1=1, alt2=1),
    choiceVar    = choice
  )

  ### Loop over classes
  for(s in 1:length(pi_values)){

    ### Compute class-specific utilities
    V=list()
    V[['alt1']]  = asc_1 + beta_tc[[s]]*tc1 + beta_tt[[s]]*tt1 + beta_hw[[s]]*hw1 + beta_ch[[s]]*ch1
    V[['alt2']]  = asc_2 + beta_tc[[s]]*tc2 + beta_tt[[s]]*tt2 + beta_hw[[s]]*hw2 + beta_ch[[s]]*ch2

    mnl_settings$V = V
    mnl_settings$componentName = paste0("Class_",s)

    ### Compute within-class choice probabilities using MNL model
    P[[paste0("Class_",s)]] = apollo_mnl(mnl_settings, functionality)

    ### Take product across observation for same individual
    P[[paste0("Class_",s)]] = apollo_panelProd(P[[paste0("Class_",s)]], apollo_inputs ,functionality)

  }

  ### Compute latent class model probabilities
  lc_settings   = list(inClassProb = P, classProb=pi_values)
  P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 6.8: Implementing choice probabilities for Latent Class

We then incorporate a loop over classes (where the number of classes is given by the number of entries in `pi_values`), where we calculate the utilities for the two alternatives in each class, using the appropriate values for those parameters that vary across classes. In each class, we update

`mnl_settings` to use the utilities in that specific class, and we also define a name for each class in the `componentName` setting, where we use the `paste0` function in `R` to combine the string `Class_` with the index for the class. This step is not compulsory but helps with interpreting the outputs. We finally create one element in the `P` list for each class, where we again give these a name using `paste0` rather than just an index. Again, this is not compulsory, and if a simple index is given (i.e. using `P[[s]]`), the class-specific fits will be referred to as `component_1`, `component_2`, etc. The reader will note that in class s, we are using the coefficients for that class (e.g. `beta_tc[[s]]` uses the sth element in `beta_tc` created in `apollo_lcPars`), and the call to `apollo_mnl` in each class uses the appropriate utilities for that class as these are updated inside the overall `mnl_settings` using `mnl_settings$V = V` in each step of the loop. In our example, we calculate the within class probabilities using a MNL model, where it would again also be possible to use different models inside the Latent Class structure, e.g. Nested Logit. In preparation for the averaging across classes, we take the product across choices for the same individual in each class, using `apollo_panelProd`, in line with Equation 6.17.

We now have the likelihoods in each class, i.e. for class s, we have $L_{n,s} = \prod_{t=1}^{T_n} P_{j^*_{n,t}}(\beta_s)$. The remaining step is to take the weighted average across classes, i.e. $\sum_{s=1}^{S} \pi_{n,s} L_{n,s}$. This is achieved by the `apollo_lc` function, which uses the within class probabilities contained in the S existing elements of `P`, multiplies each one by the appropriate class allocation probability in `pi_values`, and then sums across classes. This function is called as:

```
P[["model"]] = apollo_lc(lc_settings,
                         apollo_inputs,
                         functionality)
```

The list `lc_settings` contains two elements, namely:

`inClassProb`: A list of in class probabilities, i.e. the $L_{n,s}$ for different classes. These need to already have all continuous random heterogeneity averaged out and contain one entry per individual, i.e. having been multiplied across observations for the same individual.

`classProb`: A list of class allocation probabilities, which can be either scalars (if constant across people), vectors (if using only deterministic heterogeneity) or matrices or cubes (if including continuous random heterogeneity in the class allocation probabilities, a point we return to in Section 6.3).

The output from this function is the actual Latent Class model probability in Equation 6.17 and is stored in the `model` component of the list `P`.

One final point to note here relates to the class specific model components. In the discussion here, the probabilities for individual classes are stored inside `P`, in our case as `P[["Class_1"]]` and `P[["Class_2"]]`, while the probability for the model is stored in `P[["model"]]`. The output from model estimation will then also report the log-likelihood from the individual classes. When using a Latent Class model inside a hybrid structure, the within-class probabilities should however be stored in a separate list from `P`, a point we return to in Section 7.

The focus in our discussion so far has been on Latent Class models rather than Discrete Mixtures (cf. Hess et al., 2007a). In a Discrete Mixture model, we have S_k values for parameter β_k, where the number of possible values S_k can vary across parameters (and can be 1 for some). A weight $\pi_{n,k,s}$ is assigned to the sth value for β_k for person n, with $\sum_{s=1}^{S_k} \pi_{n,k,s} = 1, \forall\, n, k$. We thus have $\sum_{k=1}^{K} S_k$ possible values across the K different β parameters, and each combination is possible in the model. This effectively means that the Discrete Mixture model can be written as a Latent Class model with $\prod_{k=1}^{K} S_k$ classes, where, for example, the first class might use the first value for each of the coefficients, and thus have a class allocation probability $\pi^*_{n,s} = \prod_{k=1}^{K} \pi_{n,k,1}$. Discrete mixtures can thus be estimated using software for Latent Class, including in *Apollo*, and an example is given in `Apollo_example_19.r`. The use of Discrete Mixture models leads to a larger number of parameters, as we now have separate $\pi_{n,k,s}$ for different β parameters, as well as generally a larger number of overall classes (and hence a more complex likelihood function) given that $S^* = \prod_{k=1}^{K} S_k$. Latent class models also provide a more natural way of capturing correlation in the heterogeneity across different coefficients.

6.3 Combining Latent Class with continuous random heterogeneity

Apollo also allows users to combine continuous random heterogeneity with Latent Classes (cf. Greene and Hensher, 2013). Continuous heterogeneity can be allowed for both in the within-class probabilities and in the class membership probabilities. Specifically, let us assume that the vector π is distributed according to $f(\pi \mid \Omega_\pi)$ where Ω_π is a vector of parameters, while the vector β_s, which contains the parameters for the within-class model in class s is distributed according to $g_s(\beta_s \mid \Omega_{\beta_s})$, where Ω_{β_s} is a vector of parameters, and where $\Omega_\beta = \langle \Omega_{\beta_1}, ..., \Omega_{\beta_S} \rangle$. We then have:

$$L_n(\Omega_\pi, \Omega_\beta) = \int_\pi \sum_{s=1}^{S} \pi_{n,s} \left(\int_{\beta_s} \prod_{t=1}^{T_n} P_{j^*_{n,t}}(\beta_s)\, g_s(\beta_s \mid \Omega_{\beta_s})\, \mathrm{d}\beta_s \right) f(\pi \mid \Omega_\pi)\, \mathrm{d}\pi. \tag{6.19}$$

The integration across the distribution for heterogeneity in the within-class model is thus carried out prior to averaging across classes, while the integration across the distribution for heterogeneity in the class-allocation model is carried out outside the averaging across classes. For estimation, this implies averaging across draws in two distinct places, as we will now illustrate.

We extend the model from Section 6.2 by allowing the travel time coefficient to follow a negative lognormal distribution, with separate parameters in the two classes. In addition, we allow the constant in the class allocation model for the first class, i.e. δ_a, to follow a Normal distribution. The specification of the random parameters is illustrated in Figure 6.9, and is available in `Apollo_example_21.r`.

We have that `draws_tt` and `draws_pi` are standard Normal draws defined in `apollo_draws` (not shown here). We use a negative Lognormal distribution for `tt_a` and `tt_b`, and a Normal distribution for `delta_a`. These random time coefficients are then also used inside `apollo_lcPars` when defining `lcPars[["tt"]]`, while the randomly distributed `delta_a` is used when defining `V[["class_a"]]`.

```
apollo_randCoeff = function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["tt_a"]] = -exp(log_tt_a_mu + log_tt_a_sig*draws_tt)
  randcoeff[["tt_b"]] = -exp(log_tt_b_mu + log_tt_b_sig*draws_tt)
  randcoeff[["delta_a"]] = delta_a_mu  + delta_a_sig*draws_pi

  return(randcoeff)
}

apollo_lcPars = function(apollo_beta, apollo_inputs){
  lcpars = list()
  lcpars[["tt"]] = list(tt_a, tt_b)
  lcpars[["tc"]] = list(tc_a, tc_b)
  lcpars[["hw"]] = list(hw_a, hw_b)
  lcpars[["ch"]] = list(ch_a, ch_b)

  V=list()
  V[["class_a"]] = delta_a      + gamma_commute_a*commute + gamma_car_av_a*car_availability
  V[["class_b"]] = delta_b      + gamma_commute_b*commute + gamma_car_av_b*car_availability

  mnl_settings = list(
    alternatives = c(class_a=1, class_b=2),
    avail        = 1,
    choiceVar    = NA,
    V            = V
  )
  lcpars[["pi_values"]] = apollo_mnl(mnl_settings, functionality="raw")

  lcpars[["pi_values"]] = apollo_firstRow(lcpars[["pi_values"]], apollo_inputs)

  return(lcpars)
}
```

Figure 6.9: The `apollo_randCoeff` and `apollo_lcPars` functions for a Latent Class model with continuous random heterogeneity

The definition of the model probabilities differs from that of the simple Latent Class model in Figure 6.8 in only two ways. In particular, as seen in Figure 6.10, in line with Equation 6.19, we now average across the random draws in the within class likelihoods via `P[[s]] = apollo_avgInterDraws(P[[s]],apollo_inputs,functionality)`, after taking the product across observations for the same individual using `apollo_panelProd`. This gives one likelihood for the observed choices for each person within each class. We then perform the weighted summation across classes using `P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)`. As `pi_values` again incorporates random terms, we now have a version of the combined model likelihood for each draw for each individual. As a final step ,we then average across the continuous heterogeneity in the class allocation probabilities, using `P[["model"]] = apollo_avgInterDraws(P[["model"]], apollo_inputs, functionality)` to again give one value per person.

6.4 Multi-threading capabilities

Apollo allows for multi-threaded estimation for classical estimation[3], leading to significant estimation speed improvements for some models. This can easily be activated by specifying the number of threads to use in `apollo_control$nCores`. The recommended number of threads is equal to

[3]When using Bayesian estimation, the reliance on `RSGHB` means only single core processing is possible.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define settings for MNL model component that are generic across classes
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2),
    avail        = list(alt1=1, alt2=1),
    choiceVar    = choice
  )

  ### Loop over classes
  for(s in 1:length(pi_values)){

    ### Compute class-specific utilities
    V=list()
    V[['alt1']]   = asc1 + tc[[s]]*tc1 + tt[[s]]*tt1 + hw[[s]]*hw1 + ch[[s]]*ch1
    V[['alt2']]   = asc2 + tc[[s]]*tc2 + tt[[s]]*tt2 + hw[[s]]*hw2 + ch[[s]]*ch2

    mnl_settings$V = V
    mnl_settings$componentName = paste0("Class_",s)

    ### Compute within-class choice probabilities using MNL model
    P[[paste0("Class_",s)]] = apollo_mnl(mnl_settings, functionality)

    ### Take product across observation for same individual
    P[[paste0("Class_",s)]] = apollo_panelProd(P[[paste0("Class_",s)]], apollo_inputs ,functionality)

    ### Average across inter-individual draws within classes
    P[[paste0("Class_",s)]] = apollo_avgInterDraws(P[[paste0("Class_",s)]], apollo_inputs, functionality
        ↪ )

  }

  ### Compute latent class model probabilities
  lc_settings   = list(inClassProb = P, classProb=pi_values)
  P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)

  ### Average across inter-individual draws in class allocation probabilities
  P[["model"]] = apollo_avgInterDraws(P[["model"]], apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 6.10: Implementing choice probabilities for Latent Class with continuous random heterogeneity

the number of available processor cores in the machine minus one, which can be determined by typing `parallel::detectCores()` in the R console. The use of multi-threaded estimation comes with some restrictions.

apollo_probabilities can only access its arguments: In other words, the likelihood can only use data stored inside `apollo_beta` and `apollo_inputs`, where the latter combines `database`, `apollo_control`, `draws`, `apollo_randCoeff` and `apollo_lcPars`. All other variables created by the user in the global environment before estimation cannot be accessed. This issue is easily avoided by creating any new variables inside the `database` object prior to calling `apollo_validateInputs`, which is good practice anyway.

Data splitting: The dataset is split among several threads, so statistics such as the mean, maximum and minimum of variables, among others, will not be reliably calculated during

estimation when using multi-threading. To avoid this issue, any such statistic (for example the mean income in our MNL example in Section 4.2) need to be calculated before estimation and saved as a new variable inside `database`[4].

Increased memory consumption: Memory consumption is increased when using multi-threading. This is because the dataset and draws (usually the biggest objects in memory) need to split, and copied into several threads.

Speed gains are dependent on the model: In general, models using few iterations that each take a long time will benefit the most. This applies to models using big datasets or a large number of draws in the case of mixture models. Speed gains also decrease with the number of threads used. For small models, speed gains due to multi-threading might be negligible, or even negative due to overhead.

To help decide how many cores to use, we provide the function `apollo_speedTest`, which calculates the loglikelihood function several times using different number of threads and draws, and reports both the calculation time and the memory usage. This function is called as:

```
apollo_speedTest(apollo_beta,
                 apollo_fixed,
                 apollo_probabilities,
                 apollo_inputs,
                 speedTest_settings)
```

The final argument, `speedTest_settings`, is optional and allows the user to change the following settings:

`nDrawsTry`: A vector with the number of draws to try (default is `c(100, 200, 500)`). Note that this may need to be reduced for very complex models if memory issues arise.

`nCoresTry`: A vector with the number of threads to try (default is to try all cores present in the machine).

`nRep`: An integer setting the number of times `apollo_probabilities` is calculated for each possible pair of elements from `nDrawsTry` and `nCoresTry` (default is `10`), ensuring stable results for the calculation of runtimes.

We illustrate the use of this function in Figure 6.11 for example `apollo_example_16.r`, which shows a big benefit especially by using a second core. When running `apollo_speedTest`, progress and results of the test are printed to the console. Each row displays the set-up, progress, and results of a given configuration. The first column (*nCores*) indicates the number of computational threads in use, i.e. how many processor cores are being used simultaneously by `R`. The second

[4]To illustrate this issue, in our earlier example in Section 4.5.2, we created a variable called `mean_income` inside the database, prior to calling `apollo_validateInputs`, by calling `database$mean_income = mean(database$income)`. This ensures that with multi-threading, the same mean income would be used in each core, while, if the variable had been created inside `apollo_probabilities`, a different mean income would have been used across cores.

(*inter*) and third (*intra*) columns indicate the number of inter-individual and intra-individual draws used. The third column (*progress*) indicates the progress of the test for each set-up, each dot representing 10% of the repetitions requested. The fifth column (*sec/LLCal*) indicates the average time in seconds required to complete one evaluation of the `apollo_probabilities` function. The sixth and last column (*RAM(MB)*) presents a lower bound of the memory required to evaluate the `apollo_probabilities` function. After completing the test, results are summarised in a table indicating the time required to evaluate `apollo_probabilities` under each configuration, as well as in a plot.

```
> speedTest_settings=list(nDrawsTry = c(50, 75, 100),nCoresTry = 1:3,nRep = 10)

> apollo_speedTest(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, speedTest_settings)

         ___Draws___                  sec/
nCores  inter  intra   progress    LLCall RAM(MB)
      1     50     50 ..........   1.42    692.8
      2     50     50 ..........   0.84   1217.6
      3     50     50 ..........   0.66   1346.7
      1     75     75 ..........   4.59   1025.8
      2     75     75 ..........   2.13   1883.6
      3     75     75 ..........   1.53   2012.8
      1    100    100 ..........   6.62   1492.0
      2    100    100 ..........   4.30   2816.0
      3    100    100 ..........   4.07   2945.2

Summary of results (sec. per call to LL function)
        draws50 draws75 draws100
cores1   1.4235  4.5884   6.6190
cores2   0.8409  2.1271   4.2973
cores3   0.6619  1.5347   4.0708
```

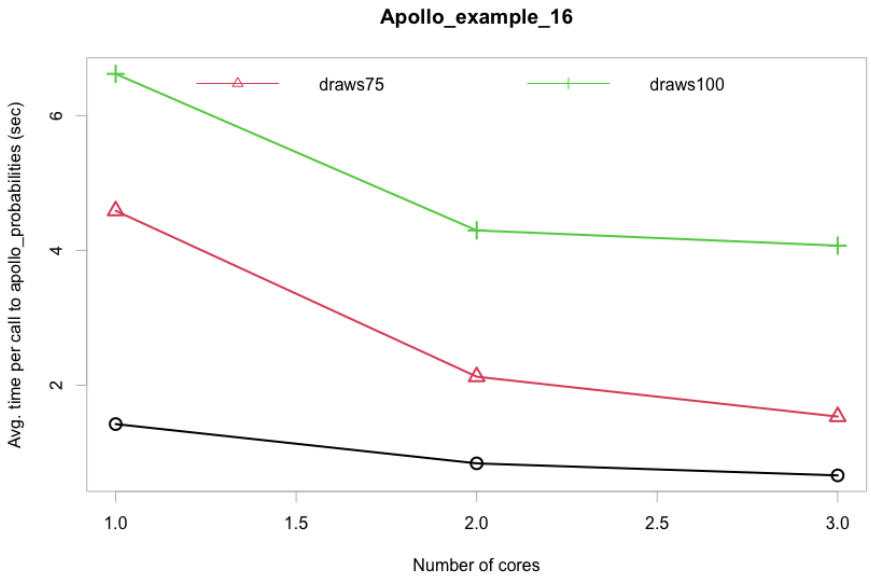

Figure 6.11: Running `apollo_speedTest`

Chapter 7

Joint estimation of multiple model components

The use of models made up of several separate components is made possible by the function `apollo_combineModels`, which is called as follows:

```
P = apollo_combineModels(P,
                         apollo_inputs,
                         functionality,
                         components,
                         asList)
```

This function takes the list `P` which contains several individual model components and produces a combined model. There is no limit on the number of subcomponents. The obvious case is estimation, where, with $L_{n,m}$ giving the likelihood of model component m for person n, the overall likelihood for person n is given by $L_n = \prod_{m=1}^{M} L_{n,m}$ (not showing here the presence of any integration over random terms, which would be carried out outside the product). The function `apollo_combineModels` creates the `model` object inside `P` as the product across individual components - when working with multiple model components, the individual components should thus not be called `model` themselves. The optional argument `components` allows the user to instruct the function to only use a subset of the components, while the optional argument `asList` (set to TRUE by default) allows the user to drop all components after taking the product (if `asList` is set to FALSE).

The most widely used case in recent years of models with multiple components is that of hybrid choice models. Before we turn to that example, we illustrate the joint estimation capabilities of *Apollo* by looking at two simpler cases of combining two models, namely the case of joint estimation on RP and SP data, and the estimation of best-worst data.

One important point should be mentioned already here. The function `apollo_combineModels` uses all elements inside the list `P`. Thus, the user needs to be careful that only components that

should be multiplied together for the overall model are included in `P`. In general, this will always be the case. However, a distinction arises in the presence of Latent Class models. As discussed in Section 6.2, for Latent Class, the user needs to first create the within-class models, before the weighted average of these is taken by `apollo_lc`, using the class allocation probabilities as weights. The within-class probabilities are in the simplest case stored inside `P`, as in our example in Section 6.2, and this has benefits in terms of showing the within-class likelihood in the output and facilitating the calculation of posteriors (cf. Section 9.11.2). However, when a Latent Class model is one of several components in a model, the use of `apollo_combineModels` would mean that the within-class probabilities are treated as a separate model component in creating the combined likelihood across models. To avoid this, the user can simply use a different list for the within class probabilities, e.g. `P_within_class` so that they are not stored in `P`.

7.1 Joint estimation on RP and SP data

The example `Apollo_example_22.r` uses the mode choice data we already covered for the earlier MNL model (cf. Section 4.5.2) but combines the RP and SP data. To allow for scale differences between the two data sources (Bradley and Daly, 1996; Hensher et al., 1998), we incorporate separate scale parameters μ_{RP} and μ_{SP} where the former is kept fixed to 1 for normalisation.

The basic setup is the same as in Section 4.5.2 with the exception that we omit `database = subset(database,database$SP==1)` used earlier in Figure 4.3 as we now utilise the entire sample. The earlier part of the code remains the same as in Figure 4.7 and is largely omitted here for conciseness of presentation - this includes the definition of `mu_RP` and `mu_SP` in `apollo_beta`, and the inclusion of `mu_RP` in `apollo_fixed`.

The key differences arise in the `apollo_probabilities` function, where we illustrate this in Figure 7.1. The definition of the utilities remains the same, with the difference that they are now calculated for all rows in the data, i.e. for RP rows as well as SP rows. In the example used here, the service quality attributes are coded as zero for the RP data and thus do not enter into the utility calculation for these rows.

We first calculate the probabilities for RP choices. The definition of `mnl_settings` differs from that in the SP model in Figure 4.7 in that we multiply the utilities by the RP scale parameter, μ_{RP}, e.g. $V_{i,n,t,RP} = \mu_{RP} V_{i,n,t}$, where we use `lapply` to cycle over the list of utilities. RP probabilities should only be calculated for RP rows in the data, and we thus include `rows=(RP==1)`, meaning that for SP rows in the data, the probability of RP choices is simply fixed to 1 so as not to contribute to model estimation. We then make the call to `apollo_mnl`, saving the output not in `P[["model"]]` which is reserved for the overall model, but in a subcomponent called `P[["RP"]]`. For the SP part of the data, we only need to change two components[1] in `mnl_settings`, namely using `lapply(V, "*", mu_SP)` instead of `lapply(V, "*", mu_RP)`, and `rows = (SP==1)` instead of `rows = (RP==1)`. We then calculate the probabilities for the SP rows in the data. For both RP and SP, we also use the `componentName` setting inside `mnl_settings` - this simply leads to labelling of model outputs using the names defined by the user for each component.

[1] As in previous examples, we do not rewrite the entire `mnl_settings` but just update individual components inside it.

The probability for the combined model is obtained by multiplying the RP and SP components together in `P[["model"]]`, which is the component used for estimation. Rather than doing this manually, we use `P = apollo_combineModels(P,apollo_inputs,functionality)`, as this function also prepares different output depending on the setting of `functionality`, allowing the use of joint models also in prediction, for example. The multiplication of probabilities across all observations for the same respondent happens after combining the two model components, using `apollo_combineModels`.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Create alternative specific constants and coefficients using interactions with socio-demographics
  asc_bus_value    = asc_bus  + asc_bus_shift_female * female
  asc_air_value    = asc_air  + asc_air_shift_female * female
  asc_rail_value   = asc_rail + asc_rail_shift_female * female
  b_tt_car_value   = b_tt_car + b_tt_shift_business * business
  b_tt_bus_value   = b_tt_bus + b_tt_shift_business * business
  b_tt_air_value   = b_tt_air + b_tt_shift_business * business
  b_tt_rail_value = b_tt_rail + b_tt_shift_business * business
  b_cost_value     = ( b_cost +  b_cost_shift_business * business ) * ( income / mean_income ) ^
      ↪ cost_income_elast

  ### List of utilities (before applying scales): these must use the same names as in mnl_settings,
      ↪ order is irrelevant
  V = list()
  V[['car']]  = asc_car        + b_tt_car_value  * time_car                          + b_cost_value *
      ↪ cost_car
  V[['bus']]  = asc_bus_value  + b_tt_bus_value  * time_bus  + b_access * access_bus  + b_cost_value *
      ↪ cost_bus
  V[['air']]  = asc_air_value  + b_tt_air_value  * time_air  + b_access * access_air  + b_cost_value *
      ↪ cost_air   + b_no_frills * ( service_air == 1 )  + b_wifi * ( service_air == 2 )  + b_food * (
      ↪ service_air == 3 )
  V[['rail']] = asc_rail_value + b_tt_rail_value * time_rail + b_access * access_rail + b_cost_value *
      ↪ cost_rail  + b_no_frills * ( service_rail == 1 ) + b_wifi * ( service_rail == 2 ) + b_food * (
      ↪ service_rail == 3 )

  ### Compute probabilities for the RP part of the data using MNL model
  mnl_settings = list(
    alternatives  = c(car=1, bus=2, air=3, rail=4),
    avail         = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar     = choice,
    V             = lapply(V, "*", mu_RP),
    rows          = (RP==1)
  )

  P[['RP']] = apollo_mnl(mnl_settings, functionality)

  ### Compute probabilities for the SP part of the data using MNL model
  mnl_settings$V = lapply(V, "*", mu_SP)
  mnl_settings$rows = (SP==1)

  P[['SP']] = apollo_mnl(mnl_settings, functionality)

  ### Combined model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 7.1: Joint RP-SP model on mode choice data

A subset of the model output is shown in Figure 7.2. We see that the model reports the

joint log-likelihood as well as the subcomponents for the two separate model components, and we obtain a scale parameter for the SP data (μ_{SP}) which is significantly larger than 1. It should be noted that in models with multiple components, the output in terms of diagnostics (cf. Figure 4.8) can become quite verbose as this information is reported for each model component, and a user may thus set `noDiagnostics` to FALSE in `apollo_control`. The diagnostics are reported in the order that the model components appear in the overall structure, in this case RP before SP. This part of the output uses any names defined by the user in `mnl_settings$componentName`. The benefit of having defined names here as `MNL-RP` and `MNL-SP` is that they are not both shown as simply coming from `MNL` components.

7.2 Joint best-worst model

Many stated choice surveys ask respondents for the most and least preferred alternatives, otherwise known as best-worst or BW (Lancsar et al., 2013). Although there is evidence that the behaviour in these two stages is not necessarily symmetrical (Giergiczny et al., 2017), such data is commonly analysed jointly, using in the simplest form a model where the probability for a given person n in choice task t is given by:

$$P_{n,t} = \frac{e^{V_{b_{n,t}}}}{\sum_{j=1} e^{V_{j,n,t}}} \cdot \frac{e^{-\mu_w V_{w_{n,t}}}}{\sum_{j \neq b_{n,t}} e^{-\mu_w V_{j,n,t}}}, \tag{7.1}$$

where $b_{n,t}$ is the most preferred alternative for respondent n in choice situation t while $w_{n,t}$ is the least preferred option. The above specification assumes that the best option is chosen first, and the worst is then chosen from the remaining set of alternatives, where the alternative with the lowest utility has the highest probability of being chosen (given the multiplication of the utilities by -1). A scale difference between the two stages is allowed for with the estimation of μ_w.

We illustrate the estimation of best-worst choice models on the drug choice data (cf. Figure 7.3), using the same utility specification as in 5.3.1, but looking at the best and worst choice stages only, where we allow for a difference in scale between the two stages. This example is available in `Apollo_example_23.r`, where we do not repeat the code showing the specification of the utilities, which is the same as in Figure 5.8. The model component for the first preference (best) is as in the Exploded Logit model. For the worst stage, we make three changes. We first adapt the availabilities by making the alternative chosen as the best alternative in the first stage unavailable in the second stage (assuming sequential choices). We next change the dependent variable to be `worst` rather than `best`, before multiplying the utilities by the negative of μ_{RP}, as in Equation 7.1. Of course, the same result could be achieved by making use of the `apollo_el` function with two stages, using the best and worst outcomes and a negative scale multiplier for the second stage. We show the use of two separate components here as this would also allow a user to change the actual utility function between the best and worst stage by for example allowing for differences in individual β terms going beyond just a generic scale difference (by redefining the utilities for the worst stage). This is not possible with `apollo_el` which allows for scale differences only. We again define names for the individual components using `mnl_settings$componentName`.

```
Model name                          : Apollo_example_22
Model description                   : RP-SP model on mode choice data
Model run at                        : 2021-02-04 19:03:44
Estimation method                   : bfgs
Model diagnosis                     : successful convergence
Number of individuals               : 500
Number of rows in database          : 8000
Number of modelled outcomes         : 8000
                                RP: 1000
                                SP: 7000

Number of cores used                :  1
Model without mixing

LL(start)                           : -9366.881
LL(0, whole model)                  : -9366.881
LL(final, whole model)              : -5802.644
Rho-square (0)                      :  0.3805
Adj.Rho-square (0)                  :  0.3786
AIC                                 :  11641.29
BIC                                 :  11767.06

LL(0,RP)                            : -1170.86
LL(final,RP)                        : -971.2441
LL(0,SP)                            : -8196.021
LL(final,SP)                        : -4831.4

Estimates:
                          Estimate        s.e.   t.rat.(0)    Rob.s.e. Rob.t.rat.(0)
asc_car                   0.000000          NA          NA          NA           NA
asc_bus                   0.124860    0.281042      0.4443    0.262032       0.4765
asc_air                  -0.396083    0.183661     -2.1566    0.177781      -2.2279
asc_rail                 -0.978683    0.180545     -5.4207    0.177525      -5.5129
asc_bus_shift_female      0.181337    0.064657      2.8046    0.071407       2.5395
asc_air_shift_female      0.134505    0.045471      2.9580    0.047351       2.8406
asc_rail_shift_female     0.098187    0.036644      2.6795    0.038414       2.5561
b_tt_car                 -0.006424  5.0942e-04    -12.6114  4.9442e-04     -12.9939
b_tt_bus                 -0.010507  9.6111e-04    -10.9321  8.7337e-04     -12.0304
b_tt_air                 -0.008668    0.001466     -5.9139    0.001432      -6.0517
b_tt_rail                -0.003838  9.2074e-04     -4.1682  8.9506e-04      -4.2878
b_tt_shift_business      -0.003203  3.4648e-04     -9.2440  3.4887e-04      -9.1806
b_access                 -0.010545    0.001531     -6.8857    0.001462      -7.2143
b_cost                   -0.038234    0.002461    -15.5356    0.002429     -15.7376
b_cost_shift_business     0.016656    0.001634     10.1910    0.001551      10.7398
cost_income_elast        -0.613155    0.029204    -20.9954    0.029760     -20.6033
b_no_frills               0.000000          NA          NA          NA           NA
b_wifi                    0.523122    0.043045     12.1530    0.043595      11.9995
b_food                    0.220074    0.030836      7.1369    0.031546       6.9764
mu_RP                     1.000000          NA          NA          NA           NA
mu_SP                     1.994746    0.126412     15.7798    0.122886      16.2325


Overview of choices for MNL model component RP:
                                    car     bus     air    rail
Times available                  778.00  902.00  752.00  874.00
Times chosen                     332.00  126.00  215.00  327.00
Percentage chosen overall         33.20   12.60   21.50   32.70
Percentage chosen when available  42.67   13.97   28.59   37.41


Overview of choices for MNL model component SP:
                                    car     bus     air     rail
Times available                 5446.00 6314.00 5264.00 6118.00
Times chosen                    1946.00  358.00 1522.00 3174.00
Percentage chosen overall         27.80    5.11   21.74   45.34
Percentage chosen when available  35.73    5.67   28.91   51.88
```

Figure 7.2: On screen output for RP-SP model

7.3 Hybrid choice model

We next turn to the use of *Apollo* for hybrid choice models (see Abou-Zeid and Ben-Akiva, 2014, for a recent overview), where we look at a simple implementation of a model with a single latent

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ...

  ### Compute probabilities for 'best' choice using MNL model
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2, alt3=3, alt4=4),
    avail        = list(alt1=1, alt2=1, alt3=1, alt4=1),
    choiceVar    = best,
    V            = V,
    componentName = "best"
  )
  P[['choice_best']] = apollo_mnl(mnl_settings, functionality)

  ### Compute probabilities for 'worst' choice using MNL model
  mnl_settings$avail        = list(alt1=(best!=1), alt2=(best!=2), alt3=(best!=3), alt4=(best!=4))
  mnl_settings$choiceVar    = worst
  mnl_settings$V            = lapply(V,"*",-mu_worst)
  mnl_settings$componentName = "worst"

  P[['choice_worst']] = apollo_mnl(mnl_settings, functionality)

  ### Combined model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 7.3: Best-Worst model on drug choice data

variable on the drug choice data described in Section 3.3. We use a dummy coded specification for the three categorical variables, along with a continuous specification for risk and cost. We specify a structural model for the latent variable that uses the three socio-demographic characteristics included in the data, and then use this latent variable in the utilities for the two branded alternatives as well as in the measurement models for the four attitudinal indicators. In our example, we do not incorporate additional random heterogeneity not linked to the latent variable, but this is entirely straightforward to do by including additional terms in `apollo_randCoeff`. Similarly, it is possible to combine latent variables with Latent Class structures. Indeed, latent variables are simply additional random components in a model.

Two different versions of the model are provided. The first of these, `Apollo_example_24.r` uses an Ordered Logit model for the indicators, as discussed by Daly et al. (2012b). The second example, `Apollo_example_25.r` uses the common simplification of treating the indicators as being normally distributed. We will now look at these two models in turn.

Specifically, we have that the latent variable for individual n is given by:

$$\alpha_n = \gamma' z_n + \eta_n, \tag{7.2}$$

where z_n is a vector combining the three socio-demographic variables for individual n, γ is a vector of estimated parameters capturing the impact of these variables on α_n and η_n is a random disturbance which follows a standard Normal distribution across individuals, i.e. $\eta_n \sim N(0,1)$.

The utility for alternative j in choice situation t for individual n is given by:

$$\begin{aligned} V_{j,n,t} = & \sum_{s=1}^{5} \beta_{brand_s} \cdot \left(x_{brand_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{6} \beta_{country_s} \cdot \left(x_{country_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{3} \beta_{characteristic_s} \cdot \left(x_{characteristic_{j,n,t}} == s\right) \\ & + \beta_{side\_effects} \cdot x_{side\_effects_{j,n,t}} \\ & + \beta_{price} \cdot x_{price_{j,n,t}} \\ & + \lambda \cdot \alpha_n \cdot (j \leq 2)\,. \end{aligned} \tag{7.3}$$

For the first five rows, the same specification as in Section 5.3.1 and Section 7.2 is used, with dummy coding for the categorical variables and a continuous treatment of risk and price. Finally, the inclusion of the latent variable, i.e. $\lambda \cdot \alpha_n$ only applies to the first two alternatives, i.e. the branded products. We thus get that the likelihood of the observed sequence of T_n choices for person n, conditional on β and α_n, is given by:

$$L_{C_n}\left(\beta, \alpha_n\right) = \prod_{t=1}^{T_n} \frac{e^{V_{j^*_{n,t}}}}{\sum_{j=1}^{4} e^{V_{j,n,t}}}, \tag{7.4}$$

where $j^*_{n,t}$ is the alternative chosen by respondent n in task t.

The latent variable is also used to explain the value of the four attitudinal questions, where two different specifications are used in our example.

With the Ordered Logit model, we have that:

$$L_{I_n,ordered}\left(\tau, \zeta, \alpha_n\right) = \prod_{i=1}^{4} \left(\sum_{s=1}^{S} \delta_{(I_{n,i}=s)} \left[\frac{e^{\tau_{i,s} - \zeta_i \alpha_n}}{1 + e^{\tau_{i,s} - \zeta_i \alpha_n}} - \frac{e^{\tau_{i,s-1} - \zeta_i \alpha_n}}{1 + e^{\tau_{i,s-1} - \zeta_i \alpha_n}} \right] \right), \tag{7.5}$$

where ζ_i is an estimated parameter that measures the impact of α_n on the attitudinal indicator I_i, and $\tau_{i,.}$ is a vector of threshold parameters for this indicator.

With the continuous measurement model, we instead have that:

$$L_{I_n,normal}\left(\sigma, \zeta, \alpha_n\right) = \prod_{i=1}^{4} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{\left(I_{n,i} - \bar{I}_i - \zeta_i \alpha_n\right)^2}{2\sigma_i^2}} \tag{7.6}$$

where ζ_i is an estimated parameter that measures the impact of α_n on the attitudinal indicator I_i, and σ_i is an estimated standard deviation. By subtracting the mean of the indicator across

the sample, i.e. using $I_{n,i} - \bar{I}_i$, we avoid the need to estimate the mean of the normal density. This is most easily done as a data transformation straight after loading the data, as shown in `Apollo_example_25.r` .

The combined log-likelihood for the model is then given by:

$$LL\left(\gamma, \zeta, \tau, \beta\right) = \sum_{n=1}^{N} log \int_{\eta_n} L_{C_n}\left(\beta, \alpha_n\right) L_{I_n, ordered}\left(\tau, \zeta, \alpha\right) \phi\left(\eta_n\right) \mathrm{d}\eta_n, \tag{7.7}$$

with the ordered model, where we would replace $L_{I_n,ordered}\left(\tau, \zeta, \alpha\right)$ by $L_{I_n,normal}\left(\sigma, \zeta, \alpha\right)$ for the continuous measurement model (Equation 7.6 instead of Equation 7.5). This log-likelihood function requires integration over the random component in the latent variable, where this integral is then approximated using numerical simulation.

For conciseness, we do not here reproduce the obvious parts of the code relating to the definition of parameters or basic settings. In Figure 7.4, we start by creating 100 inter-individual standard Normal draws for η based on Halton draws. We then define a single random component inside `apollo_randCoeff`, where this is for the latent attitude α_n, in line with Equation 7.2, which includes deterministic heterogeneity through the inclusion of socio-demographic effects.

```
### Set parameters for generating draws
apollo_draws = list(
  interDrawsType="halton",
  interNDraws=100,
  interUnifDraws=c(),
  interNormDraws=c("eta")
)

### Create random parameters
apollo_randCoeff=function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["LV"]] = gamma_reg_user*regular_user + gamma_university*university_educated + gamma_age_50*
      ↪ over_50 + eta

  return(randcoeff)
}
```

Figure 7.4: Hybrid choice model: draws and latent variable

Figure 7.5 shows the implementation of the hybrid model in the `apollo_probabilities` function for the example with an ordered measurement model, i.e. `Apollo_example_24.r`. We create a list `P` which will in the end have five components, namely the probabilities of the four measurement models and the probabilities from the choice model. We first compute the probabilities for the four Ordered Logit measurement models, one for each attitudinal statement, where these explain the values for the attitudinal indicators as a function of the latent variable, as detailed in Equation 7.5. For details on the syntax of, refer to 5.3.2. One point to note here is the inclusion of `rows=(task==1)` which ensures that the measurement model is only used once for each attitudinal statement and for each individual, rather than contributing to the overall model likelihood in each row for that person. This is in line with the rows in the data being for choice tasks, and the answers to attitudinal questions being repeated in the data in each row.

We next turn to the calculation of the probabilities for the choice model component of the hybrid model. The definition of `alternatives`, `avail` and `choiceVar` is as before. The core

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Likelihood of indicators
  ol_settings1 = list(outcomeOrdered = attitude_quality,
                      V              = zeta_quality*LV,
                      tau            = list(tau_quality_1, tau_quality_2, tau_quality_3, tau_quality_4),
                      rows           = (task==1),
                      componentName  = "indic_quality")
  ...
  ol_settings4 = list(outcomeOrdered = attitude_dominance,
                      V              = zeta_dominance*LV,
                      tau            = list(tau_dominance_1, tau_dominance_2, tau_dominance_3,
                           ↪ tau_dominance_4),
                      rows           = (task==1),
                      componentName  = "indic_dominance")
  P[["indic_quality"]]     = apollo_ol(ol_settings1, functionality)
  P[["indic_ingredients"]] = apollo_ol(ol_settings2, functionality)
  P[["indic_patent"]]      = apollo_ol(ol_settings3, functionality)
  P[["indic_dominance"]]   = apollo_ol(ol_settings4, functionality)

  ### Likelihood of choices
  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V = list()
  V[['alt1']] = ( b_brand_Artemis*(brand_1=="Artemis") + b_brand_Novum*(brand_1=="Novum")
                + b_country_CH*(country_1=="Switzerland") + b_country_DK*(country_1=="Denmark") +
                    ↪ b_country_USA*(country_1=="USA")
                + b_char_standard*(char_1=="standard") + b_char_fast*(char_1=="fast acting") +
                    ↪ b_char_double*(char_1=="double strength")
                + b_risk*side_effects_1
                + b_price*price_1
                + lambda*LV )
  V[['alt2']] = ( b_brand_Artemis*(brand_2=="Artemis") + b_brand_Novum*(brand_2=="Novum")
                + b_country_CH*(country_2=="Switzerland") + b_country_DK*(country_2=="Denmark") +
                    ↪ b_country_USA*(country_2=="USA")
                + b_char_standard*(char_2=="standard") + b_char_fast*(char_2=="fast acting") +
                    ↪ b_char_double*(char_2=="double strength")
                + b_risk*side_effects_2
                + b_price*price_2
                + lambda*LV )
  V[['alt3']] = ( b_brand_BestValue*(brand_3=="BestValue") + b_brand_Supermarket*(brand_3=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_3=="PainAway")
                + b_country_USA*(country_3=="USA") + b_country_IND*(country_3=="India") + b_country_RUS
                    ↪ *(country_3=="Russia") + b_country_BRA*(country_3=="Brazil")
                + b_char_standard*(char_3=="standard") + b_char_fast*(char_3=="fast acting")
                + b_risk*side_effects_3
                + b_price*price_3 )
  V[['alt4']] = ( b_brand_BestValue*(brand_4=="BestValue") + b_brand_Supermarket*(brand_4=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_4=="PainAway")
                + b_country_USA*(country_4=="USA") + b_country_IND*(country_4=="India") + b_country_RUS
                    ↪ *(country_4=="Russia") + b_country_BRA*(country_4=="Brazil")
                + b_char_standard*(char_4=="standard") + b_char_fast*(char_4=="fast acting")
                + b_risk*side_effects_4
                + b_price*price_4 )

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2, alt3=3, alt4=4),
    avail        = list(alt1=1, alt2=1, alt3=1, alt4=1),
    choiceVar    = best,
    V            = V,
    componentName= "choice"
  )

  ### Compute probabilities for MNL model component
  P[["choice"]] = apollo_mnl(mnl_settings, functionality)

  ### Likelihood of the whole model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Average across inter-individual draws
  P = apollo_avgInterDraws(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 7.5: Hybrid choice model with ordered measurement model: defining probabilities

```
### Load data
database <- read.csv("apollo_drugChoiceData.csv")

database$attitude_quality=database$attitude_quality-mean(database$attitude_quality)
database$attitude_ingredients=database$attitude_ingredients-mean(database$attitude_ingredients)
database$attitude_patent=database$attitude_patent-mean(database$attitude_patent)
database$attitude_dominance=database$attitude_dominance-mean(database$attitude_dominance)

...

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ...
  ### Likelihood of indicators
  normalDensity_settings1 = list(outcomeNormal = attitude_quality,
                                 xNormal       = zeta_quality*LV,
                                 mu            = 0,
                                 sigma         = sigma_qual,
                                 rows          = (task==1),
                                 componentName = "indic_quality")
  ...
  normalDensity_settings4 = list(outcomeNormal = attitude_dominance,
                                 xNormal       = zeta_dominance*LV,
                                 mu            = 0,
                                 sigma         = sigma_domi,
                                 rows          = (task==1),
                                 componentName = "indic_dominance")
  P[["indic_quality"]]     = apollo_normalDensity(normalDensity_settings1, functionality)
  P[["indic_ingredients"]] = apollo_normalDensity(normalDensity_settings2, functionality)
  P[["indic_patent"]]      = apollo_normalDensity(normalDensity_settings3, functionality)
  P[["indic_dominance"]]   = apollo_normalDensity(normalDensity_settings4, functionality)

  ...
}
```

Figure 7.6: Hybrid choice model with continuous measurement model: zero-centering indicators and defining probabilities

part of the utility functions is as in Figure 5.8, with the addition that the latent variable α_n is introduced into the utilities for the first two alternatives only, multiplied by a common parameter (λ), as shown in Equation 7.3.

The list `P` now contains five individual components, and the call to `apollo_combineModels` combines these into a joint model, prior to multiplying across choices for the same individual and averaging across draws, using the by now well known functions.

In `Apollo_example_25.r`, we use a continuous measurement model for the indicators. In line with Equation 7.6, we wish to avoid the estimation of the means for the latent variable. This is achieved by zero-centering the indicators, a process that needs to take place at the database level, prior to the call to `apollo_validateInputs` to ensure that these new variables are identical across cores in a multi-core setting. We show this part of the code in Figure 7.6, along with the part of `apollo_probabilities` which changes, which is only the treatment of the indicators, where we now use the function `apollo_normalDensity`, with details available in Section 5.3.3.

Chapter 8

Bayesian estimation

Apollo allows the user to replace classical estimation by Bayesian estimation, for all models. We do not provide details here on Bayesian theory but instead refer the reader to Lenk (2014) and the references therein. Bayesian estimation in *Apollo* makes use of the `RSGHB` package, and the user is referred to the documentation in Dumont and Keller (2019) for `RSGHB`-specific settings. The key advantage for the user is that *Apollo* provides a wrapper around `RSGHB` so that the syntax in `apollo_probabilities` does not change when a user moves from classical to Bayesian estimation. In addition, *Apollo* implements a large number of checks, most notably ensuring that all parameters defined by the user actually impact on the likelihood of the model. It also reports whether `RSGHB` applied any censoring to the probabilities.

To explain the process, we now look at the estimation of a Mixed Logit version of the mode choice model from Section 4.5.2, which is included in `Apollo_example_26.r`. We use Normal distributions for the three ASCs, negative Lognormal distributions for the time and cost coefficients, censored Normal distributions (with negative values fixed to zero) for the wifi and food parameters, and non-random parameters for all other terms.

The first steps in the model definition are shown in Figure 8.1, where we define the individual coefficients and their starting values in `apollo_beta`. In Bayesian estimation, the values given here are the means of the underlying Normal distributions. As a result, we use starting values of -3 for the underlying mean of the Lognormally distributed coefficients, i.e. the mean of the logarithm of the coefficients. The definition of `apollo_fixed` is also as in the MNL model. However, only parameters that are non-random across individuals can be kept fixed via `apollo_fixed`. `RSGHB` also allows users to have random parameters where the mean and/or standard deviation are fixed (cf. Dumont and Keller, 2019).

We next create a list called `apollo_HB`, containing settings for the Bayesian estimation of the model. The only requirement when using Bayesian estimation in *Apollo* is that this list must contain an element called `hbDist`. It can also include any other setting as described in the documentation of the `RSGHB` package (for more info, type `?RSGHB::doHB` in the `R` console). The following is a non-exhaustive list of the most relevant setting to be included in `apollo_HB`:

`hbDist:` This is the only mandatory setting to be included in `apollo_HB`. It is a vector giving the name of each model parameter and indicating the distribution to be used. This replaces

```
apollo_beta=c(asc_car                = 0,
              asc_bus                = 0,
              asc_air                = 0,
              asc_rail               = 0,
              asc_bus_shift_female   = 0,
              asc_air_shift_female   = 0,
              asc_rail_shift_female  = 0,
              b_tt_car               =-3,
              b_tt_bus               =-3,
              b_tt_air               =-3,
              b_tt_rail              =-3,
              b_tt_shift_business    = 0,
              b_access               =-3,
              b_cost                 =-3,
              b_cost_shift_business  = 0,
              cost_income_elast      = 0,
              b_no_frills            = 0,
              b_wifi                 = 0,
              b_food                 = 0)

apollo_fixed = c("asc_car","b_no_frills")

apollo_HB = list(
  hbDist          = c(asc_car                = "F",
                      asc_bus                = "N",
                      asc_air                = "N",
                      asc_rail               = "N",
                      asc_bus_shift_female   = "F",
                      asc_air_shift_female   = "F",
                      asc_rail_shift_female  = "F",
                      b_tt_car               = "LN-",
                      b_tt_bus               = "LN-",
                      b_tt_air               = "LN-",
                      b_tt_rail              = "LN-",
                      b_tt_shift_business    = "F",
                      b_access               = "LN-",
                      b_cost                 = "LN-",
                      b_cost_shift_business  = "F",
                      cost_income_elast      = "F",
                      b_no_frills            = "F",
                      b_wifi                 = "CN+",
                      b_food                 = "CN+"),
  gNCREP          = 50000, # burn-in iterations
  gNEREP          = 20000, # post burn-in iterations
  gINFOSKIP       = 500)
```

Figure 8.1: Bayesian estimation in *Apollo*: model settings

the vector `gdist` in `RSGHB`, which requires the user to use numeric coding for distributions. All parameters in `apollo_beta` should be included in the `hbDist` vector. There are seven possible distributions, as follows.

- `"NR"`: non-random (*fixed*) parameters. This setting should also be used for parameters that are included in `apollo_fixed`.
- `"N"`: normally distributed random parameters.
- `"LN+"`: positive lognormally distributed random parameters.
- `"LN-"`: negative lognormally distributed random parameters;.
- `"CN+"`: normally distributed random parameters, bounded below at 0.
- `"CN-"`: normally distributed random parameters, bounded above at 0.
- `"JSB"`: Johnson SB distributed random parameters.

`gNCREP`: number of burn-in iterations to use prior to convergence (default=10^5).

`gNEREP`: number of iterations to keep for averaging after convergence has been reached (default=10^5).

`gINFOSKIP`: number of iterations between printing/plotting information about the iteration process (default=250).

`constraintNorm`: a character vector with constraints on random coefficients. For example, `c("b1>b2", "b1<0")` indicates that all draws of parameter `b1` must be bigger than the corresponding draw of `b2`, and that all draws from `b1` should be smaller than zero. Supported constraints are of the form `"b1>b2"`, `"b1<b2"`, `"b1>0"`, and `"b1<0"`, where `b1` and `b2` are the names of parameters. Constraints can also be expressed using numerical coding of the parameters as described in the documentation of the `RSGHB` package.

`fixedA`: Named numeric vector. Contains the names and fixed mean values of random parameters. For example, `c(b1=0)` fixes the mean of parameter `b1` to zero.

`fixedD`: Named numeric vector. Contains the names and fixed variance of random parameters. For example, `c(b1=1)` fixes the variance of `b1` to zero.

Additional settings can be included in `apollo_HB` as described in the documentation of the `RSGHB` package. For more information type `?RSGHB::doHB` in the `R` console. Settings `modelname`, `gVarNamesFixed`, `gVarNamesNormal`, `gDIST`, `svN` and `FC` should *not* be included in `apollo_HB`, as these are automatically set by *Apollo*.

The `apollo_probabilities` function is exactly the same as for the MNL model shown in Figure 4.7 and is thus not reproduced here. When using Bayesian estimation, the use of `apollo_avgInterDraws` and `apollo_avgIntraDraws` does not apply even in the presence of random coefficients and these functions should not be used. In addition, the call to `apollo_panelProd` should not be made as `RSGHB` automatically groups together observations for the same individual.

The call to `apollo_estimate` is made in exactly the same way as with classical estimation. The estimation process is illustrated in Figure 8.2 for the text output and Figure 8.3 for a graphical output of the chains. In the text output, we show the first and final iteration, where this also highlights the way in which `RSGHB` confirms the distributions used at the outset. Note that `RSGHB` refers to non-random parameters as *fixed*, even if their values are estimated - the terminology refers to them being fixed across individuals.

The post-estimation output from a model using Bayesian estimation is substantially different from that with classical estimation, and is summarised in Figure 8.4. The early information on model name etc is the same as with classical estimation. This is followed by average model fit statistics across the post burn-in iterations. Next, we have convergence reports for the parameter chains, where these use the Geweke test (Geweke, 1992). The next four parts of the output look at summaries of the parameter chains, each time giving the mean and standard deviation across the post burn-in iterations for each parameter, where these results are divided into the non-random coefficients, the means for the underlying Normals, and the covariance matrix (split across two tables, with the mean and standard deviations of each entry in the covariance matrix). Finally, the output reports the means and standard deviations for the posteriors, where these are for the actual coefficients, i.e. taking into account the distributions used, rather than looking at the underlying Normals. All the values used for these components are also available in the `model` object after estimation and can be used for plotting. The use of `apollo_saveOutput` operates as before, but if `saveEst==TRUE`, the code additionally saves the output files produced by `RSGHB`, which can be very large in size (cf. Dumont and Keller, 2019).

```
> model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)



Diagnostic checks passed. Please review before proceeding
------------------------------------------------------------

                     Number of Individuals:          500
                    Number of Observations:         7000
                             Prior variance:            2
                 Target Acceptance (Fixed):          0.3
                Target Acceptance (Normal):          0.3
                         Degrees of Freedom:            5
 Avg. Number of Observations per Individual:           14
                    Initial Log-Likelihood:  -19242.47502

------------------------------------------------------------

      Fixed Parameters Start
  asc_bus_shift_female     0
  asc_air_shift_female     0
 asc_rail_shift_female     0
   b_tt_shift_business     0
 b_cost_shift_business     0
     cost_income_elast     0

------------------------------------------------------------

 Random Parameters Start Dist.
           asc_bus     0     N
           asc_air     0     N
          asc_rail     0     N
          b_tt_car    -3   LN-
          b_tt_bus    -3   LN-
          b_tt_air    -3   LN-
         b_tt_rail    -3   LN-
          b_access    -3   LN-
            b_cost    -3   LN-
            b_wifi     0   CN+
            b_food     0   CN+

------------------------------------------------------------

...

------------------------------------------------------------
Iteration:      70000
------------------------------------------------------------

              RHO (Fixed): 2.552501553e-06
  Acceptance Rate (Fixed):            0.27
             RHO (Normal):    0.3597337714
 Acceptance Rate (Normal):           0.274
            Parameter RMS:    0.8041928485
            Avg. Variance:    0.6332235963
           Log-Likelihood:    -4679.250243
                      RLH:    0.5321134338

------------------------------------------------------------

       Fixed Parameters     Estimate
  asc_bus_shift_female:  0.155267594
  asc_air_shift_female:  0.179787768
 asc_rail_shift_female:  0.049740490
   b_tt_shift_business: -0.007606631
 b_cost_shift_business:  0.025081677
     cost_income_elast: -0.657755249

------------------------------------------------------------

 Random Parameters    Estimate
           asc_bus: -1.26844325
           asc_air: -0.88899095
          asc_rail: -1.68400949
          b_tt_car: -4.60350381
          b_tt_bus: -4.17958902
          b_tt_air: -4.79469223
         b_tt_rail: -5.54448074
          b_access: -4.15475692
            b_cost: -2.69130377
            b_wifi:  0.86761015
            b_food:  0.04980724

------------------------------------------------------------
Time per iteration: 0.0497 secs
Time to completion: 0 mins
------------------------------------------------------------
Estimation complete.
WARNING: RSGHB has censored the probabilities. Please note that in
  at least some iterations RSGHB has avoided numerical issues by
  left censoring the probabilities. This has the side effect of
  zero or negative probabilities not leading to failures
```

Figure 8.2: Bayesian estimation in *Apollo*: estimation process

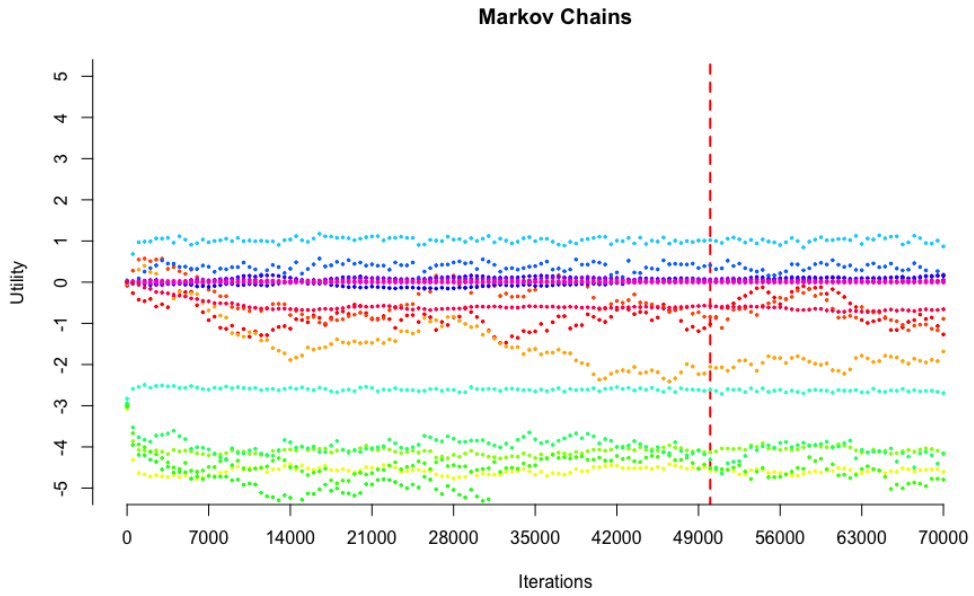

Figure 8.3: Bayesian estimation in *Apollo*: estimation process (parameter chains)

In classical estimation, *Apollo* creates an object `estimates` in the `model` list created after estimation, containing the final parameter values. When using Bayesian estimation, `model$estimates` is also produced, combining non-random parameters with individual specific posteriors for random parameters. This allows the user to use `apollo_prediction` and `apollo_llCalc` on such outputs, where care is of course required in interpretation of outputs based on posterior means.

```
> apollo_modelOutput(model)

Model run using Apollo for R, version 0.2.4 on Darwin by stephane.hess
www.ApolloChoiceModelling.com

Model name                              : Apollo_example_26
Model description                       : HB model on mode choice SP data
Estimation method                       : Hierarchical Bayes

Average post. LL post burn-in           : -4629.457
Average post. RLH post burn-in          : 0.5362


Chain convergence report (Geweke test)

Fixed (non random) parameters (t-test value for Geweke test)
 asc_bus_shift_female  asc_air_shift_female asc_rail_shift_female   b_tt_shift_business
              -4.9291                -5.7769                7.5432               -0.1500
...

Random parameters (t-test value for Geweke test)
  asc_bus   asc_air  asc_rail  b_tt_car  b_tt_bus  b_tt_air b_tt_rail  b_access    b_cost    b_wifi
   0.2384    0.0696   -3.2800    0.0959    0.0235    2.4283   -1.2188   -3.8897    0.5057   -0.3900
...

Covariances of random parameters (t-test value for Geweke test)
    asc_bus_asc_bus      asc_air_asc_bus      asc_air_asc_air     asc_rail_asc_bus     asc_rail_asc_air
             0.4194               1.2288               0.8878               2.0205              -0.9210
...

Summary of parameter chains

Non-random coefficients
                          Mean     SD
asc_car                 0.0000     NA
asc_bus_shift_female    0.0834 0.0307
...

Upper level model results for mean parameters for underlying Normals
             Mean     SD
asc_bus   -0.5754 0.3610
asc_air   -0.7175 0.2733
...

Upper level model results for covariance matrix for underlying Normals (means across iterations)
          asc_bus asc_air asc_rail b_tt_car b_tt_bus b_tt_air b_tt_rail b_access  b_cost  b_wifi  b_food
asc_bus    0.2204 -0.0177  -0.0058   0.0017   0.0310  -0.0131   -0.0138  -0.0033 -0.0010 -0.0136  0.0149
asc_air   -0.0177  0.1471   0.0442  -0.0125  -0.0072   0.0166   -0.0172   0.0212  0.0089 -0.0059 -0.0135
...

Upper level model results for covariance matrix for underlying Normals (SD across iterations)
          asc_bus asc_air asc_rail b_tt_car b_tt_bus b_tt_air b_tt_rail b_access b_cost b_wifi b_food
asc_bus    0.0787  0.0515   0.0491   0.0202   0.0207   0.0485    0.0432   0.0611 0.0158 0.0577 0.0577
asc_air    0.0515  0.0440   0.0339   0.0149   0.0176   0.0420    0.0474   0.0456 0.0141 0.0440 0.0435
...

Summary of distributions of random coeffients (after distributional transforms)
             Mean     SD
asc_bus   -0.5767 0.4664
asc_air   -0.7213 0.3814
...

Covariance matrix of random coeffients (after distributional transforms)
          asc_bus asc_air asc_rail b_tt_car b_tt_bus b_tt_air b_tt_rail b_access b_cost  b_wifi  b_food
asc_bus    0.2175 -0.0165  -0.0037    0e+00   -5e-04    1e-04         0    0e+00  1e-04 -0.0114  0.0131
asc_air   -0.0165  0.1455   0.0431    1e-04    1e-04   -2e-04         0   -4e-04 -6e-04 -0.0066 -0.0087
...

Correlation matrix of random coeffients (after distributional transforms)
          asc_bus asc_air asc_rail b_tt_car b_tt_bus b_tt_air b_tt_rail b_access  b_cost  b_wifi  b_food
asc_bus    1.0000 -0.0929  -0.0212  -0.0186  -0.2784   0.0670    0.0651   0.0093  0.0147 -0.0587  0.0765
asc_air   -0.0929  1.0000   0.3010   0.1369   0.0760  -0.1130    0.0965  -0.1184 -0.1026 -0.0418 -0.0619
...

Results for posterior means for random coefficients
             Mean     SD
asc_bus   -0.5754 0.0356
asc_air   -0.7173 0.0485
...
```

Figure 8.4: Bayesian estimation in *Apollo*: output (extracts)

Chapter 9

Pre and post-estimation capabilities

A large number of additional functions are provided in *Apollo* to allow the user to analyse the results after estimation. The outputs from these functions are not saved in the model output files, and it is then helpful for a user to dump the additional output to a text file, for example using `sink(paste(model$apollo_control$modelName,"_additional_output.txt",sep=""), split=TRUE)` which produces a new text file using the name of the current model. Outputs in the console are then also written into this text file, and writing to file can be stopped via `if(sink.number()>0) sink()`. We will now look at these various functions in turn.

9.1 Pre-estimation analysis of choices

With labelled choice data (or even unlabelled data where there may be strong left-right effects), it can be useful to analyse the choices before model estimation to determine whether the characteristics of individuals choosing specific alternatives differ across alternatives. This is made possible by the function called `apollo_choiceAnalysis`, which is called as follows:

```
apollo_choiceAnalysis(choiceAnalysis_settings,
                      apollo_control,
                      database)
```

where `choiceAnalysis_settings` has the following contents:

- `alternatives`: A named vector containing the names of the alternatives, as in e.g. an MNL model.
- `avail`: A list containing one element with availabilities per alternative, as in e.g. an MNL model, but where reference to database needs to be made given that we are operating outside `apollo_probabilities` (cf. Figure 9.1).
- `choiceVar`: A vector of length equal to the number of observations, containing the chosen alternative for each observation.

`explanators`: A dataframe containing a set of variables, one per column and one entry per choice observation, that are to be used to analyse the choices. This could include explantory variables describing the alternatives but is most useful for characteristics of the decision makers. In order to be able to define this object outside `apollo_probabilities`, reference to the database again needs to be made(cf. Figure 9.1).

`rows`: This is an optional argument which is missing by default. It allows the user to specify a vector called `rows` of the same length as the number of rows in the data. This vector needs to use logical statements to identify which rows in the data are to be used for the analysis of choices.

```
choiceAnalysis_settings <- list(
  alternatives = c(car=1, bus=2, air=3, rail=4),
  avail        = list(car=database$av_car, bus=database$av_bus, air=database$av_air, rail=
      ↪ database$av_rail),
  choiceVar    = database$choice,
  explanators  = database[,c("female","business","income")],
  rows         = database$income>30000
)

apollo_choiceAnalysis(choiceAnalysis_settings, apollo_control, database)
```

	Mean for female if chosen	Mean for female if not chosen	t-test for difference
car	0.4858	0.5401	1.29
bus	0.5309	0.4842	-0.79
air	0.4914	0.4826	-0.19
rail	0.4821	0.5012	0.48

Figure 9.1: Running `apollo_choiceAnalysis` (syntax and excerpt of output)

The function produces a *csv* file with one row per alternative, and three columns per variable included in `explanators`. In a given row, i.e. for a given alternative, these three columns contain the mean value for the given explanatory variable for those choices where the alternative is chosen, the mean value where it is not chosen (but available), and the test statistic for the two-sample t-test comparing the means in these two groups (where the null hypothesis states that the difference between the means is equal to 0, and the alternative hypothesis says that it is different from zero.). These value are also returned silently by the function, so they can be stored in a variable, by using e.g. `output=apollo_choiceAnalysis(choiceAnalysis_settings,apollo_control,database)`. An example application of this function is included in the MNL model estimated on the RP mode choice data (`Apollo_example_1.r`), where we show the results for the first explanator only.

9.2 Reading in a previously saved `model` object

As mentioned in Section 4.7, the call to `apollo_saveOutput` (with default settings) saves the `model` object in a *.rds* file. It is then possible to read this in as a model object using the function

apollo_loadModel which is called as:

```
oldModel = apollo_loadModel(modelName)
```

where modelName needs to be replaced by the name (as a string, i.e. with quotation marks) of previously run model (for which the output was saved in the current directory). The output from this function is a model object, which in this case is saved into oldModel. The benefit of this function is that it is then easy for a user to return to a previously estimated model, and compute additional output with the estimates from that model and having access to the full covariance matrix without needing to reestimate the model. An example is included in apollo_example_16.r.

9.3 Calculating model fit for given parameter values

Especially with complex models, it can be useful for testing purposes to calculate the log-likelihood of the model (and subcomponents) for given parameter values, before or after estimation. This is made possible by the function apollo_llCalc, which is called as follows:

```
apollo_llCalc(apollo_beta,
              apollo_probabilities,
              apollo_inputs)
```

where we illustrate this in Figure 9.2 for the case of the hybrid choice model (Apollo_example_24.r) from Section 7.3. It should be noted that when calling this function, the format of apollo_beta needs to be compatible with what is used inside apollo_probabilities. All parameters used inside the model need to be included, with either one value per parameter (classical estimation) or one value per parameter and per observation (Bayesian estimation).

```
> apollo_llCalc(apollo_beta, apollo_probabilities, apollo_inputs)
Updating inputs... Done.
Calculating LL of each model component...Done.
$model
[1] -19734.47

$indic_quality
[1] -1505.83

$indic_ingredients
[1] -1531.451

$indic_patent
[1] -1543.356

$indic_dominance
[1] -1465.061

$choice
[1] -13314.75
```

Figure 9.2: Running apollo_llCalc

Note that apollo_llCalc use the copy of the database stored within apollo_inputs, and not the version of database that may be stored in the user environment (i.e. the Global Environment). If the user wishes to calculate the log-likelihood for a modified database (for example, to

study how the log-likelihood changes in a new scenario) we recommend doing it in three stages. First, modify `database` in whatever way is needed. Then update `apollo_inputs` by running `apollo_inputs = apollo_validateInputs()`, and finally calling `apollo_llCalc`.

9.4 Likelihood ratio tests against other models

A core step in many model fitting exercises is the comparison of models of different levels of complexity. When comparing two models where one model is a more general version of a base model, i.e. the *base* model is nested within the *general* model, a likelihood-ratio test can be used to compare the two models (cf. Train, 2009, Section 3.8.2.).

This test is implemented in the function `apollo_lrTest`, which is called as follows:

```
apollo_lrTest(model1,
              model2)
```

```
> apollo_lrTest("Apollo_example_3", "Apollo_example_5")

                      LL par
Apollo_example_4 -4774.55  18
Apollo_example_5 -4770.48  19
Difference           4.07   1

Likelihood ratio test-value:    8.14
Degrees of freedom:             1
Likelihood ratio test p-value:  0.00433

> apollo_lrTest("Apollo_example_3", model)

                      LL par
Apollo_example_4 -4774.55  18
Apollo_example_5 -4770.48  19
Difference           4.07   1

Likelihood ratio test-value:    8.14
Degrees of freedom:             1
Likelihood ratio test p-value:  0.00433
```

Figure 9.3: Running `apollo_lrTest`

The function `apollo_lrTest` is flexible in the input it receives for `model1` and `model2`. It can either receive the names of models (i.e. as a character string) for which the output has previously been saved to disk, or the name of model objects stored in memory, i.e. as produced by `apollo_estimate`. A mix of the two is also permissible, i.e. one previously stored model output along with one kept in memory. The `apollo_lrTest` function will determine whether the models have been estimated on the same data (in terms of observations and alternatives), and whether one model uses fewer parameters and has a lower fit. The function is not able to determine whether one model specification is in fact nested within the other, i.e. the user needs to decide whether a likelihood ratio test is appropriate. The order in which the two models are given is not relevant, as `apollo_lrTest` will reorder them in the appropriate way. In Figure 9.3, we illustrate the function by comparing the MNL model from Section 4.5.2, i.e. `Apollo_example_3`, and the NL model from Section 5.1.1, i.e. `Apollo_example_5`. We show both the version where the outputs for both models have been saved earlier[1] as well as the version where the more general

[1]Note that the output needs to have been saved in the same directory.

model has just been run and still exists in the R environment. This function is not suitable when for example comparing a joint model with two separate models (e.g. RP-SP vs separate RP and SP models) and the user in that case needs to calculate the LR test manually, which is of course trivial.

9.5 Ben-Akiva & Swait test

A well known example of a non-nested test for model comparison is the Ben-Akiva & Swait test (Ben-Akiva and Swait, 1986), which is based on adjusted ρ^2. Specifically, we have that

$$P(\bar{\rho}_1^2 - \bar{\rho}_2^2 \geq z) \leq \Phi\left[-\left(2zLL_0 + df_1 - df_2\right)^{\frac{1}{2}}\right], \tag{9.1}$$

where z is the observed difference in adjusted ρ^2, Φ is the cumulative standard normal distribution, and df_1 and df_2 are the number of parameters for models 1 and 2. The two models need to both be discrete choice, and estimated on the same data.

This test is implemented in *Apollo* using the function `apollo_basTest`, which is called as:

```
apollo_basTest(model1,
               model2)
```

The function `apollo_baseTest` is flexible in the input it receives for `model1` and `model2`. It can either receive the names of models (i.e. as a character string) for which the output has previously been saved to disk, or the name of model objects stored in memory, i.e. as produced by `apollo_estimate`. A mix of the two is also permissible, i.e. one previously stored model output along with one kept in memory. The `apollo_basTest` function will determine whether the models have been estimated on the same data, and whether a ρ^2 measure is available for both models. The order in which the two models are given is not relevant, as `apollo_basTest` will reorder them in the appropriate way. It then calculates the p-value for the Ben-Akiva & Swait test.

In Figure 9.4, we illustrate the function by comparing the three-level NL model from Section 5.1.1, i.e. `Apollo_example_5`, and the CNL model from Section 5.1.2, i.e. `Apollo_example_6`. We show both the version where the outputs for both models have been saved earlier[2] as well as the version where the more general model has just been run and still exists in the R environment.

9.6 Model predictions

A core capability of *Apollo* is that it covers model application (i.e. prediction) in addition to estimation. This is implemented in the function `apollo_prediction`. The function is called as

[2]Note that the output needs to have been saved in the same directory.

```
> apollo_basTest("Apollo_example_5","Apollo_example_6")


                        LL0       LL par adj.rho2
Apollo_example_5 -8196.02 -4770.48  19   0.4156
Apollo_example_6 -8196.02 -4742.25  20   0.4190
Difference           0.00    28.23   1   0.0034

p-value for Ben-Akiva & Swait test: 2.496e-14

> apollo_basTest("Apollo_example_5",model)

                        LL0       LL par adj.rho2
Apollo_example_5 -8196.02 -4770.48  19   0.4156
Apollo_example_6 -8196.02 -4742.25  20   0.4190
Difference           0.00    28.23   1   0.0034

p-value for Ben-Akiva & Swait test: 2.496e-14
```

Figure 9.4: Running `apollo_basTest`

follows:

```
forecast = apollo_prediction(model,
                             apollo_probabilities,
                             apollo_inputs,
                             prediction_settings)
```

The majority of these arguments have been discussed already. The only additional new argument is `prediction_settings`, which is an optional list that can contain two entries:

- `modelComponent`: name of the model component for which predictions are requested. This argument is required for models with multiple components, and needs to be the name (string) of one of the elements in the list `P` used inside `apollo_probabilities`.
- `nRep`: number of Monte Carlo iterations used for models that require simulation for prediction (e.g. MDCEV) (default is 100).
- `runs`: number of runs of the prediction algorithm to use with random draws from the set of parameters (default=1)

The application of this function to the `Apollo_example_3.r` mode choice model is illustrated in Figure 9.5, which also shows how to look at changes in choices following a change in an explanatory variable, as well as how elasticities can be calculated. We first run a prediction using the model on the original data, in addition to 30 runs of the prediction on sets of random draws from the vector of parameter estimates, where these are drawn using the full covariance matrix. This allows us to get an indication of the level of uncertainty in the predictions. This process shows the aggregated predicted demand (summing probabilities across observations) at the model estimates, and also a 95% confidence interval for it, based on the additional repetitions. When not including the setting for `runs`, the use of `apollo_prediction` produces a list of matrices with one element per model component (for multi-component models), or a single matrix (for single-component models). The matrix or matrices returned have one row per observation, and with the following sets of columns:

- ID of the individual
- Index of observations for the individual
- Set of J columns with predicted probabilities, one per alternative
- Probability for the chosen alternative

The output of `apollo_prediction` is slightly different for MDCEV models, including the following columns.

- ID of the individual
- Index of observations for the individual
- Set of J columns with predicted continuous consumption, one per alternative
- Set of J columns with the s.d. of the predicted continuous consumption, one per alternative
- Set of J columns with predicted probability of (discrete) consumption, one per alternative
- Set of J columns with the s.d. of the predicted probability of (discrete) consumption, one per alternative
- Set of J columns with predicted expenditure, one per alternative
- Set of J columns with the s.d. of the predicted expenditure, one per alternative

When including a value for `runs` that is larger than 1, `apollo_prediction` returns a list. The first element in the list will be the matrix of prediction at the parameter estimates, using the format above. This element is called `at_estimates`. The second element in the list, called `draws`, is a 3-dimensional array, with dimensionality number of observations x 6J x `runs`. Each slide on the first two dimensions has the same columns described above, except for the ID of the individual and the index of the observation (i.e. first two columns in the list above are missing). Each of these predictions come from different draws from the sets of parameter estimates.

Note that `apollo_prediction` uses the copy of the `database` stored within `apollo_inputs`, and not the version of `database` that may be stored in the user environment (i.e. the Global Environment). If the user wishes to calculate predictions for a modified `database` (for example, to study how predictions change in a new scenario) we recommend doing it in three stages. First, modify `database` in whatever way is needed. Then update `apollo_inputs` by running `apollo_inputs = apollo_validateInputs()`, and finally call `apollo_prediction`.

We next run a prediction (without separate runs for confidence interval computation) for a scenario where the cost of rail is increased by 1% (after which we reverse that change in the data). This leads to a drop in the demand for rail, and an increase in the demand for other modes.

We next compare the before and after probabilities at the level of individual observations. For the base predictions, we first need to extract the first element of the list, i.e. the predictions at the actual model estimates. We then look at changes for individual people in the data, summaries of changes, including for subsets of the data, before computing elasticities.

The output of `apollo_prediction` will depend on the underlying model component, but will always include the ID and choice situation index for each row. In particular:

- For `MNL`, `NL`, `CNL`, `DFT`, `OL` and `OP` models, `apollo_prediction` will return the probability of the chosen alternative, as well as the probability of each alternative at the observation level (rather than person level). In particular, these models return a list containing one vector

```
> predictions_base = apollo_prediction(model, apollo_probabilities, apollo_inputs, prediction_settings=list(runs=30))
Updating inputs...Done.

Running predictions from model using parameter estimates...
Running predictions across draws from the asymptotic distribution for maximum likelihood estimates.
Predicting for set of draws 1/30...
...
Predicting for set of draws 30/30...

Predicted aggregated demand
     At estimates Std.dev. Quantile 0.025 Quantile 0.975
car        1946.01     49.03         1871.29         2038.45
bus         358.00     17.67          323.56          384.08
air        1522.00     31.38         1476.61         1593.72
rail       3173.98     34.11         3114.28         3220.78

The output from apollo_prediction is a list with two elements: a matrix containing the predictions at
    ↪ the
  estimated values, and an array with predictions for different values of the parameters drawn from
      ↪ their
  asymptotic distribution.
Several observations per individual detected based on the value of ID. Setting panelData in
    ↪ apollo_control set
  to TRUE.
All checks on apollo_control completed.
All checks on database completed.
Running predictions from model using parameter estimates...
Predicted aggregated demand at model estimates
          car     bus     air     rail
Demand 1967.4  362.98  1535.1  3134.53

> ### Now imagine the cost for rail increases by 1%
> database$cost_rail = 1.01*database$cost_rail
> ### Rerun predictions with the new data
> apollo_inputs = apollo_validateInputs()
> predictions_new = apollo_prediction(model, apollo_probabilities, apollo_inputs)
Updating inputs...Done.

Running predictions from model using parameter estimates Done.

Predicted aggregated demand at model estimates
          car     bus     air     rail
Demand 1967.4  362.98  1535.1  3134.53
> ### Return to original data
> database$cost_rail = 1/1.01*database$cost_rail

> ### work with predictions at estimates
> predictions_base=predictions_base[["at_estimates"]]
> ### Compute change in probabilities
> change=(predictions_new-predictions_base)/predictions_base

### Not interested in chosen alternative now, so drop last column
> change=change[,-ncol(change)]
### First two columns (change in ID and task) also not needed
> change=change[,-c(1,2)]
>### Look at person 9, who has all 4 modes available
> change[database$ID==9,]
              car          bus          air          rail
 [1,] 0.010361469 0.010361469 0.010361469 -0.018685663
 [2,] 0.012488949 0.012488949 0.012488949 -0.005077821
...
[14,] 0.015476546 0.015476546 0.015476546 -0.006015603
>### Look at mean changes for subsets of the data, ignoring NAs
> colMeans(change,na.rm=TRUE)
         car          bus          air          rail
 0.01383311   0.01521340   0.01439624 -0.02132516
> colMeans(subset(change,database$business==1),na.rm=TRUE)
         car          bus          air          rail
 0.01246518   0.01281052   0.01070145 -0.01099004
> colMeans(subset(change,database$business==0),na.rm=TRUE)
         car          bus          air          rail
 0.01451181   0.01643493   0.01604469 -0.02640454
> colMeans(subset(change,(database$income<quantile(database$income,0.25))),na.rm=TRUE)
         car          bus          air          rail
 0.01748663   0.01906556   0.01873898 -0.03358834
> colMeans(subset(change,(database$income>=quantile(database$income,0.25))|(database$income<=quantile(database$income,0.75))), na.rm=TRUE)
         car          bus          air          rail
 0.01383311   0.01521340   0.01439624 -0.02132516
> colMeans(subset(change,(database$income>quantile(database$income,0.75))),na.rm=TRUE)
         car          bus          air          rail
 0.01080122   0.01171832   0.01068467 -0.01432130

>### Own elasticity for rail:
> log(sum(predictions_new[,4])/sum(predictions_base[,4]))/log(1.1)
[1] -1.257109

>### Cross-elasticities for other modes
> log(sum(predictions_new[,1])/sum(predictions_base[,1]))/log(1.1)
[1] 1.098718
> log(sum(predictions_new[,2])/sum(predictions_base[,2]))/log(1.1)
[1] 1.386133
> log(sum(predictions_new[,3])/sum(predictions_base[,3]))/log(1.1)
[1] 0.8607355
```

Figure 9.5: Running `apollo_prediction`

per alternative plus an additional vector for the chosen alternative, where each vector is as long as the number of observations in the `database`, contain the probability of that alternative. In the presence of continuous random heterogeneity, the draws are averaged out before presenting the results.

- The discrete continuous models `MDCEV` and `MDCNEV` do not return probabilities, but instead expected values of consumption for each alternative at the observation level. In particular, they return a matrix detailing the expected (continuous) consumption for each alternative, and a proxy for the probability of consuming each alternative (discrete choice), as well as the standard deviations for both of these measurements. These outputs are calculated using the efficient forecasting method proposed by Pinjari and Bhat 2010b, and its modification for the **MDCNEV** model by Calastri et al. 2017. These methods are based on simulation (200 repetitions are used), and can therefore be computationally demanding. The probability of consuming each alternative is calculated as the percentage of simulation repetitions in which the alternative is consumed, and is not calculated using an analytical formula. Again, in the presence of continuous random coefficients, the results are averaged across draws. If using the optional argument `runs` in `prediction_settings`, the output will present standard deviations across runs for both the mean predictions and the standard deviations due to the use of draws inside the Pinjari and Bhat (2010b) algorithm.
- The **EL** (Exploded Logit) and **Normal Density** models do not return any predictions, as it is not evident what precise outcome would be the most useful for the biggest share of users.

9.7 Market share recovery for subgroups of data

With labelled choice data (or even unlabelled data where there may be strong left-right effects), it can be useful to test after model estimation how well the choice shares in the data are recovered by the model. With a full set of ASCs, a linear in attributes MNL model will perfectly recover market shares at the sample level (see e.g. Train, 2009, Section 2.6.1.). This is however likely not the case in subsets of the data, or indeed for more complex models, and this test can thus be a useful input for model refinements. The function `apollo_sharesTest` is based on the "apply" tables approach in ALogit (ALogit, 2016). It can be used for any of the discrete choice models implemented in *Apollo*, and is called as follows:

```
apollo_sharesTest(model,
                  apollo_probabilities,
                  apollo_inputs,
                  sharesTest_settings)
```

The list `sharesTest_settings` has the following components:

`alternatives:` A named vector containing the names of the alternatives as defined by the user, and for each alternative, giving the value used in the dependent variable in the data.

choiceVar: A variable indicating the column in the database which identifies the alternative chosen in a given choice situation. This is not a character variable (i.e. text) but the name use to identify a column in the database. As we are now operating outside `apollo_probabilities`, we need to use `database$choice` for example.

subsamples: The list `subsamples` is an optional input which contains one column for each subset of the data to be used in the test, where it is possible for a given row to be included in multiple subsets, i.e. the sum of the values across column vectors in `subsamples` may exceed 1.

modelComponent: The name of the model component for which predictions are requested. This argument is required for models with multiple components, and needs to be the name (string) of one of the elements in the list `P` used inside `apollo_probabilities`.

newAlts: Optional list describing the new alternatives to be used by `apollo_sharesTest`. This should have as many elements as new alternatives, with each entry being a matrix of 0-1 entries, with one row per observation, and one column per alternative used in the model.

newAltsOnly: An optional boolean variable, which, if set to TRUE, means that results will only be printed for the 'new' alternatives defined in `newAlts`, not the original alternatives used in the model. Set to FALSE by default.

```
> sharesTest_settings = list()
> sharesTest_settings=list()
> sharesTest_settings[["alternatives"]] = c(car=1, bus=2, air=3, rail=4)
> sharesTest_settings[["choiceVar"]] = database$choice
> sharesTest_settings[["subsamples"]] = list(business=(database$business==1),+ leisure=(database$business==0))

> apollo_sharesTest(model,apollo_probabilities,apollo_inputs,sharesTest_settings)
Updating inputs... Done.
Running predictions from model... Done.

Running share prediction tests

Prediction test for group: business (2310 observations)

                              car     bus      air      rail   All
Times chosen (data)       366.000   8.000 771.000 1165.000 2310
Times chosen (prediction) 350.443  24.348 739.725 1195.484 2310
Diff (prediction-data)    -15.557  16.348 -31.275   30.484    0
t-ratio                    -1.093   3.463  -1.846    1.612   NA
p-val                       0.275   0.001   0.065    0.107   NA

Prediction test for group: leisure (4690 observations)

                              car     bus      air      rail   All
Times chosen (data)       1580.000 350.000 751.000 2009.000 4690
Times chosen (prediction) 1595.563 333.656 782.280 1978.501 4690
Diff (prediction-data)      15.563 -16.344  31.280  -30.499    0
t-ratio                      0.606  -1.075   1.601   -1.160   NA
p-val                        0.544   0.282   0.109    0.246   NA

Prediction test for group: All data (7000 observations)

                              car     bus      air      rail   All
Times chosen (data)       1946.000 358.000 1522.000 3174.000 7000
Times chosen (prediction) 1946.006 358.004 1522.005 3173.984 7000
Diff (prediction-data)       0.006   0.004    0.005   -0.016    0
t-ratio                      0.000   0.000    0.000    0.000   NA
p-val                        1.000   1.000    1.000    1.000   NA
```

Figure 9.6: Running `apollo_sharesTest`

The function produces one table per column in `subsamples`, along with an overall table for the entire sample. In each table, the code reports the number of times an alternative is chosen in the

data, the number of times the model predicts it to be chosen, the difference between prediction and data, and a t-ratio and p-value for this difference. An example application of this function to the SP mode choice data is included in `Apollo_example_3.r`. As we can see from Figure 9.6, in our example, the model significantly overpredicts the rate at which business travellers choose bus and underpredicts the rate at which they choose air. A revised model specification may thus incorporate shifts in these ASCs for business travellers. In addition to screen output, `apollo_sharesTest` also invisibly returns the output so it can be saved into a data.frame, using e.g. `sharesTest_output=apollo_sharesTest(model,apollo_probabilities,apollo_inputs,sharesTest_set`

9.8 Comparison of model fit across subgroups of data

An additional function is implemented to compare the performance of the estimated model for different subsets of the data. The function `apollo_fitsTest` can be used for any models estimated in *Apollo*, and is called as follows:

```
apollo_fitsTest(model,
                apollo_probabilities,
                apollo_inputs,
                fitsTest_settings)
```

The list `fitsTest_settings` has the following component:

`subsamples`: The list `subsamples` is an optional input which contains one column for each subset of the data to be used in the test, where it is possible for a given row to be included in multiple subsets, i.e. the sum of the values across column vectors in `subsamples` may exceed 1.

The function calculates various statistics for the log-likelihood, as illustrated in Figure 9.7 for the mode choice MNL model `Apollo_example_3.r`, where the last row in the output compares the mean log-likelihood in the specific subsample to the mean in all other subsamples. Users need to exercise caution when using this function in the case where the choice set size varies across individuals in a manner that is correlated with the subgroups as the prediction performance for individuals with smaller choice sets will be likely to be larger, all else being equal.

9.9 Functions of model parameters and associated standard errors

A key use of estimates from choice models is the calculation of functions of these estimates, for example in the form of ratios of coefficients, leading to marginal rates of substitution, and in the case of a cost coefficient being used as the denominator, willingness-to-pay (WTP) measures. It is then important to be able to calculate standard errors for these derived measures, where this can be done straightforwardly and accurately with the Delta method, as discussed by Daly et al.

```
> fitsTest_settings = list()

> fitsTest_settings[["subsamples"]] = list()
> fitsTest_settings$subsamples[["business"]] = database$business==1
> fitsTest_settings$subsamples[["leisure"]] = database$business==0
> apollo_fitsTest(model,apollo_probabilities,apollo_inputs,fitsTest_settings)

                              All data business leisure
Min LL per obs                   -1.45    -1.29   -1.45
Mean LL per obs                  -0.69    -0.59   -0.74
Median LL per obs                -0.70    -0.61   -0.74
Max LL per obs                   -0.01    -0.01   -0.06
SD LL per obs                     0.28     0.28    0.27
mean vs mean of all others          NA     0.14   -0.14
```

Figure 9.7: Running `apollo_fitsTest`

(2012a). The function `apollo_deltaMethod` is implemented for this purpose for a limited number of operations, and is called as follows:

```
apollo_deltaMethod(model,
                   deltaMethod_settings)
```

The list `deltaMethod_settings` has the following components:

`operation`: A character object `operation`, which determines which function is to be applied to the parameters. Possible values are:

- `sum`: two-parameter function, with $f(\beta_1, \beta_2) = \beta_1 + \beta_2$
- `diff`: two-parameter function, with $f(\beta_1, \beta_2) = \beta_1 - \beta_2$
- `ratio`: two-parameter function, with $f(\beta_1, \beta_2) = \frac{\beta_1}{\beta_2}$
- `exp`: one-parameter function, with $f(\beta_1) = e^{\beta_1}$
- `logistic`: either one-parameter function, with $f_1(\beta_1) = \frac{e^{\beta_1}}{e^{\beta_1}+1}$ and $f_2(\beta_1) = \frac{1}{e^{\beta_1}+1}$, or two-parameter function, with $f_1(\beta_1, \beta_2) = \frac{e^{\beta_1}}{e^{\beta_1}+e^{\beta_2}+1}$, $f_2(\beta_1, \beta_2) = \frac{e^{\beta_2}}{e^{\beta_1}+e^{\beta_2}+1}$, and $f_3(\beta_1, \beta_2) = \frac{1}{e^{\beta_1}+e^{\beta_2}+1}$.
- `lognormal`: two-parameter function giving the mean and standard deviation for a Lognormal distribution on the basis of the mean and standard deviation for the logarithm of the coefficient, i.e. with $\beta = e^{N(\beta_1,\beta_2)}$, we have $f_1(\beta_1, \beta_2) = \mu_\beta = e^{\beta_1 + \frac{\beta_2^2}{2}}$ and $f_2(\beta_1, \beta_2) = \sigma_\beta = \mu_\beta\sqrt{e^{\beta_2^2} - 1}$

`parName1`: A character object giving the name of the first parameter.

`parName2`: A character object giving the name of the second parameter, optional if `operation=logistic`.

`multPar1`: An optional numerical value used to multiply the first parameter, set to 1 if omitted.

`multPar2`: An optional numerical value used to multiply the second parameter, set to 1 if omitted.

An example application of this function is included in `Apollo_example_3.r`, and is illustrated in Figure 9.8 for the car value of travel time, i.e. the ratio between the car travel time and cost

coefficients (in both minutes and hours) as well as for the difference between the car and rail travel time coefficients. The values here are all calculated for an individual in the base socio-demographic group.

```
> deltaMethod_settings=list(operation="ratio",parName1="b_tt_car",parName2="b_cost")
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                              Value Robust s.e. Rob t-ratio (0)
Ratio of b_tt_car and b_cost:  0.172     0.009719          17.7
> deltaMethod_settings=list(operation="ratio",parName1="b_tt_car",parName2="b_cost",multPar1 = 60)
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                                                 Value Robust s.e. Rob t-ratio (0)
Ratio of b_tt_car (multiplied by 60) and b_cost:  10.32       0.5832          17.7

> deltaMethod_settings=list(operation="diff",parName1="b_tt_car",parName2="b_tt_rail")
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                                                Value Robust s.e. Rob t-ratio (0)
Difference between b_tt_car and b_tt_rail:  -0.006056     0.001906         -3.178
```

Figure 9.8: Running `apollo_deltaMethod`

Only a limited number of functions of parameters are covered by `apollo_deltaMethod`. Rather than relying on sampling based approaches such as the Krinsky & Robb method (Krinsky and Robb, 1986) for calculating the standard errors for more complex functions, users who wish to compute standard errors of other functions can for example use the `R` function `deltamethod` from the `alr3` package (Weisberg, 2005). This uses symbolic differentiation of the user provided function.

9.10 Unconditionals for random parameters

After model estimation, it may be useful to an analyst to have at their disposal the actual values used for random coefficients, especially if these included interactions with socio-demographics or (non-linear) transforms that may lead to a requirement for simulation to calculate moments (as in the semi-non-parametric approach of Fosgerau and Mabit 2013 used in Section 6.1.2). We look separately at continuous random parameters and Latent Class.

9.10.1 Continuous random heterogeneity

For continuous random coefficients, the function `apollo_unconditionals` is called as follows:

```
unconditionals = apollo_unconditionals(model,
                                       apollo_probabilities,
                                       apollo_inputs)
```

The function produces a list as output, with one element per random coefficient, where this is a matrix for inter-individual draws, and a cube with inter and intra-individual draws. Each time, there is one row per individual, rather than one row per observation. The outputs from this function can then readily be used for summary statistics or to produce plots. An example of

this is included in `apollo_example_16.r`, and also illustrated in the discussion of conditionals in Section 9.11.1.

Note that `apollo_unconditionals` uses the copy of the `database` stored within `apollo_inputs`, and not the version of `database` that may be stored in the user environment (i.e. the Global Environment). If the user wishes to calculate the unconditionals for a modified `database` (for example, to study how unconditionals change in a new scenario) we recommend doing it in three stages. First, modify `database` in whatever way is needed. Then update `apollo_inputs` by running `apollo_inputs = apollo_validateInputs()`, and finally call `apollo_unconditionals`.

9.10.2 Latent class

For Latent Class models, the function `apollo_lcUnconditionals` is called as follows:

```
unconditionals = apollo_lcUnconditionals(model,
                                         apollo_probabilities,
                                         apollo_inputs)
```

The `apollo_lcUnconditionals` produces a list which has one element for each model parameter that varies across classes, where these are given by lists, with one element per class. The entry for each class could be either a scalar (if fixed coefficients are used inside the classes), a vector (if interactions with socio-demographics are used), or a matrix or cube if continuous heterogeneity is also incorporated. For the latter two cases, there is one row per individual, rather than one row per observation. The final component in the list produced by `apollo_lcUnconditionals` is a list containing the class allocation probabilities, with one element per class, where these could again be scalars, vectors, matrices or cubes, depending on the extent of heterogeneity allowed for by the user. An example of this is included in `apollo_example_20.r`, and also illustrated in the discussion of conditionals in Section 9.11.2.

Note that `apollo_lcUnconditionals` uses the copy of the `database` stored within `apollo_inputs`, and not the version of `database` that may be stored in the user environment (i.e. the Global Environment). If the user wishes to calculate the unconditionals for a modified `database` (for example, to study how unconditionals change in a new scenario) we recommend doing it in three stages. First, modify `database` in whatever way is needed. Then update `apollo_inputs` by running `apollo_inputs = apollo_validateInputs()`, and finally call `apollo_lcUnconditionals`.

9.11 Conditionals for random coefficients

There is extensive interest by choice modellers in posterior model parameter distributions, as discussed in Train (2009, chapter 11) for continuous mixture models and Hess (2014) for Latent Class. We implement functions for this for both continuous Mixed Logit and Latent Class models.

9.11.1 Continuous random coefficients

Let β give a vector of taste coefficients that are jointly distributed according to $f(\beta \mid \Omega)$, where Ω is a vector of distributional parameters that is to be estimated from the data. Let Y_n give the sequence of observed choices for respondent n (which could be a single choice), and let $L(Y_n \mid \beta)$ give the probability of observing this sequence of choices with a specific value for the vector β. Then it can be seen that the probability of observing the specific value of β given the choices of respondent n is equal to:

$$L(\beta \mid Y_n) = \frac{L(Y_n \mid \beta) f(\beta \mid \Omega)}{\int_\beta L(Y_n \mid \beta) f(\beta \mid \Omega) \, \mathrm{d}\beta} \tag{9.2}$$

The integral in the denominator of Equation 9.2 does not have a closed form solution, such that its value needs to be approximated by simulation. This is a simple (albeit numerically expensive) process, with as an example the mean for the conditional distribution for respondent n being given by:

$$\widehat{\beta_n} = \frac{\sum_{r=1}^{R} [L(Y_n \mid \beta_r) \beta_r]}{\sum_{r=1}^{R} L(Y_n \mid \beta_r)}, \tag{9.3}$$

where β_r with $r = 1, \ldots, R$ are independent multi-dimensional draws[3] with equal weight from $f(\beta \mid \Omega)$ at the estimated values for Ω. Here, $\widehat{\beta_n}$ gives the most likely value for the various marginal utility coefficients, conditional on the choices observed for respondent n.

It is important to stress that the conditional estimates for each respondent themselves follow a random distribution, and that the output from Equation 9.3 simply gives the expected value of this distribution. As such, a distribution of the output from Equation 9.3 across respondents should not be seen as a conditional distribution of a taste coefficient across respondents, but rather a distribution of the means of the conditional distributions (or conditional means) across respondents. Here, it is similarly possible to produce a measure of the conditional standard deviation, given by:

$$\widetilde{\beta_n} = \sqrt{\frac{\sum_{r=1}^{R} \left[L(Y_n \mid \beta_r) \left(\beta_r - \widehat{\beta_n} \right)^2 \right]}{\sum_{r=1}^{R} L(Y_n \mid \beta_r)}}, \tag{9.4}$$

with $\widehat{\beta_n}$ taken from Equation 9.3.

The calculation of posteriors for models with continuous random heterogeneity is implemented in the function `apollo_conditionals`, which is called as follows:

```
conditionals = apollo_conditionals(model,
                                   apollo_probabilities,
                                   apollo_inputs)
```

[3] The term *independent* relates to independence across different multivariate draws, where the individual multivariate draws allow for correlation between univariate draws.

The function produces a list object with one component per continuous random coefficient (element defined in `apollo_randCoeff`). Each of these components is a matrix with one row per individual, containing the ID for that individual, the mean of the posterior distribution for that individual for the coefficient in question, and the standard deviation. As `apollo_conditionals` uses the contents of `apollo_randCoeff`, any socio-demographic interactions included in `apollo_randCoeff` will also be included in the calculation for the conditionals, where, if these vary across observations for the same individual, they will be averaged across observations. Similarly, any intra-individual random heterogeneity will also be averaged out. This function is only applicable for models using classical estimation, and also only models where `apollo_control$workInLogs==FALSE`.

Note that `apollo_conditionals` uses the copy of the `database` stored within `apollo_inputs`, and not the version of `database` that may be stored in the user environment (i.e. the Global Environment). If the user wishes to calculate the conditionals for a modified `database` (for example, to study how conditionals change in a new scenario) we recommend doing it in three stages. First, modify `database` in whatever way is needed. Then update `apollo_inputs` by running `apollo_inputs = apollo_validateInputs()`, and finally call `apollo_conditionals`.

```
> unconditionals = apollo_unconditionals(model, apollo_probabilities, apollo_inputs)
Unconditional distributions computed

> conditionals <- apollo_conditionals(model, apollo_probabilities, apollo_inputs)
Your model contains intra-individual draws which will be averaged over for conditionals

> mean(unconditionals[["v_tt"]])
[1] 0.418239
> sd(unconditionals[["v_tt"]])
[1] 0.4741362

> summary(conditionals[["v_tt"]])
       ID          post. mean        post. sd
 Min.   : 2439   Min.   :0.1167   Min.   :0.04665
 1st Qu.:15308   1st Qu.:0.2695   1st Qu.:0.13136
 Median :18533   Median :0.3361   Median :0.16234
 Mean   :22181   Mean   :0.4162   Mean   :0.19752
 3rd Qu.:21948   3rd Qu.:0.4663   3rd Qu.:0.22126
 Max.   :84525   Max.   :2.1597   Max.   :1.31119

> income_n=apollo_firstRow(database$hh_inc_abs, apollo_inputs)


Call:
lm(formula = conditionals[["v_tt"]][, 2] ~ income_n)

Residuals:
     Min       1Q   Median       3Q      Max
-0.31753 -0.14661 -0.07131  0.05278  1.71282

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.510e-01  2.623e-02  13.380  < 2e-16 ***
income_n    8.527e-07  2.966e-07   2.875  0.00427 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2592 on 386 degrees of freedom
Multiple R-squared:  0.02096,    Adjusted R-squared:  0.01842
F-statistic: 8.263 on 1 and 386 DF,  p-value: 0.00427
```

Figure 9.9: Running `apollo_unconditionals` and `apollo_conditionals`

Figure 9.9 illustrates the use of this function for the value of travel time coefficient in the Swiss route choice MMNL (`Apollo_example_16.r`) example, where we show how the conditional means can then for example also be used in regression analysis against characteristics of the individual,

as discussed by Train (2009, chapter 11), in our case showing a significant impact of income on the conditionals for the VTT. Note that as `apollo_conditionals` produces one value per individual, we also need to reduce the dimensionality of the income variable to one per individual, using `apollo_firstRow`. We also include a comparison with the unconditionals.

9.11.2 Latent class

It is similarly possible to calculate a number of posterior measures from Latent Class models. A key example comes in the form of posterior class allocation probabilities, where the posterior probability of individual n for class s is given by:

$$\widehat{\pi_{ns}} = \frac{\pi_{ns} L_n(\beta_s)}{L_n(\beta, \pi_n)}, \tag{9.5}$$

where $L_n(\beta_s)$ gives the likelihood of the observed choices for individual n, conditional on class s.

To explain the benefit of these posterior class allocation probabilities, let us assume that we have calculated for each class in the model a given measure $w_s = \frac{\beta_{s1}}{\beta_{s2}}$, i.e. the ratio between the first two coefficients. Using $\overline{w_n} = \sum_{s=1}^{S} \pi_{ns} w_s$ simply gives us a sample level mean for the measure w for an individual with the specific observed characteristics of person n. These characteristics (in terms of socio-demographics used in the class allocation probabilities) will however be common to a number of individuals who still make different choices, and the most likely value for w for individual n, conditional on his/her observed choices, can now be calculated as $\widehat{w_n} = \sum_{s=1}^{S} \widehat{\pi_{ns}} w_s$.

Finally, it might also be useful to produce a profile of the membership in each class. From the parameters in the class allocation probabilities, we know which class is more or less likely to capture individuals who posses a specific characteristic, but this is not taking into account the multivariate nature of these characteristics. Let us for example assume that a given socio-demographic characteristic z_c is used in the class allocation probabilities, with associated parameter γ_c, and using a linear parameterisation in Equation 6.18. We can then calculate the likely value for z_c for an individual in class s as:

$$\widehat{z_{cs}} = \frac{\sum_{n=1}^{N} \widehat{\pi_{ns}} z_{cn}}{\sum_{n=1}^{N} \widehat{\pi_{ns}}}, \tag{9.6}$$

where we again use the posterior probabilities to take into account the observed choices. Alternatively, we can also calculate the probability of an individual in class s having a given value κ for z_c by using:

$$P(\widehat{z_{cs} = \kappa}) = \frac{\sum_{n=1}^{N} \widehat{\pi_{ns}} (z_{cn} = \kappa)}{\sum_{n=1}^{N} \widehat{\pi_{ns}}}. \tag{9.7}$$

The calculation of posteriors for Latent Class models is implemented in the function `apollo_lcConditionals`, which is called as follows:

```
conditionals = apollo_lcCconditionals(model,
                                      apollo_probabilities,
                                      apollo_inputs)
```

This function is only applicable for Latent Class models that do not incorporate additional continuous random heterogeneity. The function produces a matrix, with one row per individual, one column with the individual id, and one column per class, containing the individual-specific posterior class allocation probabilities.

Note that `apollo_lcConditionals` uses the copy of the `database` stored within `apollo_inputs`, and not the version of `database` that may be stored in the user environment (i.e. the Global Environment). If the user wishes to calculate the conditionals for a modified `database` (for example, to study how conditionals change in a new scenario) we recommend doing it in three stages. First, modify `database` in whatever way is needed. Then update `apollo_inputs` by running `apollo_inputs = apollo_validateInputs()`, and finally call `apollo_lcConditionals`.

```
apollo_inputs = apollo_validateInputs()
```

```
> unconditionals=apollo_lcUnconditionals(model, apollo_probabilities, apollo_inputs)

> vtt_class_a=unconditionals[["beta_tt"]][[1]]/unconditionals[["beta_tc"]][[1]]
> vtt_class_b=unconditionals[["beta_tt"]][[2]]/unconditionals[["beta_tc"]][[2]]
> vtt_unconditional=unconditionals[["pi_values"]][[1]]*vtt_class_a+unconditionals[["pi_values"]][[2]]*vtt_class_b

> conditionals=apollo_lcConditionals(model, apollo_probabilities, apollo_inputs)
> summary(conditionals[,2:3])
     Class 1           Class 2
 Min.   :0.000003   Min.   :0.0000
 1st Qu.:0.151559   1st Qu.:0.1209
 Median :0.381015   Median :0.6190
 Mean   :0.483881   Mean   :0.5161
 3rd Qu.:0.879099   3rd Qu.:0.8484
 Max.   :1.000000   Max.   :1.0000
> summary(as.data.frame(unconditionals[["pi_values"]]))
     class_a           class_b
 Min.   :0.3924   Min.   :0.4140
 1st Qu.:0.4467   1st Qu.:0.4140
 Median :0.4467   Median :0.5533
 Mean   :0.4839   Mean   :0.5161
 3rd Qu.:0.5860   3rd Qu.:0.5533
 Max.   :0.5860   Max.   :0.6076
> vtt_conditional=conditionals[,2]*vtt_class_a+conditionals[,3]*vtt_class_b

> summary(vtt_unconditional)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.4221  0.4544  0.4544  0.4766  0.5374  0.5374
> summary(vtt_conditional)
    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.1885  0.2787  0.4153  0.4766  0.7118  0.7838

> commute_n = apollo_firstRow(database$commute,apollo_inputs)
> car_availability_n = apollo_firstRow(database$car_availability,apollo_inputs)

> post_commute=colSums(commute_n*conditionals)/colSums(conditionals)
> post_car_availability=colSums(car_availability_n*conditionals)/colSums(conditionals)

> post_commute
   Class 1    Class 2
0.2629875 0.3077349
> post_car_availability
   Class 1    Class 2
0.4465377 0.3154211
```

Figure 9.10: Running `apollo_lcUnconditionals` and `apollo_lcConditionals`

Figure 9.10 illustrates the use of this for the Swiss mode choice LC model (`Apollo_example_20.r`). We first produce the output from the `apollo_unconditionals_lc` function to compare to the conditionals later on, and also calculate the value of travel time (VTT) in each class, e.g. $VTT_a = \frac{\beta_{t,a}}{\beta_{c,a}}$, where $\beta_{t,a}$ and $\beta_{c,a}$ are the time and cost coefficients, respectively,

in class a. We then calculate the unconditional VTT obtained by taking the weighted average across classes, where this varies across individuals as the class allocation probabilities do, i.e. $VTT_n = \pi_{n,a}VTT_a + \pi_{n,b}VTT_b$. We next calculate the conditional class allocation probabilities using `apollo_lcConditionals`. As can be seen from the output, the means of the conditionals is identical to the mean of the unconditionals, but the range is much wider. Similarly, when we calculate the conditional VTT, we see a wider range for that too.

We finally use the conditional class allocation probabilities to calculate some posterior statistics for class membership. To do this, we first retain only one value for the two socio-demographic variables `commute` and `car_availability` for each individual (by using `apollo_firstRow`) to make the dimensionality the same as the conditionals, before using the formula in Eq. 9.6 to calculate the most likely value for these two variables for individuals in the two classes, given the posterior class allocation probabilities. These posteriors class allocation probabilities can of course then also be used in regression.

9.12 Summary of results for multiple models

It is often useful to produce an output file combining the estimates from multiple models run on the same data. This is facilitated by the function `apollo_combineResults`. This function allows the user to combine the results from a number of models (which can be larger than 2) for which the outputs have all been saved in the same directory. The function is called as follows:

```
apollo_combineResults(combineResults_settings)
```

where the list `combineResults_settings` has the following contents:

`modelNames`: a vector of model names, e.g. `c("Apollo_example_1", "Apollo_example_2", "Apollo_example_3")`. If this argument is not given, all models within the directory are used.
`printClassical`: if set to TRUE, the code will save classical standard errors as well as robust standard errors, computed using the sandwich estimator (cf. Huber, 1967). This setting then also affects the reporting of t-ratios and p-values (default is FALSE).
`printPVal`: if set to TRUE, p-values are saved (default is FALSE).
`printT1`: if set to TRUE, t-ratios against 1 are saved in addition to t-ratios against 0 (default is FALSE).
`estimateDigits`: number of digits used for estimates (default set to 4).
`tDigits`: number of digits used for t-ratios (default set to 2).
`pDigits`: number of digits used for p-values (default set to 2).
`sortByDate`: sort models by date of estimation. If FALSE, order is given by user or alphabetical if using all files (default is TRUE).

The function produces a *csv* file with the name `model_comparison_time` where `time` is a numerical value defined by the current date and time. The file contains for each model the name, the model description, the number of individuals and observations, the number of estimated parameters, as well as four model fit statistics, namely the final log-likelihood, the adjusted ρ^2

measure, the AIC and the BIC. Note that not all these measures will be reported for all models, e.g. ρ^2 is only calculated for discrete choice models. The actual model outputs are then included in a number of columns where this depends on the level of detail requested by the user as described above (e.g. including classical t-ratios).

The function can be called as `apollo_combineResults()`, i.e. without any arguments. In that case, the default settings are used for all arguments, and all model files within the directory are combined into the output. An example of how to call this function is included in `apollo_example_6.r`, combining the MNL, NL and CNL results.

Chapter 10

Debugging

Especially with advanced models, it is easy to make mistakes that lead to failures, most commonly in estimation. A first check is to ensure that all the pre-estimation parts of the code were run and that the memory was cleared before running the code. If that has been done, but the model still fails, then the issue is likely in `apollo_probabilities`. While *Apollo* does spot and report some errors, for others, the calculations will simply fail. This is simply a result of the fact that not all user and data errors can be anticipated by programmers. Debugging a model then becomes an important skill.

A good approach to debugging consists of following these three steps:

Step 1: Find the location of the problem
Step 2: Find out why it fails
Step 3: Solve the problem

To illustrate the process of debugging, `apollo_example_24_bug.r` is a version of the hybrid choice model with ordered measurement models that contains a bug. Figure 10.1 shows that the calculation of the initial log-likelihood fails for a large number of individuals.

```
> model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)
Testing likelihood function...
...
Log-likelihood calculation fails at starting values
Affected individuals:
  ID  LL
   3 NaN
   5 NaN
  16 NaN
  21 NaN
  32 NaN
  35 NaN
  ...
 996 NaN
Error in apollo\_estimate(apollo\_beta, apollo\_fixed, apollo\_probabilities,  :
  Log-likelihood calculation fails at values close to the starting values
In addition: Warning message:
In log(P[[j]]) : NaNs produced
```

Figure 10.1: Example of failure during model estimation

The first step is to identify the location of the failure. Unless *Apollo* reports an error message relating to specific coding errors, the source of the problem is usually to do with calculations

leading to numerical problems, either probabilities at zero or not a real number. The function `apollo_probabilities` uses an argument `functionality` as input, which can take different values that are controlled by the functions that call `apollo_probabilities`. The value most useful for debugging is `"estimate"` if this is where the problems occur.

The first step is to directly call `apollo_probabilities` from the console, with `functionality="estimate"`. This is illustrated in Figure 10.2. The output from this process is a vector with the the contribution to the likelihood function for each individual. We see the same warning message we obtained in estimation, i.e. `In log(P) : NaNs produced`. This relates to the use of log of `P` inside `apollo_probabilities`, which works with the exponential of the sum of logarithms of probabilities instead of the product of probabilities, as the former can avoid some numerical issues.

To get some initial insights, we look at the first 30 individuals. This clearly shows us that we obtain a likelihood that is not a real number for many individuals, which will then lead to a failure to calculate the initial log-likelihood.

```
> L=(apollo_probabilities(apollo_beta, apollo_inputs, functionality="estimate"))
> L[1:30]
Warning message:
In log(P) : NaNs produced
> L[1:30]
            1             2             3             4             5             6             7
7.989858e-09 5.643820e-09           NaN 1.183843e-10           NaN 3.415758e-10 7.045842e-07
            8             9            10            11            12            13            14
4.116410e-09 9.571308e-09 7.218330e-11 1.255858e-10 4.239593e-10 1.048240e-07 7.148491e-09
           15            16            17            18            19            20            21
7.272700e-13           NaN 3.757502e-10 9.788657e-12 8.996220e-11 2.404696e-07           NaN
           22            23            24            25            26            27            28
9.737283e-11 3.491144e-12 6.183129e-10 2.278400e-09 5.485415e-10 4.065503e-11 8.689893e-09
           29            30
6.857815e-07 3.646095e-10
```

Figure 10.2: Debugging step 1: testing with `functionality="estimate"`

A first step is to now identify those individuals for which the likelihood calculation fails, and study their choices. This is illustrated in Figure 10.3, where we only report the first row for each of the attitudinal indicators per individual (given that these are repeated across the rows for different choice tasks). We look at the choices for any individuals whose contribution to the log-likelihood is NA, which could mean that the likelihood is zero, negative or not a real number. We clearly see that all concerned individuals always give level 4 for the final indicator, and this provides our first clue to the source of the problem.

The core step in debugging is now to identify the source of the failure. For this, we run the code inside `apollo_probabilities` line by line. We can do this after setting a value for `functionality`, in our case `estimate`, as this is where it fails. We do this up until the point before the separate model components are combined into a joint model, to allow us to identify which component fails. The process is illustrated in Figure 10.4. This clearly highlights issues with the final measurement model, with negative probabilities being obtained for individuals who select level 4 for this indicator. Negative probabilities in an Ordered Logit model arise when the thresholds do not rise monotonically, and the solution to the problem in this model is thus to correct the starting values for the thresholds for the final indicator (which were set to -2, -1, 2, 1).

In this section, we have looked at a very specific case of failure, where this was a clear user error. There are many reasons why a model could fail, and in other cases, it could be due to the data,

```
> IDs=unique(database$ID)
> failures=IDs[is.na(log(L))]

> database$best[database$ID%in%failures]
> failures=IDs[is.na(log(L))]

> database$best[database$ID%in%failures]
  [1] 3 2 3 2 3 3 4 2 4 4 2 2 4 2 3 1 2 2 2 1 3 1 3 3 1 4 4 4 1 4 3 4 2 1 1 4 1 3 2 4 3 2 4
 [44] 3 2 4 3 4 3 1 1 1 3 1 1 2 3 3 2 2 4 1 2 3 4 2 3 4 1 1 2 1 4 2 3 2 3 3 1 1 3 3 1 2 3 2
 [87] 3 1 3 3 2 2 4 2 2 2 1 1 3 4 2 1 4 1 2 2 3 3 1 1 2 2 1 1 4 1 4 1 1 4 2 3 1 4 4 4 4 2 4
[130] 3 1 4 2 4 1 1 1 3 2 3 3 1 2 1 2 2 1 1 1 4 1 1 2 2 2 1 1 1 3 2 3 3 1 2 2 2 4 3 1 2 1 4
[173] 4 1 2 1 1 2 1 2 1 2 2 2 2 2 1 1 1 2 1 4 3 1 1 3 1 1 2 1 3 1 2 4 1 1 1 3 2 1 1 4 3 2 4
...

> database$attitude_quality[database$ID%in%failures&database$task==1]
  [1] 1 4 1 2 2 3 1 3 2 1 3 3 3 1 3 5 5 5 4 3 1 3 3 3 1 3 3 4 1 3 3 2 1 3 3 3 4 4 3 5 1 5 1 3
 [45] 1 3 3 3 4 1 1 3 4 1 1 1 4 5 1 1 1 3 2 1 3 3 3 3 3 3 3 1 3 1 3 1 1 1 2 3 5 2 2 3 3 1 5 1
 [89] 3 5 1 3 2 3 2 1 3 4 5 3 1 3 1 5 4 4 3 4 5 3 4 5 5 2 1 5 2 3 5 3 2 3 3 1 3 3 3 3 1 5 1 3
[133] 3 3 4 3 3 3 3 1 1 3 1 3 1 2 2 3 4 1 3 1 3 3 1 1 1 4 3 3 2 2 2 3 3 4 3 1 1 4

> database$attitude_ingredients[database$ID%in%failures&database$task==1]
  [1] 5 2 5 5 5 3 5 5 4 5 4 1 5 3 4 5 4 1 3 5 3 5 3 5 5 5 3 4 3 4 2 5 5 3 3 3 1 3 3 2 3 5 1 1
 [45] 3 4 2 5 3 2 5 3 3 5 1 3 3 4 5 4 5 2 3 5 3 3 3 3 4 2 2 3 4 3 2 5 3 3 5 4 1 3 4 3 3 4 5 3
 [89] 1 3 3 3 5 3 4 5 2 1 4 3 3 3 4 5 4 2 1 3 4 2 5 2 4 3 5 3 5 5 2 3 5 5 4 4 3 3 4 2 5 4 4 3
[133] 5 3 2 4 5 4 4 3 1 3 5 2 4 4 5 2 5 3 4 3 4 2 5 5 5 2 3 3 4 5 5 2 3 3 3 4 5 3

> database$attitude_patent[database$ID%in%failures&database$task==1]
  [1] 1 4 1 2 1 2 1 1 3 2 4 5 1 1 2 1 4 5 4 3 3 2 2 3 1 1 3 1 2 3 1 2 2 1 1 3 4 3 3 5 3 3 3 4
 [45] 4 1 3 5 3 1 3 1 3 1 1 3 3 3 1 3 1 3 3 2 1 3 4 1 3 4 3 1 1 2 1 2 3 3 1 1 3 1 3 2 1 4 1 5
 [89] 3 3 1 3 3 3 3 1 4 5 5 5 2 4 1 2 3 1 5 1 4 3 3 5 2 1 3 4 3 5 1 2 1 3 5 1 1 5 5 2 2 3 1 1
[133] 1 1 3 2 3 3 1 3 4 2 2 4 4 2 2 5 3 2 2 3 2 4 1 2 1 3 3 3 1 2 3 3 1 2 5 1 2 1

> database$attitude_dominance[database$ID%in%failures&database$task==1]
  [1] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
 [45] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
 [89] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
[133] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

Figure 10.3: Debugging step 2: analysing choices for respondents with zero likelihood value

```
> functionality="estimate"
> apollo_attach(apollo_beta, apollo_inputs)
...
> P[["choice"]] = apollo_mnl(mnl_settings, functionality)

> summary(P[["indic_quality"]][database$ID%in%failures&database$task==1])
    Min.  1st Qu.   Median     Mean 3rd Qu.     Max.
0.00000 0.03051 0.21219 0.27807 0.53681 0.91847
> summary(P[["indic_ingredients"]][database$ID%in%failures&database$task==1])
    Min.  1st Qu.   Median     Mean 3rd Qu.     Max.
0.00000 0.02925 0.17639 0.25340 0.45383 0.92321
> summary(P[["indic_patent"]][database$ID%in%failures&database$task==1])
    Min.  1st Qu.   Median     Mean 3rd Qu.     Max.
0.00000 0.02301 0.15064 0.24031 0.41793 0.90898
> summary(P[["indic_dominance"]][database$ID%in%failures&database$task==1])
       Min.     1st Qu.      Median        Mean     3rd Qu.        Max.
-0.3829249 -0.2737148 -0.1352911 -0.1612505 -0.0433236 -0.0000036
> summary(P[["choice"]][database$ID%in%failures])
    Min.  1st Qu.   Median     Mean 3rd Qu.     Max.
0.01573 0.16881 0.25001 0.25004 0.33119 0.48516
```

Figure 10.4: Debugging step 3: actual debugging

or the size of the model. The majority of issues manifest themselves through zero (or negative) probabilities, either for some observations or some individuals, leading to a failure to calculate the log-likelihood. There are three broad cases where this happens:

- The zero/negative probabilities could arise for some individual observations, where this becomes clear by looking at e.g. the output of `apollo_mnl`, where probabilities are at the

observation level. This may be related to data or starting values.

- The zero/negative probabilities could also arise at the level of individuals, where they may arise only after calling `apollo_panelProd`, where many small numbers will get multiplied together, potentially leading to a product of zero. This issue can often be resolved by using `workInLogs=TRUE` in `apollo_control`, which works with the sum of log-probabilities, rather than the product of probabilities.
- Finally, in models with multiple components (e.g. hybrid choice), problems may apply to some models only, as seen in our example, but the product across models that all produce probabilities that are greater than 0 may similarly lead to problems in case of many components, where `workInLogs=TRUE` may again help.

Chapter 11

Extensions

11.1 Starting value search

In classical estimation, convergence to the global maximum of the likelihood function is not guaranteed by any optimization algorithm. While this is not a problem for simple linear in attributes MNL models due to their concave likelihood function, it might be for other more complex models, such as Mixed Logit or Latent Class models. A popular approach to reduce the probability of reaching a poor local maximum is to start the optimization process from several different candidate points (i.e. sets of parameters), and keep the solution with the highest likelihood. However, this approach is very computationally intensive. To reduce its cost, algorithms have been proposed to dynamically eliminate unpromising candidates.

The function `apollo_searchStart` implements a simplified version of the algorithm proposed by Bierlaire et al. (2010), where the main difference in our implementation lies in the fact that `apollo_searchStart` uses only two out of three tests on the candidates described by Bierlaire et al. (2010). The implemented algorithm has the following steps, where these use a number of inputs, which we will define below:

1. Randomly draw `nCandidates` candidates from an interval given by the user.
2. Label all candidates with a valid log-likelihood (LL) as *active*.
3. Apply `bfgsIter` iterations of the BFGS algorithm to each active candidate.
4. Apply the following tests to each candidate:
   (a) Has the BGFS search converged?
   (b) Are the candidate parameters after BFGS closer than `dTest` from any other candidate with higher LL?
   (c) Is the LL of the candidate after BFGS further than `distLL` from a candidate with better LL, and is its gradient smaller than `gTest`?
5. Mark any candidates for which at least one of these tests is passed as *inactive*.
6. Go back to step 3 for the remaining candidates.

The apollo_searchStart function is called as follows:

```
apollo_beta = apollo_searchStart(apollo_beta,
                                 apollo_fixed,
                                 apollo_probabilities,
                                 apollo_inputs,
                                 searchStart_settings)
```

The function returns an updated list of starting values. The list searchStart_settings has the following contents:

apolloBetaMin: a vector of the minimum possible value for each parameter (default is apollo_beta-0.5).
apolloBetaMax: a vector of the maximum possible value for each parameter (default is apollo_beta+0.5).
nCandidates: the number of initial candidates (default is 100).
maxStages: the maximum number of iterations of the algorithm, i.e. maximum number of times the algorithm jumps from step 6 to 3 described below (default is 10).
bfgsIter: the maximum number of BFGS iterations to apply to each candidate in each iteration of the main algorithm (default is 10).
smartStart: if TRUE, the Hessian of apollo_probabilities is calculated at apollo_beta, and the initial candidates are drawn with a higher probability from the area where the Hessian indicates an improvement in the likelihood. This adds a significant amount of time to the initialization of the algorithm (default is FALSE).
dTest: the tolerance of test 4.2 described below (default is 1).
gTest: the tolerance for the gradient in test 4.3 described below (default is 10^{-3}).
llTest: the tolerance for the LL in test 4.3 described below (default is 3).

The performance of the function varies across models and datasets and depends on the settings used. In particular, we advise to adjust the bfgsIter, dTest, distLL and gTest parameters to suit each user's particular model characteristics, as their default values might not be suitable for some model specifications. The running of the function is illustrated in Figure 11.1 for example apollo_example_20.r, where only a small part of the output is shown.

11.2 Out of sample fit (Cross validation)

A common method to test for overfitting of a model is to measure its fit on a sample not used during estimation, i.e. measuring out-of-sample fit. A simple way to do this is to split the available dataset into two parts: an estimation sample, and a validation sample. The model of interest is estimated using only the estimation sample, and then those estimated parameters are used to measure the fit of the model (e.g. the log-likelihood of the model) on the validation sample. Doing this with only one validation sample may however lead to biased results, as a particular validation sample need not be representative of the population. One way to minimise this issue is

```
> apollo_beta=apollo_searchStart(apollo_beta, apollo_fixed,apollo_probabilities, apollo_inputs)
...
Creating initial set of 100 candidate values.
Calculating LL of candidates 0%....50%.....100%

Stage 1, 100 active candidates.
 Estimating 20 BFGS iteration(s) for each active candidate.
 Candidate..... LLstart ..... LLfinish ..... GradientNorm ... Converged
         1         -1756         -1562            78.273           0
         2         -1804         -1551           350.796           0
...
       100         -1867         -1562             8.991           0
 Candidate 1 dropped.
  Failed test 1: Too close to 22 26 29 40 44 46 58 72 75 78 91 95 97 98 100 in parameter space.
 Candidate 3 dropped.
...
 Candidate 100 dropped.
  Failed test 1: Too close to 29 40 44 72 91 95 98 in parameter space.
 Best candidate so far (LL=-1551.2)
...
...
Stage 3, 8 active candidates.
 Estimating 20 BFGS iteration(s) for each active candidate.
 Candidate..... LLstart ..... LLfinish ..... GradientNorm ... Converged
        15         -1587         -1580           404.556           0
        42         -1582         -1549             2.635           0
        61         -1574         -1562             0.517           0
        76         -1583         -1562             9.288           0
        77         -1606         -1599           220.242           0
        83         -1549         -1549                 0           1
        86         -1589         -1566            424.43           0
        89         -1623         -1563           169.132           0
 Candidate 15 dropped.
  Failes test 2: Converging to a worse solution than 83
...
 Best candidate so far (LL=-1549.1)
                    [,1]
asc_1           -0.0567
asc_2            0.0000
beta_tt_a       -0.0553
beta_tt_b       -0.3649
beta_tc_a       -0.0787
beta_tc_b       -2.2416
beta_hw_a       -0.0423
beta_hw_b       -0.0554
beta_ch_a       -0.9951
beta_ch_b       -2.8356
delta_a          0.7372
gamma_commute_a -0.5892
gamma_car_av_a   0.5894
delta_b          0.0000
gamma_commute_b  0.0000
gamma_car_av_b   0.0000
```

Figure 11.1: Running `apollo_searchStart`

to randomly draw several pairs of estimation and validation samples from the complete dataset, and apply the procedure to each pair. This also allows the calculation of a confidence interval for the out-of-sample measure of fit.

The function `apollo_outOfSample` implements the process described above. It is called as

follows:

```
apollo_outOfSample(apollo_beta,
                   apollo_fixed,
                   apollo_probabilities,
                   apollo_inputs,
                   estimate_settings,)
                   outOfSample_settings)
```

The only new input here is `outOfSample_settings`, which has the following contents:

`nRep`: Number of times a different pair of estimation and validation sets are to be extracted from the full database (default is 10).

`validationSize`: Size of the validation sample. It can be provided as a fraction of the whole database (number between 0 and 1), or a number of individuals (number bigger than 1). The splitting of the database is done at the individual level, not at the observation level (default is 0.1).

`samples`: An optional numeric matrix or data.frame with as many rows as observations in the `database`, and as many columns as number of repetitions wanted. Each column represents a re-sample, and each element must be a 0 if the observation should be assigned to the estimation sample, or 1 if the observation should be assigned to the prediction sample. If this argument is provided, then `nRep` and `validationSize` are ignored. Note that this allows sampling at the observation rather than the individual level.

`apollo_outOfSample` saves to disk a file called `name_outOfSample_params.csv`, where `name` is the name of the model as defined in `apollo_control`. This file contains the estimates from each of the `nRep` estimation runs, as well as the estimation and out of sample log-likelihoods. It also saves a file `name_outOfSample_samples.csv` which contains information on the samples. In addition, the function prints to screen the per observation log-likelihood for each subsample, both for the estimation sample and the holdout sample. The running of the function is illustrated in Figure 11.2 for example `apollo_example_18.r`.

Before running, `apollo_outOfSample` will look for existing `name_outOfSample_params.csv` and `name_outOfSample_samples.csv` files in the working directory. If they are found, and are consistent with the model at hand, then new repetitions will be added to those files. This means, for example, that if the function is run twice, the output files will hold results for 20 repetitions. This is a useful feature in case the cross-validation process gets interrupted, or if the user wants to increase the number of repetitions a posteriori.

11.3 Bootstrap estimation

Apollo also allows the user to use bootstrap estimation. Given a number of repetitions, this function generates as many new samples as requested, by sampling individuals (i.e. blocks of

```
> apollo_outOfSample(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

10 separate runs will be conducted, each using a random subset of 90% for estimation and the remainder
  for validation.
Number of individuals
- for estimation : 349
- for forecasting : 39
- in sample (total): 388
Preparing loop.
Estimated parameters and log-likelihoods for each sample will be written to:
  Apollo_example_18_outOfSample_params.csv
The matrix defining the observations used in each repetition will be written to:
  Apollo_example_18_outOfSample_samples.csv

Estimation cycle 1 (3141 obs.)
Estimation results written to file.
Estimation cycle 2 (3141 obs.)
Estimation results written to file.
Estimation cycle 3 (3141 obs.)
Estimation results written to file.
Estimation cycle 4 (3141 obs.)
Estimation results written to file.
Estimation cycle 5 (3141 obs.)
Estimation results written to file.
Estimation cycle 6 (3141 obs.)
Estimation results written to file.
Estimation cycle 7 (3141 obs.)
Estimation results written to file.
Estimation cycle 8 (3141 obs.)
Estimation results written to file.
Estimation cycle 9 (3141 obs.)
Estimation results written to file.
Estimation cycle 10 (3141 obs.)
Estimation results written to file.
Processing time: 1.725478 mins

      LL per obs in estimation sample LL per obs in validation sample
 [1,]                      -0.4507688                       -0.4252647
 [2,]                      -0.4431325                       -0.4959779
 [3,]                      -0.4506121                       -0.4252124
 [4,]                      -0.4415683                       -0.5075583
 [5,]                      -0.4442535                       -0.4805690
 [6,]                      -0.4415376                       -0.4739547
 [7,]                      -0.4438256                       -0.4906856
 [8,]                      -0.4422680                       -0.4652911
 [9,]                      -0.4424068                       -0.5012229
[10,]                      -0.4412086                       -0.5083646
```

Figure 11.2: Running `apollo_outOfSample`

observations) *with replacement* from the original dataset. Parameters are then estimated for each of these new samples. Finally, the covariance matrix of the sequence of estimated parameters is calculated. This matrix is in itself an estimator of the covariance matrix of the parameter estimates.

The function `apollo_bootstrap` implements the process described above. It is called as follows:

```
apollo_bootstrap(apollo_beta,
                 apollo_fixed,
                 apollo_probabilities,
                 apollo_inputs,
                 estimate_settings,
                 bootstrap_settings)
```

The only new input here is `bootstrap_settings`, which has the following contents:

`nRep`: Number of bootstrap samples to use (default is 30).
`samples`: An optional numeric matrix or data.frame with as many rows as observations in the `database`, and as many columns as number of repetitions wanted. Each column represents a re-sample, and each element must be a 0 or a positive integer representing the number of times that row is used in that given sample. If this argument is provided, then `nRep` is ignored. Note that this allows sampling at the observation rather than the individual level.
`seed`: An optional positive integer used as seed for the bootstrap sampling generation process. Default is 24. It is only used if `samples` is `NA`. Changing the seed allows drawing new samples when re-starting a bootstrap process. This is useful when a bootstrap process has been interrupted, or when additional repetitions are needed.

`apollo_bootstrap` returns a list with three elements. The first element is a matrix containing the estimated parameters in each repetition, with as many rows as repetitions, and as many columns as parameters. The second element is the correlation matrix between the estimated parameters across repetitions. The third element is a vector with the log-likelihood at convergence in each repetition.

`apollo_bootstrap` saves to disk a file called `name_bootstrap_params.csv`, where `name` is the name of the model as defined in `apollo_control$modelName`. This file contains the estimates from each of the `nRep` estimation runs, as well as the log-likelihoods. It also saves a file `name_bootstrap_samples.csv` which contains information on the samples and a file called `name_bootstrap_vcov.csv` containing the covariance matrix. If the files already exist, for example because the process got interrupted, the new repetitions will be added to the existing files. So, if the function is run twice with the default settings, the `name_bootstrap_params.csv` file will contain 60 sets of parameters. In addition, the function prints to screen the covariance matrix. The running of the function is illustrated in Figure 11.3 for example `apollo_example_18.r`.

The function can also be called directly during estimation, as described in Section 4.6, by adjusting the setting `estimate_settings$bootstrapSE`.

11.4 Expectation-maximisation (EM) algorithm

Apollo allows the user to estimate models using Expectation - Maximisation (EM) algorithms. These are iterative algorithms were the updating of the parameters is usually achieved through the maximization of a simplified version of the model likelihood function. EM algorithms do not provide standard error estimates for the parameters. To obtain them, a Maximum Likelihood estimation with the EM estimated parameters as the starting values is typically run afterwards. This guarantees quick convergence and standard errors for all parameters. For a detailed discussion of EM algorithms, see Train (2009, ch. 14).

The precise steps of these algorithms change depending on the kind of model, making them hard to generalize and implement in a flexible way. *Apollo* includes EM routines for two types of models, namely Latent Class (LC) models where all parameters vary across classes, and Mixed Multinomial Logit (MMNL) models where all parameters are random, with a full covariance

```
> apollo_bootstrap(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

30 new datasets will be constructed by randomly sampling 388 individuals with replacement from the
  original dataset.
Preparing bootstrap.
Parameters and LL in each repetition will be written to: Apollo_example_18_bootstrap_params.csv
Vectors showing sampling rate for each observation in each repetition written to:
  Apollo_example_18_bootstrap_samples.csv

Estimation cycle 1 (3492 obs)
Estimation results written to file.
...
Estimation cycle 30 (3492 obs)
Estimation results written to file.

Finished bootstrap runs.
Parameters and LL for each repetition written to: Apollo_example_18_bootstrap_params.csv
Vectors showing sampling rate for each observation in each repetition written to:
  Apollo_example_18_bootstrap_samples.csv
Covariance matrix of parameters written to: Apollo_example_18_bootstrap_vcov.csv

Mean LL across runs: -1553.99
Mean parameter values across runs:
          Estimate
asc_1      -0.0407
asc_2       0.0000
beta_tt_a  -0.0875
beta_tt_b  -0.1201
beta_tc_a  -0.1049
beta_tc_b  -0.7484
beta_hw_a  -0.0403
beta_hw_b  -0.0494
beta_ch_a  -0.9999
beta_ch_b  -2.0265
delta_a     0.1838
delta_b     0.0000

Covariance matrix across runs:
              asc_1  beta_tt_a  beta_tt_b ...
    asc_1  3.119e-03 -4.156e-04 -0.0006079 ...
beta_tt_a -4.156e-04  1.032e-03 -0.0015649 ...
beta_tt_b -6.079e-04 -1.565e-03  0.0070312 ...
...
              delta_a
    asc_1 -6.761e-05
beta_tt_a  5.163e-03
beta_tt_b -2.302e-02
beta_tc_a  2.088e-03
beta_tc_b -2.137e-01
beta_hw_a -1.398e-03
beta_hw_b  1.458e-04
beta_ch_a -1.168e-01
beta_ch_b -1.524e-02
  delta_a  2.610e-01
Bootstrap processing time: 5.735613 mins
```

Figure 11.3: Running `apollo_bootstrap`

matrix being estimated. These functions were added in *Apollo* v0.2.0 and users interested in the manual coding of these routines are referred to the manual for earlier versions of *Apollo*.

11.4.1 EM algorithm for LC model

Apollo incorporates the function `apollo_lcEM` for latent class models, where this can be used for models with any functional form inside the latent classes (i.e. not just MNL), as long as all parameters vary across classes (i.e. no generic parameters). The function `apollo_lcEM` is called

via:

```
model = apollo_lcEM(apollo_beta,
                    apollo_fixed,
                    apollo_probabilities,
                    apollo_inputs,
                    lcEM_settings,
                    estimate_settings)
```

where we have already covered the first four arguments, as well as the final one. The further optional argument `lcEM_settings` can contain the following entries:

`stoppingCriterion:` A numeric convergence criterion. The EM process will stop when improvements in the log-likelihood fall below this value (default is 10^-5).

`EMmaxIterations:` An integer setting a maximum number of iterations of the EM algorithm before stopping (default is 100).

`postEM:` A scalar which determines the number of tasks performed by this function after the EM algorithm has converged. Can take values 0, 1 or 2 only. If value is 0, only the EM algorithm will be performed, and the results will be a model object without a covariance matrix (i.e. estimates only.). If value is 1, after the EM algorithm the covariance matrix of the model will be calculated as well, and the result will be a model object with a covariance matrix. If value is 2, after the EM algorithm, the estimated parameter values will be used as starting value for a maximum likelihood estimation process, which will render a model object with a covariance matrix. Performing maximum likelihood estimation after the EM algorithm is useful, as there may be room for further improvement (default is 2).

`silent:` A boolean variable, which, when set to TRUE, means that no information is printed to the screen during estimation (default is set to FALSE).

In addition to ensuring that all parameters vary across classes (or are included in `apollo_fixed`, the only other requirement for using the `apollo_lcEM` function is that inside the `apollo_probabilities` function, the loop over classes is defined as

```
for(s in 1 : length(pi_values)){
                                ...
                                }
```

We first describe the EM estimation of a choice model with S different classes. The conditional choice probability of class s (i.e. the in-class probability) is determined by a MNL model. All preference parameters β are allowed to vary across classes. The class allocation is determined by a MNL model using covariates Z_n such as an individual's income, interacted with a vector of parameters γ_s in class s, causing the allocation probabilities $\pi_{n,s}$ to change from one individual to the next. Estimation is achieved through an iterative five step process detailed below (cf. Train, 2009, ch. 14), drawing also on Bhat (1997).

1. Definition of starting values for γ_s^0 and β_s^0 parameters, where β_s^0 should be different across classes.
2. Calculate the likelihood of the whole model, using $\gamma^0 = \langle \gamma_1^0, ..., \gamma_S^0 \rangle$, which gives $\pi^0 = \langle \pi_1^0, ..., \pi_S^0 \rangle$, and $\beta^0 = \langle \beta_1^0, ..., \beta_S^0 \rangle$ and store this as L_0.
3. Calculate class allocation probabilities conditional on observed choices for each class $s \in \{1, ..., S\}$, using the following expression.

$$h_{n,s}^0 = \frac{\pi_{n,s}^0(\gamma^0) L_{n,s}(\beta_s^0)}{\sum_{s=1}^S \pi_{n,s}^0(\gamma^0) L_{n,s}(\beta_s^0)} \tag{11.1}$$

   where $L_{n,s}(\beta_s^0)$ is the likelihood of the observed choice for individual n assuming class s, and where $\pi_{n,s}^0(\gamma^0)$ is the class allocation probability for individual n for class s, using γ^0 as parameters.
4. Update the parameters γ used in the class allocation model by maximising the allocation probabilities weighted by $h_{n,s}^0$.

$$\gamma^1 = argmax_\gamma (\sum_{n=1}^N \sum_{s=1}^S h_{n,s}^0 log(\pi_{n,s})), \tag{11.2}$$

   where this shows the maximisation of the log-likelihood of this component, where the likelihood is given by $\prod_{n=1}^N \prod_{s=1}^S \pi_{n,s}^{h_{n,s}^0}$ (cf. Bhat, 1997).
5. Update the parameters β_s for the within class model for each class by estimating separate weighted MNL models, just as in the procedure in Section **??**. Estimation of each MNL model can be done using Maximum Likelihood, using $h_{n,s}^0$ as weights.

$$\beta_s^1 = argmax_{\beta_s} (\sum_{n=1}^N h_{n,s}^0 log(L_{n,s})) \; \forall s \tag{11.3}$$

6. Calculate the likelihood of the whole model, using $\gamma^1 = \langle \gamma_1^1, ..., \gamma_S^1 \rangle$, which gives $\pi^1 = \langle \pi_1^1, ..., \pi_S^1 \rangle$, and $\beta^1 = \langle \beta_1^1, ..., \beta_S^1 \rangle$ and store this as L_1. If $L_1 - L_0 < c$, where c is a convergence limit, say 10^-5, then convergence has been reached. If convergence is not achieved, set $\gamma^0 = \gamma^1$ and $\beta^0 = \beta^1$, and return to step 2.

We illustrate this example using a model similar to `Apollo_example_20.r`, , i.e. the two-class LC model on the Swiss route choice data, with the difference being that the ASCs are now also class-specific. This example is available in `Apollo_example_28.r`. The implementaion is shown in Figure 11.4, with the estimation process in Figure 11.5.

11.4.2 MMNL model with full covariance matrix for random coefficients

In this subsection, we describe the EM estimation of a MMNL model in which all parameters are random and where we estimate a full covariance matrix between them. More formally, we assume the preference parameters to follow a joint random normal distribution $\beta \sim N(\mu, \Sigma)$, where transformations to other distributions are straightforward, as explained below.

The iterative process is described below (cf. Train, 2009, chapter 11).

```
apollo_beta = c(asc_1_a          = 0,
                asc_1_b          = 0,
                ...)

apollo_lcPars=function(apollo_beta, apollo_inputs){
  lcpars = list()
  lcpars[["asc_1"]] = list(asc_1_a, asc_1_b)
  ...

  return(lcpars)
}

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ...
  ### Loop over classes
  for(s in 1:length(pi_values)){

    ### Compute class-specific utilities
    V=list()
    V[['alt1']]   = asc_1[[s]] + beta_tc[[s]]*tc1 + beta_tt[[s]]*tt1 + beta_hw[[s]]*hw1 + beta_ch[[s]]*
        ↪ ch1
    V[['alt2']]   = asc_2      + beta_tc[[s]]*tc2 + beta_tt[[s]]*tt2 + beta_hw[[s]]*hw2 + beta_ch[[s]]*
        ↪ ch2

    mnl_settings$V = V
    mnl_settings$componentName = paste0("Class_",s)

    ### Compute within-class choice probabilities using MNL model
    P[[paste0("Class_",s)]] = apollo_mnl(mnl_settings, functionality)

    ### Take product across observation for same individual
    P[[paste0("Class_",s)]] = apollo_panelProd(P[[paste0("Class_",s)]], apollo_inputs ,functionality)
  }

  ### Compute latent class model probabilities
  lc_settings   = list(inClassProb = P, classProb=pi_values)
  P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 11.4: EM algorithm for Latent Class: setup

1. Define starting values for μ^0 and Σ^0.
2. Generate R multivariate draws for each individual n, say $\beta^0_{n,r}$ for draw r, where $\beta^0_{n,r}$ contains one value for each random parameter.
3. Calculate the model likelihood at the individual level for each draw, i.e. $L_0 = L_{n,r}(\beta^0_{n,r})$.
4. Calculate weights for each draw and for each individual using the following expression.

$$w^0_{n,r} = \frac{L_{n,r}}{\frac{\sum_r L_{n,r}}{R}} \; \forall r, n \tag{11.4}$$

5. Update the means of the random parameters.

$$\mu^1 = \frac{w^0_{n,r}\beta^0_{n,r}}{RN} \tag{11.5}$$

   Where N is the number of individuals in the sample.
6. Update the covariance matrix of the random parameters, given by

$$\Sigma^1 = \frac{w^0_{n,r}\Sigma^0_{n,r}}{RN}, \tag{11.6}$$

```
> model=apollo_lcEM(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)
The use of apollo_lcEM has a number of requirements. No checks are run for these, so the
  user needs to ensure these conditions are met by their model:
1:This function is only suitable for single component models, i.e. no use of
  apollo_combineModels or manual multiplication of model components.
2:Any parameters that vary across classes need to be included in the definition of random
  parameters in apollo_lcPars.
3:The entries in the lists in apollo_lcPars need to be individual parameters, not
  functions thereof.

Validating inputs of likelihood function (apollo_probabilities)

Validating inputs of likelihood function (apollo_probabilities)

Initialising EM algorithm
Starting iteration: 1
Current LL : -1755.505

...

Starting iteration: 35
Current LL : -1558.427
New LL     : -1558.427
Improvement: 9.141917e-06

EM algorithm stopped: improvements in LL smaller than convergence criterion.
Continuing with classical estimation...

Testing likelihood function...

Pre-processing likelihood function...

Starting main estimation
Initial function value: -1558.427
Initial gradient value:
       asc_1_a         asc_1_b       beta_tt_a       beta_tt_b       beta_tc_a       beta_tc_b
-0.025103191    0.018312903    -0.246172021    -0.539511348     0.180421239     0.088650950
     beta_hw_a       beta_hw_b       beta_ch_a       beta_ch_b         delta_a gamma_commute_a
 0.323793074    -0.217498382    -0.078109679    -0.036779284     0.002144134    -0.002519300
gamma_car_av_a
-0.003000423
initial  value 1558.426648
iter   1 value 1558.426644
final  value 1558.426644
converged
Estimated parameters:
                  Estimate
asc_1_a            -0.08345
...
gamma_car_av_b      0.00000

Computing covariance matrix using numerical methods (numDeriv).
 0%....25%....50%....75%....100%
Negative definite Hessian with maximum eigenvalue: -8.782019
Computing score matrix...

Summary of class allocation for LC model component :
        Mean prob.
Class_1     0.4702
Class_2     0.5298

Calculating LL(0)...
Calculating LL of each model component...
```

Figure 11.5: EM algorithm for Latent Class: estimation

where this requires calculating the covariance matrix $\Sigma^0_{n,r}$ at the individual draw level, given by:

$$\Sigma^0_{n,r} = \left(\beta^0_{n,r} - \mu^1\right) \cdot \left(\beta^0_{n,r} - \mu^1\right)'. \tag{11.7}$$

7. Calculate the likelihood of the whole model and check for convergence. Convergence is

achieved when the change on this likelihood is smaller than an pre-defined value. If convergence is not achieved, return to step 2.

Apollo incorporates the function `apollo_mixEM` for MMNL models, where this can be used for models with all parameters being random, and a full covariance matrix being estimated. The function `apollo_mixEM` is called via:

```
model = apollo_mixEM(apollo_beta,
                     apollo_fixed,
                     apollo_probabilities,
                     apollo_inputs,
                     lcEM_settings,
                     estimate_settings)
```

A specific consideration applies in that for `apollo_mixEM`, the entries in `apollo_beta` need to be provided in a specific order. With K random parameters, the first K entries need to be the means for the underlying Normals, followed by the elements of the lower triangle of the Cholesky matrix, by row.

The further optional argument `mixEM_settings` can contain the following entries:

`stoppingCriterion:` A numeric convergence criterion. The EM process will stop when improvements in the log-likelihood fall below this value (default is 10^-5).

`EMmaxIterations:` An integer setting a maximum number of iterations of the EM algorithm before stopping (default is 100).

`postEM:` A scalar which determines the number of tasks performed by this function after the EM algorithm has converged. Can take values 0, 1 or 2 only. If value is 0, only the EM algorithm will be performed, and the results will be a model object without a covariance matrix (i.e. estimates only.). If value is 1, after the EM algorithm the covariance matrix of the model will be calculated as well, and the result will be a model object with a covariance matrix. If value is 2, after the EM algorithm, the estimated parameter values will be used as starting value for a maximum likelihood estimation process, which will render a model object with a covariance matrix. Performing maximum likelihood estimation after the EM algorithm is useful, as there may be room for further improvement (default is 2).

`silent:` A boolean variable, which, when set to TRUE, means that no information is printed to the screen during estimation (default is set to FALSE).

`transforms:` A list, with one entry per parameter, showing the inverse transform to return from beta to the underlying Normal. E.g. if the first parameter is specified as negative logormal inside `apollo_randCoeff`, then the entry in transforms should be `transforms[[1]]=function(x) log(-x)`. If any transforms are used, then they need to be provided for all random parameters, even untransformed ones.

Figures 11.6 and 11.7 presents code implementing this algorithm for the EM analogue of `Apollo_example_15.r`, i.e. using a MMNL model with correlated negative Lognormals on the

Swiss route choice data. This example is implemented in `Apollo_example_29.r`. The core steps to focus on are the order of parameters in `apollo_beta` and the inverse transforms for a negative Lognormal in `mixEM_settings$transforms`.

```
apollo_beta = c(mu_log_b_tt       =-3,
                mu_log_b_tc       =-3,
                mu_log_b_hw       =-3,
                mu_log_b_ch       =-3,
                sigma_log_b_tt    = 0.4880,
                sigma_log_b_tt_tc = 0,
                sigma_log_b_tc    = 1.0369,
                sigma_log_b_tt_hw = 0,
                sigma_log_b_tc_hw = 0,
                sigma_log_b_hw    = 0.8141,
                sigma_log_b_tt_ch = 0,
                sigma_log_b_tc_ch = -0.3,
                sigma_log_b_hw_ch = 0,
                sigma_log_b_ch    = -0.8298)


apollo_randCoeff = function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["b_tt"]] = -exp( mu_log_b_tt  + sigma_log_b_tt    * draws_tt )
  randcoeff[["b_tc"]] = -exp( mu_log_b_tc  + sigma_log_b_tt_tc * draws_tt + sigma_log_b_tc    * draws_tc
      ↪  )
  randcoeff[["b_hw"]] = -exp( mu_log_b_hw  + sigma_log_b_tt_hw * draws_tt + sigma_log_b_tc_hw * draws_tc
      ↪  + sigma_log_b_hw    * draws_hw )
  randcoeff[["b_ch"]] = -exp( mu_log_b_ch  + sigma_log_b_tt_ch * draws_tt + sigma_log_b_tc_ch * draws_tc
      ↪  + sigma_log_b_hw_ch * draws_hw + sigma_log_b_ch * draws_ch )

  return(randcoeff)
}


mixEM_settings = list(transforms=list(function(x) log(-x),
                                      function(x) log(-x),
                                      function(x) log(-x),
                                      function(x) log(-x)))
```

Figure 11.6: EM algorithm for Mixed Logit: setup

11.5 Iterative coding of utilities for large choice sets

In the examples shown in this manual, the user codes the utilities of all alternatives one by one. With very large choice sets, this may not be practical, and a user may create the utilities recursively, for example. We illustrate this in `Apollo_example_30.r`. For this example, we generate a large dataset with 100 alternatives, each described by two attributes: x_1 and x_2. In the database, there are 200 columns specifying the value of x_1 and x_2 for each alternative, so that -for example- column `x1_30` corresponds to the value of x_1 for alternative 30. In the example, we assume the utility of each alternative contains no alternative specific constant (ASC). Figure 11.8 shows the definition of utilities, availabilities and the names of the alternatives in a recursive way, inside `apollo_probabilities`.
In this example code, we make use of three specific `R` functions that are especially useful for this context. Firstly, `paste0` creates a string by combining the various elements that are passed to it as input. This string can be used directly inside `V` as an index. It can also be used to refer to an actual variable in combination with the function `get`. Finally, the function `setNames` is used to assign the names from V to the vector with the codings for the alternatives.

In Figure 11.8, we define availability by explicitly extracting the relevant columns from the

```
> model=apollo_mixEM(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)
The use of apollo_mixEM has a number of requirements. No checks are run for these, so the
  user needs to ensure these conditions are met by their model:
1:This function is only suitable for single component models, i.e. no use of
  apollo_combineModels or manual multiplication of model components.
2:All parameters need to be random, and a full covariance matrix needs to be
  estimated/specified in apollo_randCoeff.
3:All random parameters need to be based on Normal distributions or transformations
  thereof.
4:With K random parameters, the order of the elements in 'apollo_beta' needs to be as
  follows: K means for the underlying Normals, followed by the elements of the lower
  triangle of the Cholesky matrix, by row.

Initialising EM algorithm
Validating inputs of likelihood function (apollo_probabilities)

Initial LL: -2026.718

Starting iteration: 1
Current LL: -2026.718
New LL: -1983.751
Improvement: 42.96719

...

Starting iteration: 91
Current LL: -1410.822
New LL: -1410.84
Improvement: -0.01809646

EM algorithm stopped: improvements in LL smaller than convergence criterion.
Continuing with classical estimation...
```

Figure 11.7: EM algorithm for Mixed Logit: estimation

database. The availability columns follow a similar naming structure to the attributes, with -for example- `av_30` storing the the availability of alternative 30. Note that when referencing the database we must point to the `apollo_inputs$database`, as opposed to simply `database`. If all alternatives are always available, then we could just skip the definition of availability altogether, and `apollo_mnl` would assume full availability.

```
### List of utilities
J = 100
V = list()
for(j in 1:J) V[[paste0("alt",j)]] = b1*get(paste0("x1_",j)) + b2*get(paste0("x2_",j))

### Define settings for MNL model component
mnl_settings = list(
  alternatives  = setNames(1:J, names(V)),
  avail         = setNames(apollo_inputs$database[,paste0("av",1:J)], names(V)),
  choiceVar     = choice,
  V             = V
)
```

Figure 11.8: Defining utilities for large choice sets (`Apollo_example_30.r`)

Chapter 12

Frequently asked questions

This chapter addresses some frequently asked questions (FAQ), divided into some broad categories. Many other questions by individual users have been answered in the online forum (access via www.apollochoicemodelling.com), and readers are invited to check existing posts there before raising a new question.

12.1 General

What is *Apollo*?

Apollo is a software package for the R programming language. It is a set of tools to aid with the estimation and application of choice models in R. Users are able to write their own model functions or use a mix of already available ones. Random heterogeneity, both continuous and discrete and at the level of individuals and observations, can be incorporated for all models. There is support for both standalone models and hybrid model structures. Both classical and Bayesian estimation is available, and multiple discrete continuous models are covered in addition to discrete choice. Multi-threading processing is supported for estimation. A large number of pre and post-estimation routines, including for computing posterior (individual-level) distributions, are available. For examples, a manual, and a support forum, visit www.ApolloChoiceModelling.com. For more information on choice models see Train (2009) and Hess and Daly (2014) for an overview of the field.

12.2 Installation and updating of *Apollo*

How do I install *Apollo*?

Type `install.packages("apollo")` in the R console and press enter.

How do I update *Apollo*?

You update *Apollo* by re-installing it. Type `install.packages("apollo")`in the R console and press enter.

Why do I get an error during installation that one of the packages is not available for the R version I am running?

You are likely running an old version of R. Update R to the latest version and re-install *Apollo*. You can get the latest version of R from https://cran.r-project.org/

Why does installation work on my home laptop/desktop computer but fail in my company laptop/desktop computer

Often computers from big organisations will install R packages in shared libraries (that is in folders in the private company network). R does not like its libraries to be in shared folders. As a general recommendation, always install packages in local libraries, i.e. in a folder in the local hard drive. You can see your active libraries by typing `.libPaths()`. This will list the active libraries. If the local library is, for example, the second one in the list, you should keep only that one by typing `.libPaths(.libPaths()[2])`. Then you should try installing *Apollo* again.

12.3 Data

Why do I get the error message `Error in file(file, "rt") : cannot open the connection` when trying to open the data?

This is most often caused by the user forgetting to set the working directory meaning that R cannot find the data file. Or there could be a typo in the name of the data file.

Can I use "long" formatted data in *Apollo*?

No. *Apollo* requires data to be in a "wide" format, meaning that all necessary information to calculate the likelihood of a single observation should be contained in a single row of the data. In more practical terms, for an MNL model, this means that attributes for all alternatives should be contained in each row. This format is the more common format in choice modelling, uses less space, and is also more general in allowing for a mixture of different dependent variables in the same data.

Is there a way to transform data in "long" format into "wide" format?

No such functionality is embedded in *Apollo* but is a straightforward task that can be done in a spreadsheet tool or indeed in R.

Can I use a list of dummy variables to represent the choice? For example, for a choice between three alternatives, with the third chosen use variables alt1=0, alt2=0, alt3=1?

No. *Apollo* requires the user to encode the choice in a single variable. In this case, it would be a variable (for example called "choice") that could take only three values (for example 1, 2 or 3). This is easily created in R on the basis of separate dummy variables.

Can *Apollo* estimate models with aggregate share data?

Current versions of *Apollo* require one alternative to be chosen in each row in the data, rather than using data where in each row, each alternative has a share of the choices, with these summing to 1 across alternatives. This type of data can be accommodated either by users coding their own model probability function inside *Apollo* or by replicating each row a number of times. For example, in a binary case with shares of $65 - 35$, the user could replicate the row 100 times, with 65 rows choosing alternative A and 35 rows choosing alternative B.

Can I model "dual response" survey data with *Apollo*?
Yes, this is possible. We recommend beginning by modelling both questions separately, and if there is evidence of the parameters being similar, then estimate them jointly using a scale parameter between them. See `apollo_example_22` to learn how to conduct joint estimation in *Apollo*.

12.4 Model specification

What distributions are possible for random coefficients in *Apollo*?
There are no limits imposed on distributional assumptions. The user can specify whatever distributions they want to use. Distributions can be coded as transformations of either Normal or Uniform draws. While transformations of Normal draws can be used for Lognormal, Censored or Truncated Normals and Johnson SB, Uniform distributions open up even broader scope as an inverse cumulative distribution function can be applied to the Uniform draws for a huge set of possible distributions.

How can I capture the panel structure of my data in *Apollo*?
The treatment of panel data depends completely on the model being used. Whenever the data contains multiple choices per individual, the analyst needs to use the function `apollo_panelProd` to group them together in estimation, except if using the setting `apollo_control$panelData=FALSE`, in which case the data will be treated as if all observations came from separate individuals. In models without any random heterogeneity, such as MNL, there is no explicit modelling of the correlation across choices for the same individual. All that will happen by using `apollo_panelProd` is that the calculation of the robust standard errors recognises that the choices come from the same individual. In models with random heterogeneity, such as Mixed Logit, the analyst can more explicitly account for the panel structure, for example by specifying that the heterogeneity in any random taste coefficients is across individuals, not within individuals, and/or by including an explicity pseudo-panel effect error component. These issues are discussed in detail in the manual.

How can I avoid writing each utility function separately if I have tens or hundreds of alternatives?
If the utilities all use the same structure but with different attributes for each alternative, then the utility functions (and availabilities) can be written iteratively. For example, imagine we have 100 alternatives, with attributes `x1_j` and `x2_j` for alternative `j`, then we can use:

```
V = list()
for(j in 1:100){
V[[paste0("alt",j)]] = (b1*get(paste0("x1_",j))+b2*get(paste0("x2_",j)))}
```

What starting values should I use for the thresholds in my Ordered Logit/Ordered Probit model?
The thresholds need to be different from each other, and monotonically increasing. If the thresholds are too wide, extreme ratings will obtain very low or zero probabilities. If the thresholds are too narrow, extreme ratings will obtain very large probabilities. Either of these can lead to estimation failures. Some trial and error is often required, but a good starting

point is to have thresholds symmetrical around zero, going to extreme values of +/-3.

How many draws should I use to estimate my models with random components?

There is no correct answer to this question. More draws is always better. The likelihood of the model is given by an integral without a closed form solution, and the simulation based appraoch only offers an approximation to this integral. Using a low number of draws means that the approximation to this integral is poor. In simple words, it means that the model we *are* estimating is not the one we *think* we are estimating. The parameter estimates will then be biased for the model we actually specified.

But the log-likelihood of my model is better with fewer draws, so isn't that good?

The fact that the log-likelihood is better with fewer draws does not justify the use of fewer draws. It is simply a reflection of the fact that fewer draws offers a poor approximation to the real model. Once the number of draws is increased to a sufficient number, the model fit will stay much more stable for further increases.

Why does my model converge with a low number of draws, but fail with a high number of draws?

The fact that the model does not converge with a high number of draws shows that there is a problem with the model. It is known that using a low number of draws can mean a model that is overspecified still converges and can give every impression of being identified (Chiou and Walker, 2007).

Can I at least use fewer draws if I use quasi-Monte Carlo draws?

In theory, yes, but again, more is better. Care is also required in deciding which type of draws to use. Halton draws are an excellent option for models with a low number of random parameters, but the colinearity issues with Halton draws means they should not be used with more than say 5 random components.

To keep estimation cost under control, can I use a low number of draws in my specification search and then reestimate the final model with a large number?

This is unfortunately a rather common practice, but it is misguided to think this is a good approach or that it solves the issues arising with using low numbers of draws. The fact that low numbers are used during the specification search means that the approximation to the integral is poor at that stage. This in turn means that the decisions that are leading to the final model specification may themselves be biased. While the final specification is then estimated robustly with a large number of draws, it may in fact be inferior to a specification that would have been obtained by using a high number throughout the specification search.

Does *Apollo* allow me to separate scale heterogeneity from preference heterogeneity?

The notion that it is possible to separate out scale heterogeneity from other heterogeneity is a myth, as discussed at length by Hess and Train (2017). Many models can allow for scale heterogeneity, but no model can separate it from preference heterogeneity, and there is no need to do so.

So can *Apollo* estimate the GMNL model?
The GMNL model is in fact not a new model or a more general model. It is simply a Mixed Logit model with a very particular set of constraints applied to it. It is not more general than Mixed Logit, which is the most general RUM model (cf. McFadden and Train, 2000)]. Given that *Apollo* allows full flexibility, users can of course specify the heterogeneity in a Mixed Logit model using the GMNL style constraints, but should be mindful when it comes to interpretation of the results given the above points.

So how about scale adjusted Latent Class (SALC)?
A SALC model is affected by the same issues as discussed by Hess and Train (2017) for GMNL. It is not possible to separate out scale heterogeneity from other heterogeneity. Users of *Apollo* can produce a SALC specification, which is simply a two layer Latent Class model, by using $S_1 \cdot S_2$ classes, allowing for S_1 sets of β parameters and S_2 sets of μ (scale) parameters, with an appropriate normalisation, and with the $S_1 \cdot S_2$ classes using all combinations of β and μscales model will be more general than a Latent Class model with S_1 classes with different β, but less general than a model with $S_1 \cdot S_2$ classes with different β.

12.5 Errors and failures during estimation

Why does *Apollo* complain that some function arguments are missing or incorrect?
There are different reasons for this, but the most likely cause is that the analyst has used the wrong order of arguments. For the predefined functions, the order of arguments passed to the function should be kept in the order specified for the function. For example, if a function is defined to take two inputs, namely `dependent` and `explanatory`, e.g. `model_prob(dependent,explanatory)`, and the user wants to use `choice` and `utility` as the inputs, then the function can be called as `model_prob(choice,utility)` but not as `model_prob(utility,choice)`. The latter change in order is only possible if the function is called explicitly as `model_prob(explanatory=utility,dependent=choice)`, which is the same as `model_prob(dependent=choice,explanatory=utility)`.

Why is my estimation failing with the message "Log-likelihood calculation fails at starting values"?
This happens when, at the starting values, the likelihood of the model is zero or or cannot be calculated for at least some people in the data. *Apollo* will report the IDs of these individuals. Three common reasons exist for this problem:

Are the starting values feasible/appropriate for the model?
The most common reason for the initial likelihood calculation to fail is a problem with the values used in `apollo_beta`. The starting values of some parameters may be invalid. For example, the molde may be dividing by a parameter with an initial value equal to zero. Also, different models have different requirements, for example the structural parameters in nested and Cross-nested Logit models should be different from zero; the α parameters in Cross-nested Logit models should be between zero and one; the γ and σ parameters in MDCEV should be greater than zero; the thresholds in Ordered Logit and Probit models

should be different and monotonically increasing; the variance of linear regressions should be positive; etc. Another potential cause (less common than the previous one) is that initial values are too poor, leading to an initial likelihood too close to zero, which in turn leads to an infinite log-likelihood. To avoid this, the user should look for better starting values, either by estimating a simpler model and using those estimates as starting values, or using the `apollo_searchStart` function.

Does the data contain many observations for each person and/or does the model use several components (i.e. hybrid choice)?

When multiplying together the probability of many individual observations at the person level (using `apollo_panelProd`), it is possible that the product becomes too close to zero for `R` to store it as a number. The same can happen when combining many individual model components using `apollo_combineModels`. The risk of this is greater in case of models with low probabilities, such as in the case of large choicesets or many observations per individual. A solution to this problem is to use set `workInLogs=TRUE` in `apollo_control`. This ensures that all calculations are made with the logarithms of probabilities, avoiding the issue of multiplying many small numbers. The use of this setting is however only recommended when necessary as it will slow down estimation.

Does the model use lognormal distributions for random coefficients?

The value of lognormally distributed coefficient is given by $\beta = exp\left(\mu_{\ln(\beta)} + \sigma_{\ln(\beta)}\xi\right)$, or $\beta = -exp\left(\mu_{\ln(-\beta)} + \sigma_{\ln(-\beta)}\xi\right)$ in the case of a negative lognormal distribution. The estimated parameters thus relate to the mean and standard deviation of the logarithm of β (or the logarithm of $-\beta$). A common mistake is to start $\mu_{\ln(\beta)}$ at zero, just as in the case of a normally distributed β. With the exponential, this will lead to a large starting value for β, which can result in numerical problems. In the case of lognormally distributed coefficients, it is thus advisable to use a large negative value for the starting value of the mean of the logarithm of the coefficient, e.g. starting $\mu_{\ln(\beta)}$ at something like -3 or lower, as this would imply starting the median of β close to zero.

Why do I get an error saying that one of my parameters does not influence the likelihood, even though I am using it in a utility function?

There may be several reasons for this. A common mistake is when writing utilities (or any other code statement) across multiple lines, the link between lines is missing and only the first one is considered. To split a statement across multiple lines, the incomplete lines should finish with an operator. For example:

```
U[["A"]] = b0 + b1*x1
+ b2*x2
```

will ignore the effect of `b2*x2`. Instead, it should be:

```
U[["A"]] = b0 + b1*x1 +
b2*x2
```

It could also be that the attribute associated with the parameter does not vary across the utility functions, or that the same constant is included in all utility functions. Much less common, it could be that the starting probabilities in your model are so small that due to rounding errors, they are equal to zero, and changes in parameter values also lead to small

probabilities not different from zero. This is more likely to happen in complex models with many observations per individual. In this case, we recommend (1) to begin by estimating a simple model (e.g. constants only) to obtain better starting values, and (2) to set the option `apollo_control$workInLogs=TRUE`. This last option will increase numerical precision at the expense of estimation speed.

Estimation of my model failed after a long time. Have I lost all the information?
If estimation was run using BFGS, *Apollo* will produce a *csv* file with the parameter values at each iteration in the working directory, using the name given to the model inside `apollo_control$modelName`.

Why does my estimation fail, saying the maximum number of iterations has been reached?
The default number of iterations for estimation is set to 200. This can be increased in `estimate_settings$maxIterations`. In general however, if a model has not converged after 200 iterations, this could be a sign of problems with the model. Inspecting the iterations file produced during estimation can help diagnose if there is a problem or if more iterations are required.

12.6 Model results

Why am I getting `Inf` or `NaN` for standard errors?
There are several main reasons why this could happen:

The model could have theoretical identification issues
To diagnose these issues is not easy, as requirements change depending on the particular structure of the model. For example, in random utility models, only difference in utility matters, so the constant of at least one alternative must be fixed to zero. Similarly, a normalisation is required for categorical variables. In Hybrid choice models, the variance of each structural equation (or the slope of one measurement equation per latent variable) should be fixed to one.

The model could be too complex for the data, leading to empirical identification issues
Many users fall into the trap of believing that choice models are easy tools and that the most complex model should always be used. Instead, analyst should always begin by estimating the simplest possible model and moving progressively towards more complex formulations. This will help troubleshooting any potential identification problems. In Mixed Logit for example, analyst should always start by introducing only a few random coefficients, leaving the rest as fixed, and progressively making the model more general.

Differences in scale between parameters can complicate the calculation of the standard errors
Calculating standard errors requires inverting the Hessian matrix at the estimated value of the parameters. Depending on the model, this Hessian is calculated using numerical derivatives, looking at small changes to either side of the estimates. Some of these could result in numerical failures. Similarly, inverting the Hessian itself can be challenging due to

numerical precision issues in this case. This can be diagnosed by looking at `model$hessian`. If it has very big and very small values, inverting it could be problematic. In these cases, using the `estimate_settings$scaling` option can help. This is an optional setting that can be given to the `apollo_estimate` function to scale parameters and therefore avoid numerical issues.

The calculation of numerical derivatives could lead to some zero probabilities

Especially with complex models, the calculation of the numerical derivatives can be affected by a small number of calculations leading to zero probabilities. Greater stability can in this case be obtained by using bootstrapping for estimating the standard errors (i.e. setting `estimate_settings$bootstrapSE=TRUE`), obviously at the cost of increased estimation time. Similarly, depending on the data used, setting `workInLogs=TRUE` in `apollo_control` could help.

Why is my estimate of the standard deviation negative?

This happens if a random coefficient is coded as `randCoeff[["beta"]]=mu+sigma*draws` where `draws` is a random variate that is symmetrical around zero.

So should I constrain the standard deviation to be positive?

There is no reason for doing so. The results will be the same if the random variate `draws` is symmetrical around zero. Imposing constraints will also make estimation harder. And of course, if a user wants to allow for correlation between individual coefficients, then the parameters multiplying the draws need to be able to be positive or negative.

Why are my structural/nesting parameters greater than one or smaller than zero in Nested or Cross-nested Logit?

Apollo does not constrain the structural parameters in Nested (NL) and Cross-nested (CNL) Logit models to be between zero and one. If, after estimation, the structural parameters are outside of this interval, this could be evidence of the nesting structure not being supported by the data. The userYouould then try a different nesting structure.

So should I constrain them to be between 0 and 1?

In general, while possible using the settings in `maxLik`, we do not recommend imposing constraints. If unconstrained estimation yields an estimate outside the bounds of the interval of acceptable values, then it is highly likely that the use of constraints will lead to an estimate that goes to one of the bounds of the interval that the parameter is constrained in. The model fit will be inferior too and the real problem will simply be masked by the constraints.

So how about constraints on other parameters, such as standard deviations or γ parameters in MDCEV?

It is common practice to use exponential transforms for parameters that are only allowed to be positive, e.g. using $\gamma = exp(\gamma_0)$, with γ_0 being estimated. We have found that this often slows down estimation and does not necessarily lead to the same solution as unconstrained estimation even if the latter finds an acceptable solution. The reason for the problem is that small changes to γ_0 will lead to large changes in γ, making estimation difficult, especially with numerical derivatives.

How do I calculate hit rates for my model in *Apollo*?

We made the decision not to include hit rates in *Apollo* outputs. They really offer a very distorted view of the results. Models give probabilities in prediction. If, for each task, an analyst just looks at what alternative has the highest probability, then they're ignoring the error term in the model. To put it succinctly, imagine you have a case with 2 alternatives, and we have 2 models. Model A gives a probability of 51% for the chosen alternative in 70% of cases in model 1, but a probability of only 10% in the remaining 30% of cases. Model B gives a probability of 49% for the chosen alternative in 70% of cases in model 1, but a probability of 90% in the remaining 30% of cases. Using a hit rate would give model A a figure of 70%, and model B a figure of 30%. But clearly model B is far superior. That's why outside marketing, choice modellers work with probabilities of correct prediction instead if they want a percent measure like this. And that would give 0.387 for model A but 0.613 for model B.

Appendix A

Apollo versions: timeline, changes and backwards compatibility

Version 0.0.6 (13 March 2019)

This is the first fully functioning release of *Apollo*.

Version 0.0.7 (8 May 2019)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Inputs changed for `apollo_choiceAnalysis`
Functions affected: `apollo_choiceAnalysis`
Detailed description: inputs changed so function can be called prior to `apollo_validateInputs`
Backwards compatibility of code: function call changed from version 0.0.7 onwards

Constraints for classical estimation
Functions affected: `apollo_estimate`
Detailed description: *Apollo* now allows the user to include a list called `constraints` in `estimate_settings` for use with BFGS for classical model estimation.
Backwards compatibility of code: no backwards compatibility issues for existing functions

Scaling of parameters during model estimation
Functions affected: `apollo_estimate`
Detailed description: scaling of model parameters can be used during estimation

Backwards compatibility of code: no backwards compatibility issues for existing functions

Validation output
Functions affected: `apollo_estimate`
Detailed description: *Apollo* no longer reports that all pre-estimation checks were passed for a model component and instead only reports if there are an issues.
Backwards compatibility of code: no backwards compatibility issues for existing functions

Bayesian estimation produces `model$estimate`
Functions affected: `apollo_estimate`, `apollo_prediction`, `apollo_llCalc`
Detailed description: until version 0.0.6, Bayesian estimation in *Apollo* did not produce a `model$estimate` output. We have retained the various existing outputs, but in addition, `model$estimate` is now produced, combining non-random parameters with individual specific posteriors for random parameters. This now allows the user to use `apollo_prediction` and `apollo_llCalc` on such outputs, where care is of course required in interpretation of outputs based on posterior means.
Backwards compatibility of code: no backwards compatibility issues for existing functions

Changes to *Apollo* examples:

Examples affected: `apollo_example_1.r` and `apollo_example_2.r`
Detailed description: Use of `apollo_choiceAnalysis` added
Backwards compatibility of examples: affected part only works from version 0.0.7 onwards

Examples affected: `apollo_example_12.r`
Detailed description: Scaling in estimation implemented
Backwards compatibility of examples: only works from version 0.0.7 onwards

Examples affected: `apollo_example_26.r`
Detailed description: HB prediction component added
Backwards compatibility of examples: affected part only works from version 0.0.7 onwards

Bug fixes:

apollo_speedTest
This function was unintentionally hidden from users in previous versions

Version 0.0.8 (9 September 2019)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Bootstrap estimation added
Functions affected: `apollo_bootstrap`, `apollo_estimate`
Detailed description: the user can now perform bootstrap estimation. This can also be called directly with `apollo_estimate` during estimation
Backwards compatibility of code: new function from version 0.0.8 onwards, new optional arguments for `apollo_estimate`, but function called in the same way

Allow user to use subset of rows for analysis of choices
Functions affected: `apollo_choiceAnalysis`
Detailed description: An additional `rows` argument can be entered into `choiceAnalysis_settings`.
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

Outputs changed for `apollo_choiceAnalysis`
Functions affected: `apollo_choiceAnalysis`
Detailed description: outputs changed so that t-test value is reported instead of p-value, and order of outputs is changed
Backwards compatibility of code: outputs changed from version 0.0.8 onwards, but function called in the same way

Allow user to change name and location of outside good in MDCEV and MDCNEV
Functions affected: `apollo_mdcev` and `apollo_mdcnev`
Detailed description: An additional `outside` argument can be entered into `mdcev_settings` and `mdcnev_settings` with the name of the outside good which can now differ from `outside`. It also no longer needs to be in first position in the list of alternatives.
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

No need to define superfluous γ for outside good in MDCEV and MDCNEV
Functions affected: `apollo_mdcev` and `apollo_mdcnev`
Detailed description: The user no longer needs to create a `gamma` term for the outside good.
Backwards compatibility of code: function called in the same way

Chosen unavailable alternatives have a likelihood of zero
Functions affected: `apollo_mnl, apollo_mdcev, apollo_mdcnev`
Detailed description: MNL, MDCEV and MDCNEV now return a likelihood equal to zero for chosen alternatives that are not available. This change is only relevant if `apollo_control$noValidation` is `TRUE`.
Backwards compatibility of code: new likelihood values for unavailable chosen alternatives on MNL, MDCEV and MDCNEV models from version 0.0.8 onwards

Allow user to specify number of outliers to report
Functions affected: `apollo_modelOutput, apollo_saveOutput`
Detailed description: In addition to specifying TRUE/FALSE for printOutliers, the user can provide the number of outliers to report (instead of the default of 20).
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

Ability to define estimation/validation subsets for `apollo_outOfSample`
Functions affected: `apollo_outOfSample`
Detailed description: the user can now provide a matrix or data.frame describing which observations are to be used in the estimation and validation subsets
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

Individual IDs and choice scenario numbers added in predictions
Functions affected: `apollo_prediction`
Detailed description: The output from `apollo_prediction` now includes the IDs and choice observation numbers as the first two columns.
Backwards compatibility of code: function called in the same way

Changes to *Apollo* examples:

Examples affected: `apollo_example_3.r`, `apollo_example_11.r`, `apollo_example_13.r`, `apollo_example_22.r`, `apollo_example_24.r`, `apollo_example_25.r`, `apollo_example_26.r`
Detailed description: `apollo_prediction` now includes the IDs and observation numbers as the first two columns, meaning some output is shifted.
Backwards compatibility of examples: use of outputs needs adjusting to reflect change in columns

Bug fixes:

apollo_combineResults
This function failed in earlier versions when using only a single model

apollo_firstRow
This function mistakenly replicated the first row for each person T_n times

apollo_estimate with scaling and HB
HB estimation failed in earlier versions for models without any random parameters

Version 0.0.9 (23 October 2019)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Additional diagnostic message for HB estimation
Functions affected: `apollo_estimate`
Detailed description: the `RSGHB` package used for Bayesian estimation left censors likelihood values at the individual level to avoid numerical issues. This has the undesired side effect of mis-specified models still running, and a warning message is now displayed when censoring has been used
Backwards compatibility of code: function called in the same way

Pre-estimation tests to ensure all parameters affect likelihood function
Functions affected: `apollo_estimate`
Detailed description: unless `apollo_control$noDiagnostics==TRUE`, a pre-estimation check is used to ensure that there are no parameters in `apollo_beta` for which changes do not lead to changes in the model likelihood
Backwards compatibility of code: function called in the same way

Changes to *Apollo* examples:

None

Bug fixes:

apollo_deltaMethod
This function had a small error in the calculation for standard errors for logistic transforms

apollo_estimate with HB and only non-random parameters
HB estimation failed in earlier versions for models without any random parameters if using scaling

Version 0.1.0 (16 March 2020)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

`R` version requirement changed to 3.6.0
Functions affected: all of *Apollo*
Detailed description: users running *Apollo* version 0.1.0 require at a minimum `R` version 3.6.0
Backwards compatibility of code: no changes to actual functions

Improved error reporting and printing
Functions affected: majority of functions
Detailed description: warnings are now generally displayed at the point they apply rather than after estimation. In addition, numerous new checks have been implemented for many of the functions.
Backwards compatibility of code: no backwards compatibility issues

Referring to `database` inside `apollo_probabilties` no longer allowed
Functions affected: all functions that call `apollo_probabilties`
Detailed description: the user is not allowed to refer to `database` by name inside `apollo_probabilties`. There is no reason for doing so as the `database` is attached and all elements therein can be referred to directly
Backwards compatibility of code: this should not affect any users. The only requirement is to now use the `get` function when attempting to retrieve an object whose name is put together via `paste0`

Can define names for individual model components
Functions affected: all existing functions for models, i.e. `apollo_mnl` etc
Detailed description: the user can now include an optional additional argument `componentName` in the settings for individual models. This is then used in the reporting of outputs as well as in any error messages
Backwards compatibility of code: no backwards compatibility issues

Tests for zero probabilities at starting values for individual model components
Functions affected: all existing functions for models, i.e. `apollo_mnl` etc
Detailed description: unless `apollo_control$noValidation==TRUE`, *Apollo* now checks the likelihood of individual model components in addition to the overall model and prints a warning if probabilities are zero for some individuals while estimation is not started if probabilities are zero for all. This is relevant for latent class models, where it is permissable

to have zero probabilities for some individuals in some classes, but where initial zero probabilities for all individuals in a class are likely to highlight problems.
Backwards compatibility of code: no backwards compatibility issues

Ability to sort results by date
Functions affected: `apollo_combineResults`
Detailed description: an additional option `sortByDate` has been included. When set to `TRUE`, the models in the summary file will be sorted by the date when the model was estimated (default set to TRUE)
Backwards compatibility of code: no backwards compatibility issues

Improved memory usage with multi-core estimation
Functions affected: `apollo_estimate`
Detailed description: Memory requirements for multi-core estimation have been reduced substantially compared to previous versions
Backwards compatibility of code: no backwards compatibility issues

Constraints in HB estimation can now use names of parameters
Functions affected: `apollo_estimate` with `apollo_control$HB==TRUE`
Detailed description: the user can now use names of parameters when creating constraints for HB estimation rather than relying on the numeric coding from `RSGHB`
Backwards compatibility of code: no backwards compatibility issues for existing functions as old format for input still permitted

Smallest absolute eigenvalue of Hessian reported
Functions affected: `apollo_estimate`, `apollo_modelOutput`, `apollo_saveOutput`
Detailed description: *Apollo* reports the eigenvalue of the Hessian that is closest to zero. Small values can indicate convergence issues. A special warning message is displayed if some of the eigenvalues are positive.
Backwards compatibility of code: no backwards compatibility issues

Check whether class allocation probabilities for latent class sum to 1
Functions affected: `apollo_lc`
Detailed description: New check to ensure that class alloaction probabilities for latent class sum to 1
Backwards compatibility of code: no backwards compatibility issues

Calculation of $LL\,(0)$
Functions affected: `apollo_modelOutput` and `apollo_saveOutput`
Detailed description: The calculation of the log-likelihood at zero has been improved for some models, as has the reporting of it. Where this measure does not apply, *Apollo* now reports "Not applicable" instead of "NA"

Backwards compatibility of code: no backwards compatibility issues

Ordered probit added
Functions affected: `apollo_op`
Detailed description: the user can now use ordered probit models via the function `apollo_op`
Backwards compatibility of code: no backwards compatibility issues

Out of sample testing can add to existing runs
Functions affected: `apollo_outOfSample`
Detailed description: `apollo_outOfSample` checks whether output files already exists and adds to those if the number of runs requested is larger than what is already stored in these files
Backwards compatibility of code: no backwards compatibility issues

Prevent use of some functions for HB models
Functions affected: `apollo_panelProd`, `apollo_avgInterdraws`, `apollo_avgIntraDraws`, `apollo_deltaMethod`, `apollo_conditionals`, `apollo_unconditionals`, `apollo_lcConditionals`, `apollo_lcUnconditionals`
Detailed description: Many of the *Apollo* functions are for classical estimation only, and their use previously led to failures. Their use is now prevented when `apollo_control$HB==TRUE`.
Backwards compatibility of code: the call to these functions was previously ignored and will now lead to a failure in any files they are still included in. The appropriate lines need commenting out.

Inputs changed for `apollo_prediction` and added ability to calculate standard errors
Functions affected: `apollo_prediction`
Detailed description: the `apollo_prediction` function now takes `prediction_settings` as an input, where this is the new location for including `modelComponent`. In addition, an optional setting called `runs` has been included that computes standard errors across multiple prediction runs based on different draws from the estimates and covariance matrix for the model parameters
Backwards compatibility of code: no backwards compatibility issues for existing functions as old format for input still permitted

Output files no longer overwritten
Functions affected: `apollo_saveOutputs`
Detailed description: the `apollo_saveOutput` function now checks whether output files for the model already exists and changes their names (by including `OLD` in the name) rather than overwritting them

Backwards compatibility of code: no backwards compatibility issues

Output file for starting value search simplified
Functions affected: `apollo_searchStart`
Detailed description: The output file for `apollo_searchStart` was simplified from v0.0.9 to v0.1.0. Now it only records starting candidate values, and their loglikelihoods throughout the stages, but not their values at each stage, as it used to.
Backwards compatibility of code: no backwards compatibility issues

Subsetting of data when some variables are factors
Functions affected: all functions making use of the `database`
Detailed description: If a dataset contains `factors`, the use of subsetting of the data in `R` would still retain levels that no longer apply. This is a feature of `R` which can have unintended consequences and we thus eliminate any missing levels.
Backwards compatibility of code: no backwards compatibility issues

Changes to *Apollo* examples:

Examples affected: `apollo_example_3`
Detailed description: include calculation of standard errors for forecasts
Backwards compatibility of examples: affected part only works from version 0.1.0 onwards

Examples affected: `apollo_example_3` and `apollo_example_26`
Detailed description: use cost increase by 1% instead of 10% for calculating elasticities
Backwards compatibility of examples: no backwards compatibility issues

Examples affected: `apollo_example_5`
Detailed description: `apollo_example_5` no longer reads in the results from `apollo_example_4` as starting values as this woudl have implied an inconsistent initial nesting structure
Backwards compatibility of examples: no backwards compatibility issues

Examples affected: `apollo_example_18`, `apollo_example_19`, `apollo_example_20`, `apollo_example_22`, `apollo_example_27` and `apollo_example_28`
Detailed description: `componentName` added to class specific models
Backwards compatibility of examples: setting ignored in earlier versions

Examples affected: `apollo_example_22`
Detailed description: in the use of `apollo_prediction`, `modelComponent` is now included in `prediction_settings` rather than as a direct input
Backwards compatibility of examples: affected part only works from version 0.1.0 onwards

Examples affected: `apollo_example_28`
Detailed description: included `apollo_probabilities` as an argument for `apollo_unconditionals`
Backwards compatibility of examples: this is a bug fix that ensures the example now runs correctly

Bug fixes:

apollo_bootstrap
Adding additional bootstrap runs with the same seed now ensures that new draws are used rather than adding the same results that already exist.

apollo_combineResults
When `combineResults_settings$modelNames` was not provided, other settings were ignored

apollo_estimate
Earlier versions still performed model validation even if `apollo_control$noValidation==TRUE`

apollo_lc
The pre-estimation diagnostic tests would fail in case the starting values were the same for multiple classes in a latent class model

apollo_lcConditionals
Conditionals from latent class used to report one row per observation, while they should have reported one row per individual

apollo_mdcev
When the `rows` option was used, any pre-estimation checks still included all rows in the data. In addition, some failures could occur in estimation.

apollo_prediction
Predictions from latent class were missing the names of the alternatives in the output

apollo_searchStart
Multiple bugs have been corrected. In previous versions, the function would ignore converged candidates and would pick the wrong candidate as "best"

Version 0.1.1 (15 September 2020)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Additional checks on draws
Functions affected: all functions with mixing
Detailed description: `Apollo` now ensures that intra-individual draws are only used in the presence of multiple observations per individual.
Backwards compatibility of code: no backwards compatibility issues

Additional checks for exploded logit
Functions affected: `apollo_el`
Detailed description: The `apollo_el` now ensures that no alternative is chosen more than once across stages and ensures that any obsolete stages have the choice variable coded as -1.
Backwards compatibility of code: no backwards compatibility issues

Parameter specific constraints in HB estimation can now use names of parameters
Functions affected: `apollo_estimate` with `apollo_control$HB==TRUE`
Detailed description: the user can now use names of parameters when creating constraints for individual parameters in HB estimation rather than relying on the numeric coding from `RSGHB`
Backwards compatibility of code: no backwards compatibility issues for existing functions as old format for input still permitted

Additional flexibility for thresholds in ordered logit and ordered probit
Functions affected: `apollo_ol` and `apollo_op`
Detailed description: The thresholds for ordered logit and ordered logit are now given as a list, with one entry per threshold, thus allowing the user to have thresholds varying across observations/individuals, with possible additional random heterogeneity.
Backwards compatibility of code: no backwards compatibility issues as the function can still be called with the thresholds as a simple vector

Check for use of reserved names
Functions affected: `apollo_validateInputs`
Detailed description: For internal consistency, some core names, such as `alternatives`, `avail` and `sigma` are now protected and cannot be used in the data, `apollo_beta`, `apollo_lcPars` and `apollo_randCoeff`.

Backwards compatibility of code: backwards compatibility for some model files, but can be easily addressed by changing names

Changes to *Apollo* examples:

Examples affected: `apollo_example_24.r`
Detailed description: Uses new specification of thresholds as a list for `apollo_ol`
Backwards compatibility of examples: old format still accepted

Examples affected: `apollo_example_11.r`, `apollo_example_12.r` and `apollo_example_17.r`
Detailed description: `sigma` replaced by `sig` to avoid protected names issue
Backwards compatibility of examples: change required for v 0.1.1 onwards

Bug fixes:

apollo_conditionals with apollo_control$workInLogs==TRUE
Fixed a bug where Apollo allowed the use of `apollo_conditionals` despite having `apollo_control$workInLogs==TRUE`. This led to a failure as the probabilities are not available at the draw level when working in logs. This functionality is consequently not available now.

apollo_estimate with HB and constraints
Fixed a bug that led to a failure if a constraint was imposed on the final element of `apollo_beta`.

apollo_estimate with weights
Fixed a bug where in older versions, the log-likelihood at zero (LL0) was calculated without the weights.

apollo_estimate with apollo_control$workInLogs==TRUE
Fixed a bug that led to a failure if the probabilities were zero with some draws. Also fixed a bug when using latent class models with `apollo_control$workInLogs==TRUE` which led to a bias in the calculation of probabilities.

apollo_ol and **apollo_op**
Fixed a bug that led to a failure when using a response variable with fewer or more than 5 levels

apollo_sharesTest
Fixed a bug that led to a failure when using `apollo_sharesTest` with `sharesTest_settings$subsamples==NA` and when using `rows` inside the model

apollo_inputs$apollo_control
Fixed a bug to ensure that when `apollo_control` is used by a function, it is fetched from `apollo_inputs$apollo_control`, making sure that when debugging, the correct (i.e. processed) `apollo_control` is used.

apollo_mdcev
Fixed a bug that led to an erroneous error message and halt to estimation with MDCEV models when using the `mdcev_settings$minConsumption` setting.

LL(0) with some models using random coefficients
Fixed a bug that led to incorrect LL(0) for some models using random coefficients, notably for latent class models with continuous random coefficients in the class allocation model.

Version 0.2.0 (19 October 2020)

Changes to *Apollo* code:

General
Major improvements to efficiency, stability and reporting of user errors.

Ben-Akiva & Swait test added
Functions affected: `apollo_basTest`
Detailed description: the user can now perform the Ben-Akiva & Swait test
Backwards compatibility of code: new function from version 0.2.0 onwards

New outputs for bootstrap, and changes to storage of repetitions
Functions affected: `apollo_bootstrap`
Detailed description: `apollo_bootstrap` now returns a list which contains two components, namely a list of estimated parameters and likelihood function in each repetition, and the covariance matrix. In addition, when called an additional time, the function will always add new repetitions to old files, if they exist (it won't "complete" the requested number).
Backwards compatibility of code: format of output has changed for post-processingno backwards compatibility issues

`apollo_inputs` is no longer automatically updated when calling post-processing functions
Functions affected: `apollo_conditionals`, `apollo_lcConditionals`, `apollo_lcUnconditionals`, `apollo_llCalc`, `apollo_prediction`, `apollo_unconditionals`
Detailed description: In previous versions, `apollo_inputs` would be updated when calling post-processing functions. Now *Apollo* instead checks if any inputs have changed and alerts the user to revalidate `apollo_inputs`.
Backwards compatibility of code: examples where changes to the data were made post estimation require a call to `apollo_validateInputs`

Model description added to output of `apollo_combineResults`
Functions affected: `apollo_combineResults`
Detailed description: The `csv` file produced by `apollo_combineResults` now includes the model description defined in `apollo_control` by the user
Backwards compatibility of code: Ouput changed, but no backwards compatibility issues

Estimation time split into separate components in output
Functions affected: `apollo_modelOuptut`, whether using `apollo_estimate` or `apollo_estimateHB`
Detailed description: The time taken by `apollo_modelOuptut` and `apollo_estimate` is now split into pre-processing, estimation, and post-processing time

Backwards compatibility of code: Ouput changed, but no backwards compatibility issues

`apollo_fitsTest` uses log-likelihood instead of probabilities
Functions affected: `apollo_fitsTest`
Detailed description: In previous versions of *Apollo*, the function `apollo_fitsTest` used the probability of correct prediction, which was not appropriate for non-discrete choice models and also not for multi-component models. The function now uses the log-likelihood instead. The function also only does this for the overall model, i.e. no longer allowing the user to look separately at subcomponents of a joint model.
Backwards compatibility of code: Ouput changed.

ID included in outputs of `apollo_lcConditionals`
Functions affected: `apollo_lcConditionals`
Detailed description: The individual-specific value for `apollo_control$indivID` is now included as the first column of the output of `apollo_lcConditionals`
Backwards compatibility of code: Ouput changed.

New function for EM algorithm for latent class models
Functions affected: `apollo_lcEM`
Detailed description: Function for automatically using the EM algorithm for latent class models rather than requiring the user to code this for a specific model
Backwards compatibility of code: New function. Users interested in how to code this manually are referred to earlier manuals.

New function for EM algorithm for mixed logit
Functions affected: `apollo_mixEM`
Detailed description: Function for automatically using the EM algorithm for mixed logit models with a full covariance matrix and all parameters being random
Backwards compatibility of code: New function. Users interested in how to code this manually are referred to earlier manuals.

New input options for likelihood ratio test, and enhanced output
Functions affected: `apollo_lrTest`
Detailed description: *Apollo* now allows the user to apply a likelihood ratio test using outputs from two models that are stored in memory (rather than one model needing to have been saved to disk). The restricted model also no longer needs to be given first, as *Apollo* will determine the order of the two models. The `apollo_llTest` function now also prints the likelihoods of each model as well as their difference.
Backwards compatibility of code: No backwards compatibility issues.

User can specify number of replications to use in MDCEV forecasting
Functions affected: `apollo_mdcev` and `apollo_mdcev`

Detailed description: The user can specify the number of replications using the optional argument `nRep`
Backwards compatibility of code: No backwards compatibility issues.

New optional settings for model output
Functions affected: `apollo_modelOutput` and `apollo_saveOutput`
Detailed description: The setting `printPVal` can now take three values: 0 (no p-vals), 1 (one-sided), or 2 (two-sided). The setting `printDiagnostics` was split in two settings: `printDataReport` (logical) to print the summary of the dependant variable, and `printModelStructure` (logical) to print data on nesting structures. A new setting `printFunctions` is included to print a copy of `apollo_probabilities`, `apollo_control`, scaling (for estimation and Hessian), Hessian routines attempted, `apollo_lcPars` and `apollo_randCoeff`.
Backwards compatibility of code: No backwards compatibility issues as old input still accepted and new inputs are optional.

Changes to storage of repetitions for out of sample prediction
Functions affected: `apollo_outOfSample`
Detailed description: When called an additional time `apollo_outOfSample` will always add new repetitions to old files, if they exist (it won't "complete" the requested number).
Backwards compatibility of code: format of output has changed for post-processingno backwards compatibility issues

Improved output for predictions
Functions affected: `apollo_prediction`
Detailed description: A summary of the predictions are printed in a nice way, with special handling of MDCEV models. Automatic creation of confidence intervals when using repeated sampling.
Backwards compatibility of code: Output changed

Data no longer gets sorted by ID
Functions affected: `apollo_validateData`
Detailed description: *Apollo* no longer sorts the data by `apollo_control$ID`. Sorting is not required by the user either, but all observations from same individual need to be contiguous in the data.
Backwards compatibility of code: *Apollo* will now ask the user to group together data for the same individual. Some older models may thus require adjustment to the data.

Allow working in logs for cross-sectional data
Functions affected: `apollo_validateInputs`
Detailed description: Previous versions of *Apollo* allowed the use of `apollo_control$workInLogs` only with panel data.

Backwards compatibility of code: No backwards compatibility issues.

Changes to reserved names
Functions affected: `apollo_validateInputs`
Detailed description: The restrictions imposed in version `0.1.1` for reserved names have been relaxed, and replaced by a check that no names from `apollo_beta`, `apollo_randCoeff`, `database`, `apollo_draws` or `apollo_lcPars` are redefined elsewhere.
Backwards compatibility of code: Possible changes needed to old model files.

Ability to use a subset of model components in `apollo_combineModels`
Functions affected: `apollo_combineModels`
Detailed description: The user can now specify a subset of model components to multiply.
Backwards compatibility of code: New optional argument.

Changes to *Apollo* examples:

Examples affected: `apollo_example_2.r`
Detailed description: Uses bootstrap standard errors.
Backwards compatibility of examples: This functionality was not supported in previous versions.

Examples affected: `apollo_example_3.r` and `apollo_example_22.r`
Detailed description: Function `apollo_validateInputs` is called before every `apollo_prediction` to update `database` inside `apollo_inputs`.
Backwards compatibility of examples: If `apollo_validateInputs` is not called before prediction (as in previous versions), then the prediction will not change, even if the version of `database` in the Global Environment changes.

Examples affected: `apollo_example_6.r`
Detailed description: Added Ben-Akiva and Swait test, i.e. a call to function `apollo_basTest`.
Backwards compatibility of examples: `apollo_basTest` was not implemented in previous version, so it will fail.

Examples affected: `apollo_example_11.r` and `apollo_example_12.r`
Detailed description: Name of scale parameter is now `sigma`.
Backwards compatibility of examples: Previous versions did not allow user defined variables to be called `sigma`, so the example will fail in previous versions.

Examples affected: `apollo_example_11.r`
Detailed description: Removed line `colMeans(predictions_base)`. This is no longer necessary as `apollo_prediction` now prints an aggregate summary of the predicted values.

Backwards compatibility of examples: The aggregate summary of the prediction will not be shown in previous versions.

Examples affected: `apollo_example_14.r`
Detailed description: This example now uses 4-core multithreading for estimation.
Backwards compatibility of examples: No issues.

Examples affected: `apollo_example_18.r`, `apollo_example_20.r`, `apollo_example_27.r` and `apollo_example_28.r`
Detailed description: When looping over latent classes inside `apollo_probabilities`, the total number of classes is not hard-coded any more. Instead, the number of classes is obtained as the length of `pi_values` (part of the return of `apollo_lcPars()`)
Backwards compatibility of examples: No issues.

Examples affected: `apollo_example_20.r`
Detailed description: Dropping first column of conditionals as this now includes the ID.
Backwards compatibility of examples: Change needed from v.0.2.0. onwards.

Examples affected: `apollo_example_27.r` and `apollo_example_28.r`
Detailed description: EM estimation is now achieved using `apollo_lcEM`.
Backwards compatibility of examples: This example will not run in Apollo v0.1.1 or earlier, as function `apollo_lcEM` does not exist in those versions.

Examples affected: `apollo_example_29.r`
Detailed description: EM estimation is now achieved using `apollo_mixEM`.
Backwards compatibility of examples: This example will not run in Apollo v0.1.1 or earlier, as function `apollo_mixEM` does not exist in those versions.

Bug fixes:

apollo_choiceAnalysis
Fixed a bug where `apollo_choiceAnalysis` failed when an entry in `choiceAnalysis_settings$explanators` was empty

Seed for apollo_makeDraws
Fixed a bug where this function was looking for `apollo_control$seed_draws` instead of `apollo_control$seed`

Version 0.2.1 (28 October 2020)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Changes to *Apollo* examples:

None

Bug fixes:

apollo_lcEM
Fixed bugs where `apollo_lcEM` could not be used with multiple cores, or when not additionally computing the covariance matrix

Version 0.2.2 (5 December 2020)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

New options for shares test added added
Functions affected: `apollo_sharesTest`
Detailed description: the user can now create pseudo-alternatives, grouping together existing alternatives
Backwards compatibility of code: new optional argument, no backwards compatibility issues

User can omit `avail` when all alternatives are available
Functions affected: `apollo_mnl`, `apollo_nl`, `apollo_cnl`, `apollo_dft`, `apollo_el`, `apollo_mdcev`, `apollo_mdcnev`
Detailed description: the user can now omit the `avail` argument if all alternatives are available
Backwards compatibility of code: optional, no backwards compatibility issues

Improved counting of modelled outcomes
Functions affected: `apollo_modelOutput` and `apollo_saveOutput`
Detailed description: the output now counts modelled outcomes as opposed to rows in the data, which leads to different statistics for multi-component models

Backwards compatibility of code: outputs changed, with resulting changes in goodness-of-fit statistics

Optionally drop fixed parameters from output
Functions affected: `apollo_modelOutput` and `apollo_saveOutput`
Detailed description: the user can now optionally exclude fixed parameters from the output
Backwards compatibility of code: outputs changed

Optionally return only the model component when using `apollo_combineModels`
Functions affected: `apollo_combineModels`
Detailed description: the user can now optionally exclude subcomponents when calling `apollo_combineModels`
Backwards compatibility of code: new optional argument, no backwards compatibility issues

Individual IDs and observation numbers included in output of `apollo_bootstrap`
Functions affected: `apollo_bootstrap`
Detailed description: `apollo_bootstrap` now writes the `indivID` and `apollo_seq` to its file output .
Backwards compatibility of code: Changes to output.

Ability to include constraints in classical estimation
Functions affected: `apollo_estimate`
Detailed description: `apollo_estimate` now has an additional optional argument for constraints.
Backwards compatibility of code: New capability.

Changes to *Apollo* examples:

None

Bug fixes:

apollo_lcEM
Fixed a bug where `apollo_lcEM` was still running when generic parameters were included, and a separate bug with the calculation of the log-likelihood with multiple cores

apollo_estimate with bootstrap
Fixed a bug where `apollo_estimate` could not run post-estimation bootstrapping when using multiple cores

apollo_basTest
Fixed a bug to prevent `apollo_basTest` from running when the better fitting model has

fewer parameters

Version 0.2.3 (20 January 2021)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Changes to *Apollo* examples:

None

Bug fixes:

apollo_estimate
Fixed a bug where `apollo_estimate` could fail due to missing arguments, where this is now checked before estimation starts

Version 0.2.4 (25 February 2021)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Monte Carlo replications for MDCEV can now be set by user
Functions affected: `apollo_prediction` with `apollo_mdcev, apollo_mdcnev`
Detailed description: the user can now set the number of Monte Carlo replications to use when predicting from MDCEV models
Backwards compatibility of code: new optional argument, no backwards compatibility issues

MDCEV predictions now include expenditure
Functions affected: `apollo_prediction` with `apollo_mdcev, apollo_mdcnev`
Detailed description: the output from `apollo_prediction` for MDCEV models now includes the expenditure
Backwards compatibility of code: new output, no backwards compatibility issues

Predictions now returned as `data.frame` instead of matrix
Functions affected: `apollo_prediction`

Detailed description: the output from `apollo_prediction` is now a `data.frame` instead of a matrix, which helps avoid issues with non-numeric IDs
Backwards compatibility of code: new output format, some possible backwards compatibility issues with examples

`apollo_sharesTest` now returns output
Functions affected: `apollo_sharesTest`
Detailed description: the output from `apollo_prediction` is now not just printed to screen, but also returned
Backwards compatibility of code: new output format, no backwards compatibility issues

Changes to *Apollo* examples:

None

Bug fixes:

apollo_dft
Fixed a bug where `apollo_dft` could fail in mixture models

`apollo_prediction` with `apollo_mdcev`, `apollo_mdcnev`
Fixed a bug where `apollo_predictions` could produce errors when asking for confidence intervals with MDCEV models

`apollo_cnl`
Fixed a bug where `apollo_cnl` would not check for allocation parameters to sum to 1

Version 0.2.5 (31 July 2021)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

User can specify number of replications to use in predictions from MDCEV
Functions affected: `apollo_prediction` with `apollo_mdcev` and `apollo_mdcnev`
Detailed description: The Pinjari & Bhat forecasting algorithm uses replications, and the user can now provide this as an additional settings named `rep` in `prediction_settings`.
Backwards compatibility of code: new capability, no backwards compatibility issues

More detailed outputs for MDCEV predictions

Functions affected: `apollo_prediction` with `apollo_mdcev` and `apollo_mdcnev`
Detailed description: The prediction from MDCEV models now returns the mean and standard deviations of continuous demand (`cont_mean`, `cont_sd`), discrete demand (`disc_mean`, `disc_sd`), and expenditure (`expe_mean`, `exp_sd`).
Backwards compatibility of code: new output format, no backwards compatibility issues

Predictions with multiple component components
Functions affected: `apollo_prediction` with `apollo_combineModels`
Detailed description: `apollo_prediction` with multiple model components now returns a list of data.frame, one for each model component. Model components without prediction capabilities are omitted from the return. If runs>1, then a data.frame with the point estimates are given AND a 3-dim array with the repetitions as well (the array does not have the IDs)
Backwards compatibility of code: new output format, some ancillary code may need to be updated

Output directory can be set by the user
Functions affected: all functions reading and writing to files
Detailed description: the user can specify an optional argument `outputDirectory` in `apollo_control` which will be used for outputs.
Backwards compatibility of code: new capability, no backwards compatibility issues

Output returned by `apollo_sharesTest`
Functions affected: `apollo_sharesTest`
Detailed description: In addition to screen output, `apollo_sharesTest` also invisibly returns the output so it can be saved into a data.frame
Backwards compatibility of code: new capability, no backwards compatibility issues

Changes to *Apollo* examples:

All examples changed to use output directory.

Bug fixes:

apollo_ol and apollo_op
Fixed a bug that led to failures when users provided their own coding of levels

`apollo_cnl`
Fixed a bug that failed to correctly diagnose when allocation parameters did not sum to 1 at the starting values

Appendix B

Data dictionaries

Tables B.1 to B.4 present data dictionaries for the four datasets made available with *Apollo*.

Table B.1: Data dictionary for `apollo_modeChoiceData.csv`

Individuals	500
Observations	8,000

Variable	**Description**	**Values**
ID	Unique individual ID	1 to 500
RP	RP data identifier	1 for RP, 0 for SP
SP	SP data identifier	1 for SP, 0 for RP
RP_journey	Index for RP observations	1 to 2, NA for SP
SP_task	Index for SP observations	1 to 14, NA for RP
av_car	availability for alternative 1 (car)	1 for available, 0 for unavailable
av_bus	availability for alternative 2 (bus)	1 for available, 0 for unavailable
av_air	availability for alternative 3 (air)	1 for available, 0 for unavailable
av_rail	availability for alternative 4 (rail)	1 for available, 0 for unavailable
time_car	travel time (mins) for alternative 1 (car)	Min: 250, mean: 311.79, max: 390 (0 if not available)
cost_car	travel cost (£) for alternative 1 (car)	Min: 30, mean: 39.99, max: 50 (0 if not available)
time_bus	travel time (mins) for alternative 2 (bus)	Min: 300, mean: 370.29, max: 420 (0 if not available)
cost_bus	travel cost (£) for alternative 2 (bus)	Min: 15, mean: 25.02, max: 35 (0 if not available)
access_bus	access time (mins) for alternative 2 (bus)	Min: 5, mean: 15.02, max: 25 (0 if not available)
time_air	travel time (mins) for alternative 3 (air)	Min: 50, mean: 70.07, max: 90 (0 if not available)
cost_air	travel cost (£) for alternative 3 (air)	Min: 50, mean: 79.94, max: 110 (0 if not available)
access_air	access time (mins) for alternative 3 (air)	Min: 35, mean: 45.02, max: 55 (0 if not available)
service_air	service quality for alternative 3 (air)	1 for no-frills, 2 for wifi, 3 for food (0 if not used, RP data)
time_rail	travel time (mins) for alternative 4 (rail)	Min: 120, mean: 142.93, max: 170 (0 if not available)
cost_rail	travel cost (£) for alternative 4 (rail)	Min: 35, mean: 55.03, max: 75 (0 if not available)
access_rail	access time (mins) for alternative 4 (rail)	Min: 5, mean: 14.96, max: 25 (0 if not available)
service_rail	service quality for alternative 4 (rail)	1 for no-frills, 2 for wifi, 3 for food (0 if not used, RP data)
female	dummy variable for female individuals	1 for female, 0 otherwise
business	dummy variable for business trips	1 for business trips, 0 otherwise
income	income variable (£ per annum)	Min: 15,490, mean: 44,748.27, max: 74,891
choice	choice variable	1 for car, 2 for bus, 3 for air, 4 for rail

Table B.2: Data dictionary for `apollo_swissRouteChoiceData.csv`

Individuals	388
Observations	3,492

Variable	**Description**	**Values**
ID	Unique individual ID	2,439 to 84,525
choice	choice variable	1 for alternative 1, 2 for alternative 2
tt1	travel time (mins) for alternative 1	Min: 2, mean: 52.59, max: 389
tc1	travel cost (CHF) for alternative 1	Min: 1, mean: 19.67, max: 206
hw1	headway (mins) for alternative 1	Min: 15, mean: 32.48, max: 60
ch1	interchanges for alternative 1	Min: 0, mean: 0.94, max: 2
tt2	travel time (mins) for alternative 2	Min: 2, mean: 52.47, max: 385
tc2	travel cost (CHF) for alternative 2	Min: 1, mean: 19.69, max: 268
hw2	headway (mins) for alternative 2	Min: 15, mean: 32.38, max: 60
ch2	interchanges for alternative 2	Min: 0, mean: 0.95, max: 2
hh_inc_abs	household income (CHF per annum)	Min: 10,000, mean: 76,507.73, max: 167,500
car_availability	car availability	1 for yes, 0 otherwise
commute	dummy variable for commute trips	1 for commute trips, 0 otherwise
shopping	dummy variable for shopping trips	1 for shopping trips, 0 otherwise
business	dummy variable for business trips	1 for business trips, 0 otherwise
leisure	dummy variable for leisure trips	1 for leisure trips, 0 otherwise

Table B.3: Data dictionary for `apollo_drugChoiceData.csv`

Individuals	1,000
Observations	10,000

Variable	Description	Values
ID	Unique respondent ID	1 to 1,000
task	Index for SP choice tasks	1 to 10
best	first ranked alternative	1 to 4
second_pref	second ranked alternative	1 to 4
third_pref	third ranked alternative	1 to 4
worst	worst ranked alternative	1 to 4
brand_1	brand for first alternative	Artemis; Novum
country_1	country for first alternative	Switzerland; Denmark; USA
char_1	characteristics for first alternative	standard; fast acting; double strength
side_effects_1	rate of side effects for first alternative (out of 100,000)	Min: 1, mean: 37, max: 100
price_1	price (£) for first alternative	Min: 2.25, mean: 3.15, max: 4.5
brand_2	brand for second alternative	Artemis; Novum
country_2	country for second alternative	Switzerland; Denmark; USA
char_2	characteristics for second alternative	standard; fast acting; double strength
side_effects_2	rate of side effects for second alternative (out of 100,000)	Min: 1, mean: 37, max: 100
price_2	price (£) for second alternative	Min: 2.25, mean: 3.15, max: 4.5
brand_3	brand for third alternative	BestValue; Supermarket; PainAway
country_3	country for third alternative	USA; India; Russia; Brazil
char_3	characteristics for third alternative	standard; fast acting
side_effects_3	rate of side effects for third alternative (out of 100,000)	Min: 10, mean: 370, max: 1,000
price_3	price (£) for third alternative	Min: 0.75, mean: 1.75, max: 2.5
brand_4	brand for fourth alternative	BestValue; Supermarket; PainAway
country_4	country for fourth alternative	USA; India; Russia; Brazil
char_4	characteristics for fourth alternative	standard; fast acting
side_effects_4	rate of side effects for fourth alternative (out of 100,000)	Min: 10, mean: 370, max: 1,000
price_4	price (£) for fourth alternative	Min: 0.75, mean: 1.75, max: 2.5
regular_user	dummy variable for regular users	1 for regular users, 0 otherwise
university_educated	dummy variable for university educated	1 for university educated, 0 otherwise
over_50	dummy variable for age over 50 years	1 for age over 50 years, 0 otherwise
attitude_quality	Answer to *"I am concerned about the quality of drugs developed by unknown companies"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)
attitude_ingredients	Answer to *"I believe that ingredients are the same no matter what the brand"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)
attitude_patent	Answer to *"The original patent holders have valuable experience with their medecines"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)
attitude_dominance	Answer to *"I believe the dominance of big pharmaceutical companies is unhelpful"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)

Table B.4: Data dictionary for `apollo_timeUseData.csv`

Individuals	447
Observations	2,826

Variable	**Description**	**Values**
indivID	Unique respondent ID	19209 to 9959342
day	Index of the day for the individual (day 1 excluded from data)	2 to 14
date	Date in format yyyymmdd	20161014 to 20170308
budget	Total amount of time registered during the day (in minutes)	1440 to 1440
t_a01	time spent dropping-off or picking up other people (in minutes)	0 to 1153
t_a02	time spent working (in minutes)	0 to 1425
t_a03	time spent on educational activities (in minutes)	0 to 1050
t_a04	time spent shopping (in minutes)	0 to 1434
t_a05	time spent on private business (in minutes)	0 to 1077
t_a06	time spent getting petrol (in minutes)	0 to 896
t_a07	time spent on social or leasure activities (in minutes)	0 to 1425
t_a08	time spent on vacation or on long (intercity) travel (in minutes)	0 to 828
t_a09	time spent doing exercise (in minutes)	0 to 1416
t_a10	time spent at home (in minutes)	0 to 1440
t_a11	time spent travelling (everyday travelling) (in minutes)	0 to 1182
t_a12	Non-allocated time (in minutes)	0 to 1160
female	dummy variable for female individuals	1 for female, 0 otherwise
age	age of the respondent (in years, approximate)	21 to 80
occ_full_time	dummy for respondents working full time	1 for respondents working full time, 0 otherwise
weekend	dummy for weekend days	1 for weekend, 0 otherwise

Appendix C

Index of example files

Table C.1 presents an overview of the example files made available with *Apollo*, while Table C.2 shows which function is used with what example.

Table C.1: Index of example files

File	Description	Data file
Apollo_example_1.r	Simple MNL model on mode choice RP data	apollo_modeChoiceData.csv
Apollo_example_2.r	Simple MNL model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_3.r	MNL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_4.r	Two-level NL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_5.r	Three-level NL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_6.r	CNL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_7.r	Simple RRM model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_8.r	DFT model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_9.r	DFT model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_10.r	Exploded logit model on drug choice data	apollo_drugChoiceData.csv
Apollo_example_11.r	MDCEV model on time use data without outside good and constants only in utilities	apollo_timeUseData.csv
Apollo_example_12.r	MDCEV model on time use data with outside good and with covariates in utilities	apollo_timeUseData.csv
Apollo_example_13.r	MDCNEV model on time use data with outside good and with covariates in utilities	apollo_timeUseData.csv
Apollo_example_14.r	Mixed logit model on Swiss route choice data, uncorrelated Lognormals in utility space	apollo_swissRouteChoiceData.csv
Apollo_example_15.r	Mixed logit model on Swiss route choice data, correlated Lognormals in utility space	apollo_swissRouteChoiceData.csv
Apollo_example_16.r	Mixed logit model on Swiss route choice data, WTP space with correlated and flexible distributions, inter and intra-individual heterogeneity	apollo_swissRouteChoiceData.csv
Apollo_example_17.r	Mixed MDCEV model on time use data, alpha-gamma profile, no outside good and random constants only in utilities	apollo_timeUseData.csv
Apollo_example_18.r	Simple LC model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_19.r	Simple DM model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_20.r	LC model with class allocation model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_21.r	Latent class with continuous random parameters on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_22.r	RP-SP model on mode choice data	apollo_modeChoiceData.csv
Apollo_example_23.r	Best-worst model on drug choice data	apollo_drugChoiceData.csv
Apollo_example_24.r	ICLV model on drug choice data, using ordered measurement model for indicators	apollo_drugChoiceData.csv
Apollo_example_25.r	ICLV model on drug choice data, using continuous measurement model for indicators	apollo_drugChoiceData.csv
Apollo_example_26.r	HB model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_27.r	Simple LC model on Swiss route choice data, EM algorithm	apollo_swissRouteChoiceData.csv
Apollo_example_28.r	LC model with class allocation model on Swiss route choice data, EM algorithm	apollo_swissRouteChoiceData.csv
Apollo_example_29.r	Mixed logit model on Swiss route choice data, correlated Lognormals in utility space, EM algorithm	apollo_swissRouteChoiceData.csv
Apollo_example_30.r	Efficient coding of large choice sets	none (generated in model script)

Table C.2: Functions used by *Apollo*, with inputs and outputs

	Apollo_example_1.r	**Apollo_example_2.r**	**Apollo_example_3.r**	**Apollo_example_4.r**	**Apollo_example_5.r**	**Apollo_example_6.r**	**Apollo_example_7.r**	**Apollo_example_8.r**	**Apollo_example_9.r**	**Apollo_example_10.r**	**Apollo_example_11.r**	**Apollo_example_12.r**	**Apollo_example_13.r**	**Apollo_example_14.r**	**Apollo_example_15.r**	**Apollo_example_16.r**	**Apollo_example_17.r**	**Apollo_example_18.r**	**Apollo_example_19.r**	**Apollo_example_20.r**	**Apollo_example_21.r**	**Apollo_example_22.r**	**Apollo_example_23.r**	**Apollo_example_24.r**	**Apollo_example_25.r**	**Apollo_example_26.r**	**Apollo_example_27.r**	**Apollo_example_28.r**	**Apollo_example_29.r**	**Apollo_example_30.r**
apollo_attach	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_avgInterDraws														x	x	x	x				x			x	x				x	
apollo_avgIntraDraws																x														
apollo_basTest						x	x																							
apollo_bootstrap																		x		x										
apollo_choiceAnalysis	x	x																												
apollo_cnl						x																								
apollo_combineModels																					x	x	x	x						
apollo_combineResults						x																								
apollo_conditionals																x								x	x					
apollo_deltaMethod			x																											
apollo_detach	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_dft								x	x																					
apollo_el										x																				
apollo_estimate	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_firstRow																x		x	x	x	x							x		
apollo_fitsTest			x																											
apollo_initialise	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_lc																		x	x	x	x						x	x		
apollo_lcEM																											x	x		
apollo_lcConditionals																				x										
apollo_lcUnconditionals																				x						x				
apollo_llCalc																								x						
apollo_loadModel																x														
apollo_lrTest			x	x	x																									
apollo_mdcev											x	x					x													
apollo_mdcnev													x																	
apollo_mixEM																												x		

	Apollo_example_1.r	**Apollo_example_2.r**	**Apollo_example_3.r**	**Apollo_example_4.r**	**Apollo_example_5.r**	**Apollo_example_6.r**	**Apollo_example_7.r**	**Apollo_example_8.r**	**Apollo_example_9.r**	**Apollo_example_10.r**	**Apollo_example_11.r**	**Apollo_example_12.r**	**Apollo_example_13.r**	**Apollo_example_14.r**	**Apollo_example_15.r**	**Apollo_example_16.r**	**Apollo_example_17.r**	**Apollo_example_18.r**	**Apollo_example_19.r**	**Apollo_example_20.r**	**Apollo_example_21.r**	**Apollo_example_22.r**	**Apollo_example_23.r**	**Apollo_example_24.r**	**Apollo_example_25.r**	**Apollo_example_26.r**	**Apollo_example_27.r**	**Apollo_example_28.r**	**Apollo_example_29.r**	**Apollo_example_30.r**
apollo_mnl	x	x	x				x							x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_modelOutput	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_nl				x	x																									
apollo_normalDensity																									x					
apollo_ol																								x						
apollo_op																														
apollo_outOfSample																		x		x										
apollo_panelProd	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	
apollo_prediction			x								x		x									x		x	x	x				
apollo_prepareProb	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_readBeta			x	x		x							x		x															
apollo_saveOutput	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_searchStart																		x	x	x										
apollo_sharesTest			x																											
apollo_speedTest																x														
apollo_unconditionals																x								x	x				x	
apollo_validateInputs	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_weighting																														

Appendix D

Overview of functions, lists and elements

Table D.1 presents an overview of all *Apollo* functions, together with their inputs and outputs. Table D.2 and Table D.3 give an overview of all the lists and elements used by *Apollo*.

Table D.1: Functions used by *Apollo*, with inputs and outputs

Function	Arguments	Description	Output
apollo_addCovariance	model, apollo_inputs	Given an estimated model object and its corresponding apollo_inputs list, calculates the Hessian at the estimated parameter values, and from it calculates the classical and robust covariance matrix between the estimated parameters, as well as their standar errors.	A copy of the model argument, but with a Hessian and covariance matrices added to it.
apollo_attach	apollo_beta, apollo_inputs	Attaches parameters and data to allow users to refer to individual variables by name without reference to the object they are contained in.	Nothing
apollo_avgInterDraws	P, apollo_inputs, functionality	Averages individual-specific likelihood across inter-individual draws.	Likelihood averaged over inter-individual draws (shape depends on argument functionality).
apollo_avgIntraDraws	P, apollo_inputs, functionality	Averages observation-specific likelihood across intra-individual draws.	Likelihood averaged over intra-individual draws (shape depends on argument functionality).
apollo_basTest	model1, model2	Calculates the p-value for the Ben-Akiva & Swait test for non-nested models. Both models need to have a single discrete choice component, and be estimated on the same data.	Returns the p-value of the test invisibly
apollo_bootstrap	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, estimate_settings, bootstrap_settings	Samples individuals with replacement from the database, and estimates the model in each sample.	Covariance matrix of the nRep sets of estimated parameters. Also written to file.
apollo_choiceAnalysis	choiceAnalysis_settings, apollo_control, database	Compares market shares across subsamples in dataset, and writes results to a file.	Saves the output to a csv file in the working directory.
apollo_cnl	cnl_settings, functionality	Calculates probabilities of a cross nested logit model.	The returned object depends on the value of argument functionality.

Function	Arguments	Description	Output
apollo_combineModels	P, apollo_inputs, functionality, [components], [asList]	Combines model components to create probability for overall model.	Argument P with an extra element called "model", which is the product of all the other elements. Shape depends on argument functionality.
apollo_combineResults	combineResults_settings	Writes results from various models to a single CSV file.	Nothing, but writes a file called 'model_comparison|_[date].csv' in the working directory.
apollo_compareInputs	apollo_inputs	Compares the content of the given apollo_inputs to that in the global environment.	TRUE if the content is the same, FALSE otherwise
apollo_conditionals	model, apollo_probabilities, apollo_inputs	Calculates posterior expected values (conditionals) of random coefficients, as well as their standard deviations.	List of matrices, as many as random parameters. Reports indivID, mean and s.d. Of random parameter for each individual.
apollo_deltaMethod	model, deltaMethod_settings	Applies the delta method to calculate the standard errors of transformations of parameters. If the bootstrap covariance matrix is available, it is used. If not, the robust covariance matrix is used.	Matrix contating calue, s.e. And t-ratio resulting from the operation. This is also printed to screen.
apollo_detach	apollo_beta, apollo_inputs	Detaches variables attached by apollo_attach.	Nothing.
apollo_dft	dft_settings, functionality	Calculate probabilities of a Decision Field Theory (DFT) with external thresholds.	The returned object depends on the value of argument functionality.
apollo_el	el_settings, functionality	Calculates the probabilities of an exploded logit model and can also perform other operations based on the value of the functionality argument. The function calculates the probability of a ranking as a product of logit models with gradually reducing availability, where scale differences can be allowed for.	The returned object depends on the value of argument functionality.
apollo_estimate	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, estimate_settings	Estimates a model using the likelihood function defined by apollo_probabilities.	model object

Function	Arguments	Description	Output
apollo_firstRow	P, apollo_inputs	Given a multi-row input, keeps only the first row for each individual.	Depends on inputs. If P is a list, then it returns a list where each element has only the first row of each individual.
apollo_fitsTest	model, apollo_probabilities, apollo_inputs, fitsTest_settings	Given the predictions of a model, it compares the fit across categories of observations.	Matrix with average fit per category (invisibly).
apollo_initialise	None	Prepares environment (the global environment if called by the user) for model definition and estimation.	Nothing.
apollo_lc	lc_settings, apollo_inputs, functionality	Using the conditional likelihoods of each latent class, as well as teir classification probabilities, calculate the weighted likelihood of the whole model.	The returned object depends on the value of argument functionality
apollo_lcConditionals	model, apollo_probabilities, apollo_inputs	Calculates posterior expected values (conditionals) of class allocation probabilities for each individual.	A matrix with the posterior class allocation probabilities for each individual.
apollo_lcEM	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, [lcEM_settings], [estimate_settings]	Uses the Expectation Maximisation (EM) algorithm to estimate a latent class model.	A model object.
apollo_lcUnconditionals	model, apollo_probabilities, apollo_inputs	Returns draws (unconditionals) for random parameters in model, including interactions with deterministic covariates	List of object, one per random component and one for the class allocation probabilities.
apollo_llCalc	apollo_beta, apollo_probabilities, apollo_inputs, [silent]	Calculates the log-likelihood of each model component as well as the whole model.	A list of vectors. Each vector corresponds to the log-likelihood of the whole model (first element) or a model component.
apollo_loadModel	modelName	Loads an estimated model object from a file in the current working directory.	A model object.
apollo_lrTest	model1, model2	Calculates the likelihood ratio test and prints result.	LL ratio test statistic (invisibly)
apollo_mdcev	mdcev_settings, functionality	Calculates the likelihood of a Multiple Discrete Continuous Extreme Value (MDCEV) model.	The returned object depends on the value of argument functionality

Function	Arguments	Description	Output
apollo_mdcnev	mdcnev_settings, functionality	Calculates the likelihood of a Multiple Discrete Continuous Nested Extreme Value (MDCNEV) model with an outside good.	The returned object depends on the value of argument functionality
apollo_mixEM	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, [mixEM_settings], [estimate_settings]	Uses the Expectation Maximisation (EM) algorithm to estimate an MNL model with continuous random coefficients.	A model object.
apollo_mnl	mnl_settings, functionality	Calculates probabilities of a multinomial logit model	The returned object depends on the value of argument functionality
apollo_modelOutput	model, modelOutput_settings	Prints estimation results to console. Amount of information presented can be adjusted through arguments.	A matrix of coefficients, s.d. And t-tests (invisible)
apollo_nl	nl_settings, functionality	Calculates probabilities of a nested logit model.	The returned object depends on the value of argument functionality
apollo_normalDensity	normalDensity_settings, functionality	Calculates density from a Normal distribution at a specific value with a specified mean and standard deviation.	The returned object depends on the value of argument functionality
apollo_ol	ol_settings, functionality	Calculates the probabilities of an ordered logit model and can also perform other operations based on the value of the functionality argument.	The returned object depends on the value of argument functionality
apollo_op	op_settings, functionality	Calculates the probabilities of an ordered probit model and can also perform other operations based on the value of the functionality argument.	The returned object depends on the value of argument functionality
apollo_outOfSample	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, [estimate_settings], [outOfSample_settings]	Randomly generates estimation and validation samples, estimates the model on the first and calculates the likelihood for the second, then repeats.	A matrix with the average log-likelihood per observation for estimation and validation samples, for each repetition. Files are written to the working directory.
apollo_panelProd	P, apollo_inputs, functionality	Multiplies likelihood of observations from the same individual, or adds the log of them.	Probabilities at the individual level.

Function	Arguments	Description	Output
apollo_prediction	model, apollo_probabilities, apollo_inputs, [prediction_settings], [modelComponent]	Calculates apollo_probabilities with functionality="prediction" and extracts one element from the returned list.	A list containing predictions for modelComponent as described in apollo_probabilities. The shape of the prediction will depend on the model component.
apollo_prepareProb	P, apollo_inputs, functionality	Checks that likelihoods, i.e. Probabilities in the case of choice models, are in the appropiate format to be returned.	The returned object depends on the value of argument functionality
apollo_readBeta	apollo_beta, apollo_fixed, inputModelName, [overwriteFixed]	Reads in parameters from a previously estimated model and copies the values to the given apollo_beta vector, only for those parameters whose name matches.	Named numeric vector. Names and updated starting values for parameters.
apollo_saveOutput	model, [saveOutput_settings]	Writes files in the working directory with the estimation results.	Nothing
apollo_searchStart	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, [searchStart_settings]	Given a set of starting values and a range for them, searches for points with a better likelihood.	Named vector of model parameters. These are the best values found.
apollo_sharesTest	model, apollo_probabilities, apollo_inputs, sharesTest_settings	Prints tables comparing the shares predicted by the model with the shares observed in the data.	Nothing
apollo_speedTest	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, [speedTest_settings]		A matrix with the average time per evaluation for each number of threads and draws combination. A graph is also plotted.
apollo_unconditionals	model, apollo_probabilities, apollo_inputs	Returns draws (unconditionals) for random parameters in model, including interactions with deterministic covariates	List of matrices (inter draws) or 3-dim arrays (intra draws), one per random coefficient. One row per individual.

Function	Arguments	Description	Output
apollo_validateInputs	apollo_beta, apollo_fixed, database, apollo_control, apollo_HB, apollo_draws, apollo_randCoeff, apollo_lcPars, silent	Searches the user work space (.GlobalEnv) for all necessary input to run apollo_estimate, and packs it in a single list.	List grouping several required input for model estimation.
apollo_weighting	P, apollo_inputs, functionality	Applies weights to individual observations in likelihood function.	The likelihood of the model in the appropriate form for the given functionality, multiplied by individual-specific weights.

Table D.2: Lists used by *Apollo*

Name of list	Contents	Description
apollo_control	modelName, modelDescr, indivID, mixing, nCores, workInLogs, seed, HB, noValidation, noDiagnostics, panelData, analyticGrad	Global settings. Input for apollo_validateInput.
apollo_draws	interDrawsType, interNDraws, interUnifDraws, interNormDraws, intraDrawsType, intraNDraws, intraUnifDraws, intraNormDraws	Settings for draws generation. Input for apollo_validateInput.
apollo_HB	hbDist, gNCREP, gNEREP, gINFOSKIP, gFULLCV, constraintsNorm	Settings for Bayesian estimation. Input for apollo_validateInput.
apollo_inputs	database, apollo_control, apollo_HB, apollo_draws, apollo_randCoeff, apollo_lcPars, draws	List grouping most common inputs. Created by function apollo_validateInputs
bootstrap_settings	nRep, samples, seed	Settings for bootstraping parameter estimation. Input for apollo_bootstrap.
choiceAnalysis_settings	alternatives, avail, choiceVar, explanators, rows	Settings for describing the conditional distribution of choice. Input to apollo_choiceAnalysis.
cnl_settings	alternatives, avail, choiceVar, V, cnlNests, cnlStructure, rows	Settings for Cross Nested Logit. Input for apollo_cnl.
combineResults_settings	modelNames, printClassical, printPVal, printT1, estimateDigits, tDigits, pDigits, sortByDate	Settings for combining multiple outputs. Input for apollo_combineResults.
deltaMethod_settings	operation, parName1, parName2, multiPar1, multiPar2	Settings for delta method. Input for apollo_delta_method.
dft_settings	alternatives, avail, choiceVar, attrValues, altStart, attrWeights, attrScaling, procPars, rows	Settings for Decision Field Theory models. Input for apollo_dft.
el_settings	alternatives, avail, choiceVars, V, scales, rows	Settings for exploded logit models. Input for apollo_el.
estimate_settings	estimationRoutine, maxIterations, writeIter, hessianRoutine, printLevel, constraints, scaling, numDeriv_settings, bootstrapSE, bootstrapSeed, silent	Settings for estimation. Input for apollo_estimate.
fitsTest_settings	subsamples, modelComponent	Settings for comparing model forecast fit between subsamples of data. Input for apollo_fitsTest.
lc_settings	inClassProb, classProb, componentName	Settings for latent class modelling. Input for apollo_lc
mdcev_settings	V, alternatives, alpha, gamma, sigma, cost, avail, continuousChoice, budget, minConsumption, outside, rows, nRep	Settings for Multiple discrete-continuous models. Input for apollo_mdcev.

Name of list	Contents	Description
mdcnev_settings	V, alternatives, alpha, gamma, mdcnevNests, mdcnevStructure, cost, avail, continuousChoice, budget, minCOnsumption, outside, rows, componentName	Settings for Multiple discrete-continuous nested models. Input for apollo_mdcnev.
mnl_settings	alternatives, avail, choiceVar, V, rows	Settings for Multinomial logit models. Input for apollo_mnl.
model	apollo_beta, apollo_control, apollo_draws, apollo_fixed, apollo_lcPars, apollo_probabilities, apollo_randCoeff, avgCP, avgLL, bootstrapSE, code, componentReport, constraints, control, corrmat, eigValue, estimate, estimationRoutine, fixed, gradient, gradientObs, hessian, hessianMethodsAttempted, hessianMethodUsed, hessianScaling, iterations, last.step, LL0, Llout, LLStart, maximum, message, modelTypeList, nIndivs, nIter, nObs, nObsTot, objectiveFn, Pout, robcorrmat, robse, robvarcov, scaling, se, startTime, timeEst, timePost, timePre, timeTaken, type, varcov	Object storing an estimated model. Output of apollo_estimate.
modelOutput_settings	PrintFixed, printClassical, printPVal, printT1, printDataReport, printModelStructure, printCovar, printCorr, printOutliers, printChange, printFunctions	Settings for printing model estimation results to screen. Input for apollo_modelOutput.
nl_settings	alternatives, avail, choiceVar, V, nlNests, nlStructure, rows	Settings for Nested Logit models. Input for apollo_nl.
normalDensity_settings	outcomeNormal, xNormal, mu, sigma, rows	Settings for linear models. Input for apollo_normalDensity.
ol_settings	outcomeOrdered, V, tau, coding, rows	Settings for Ordered logit models. Input for apollo_ol.
op_settings	outcomeOrdered, V, tau, coding, rows	Settings for Ordered probit models. Input for apollo_op.
outOfSample_settings	nRep, validationSize, samples	Settings for cross validation. Input for apollo_outOfSample.
prediction_settings	modelComponent, runs, silent, nRep	Settings for forecasting. Input for apollo_prediction.
saveOutput_settings	printClassical, printPVal, printT1, printDataReport, printModelStructure, printCovar, printCorr, printOutliers, printChange, printFunctions, saveEst, saveCov, saveCorr, saveModelObject, writeF12	Settings for saving estimation results to disk. Input for apollo_saveOutput.

Name of list	Contents	Description
searchStart_settings	nCandidates, smartStart, apolloBetaMin, apolloBetaMax, maxStages, dTest, gTest, llTest, bfgsIter	Settings for estimation algorithm with multiple random starting points. Input for apollo_searchStart.
sharesTest_settings	alternatives, choiceVar, subsamples, modelComponent, newAlts, newAltsOnly	Settings for comparing model forecast fit between subsamples of data. For discrete choice models only. Input for apollo_sharesTest.
speedTest_settings	nDrawsTry, nCoresTry, nRep	Settings for measuring estimation speed. Input for apollo_speedTest.

Table D.3: Elements in lists and functions used by *Apollo*

Argument	Dimensionality	Type	Description
alpha	list	numeric	Alpha parameters of MDC(N)EV associated to each alternative, including for the outside good. As many elements as alternatives.
alternatives	vector	numeric	Names of alternatives and their corresponding value in choiceVar. Should have as many elements as V
alternatives (MDCEV, MDCNEV)	vector	character	Names of alternatives, elements must match the names in list 'V'. If using an outside good, it must include "outside".
altStart	list	numeric	A named list with as many elements as alternatives. Each elment can be a scalar or vector containing the starting preference value for the alternative.
analyticGrad	scalar	logical	TRUE for using analytical gradients during estimation (only applicable to selected models). Default is TRUE.
apollo_beta	vector	numeric	Named numeric vector. Names and values for parameters.
apolloBetaMax	vector	numeric	Minimum possible value of parameters when generating candidates. Ignored if smartStart is TRUE. Default is apollo_beta - 0.1.
apolloBetaMin	vector	numeric	Maximum possible value of parameters when generating candidates. Ignored if smartStart is TRUE. Default is apollo_beta + 0.1.
apollo_fixed	vector	character	Character vector. Names (as defined in apollo_beta) of parameters whose value should not change during estimation.
apollo_lcPars	scalar	function	Function. Used with latent class models. Constructs a list of parameters for each latent class. Receives two arguments:
apollo_probabilities	scalar	function	Function. Returns probabilities of the model to be estimated. Must receive three arguments:
apollo_randCoeff	scalar	function	Function. Used with mixing models. Constructs the random parameters of a mixing model. Receives two arguments:
attrScaling	list	numeric	A named list with as many elements as attributes, or fewer. Each element is a factor scaling the attribute value. AttrWeights and attrScalings should not be both defined for an attribute. Default is 1 for all attributes.
attrValues	list	numeric	As many elements as alternatives. Each sub-list contains the alternative attributes for each observation (usually a column from the database). All alternatives must have the same attributes (can be set to zero if not relevant).

Argument	Dimensionality	Type	Description
attrWeights	list	numeric	As many elements as attributes, or fewer. Each element is the weight of the attribute. They should add up to one for each observation and draw (if present), and will be re-scaled if they do not. AttrWeights and attrScalings should not be both defined for an attribute. Default is 1 for all attributes.
avail	list	numeric	Availabilities of alternatives, one element per alternative. Names of elements must match those in alternatives. Values can be a vector of 0 or 1.
avgCP	vector	numeric	Average likelihood for each individual at the estimated parameters.
avgLL	vector	numeric	Average log-likelihood for each individual at the estimated parameters.
model1	scalar	character/model	Either a character variable with the name of a previously estimated model, or an estimated model in memory, as returned by apollo_estimate.
model2	scalar	character/model	Either a character variable with the name of a previously estimated model, or an estimated model in memory, as returned by apollo_estimate.
bfgsIter	scalar	numeric	Number od BFGS iterations to perform at each stage to each remaining candidate. Default is 20.
bootstrapSE	scalar	numeric	Number of bootstrap samples to calculate standard errors. Default is 0, meaning no bootstrap s.e. Will be calculated. Number must zero or a positive integer. Only used if apollo_control$HB is FALSE.
bootstrapSeed	scalar	numeric	Random number generator seed to generate the bootstrap samples. Only used if bootstrapSE>0. Default is 24.
budget	vector	numeric	Budget for each observation.
choiceVar	vector	numeric	Contains choices for all observations. It will usually be a column from the database. Values are defined in alternatives.
choiceVars	list	numeric	Contain choices for each position of the ranking. The list must be ordered with the best choice first, second best second, etc. It will usually be a list of columns from the database.
classProb	vector/matrix/3-dim array	numeric	Allocation probability for each class. One element per class, in the same order as inClassProb.

Argument	Dimensionality	Type	Description
cnlNests	list	numeric	Lambda parameters for each nest. Elements must be named according to nests. The lambda at the root is fixed to 1, and therefore does not need to be defined.
cnlStructure	matrix	numeric	One row per nest and one column per alternative. Each element of the matrix is the alpha parameter of that (nest, alternative) pair.
code	scalar	numeric	Value indicating convergence (or lack of it) of an a model, as defined by maxLik.
coding	vector	numeric/character	Optional argument. Defines the order of the levels in outcomeOrdered. The first value is associated with the lowest level of outcomeOrdered, and the last one with the highest value. If not provided, is assumed to be 1:(length(tau) + 1).
componentName	scalar	character	Name given to model component.
componentReport	scalar	character	Message indicating convergence (or lack of it) of an a model, as defined by maxLik.
constraints	vector	character	Constraints on parameters to estimate. For example c('b1>0','b1 + b2>1'). Only one of >, >= and = can be used. Parameter names must be on the left. Must not include fixed parameters.
constraintsNorm	vector	character	Constraints for random coefficients in bayesian estimation. Constraints can be written as "b1>b2", "b1>0", or "b1<0".
continuousChoice	list	numeric	Amount of consumption of each alternative. One element per alternative, as long as the number of observations or a scalar. Names must match those in alternatives.
control	list	numeric	Control variables (hyperparameters) for maximum likelihood estimation, as defined by maxLik.
corrmat	matrix	numeric	Correlation matrix.
cost	list	numeric	Price of each alternative. One element per alternative, each one as long as the number of observations or a scalar. Names must match those in alternatives.
database	data.frame	numeric/character	data.frame. Data used by model.
draws	list	numeric	Generated draws. They can be either matrices for inter-individual draws, or a 3-dim array for intra-individual draws.
dTest	scalar	numeric	Tolerance for test 1. A candidate is discarded if its distance in parameter space to a better one is smaller than dTest. Default is 1.

Argument	Dimensionality	Type	Description
eigValue	scalar	numeric	Maximum eigenvalue of the Hessian matrix for an estimated model.
estimate	vector	numeric	Value of estimated parameters (including fixed parameters).
estimateDigits	scalar	numeric	Number of decimal places to print for estimates. Default is 4.
estimationRoutine	scalar	character	Estimation method. Can take values "bfgs", "bhhh", or "nr". Used only if apollo_control$HB is FALSE. Default is "bfgs".
explanators	data.frame	numeric/character	Variables determining subsamples of the database. Values in each column must describe a group or groups of individuals (e.g. Socio-demographics). Most usually a subset of columns from database.
fixed	vector	logical	Vector indicating the free parameters to estimate. Only used by maxLik, not Apollo.
functionality	scalar	character	Description of the desired output from apollo_probabilities. Can take the values: "estimate", "prediction", "validate", "zero_LL", "conditionals", "output", "raw".
gamma	list	numeric	Gamma parameters for each alternative, excluding any outside good. As many elements as inside good alternatives.
gFULLCV	scalar	logical	TRUE for estimating the full covariance matrix between random coefficients.
gINFOSKIP	scalar	numeric	A short summary of the Markov chain will be printed to screen every gINFOSKIP number of draws.
gNCREP	scalar	numeric	Number of burn-in draws to use prior to convergence. (Defaults to 100000)
gNEREP	scalar	numeric	Number of draws to keep for averaging after convergence has been reached. (Defaults to 100000)
gradient	vector	numeric	Gradient of the log-likelihood function at the estimated value of the parameters.
gradientObs	matrix	numeric	Observation-level score matrix, i.e. the gradient of the log-likelihood function at the estimated parameter values at the observation level.
gTest	scalar	numeric	Tolerance for test 2. A candidate is discarded if the norm of its gradient is smaller than gTest AND its LL is further than llTest from a better candidate. Default is 10^(-3).
HB	scalar	logical	TRUE if using RSGHB for Bayesian estimation of model.
hbDist	vector	character	Defines the distribution of each parameter to be estimated. Possible values are "DNE", "F", "N", "LN+", "LN-", "CN+", "CN-", and "JSB".

Argument	Dimensionality	Type	Description
hessian	matrix	numeric	Hessian (second derivative) of the log-likelihood function.
hessianMethodsAttempted	vector	character	Name of the methods attempted to calculate the Hessian.
hessianMethodUsed	scalar	character	Name of the methods used to calculate the Hessian reported in model$hessian.
hessianRoutine	scalar	character	Name of routine used to calculate the Hessian of the loglikelihood function after estimation. Valid values are "numDeriv" (default) and "maxLik" to use the routines in those packages, and "none" to avoid estimating the Hessian (and the covariance matrix). Only used if apollo_control$HB=FALSE.
hessianScaling	vector	numeric	Scaling used to calculate the Hessian. By default, they will be equal to the estimated value of the parameters.
inClassProb	vector/matrix/3-dim array	numeric	Conditional likelihood for each class. One element per class, in the same order as classProb.
indivID	scalar	character	Name of column in the database with each decision maker's ID.
inputModelName	scalar	character	Character. modelName for model from which results are used as starting values.
interDrawsType	scalar	character	Type of inter-individual draws ('halton','mlhs','pmc','sobol','sobolOwen', 'sobolFaureTezuka', 'sobolOwenFaureTezuka' or the name of an object loaded in memory, see manual in www.ApolloChoiceModelling.com for details).
interNDraws	scalar	numeric	Number of inter-individual draws per individual. Should be set to 0 if not using them.
interNormDraws	vector	character	Names of normaly distributed inter-individual draws.
interUnifDraws	vector	character	Names of uniform-distributed inter-individual draws.
intraDrawsType	scalar	character	Type of intra-individual draws ('halton','mlhs','pmc','sobol','sobolOwen','sobolFaureTezuka', 'sobolOwenFaureTezuka' or the name of an object loaded in memory).
intraNDraws	scalar	numeric	Number of intra-individual draws per individual. Should be set to 0 if not using them.
intraNormDraws	vector	character	Names of normaly distributed intra-individual draws.
intraUnifDraws	vector	character	Names of uniform-distributed intra-individual draws.
iterations	scalar	numeric	Number of iterations of the maximum likelihood estimation according to maxLik (not Apollo).

Argument	Dimensionality	Type	Description
last.step	list	numeric	List describing last unsuccessful iteration, if model$code=3 (i.e. if there was an error during maximum likelihood estimation).
LL0	scalar	numeric	Log-likelihood of the null (equiprobable) model, if applicable (only for models with a single MNL, NL or CNL component)
Llout	vector	numeric	Log-likelihood of each model component at the estimated parameter values. At the sample level.
LLStart	scalar	numeric	Log-likelihood value at the starting value of the parameters.
llTest	scalar	numeric	Tolerance for test 2. A candidate is discarded if the norm of its gradient is smaller than gTest AND its LL is further than llTest from a better candidate. Default is 3.
maximum	scalar	numeric	Log-likelihood value at the estimated parameter values.
maxIterations	scalar	numeric	Maximum number of iterations of the estimation routine before stopping. Used only if apollo_control$HB is FALSE. Default is 200.
maxStages	scalar	numeric	Maximum number of search stages. The algorithm will stop when there is only one candidate left, or if it reaches this number of stages. Default is 5.
mdcnevNests	list	numeric	Lambda parameters for each nest. Elements must be named with the nest name. The lambda at the root is fixed to 1, and therefore must be no be defined. The value of the estimated mdcnevNests parameters should be between 0 and 1 to ensure consistency with random utility maximization.
mdcnevStructure	matrix	numeric	One row per nest and one column per alternative. Each element of the matrix is 1 if an alternative belongs to the corresponding nest.
message	scalar	character	Message describing convergence (successful or not), as generated by maxLik.
minConsumption	list	numeric	Minimum consumption of the alternatives, if consumed. As many elements as alternatives. Names must match those in alternatives.
mixing	scalar	logical	TRUE for models that include random parameters.
modelComponent	scalar	character	Deprecated. Same as modelComponent inside prediction_settings.
modelDescr	scalar	character	Description of the model. Used in output files.
modelName	scalar	character	Name of the model to load.
modelNames	vector	character	List of names of models to combine. Use an empty vector to combine results from all models in the directory.

Argument	Dimensionality	Type	Description
modelTypeList	vector	character	Vector indicating the type of model (MNL, NL, etc.) of each model component.
mu	scalar/vector	numeric	Intercept of the linear model.
multiPar1	scalar	numeric	A value to scale parName1.
multiPar2	scalar	numeric	A value to scale parName2.
nCandidates	scalar	numeric	Number of candidate sets of parameters to be used at the start. Should be an integer bigger than 1. Default is 100.
nCores	scalar	numeric	Number of threads (processors) to use in estimation of the model.
nCoresTry	vector	numeric	Number of threads to try. Default is from 1 to the detected number of cores.
nDrawsTry	vector	numeric	Number of inter and intra-person draws to try. Default value is c(50, 100, 200).
newAlts	list	numeric	Optional list describing the new alternatives to be used by apollo_sharesTest. This should have as many elements as new alternatives, with each entry be- ing a matrix of 0-1 entries, with one row per observation, and one column per alternative used in the model.
newAltsOnly	scalar	logical	f TRUE, results will only be printed for the 'new' alternatives defined in newAlts, not the original alternatives used in the model. Set to FALSE by default.
nIndivs	scalar	numeric	Number of individuals in the database.
nIter	scalar	numeric	Number of iterations necessary for convergence, as counted by Apollo.
nlNests	list	numeric	Lambda parameters for each nest. Elements must be named with the nest name. The lambda at the root is fixed to 1 if excluded (recommended).
nlStructure	matrix	numeric	As many elements as nests, it must include the "root". Each element contains the names of the nests or alternatives that belong to it. Element names must match those in nlNests.
nObs	scalar	numeric	Number of observations in the database, counted as rows in the database.
nObsTot	vector	numeric	Number of observations per model component, not counting excluding rows.
noDiagnostics	scalar	logical	TRUE if user does not wish model diagnostics to be printed - FALSE by default.

Argument	Dimensionality	Type	Description
noValidation	scalar	logical	TRUE if user does not wish model input to be validated before estimation - FALSE by default.
nRep	scalar	numeric	Number of repetitions.
numDeriv_settings	list	numeric/character	Additional arguments to the Richardson method used by numDeriv to calculate the Hessian. See argument method.args in grad for more details.
objectiveFn	scalar	function	Loglikelihood function generated by Apollo to mask apollo_probabilities from maxLik.
operation	scalar	character	Function to calculate the delta method for. See details.
outcomeNormal	vector	normal	Numeric vector. Dependant variable.
outcomeOrdered	vector	normal	Dependant variable. The coding of this variable is assumed to be from 1 to the maximum number of different levels. For example, if the ordered response has three possible values: "never", "sometimes" and "always", then it is assumed that outcomeOrdered contains "1" for "never", "2" for "sometimes", and 3 for "always". If another coding is used, then it should be specified using the coding argument.
outside	scalar	character	Optional name of the outside good.
overwriteFixed	scalar	logical	Boolean. TRUE if starting values for fixed parameters should also be updated from input file.
P	vector/matrix/3-dim array	numeric	List of vectors, matrices or 3-dim arrays. Likelihood of the model components.
panelData	scalar	logical	TRUE if using panelData data (created automatically by apollo_validateControl).
parName1	scalar	character	Name of the first parameter.
parName2	scalar	character	Name of the second parameter. Optional depending on operation.
pDigits	scalar	numeric	Number of decimal places to print for p-values. Default is 2.
Pout	vector	numeric	Loglikelihood value of all models components at the estimated parameter values. At the sample level.
printChange	scalar	logical	TRUE for printing difference between starting values and estimates. FALSE by default.
printClassical	scalar	logical	TRUE for printing classical standard errors. TRUE by default.
printCorr	scalar	logical	TRUE for printing parameters correlation matrix. If printClassical=TRUE, both classical and robust matrices are printed. FALSE by default.

Argument	Dimensionality	Type	Description
printCovar	scalar	logical	TRUE for printing parameters covariance matrix. If printClassical=TRUE, both classical and robust matrices are printed. FALSE by default.
printDataReport	scalar	logical	TRUE for printing summary of choices in database and other diagnostics. FALSE by default.
printFixed	scalar	logical	TRUE for printing fixed parameters among esti- mated parameter. TRUE by default.
printFunctions	scalar	logical	TRUE for printing apollo_control, apollo_randCoeff (when available), apollo_lcPars (when available) and apollo_probabilities. TRUE by default.
printLevel	scalar	logical	Higher values render more verbous outputs. Can take values 0, 1, 2 or 3. Ignored if apollo_control$HB is TRUE. Default is 3.
printModelStructure	scalar	logical	TRUE for printing model structure. TRUE by default.
printOutliers	scalar	logical	TRUE for printing 20 individuals with worst average fit across observations. FALSE by default. If Scalar is given, this replaces the default of 20.
printPVal	scalar	logical	TRUE for printing p-values. FALSE by default.
printT1	scalar	logical	If TRUE, t-test for H0: apollo_beta=1 are printed. FALSE by default.
procPars	list	numeric	A list containing the four DFT 'process parameters': error_sd, timesteps, phi1, and phi2.
robcorrmat	matrix	numeric	Robust correlation matrix of estimated parameters. Excludes fixed parameters.
robse	matrix	numeric	Robust s.e. of estimated parameters. Includes fixed parameters.
robvarcov	matrix	numeric	Robust covariance matrix of estimated parameters. Excludes fixed parameters.
rows	vector	logical	Consideration of rows in the likelihood calculation, FALSE to exclude. Length equal to the number of observations (nObs). Default is "all", equivalent to rep(TRUE, nObs).
runs	scalar	numeric	Number of runs to use for computing confidence intervals of predictions.

Argument	Dimensionality	Type	Description
samples	matrix/data.frame	numeric	Optional argument. Must have as many rows as observations in the database, and as many columns as number of repetitions wanted. Each column represents a re-sample, and each element the number of times that observation must be included in the sample. If this argument is provided, then nRep is ignored. Note that this allows sampling at the observation rather than the individual level, which is not recommended for panel data.
saveCorr	scalar	logical	TRUE for saving estimated correlation matrix to a CSV file. TRUE by default.
saveCov	scalar	logical	TRUE for saving estimated correlation matrix to a CSV file. TRUE by default.
saveEst	scalar	logical	TRUE for saving estimated parameters and standard errors to a CSV file. TRUE by default.
saveModelObject	scalar	logical	TRUE to save the R model object to a file (use apollo_loadModel to load it to memory). TRUE by default.
scales	list	numeric	Scale factors of each logit model. Should have one element less than choiceVars. At least one element should be normalized to 1. If omitted, scale=1 for all positions is assumed.
scaling	vector	numeric	Names of elements should match those in apollo_beta. Optional scaling for parameters. If provided, for each parameter i, (apollo_beta[i]/scaling[i]) is optimised, but scaling[i]*(apollo_beta[i]/scaling[i]) is used during estimation. For example, if parameter b3=10, while b1 and b2 are close to 1, then setting scaling = c(b3=10) can help estimation, specially the calculation of the Hessian. Reports will still be based on the non-scaled parameters.
se	vector	numeric	Standard errors of estimated parameters. Includes fixed parameters.
seed	scalar	numeric	Seed for random number generation.
sigma	scalar	numeric	Variance or scale parameter of the random error component.
silent	scalar	logical	If TRUE, no information is printed to the console by the function. Default is FALSE.
smartStart	scalar	logical	If TRUE, candidates are randomly generated with more chances in the directions the Hessian indicates improvement of the LL function. Default is FALSE.
sortByDate	scalar	logical	If TRUE, models are ordered by date.

Argument	Dimensionality	Type	Description
startTime	scalar	POSIX	POSIX date object recording the time when the full estimation processes started.
subsamples	list	logical	Each element of the list defines whether a given observation belongs to a given subsample (e.g. By sociodemographics).
tau	list	numeric	List of numeric elements representing thresholds. As many elements as different levels in the dependent variable - 1. Extreme thresholds are assumed to be -inf and +inf. Can also be a matrix with nObs rows and nThresholds columns.
tDigits	scalar	numeric	Number of decimal places to print for t-ratios values. Default is 2.
timeEst	scalar	numeric	Time (in seconds) taken by the estimation algorithm (excludes pre and post-processing).
timePost	scalar	numeric	Time (in seconds) taken by the post-estimation processing (including the calculation of the covariance matrix).
timePre	scalar	numeric	Time (in seconds) taken by the pre-estimation processing (including validation and symbolic calculation of gradient).
timeTaken	scalar	numeric	Time (in seconds) taken by the full estimation process (pre and post-estimation calculations included).
type	scalar	character	Type of maximisation (algorithm) as defined by maxLik.
V	list	numeric	Utilities (or base utilities) of the alternatives. Names of elements must match those in alternatives.
validationSize	scalar	numeric	Size of the validation sample. Can be a percentage of the sample (0-1) or the number of individuals in the validation sample (>1). Default is 0.1.
varcov	matrix	numeric	Covariance matrix of estimated parameters. Excludes fixed parameters.
weights	scalar	character	Name of column in database containing weights for estimation.
workInLogs	scalar	logical	TRUE for higher numeric stability at the expense of computational time. Useful for panel models only. Default is FALSE.
writeF12	scalar	logical	TRUE for writing results into an F12 file (ALOGIT format). FALSE by default.
writeIter	scalar	logical	Writes value of the parameters in each iteration to a csv file. Works only if estimation_routine="bfgs". Default is TRUE.
xNormal	vector	numeric	Single explanatory variable.

Appendix E

Detailed description of `model` object

Table E.1 presents an overview of all elements inside a model object produced by *Apollo* during estimation, together with their description and for what kind of model it is generated.

Table E.1: Elements inside a model object estimated by *Apollo*

Element	Type	Description	for models
apollo_beta	Named numeric vector	Starting value of parameters	all models
apollo_control	List	Main model settings, as validated by apollo_validateInputs	all models
apollo_draws	List of numeric matrix or 3-dim arrays	Contains the draws for models with mixing.	With mixing (NA otherwise)
apollo_fixed	Character vector	Name of fixed parameters. NULL for none.	all models
apollo_lcPars	Function	Function that groups parameters per latent class and calculate class allocation probabilities. As defined by the user.	With latent classes (NA otherwise)
apollo_probabilities	Function	Function that calculates the likelihood of the whole model. As defined by the user.	all models
apollo_randCoeff	Function	Function that generates the random coefficients used in the likelihood function. As defined by the user.	With mixing (NA otherwise)
avgCP	Numeric vector	Average likelihood for each individual at the estimated parameter values. As many elements as individuals in the sample.	all models
avgLL	Numeric vector	Average log-likelihood for each individual at the estimated parameter values. As many elements as individuals in the sample.	all models
bootstrapSE	Numeric scalar	Number of bootstrap repetitions used to calculate the bootstrap standard errors.	with bootstrap s.e. (0 otherwise)
code	Numeric scalar	Success code from maxLik (and optim). 0 means succesful estimation.	all models estimated with Maximum Likelihood.
componentReport	List	Contains the report on each model component dependant variable as a list of character vectors.	all models
constraints	List	List describing the constraints used during estimation in maxLik format.	With constraints estimated using Maximum Likelihood
control	List	Settings of the maximum likelihood optimisation algorithm.	all models estimated with Maximum Likelihood.
corrmat	matrix	Correlation matrix of (non-fixed) estimated parameters	all models

Element	Type	Description	for models
eigValue	Numeric scalar	Maximum eigenvalue of the Hessian matrix at the parameter estimates	all models estimated with Maximum Likelihood.
estimate	Numeric vector	Value of estimated parameters (including fixed ones)	all models estimated with Maximum Likelihood.
estimationRoutine	Scalar character	Name of the estimation algorithm	all models estimated with Maximum Likelihood.
fixed	Named logical vector	Only contains the names of non-fixed parameters, with a FALSE value associated to them. Used by maxLik, not Apollo.	all models estimated with Maximum Likelihood.
gradient	Named numerical vector	Gradient of the log-likelihood function at the estimated value of the parameters.	all models estimated with Maximum Likelihood.
gradientObs	Numerical matrix	Observation-level score matrix, i.e. the gradient of the log-likelihood function at the estimated parameter values at the observation level.	all models
hessian	Numerical matrix	Hessian (second derivative) of the log-likelihood function.	all models
hessianMethodsAttempted	Character vector	Name of the methods attempted to calculate the Hessian.	all models
hessianMethodUsed	Scalar character	Name of the methods used to calculate the Hessian reported in model$hessian.	all models
hessianScaling	Named numeric vector	Scaling used to calculate the Hessian. By default, they will be equal to the estimated value of the parameters.	all models
iterations	Numeric scalar	Number of iterations of the maximum likelihood estimation according to maxLik (not Apollo).	all models estimated with Maximum Likelihood.
last.step	List	List describing last unsuccessful iteration, if model$code=3 (i.e. if there was an error during maximum likelihood estimation).	all models estimated with Maximum Likelihood.

Element	Type	Description	for models
LL0	Numeric scalar	Log-likelihood of the null (equiprobable) model, if applicable (only for models with a single MNL, NL or CNL component)	all models. NA if not applicable.
Llout	Named numeric vector	Log-likelihood of each model component at the estimated parameter values. At the sample level.	all models
LLStart	Numeric scalar	Log-likelihood value at the starting value of the parameters.	all models
maximum	Numeric scalar	Log-likelihood value at the estimated parameter values.	all models
message	Scalar character	Message describing convergence (successful or not), as generated by maxLik.	all models estimated with Maximum Likelihood.
modelTypeList	Character vector	Vector indicating the type of model (MNL, NL, etc.) of each model component.	all models
nIndivs	Numeric scalar	Number of individuals in the database.	all models
nIter	Numeric scalar	Number of iterations necessary for convergence, as counted by Apollo.	all models estimated with Maximum Likelihood.
nObs	Numeric scalar	Number of observations in the database, counted as rows in the database.	all models
nObsTot	Numeric vector	Number of observations per model component, not counting excluding rows.	all models
objectiveFn	Function	Loglikelihood function generated by Apollo to mask apollo_probabilities from maxLik.	all models
Pout	Named list	Loglikelihood value of all models components at the estimated parameter values. At the sample level.	all models
robcorrmat	Numeric matrix	Robust correlation matrix of estimated parameters. Excludes fixed parameters.	all models
robse	Named numeric vector	Robust s.e. of estimated parameters. Includes fixed parameters.	all models
robvarcov	Numeric matrix	Robust covariance matrix of estimated parameters. Excludes fixed parameters.	all models
scaling	Named numeric vector	Scaling parameters used during model estimation (does not include fixed parameters)	all models
se	Named numeric vector	Standard errors of estimated parameters. Includes fixed parameters.	all models
startTime	Date object	POSIX date object recording the time when the full estimation processes started.	all models

Element	Type	Description	for models
timeEst	Numeric scalar	Time (in seconds) taken by the estimation algorithm (excludes pre and post-processing).	all models
timePost	Numeric scalar	Time (in seconds) taken by the post-estimation processing (including the calculation of the covariance matrix).	all models
timePre	Numeric scalar	Time (in seconds) taken by the pre-estimation processing (including validation and symbolic calculation of gradient).	all models
timeTaken	Numeric scalar	Time (in seconds) taken by the full estimation process (pre and post-estimation calculations included).	all models
type	Scalar character	Type of maximisation (algorithm) as defined by maxLik.	all models estimated with Maximum Likelihood.
varcov	Numeric matrix	Covariance matrix of estimated parameters. Excludes fixed parameters.	all models

Bibliography

Abou-Zeid, M., Ben-Akiva, M., 2014. Hybrid choice models, in: Hess, S., Daly, A. (Eds.), Handbook of Choice Modelling. Edward Elgar. chapter 17, pp. 383–412.

ALogit, 2016. ALOGIT 4.3. ALOGIT Software & Analysis Ltd. URL: www.alogit.com.

Axhausen, K.W., Hess, S., König, A., Abay, G., Bates, J.J., Bierlaire, M., 2008. State of the art estimates of the swiss value of travel time savings. Transport Policy 15, 173–185.

Ben-Akiva, M., Swait, J., 1986. The akaike likelihood ratio index. Transportation Science 20, 133–136.

Berndt, E., Hall, B., Hall, R., Hausman, J., 1974. Estimation and inference in non-linear structural models. Annals of Economic and Social Measurement 3/4, 653–665.

Bhat, C., 1997. An endogenous segmentation mode choice model with an application to intercity travel. Transportation Science 31, 34–48.

Bhat, C.R., 2003. Simulation estimation of mixed discrete choice models using randomized and scrambled Halton sequences. Transportation Research Part B 37, 837–855.

Bhat, C.R., 2008. The multiple discrete-continuous extreme value (mdcev) model: role of utility function parameters, identification considerations, and model extensions. Transportation Research Part B: Methodological 42, 274–303.

Bierlaire, M., 2003. BIOGEME: a free package for the estimation of discrete choice models. Proceedings of the 3rd Swiss Transport Research Conference, Monte Verità, Ascona.

Bierlaire, M., Thémans, M., Zufferey, N., 2010. A heuristic for nonlinear global optimization. INFORMS Journal on Computing 22, 59–70.

Bradley, M.A., Daly, A., 1996. Estimation of logit choice models using mixed stated-preference and revealed-preference information, in: Stopher, P.R., Lee-Gosselin, M. (Eds.), Understanding Travel Behaviour in an Era of Change. Elsevier, Oxford. chapter 9, pp. 209–231.

Broyden, C.G., 1970. The convergence of a class of double-rank minimization algorithms 1. general considerations. IMA Journal of Applied Mathematics 6, 76–90.

Busemeyer, J.R., Townsend, J.T., 1992. Fundamental derivations from decision field theory. Mathematical Social Sciences 23, 255–282.

Busemeyer, J.R., Townsend, J.T., 1993. Decision field theory: a dynamic-cognitive approach to decision making in an uncertain environment. Psychological Review 100, 432.

Calastri, C., Hess, S., Daly, A., Carrasco, J.A., 2017. Does the social context help with understanding and predicting the choice of activity type and duration? an application of the multiple discrete-continuous nested extreme value model to activity diary data. Transportation Research Part A: Policy and Practice 104, 1–20.

Calastri, C., Crastes dit Sourd, R., Hess, S., 2019. We want it all: experiences from a survey seeking to capture social network structures, lifetime events and short-term travel activity planning. Transportation forthcoming.

Chiou, L., Walker, J., 2007. Masking identification of discrete choice models under simulation methods. Journal of Econometrics 141, 683–703.

Chorus, C., 2010. A new model of random regret minimization. European Journal of Transport and Infrastructure Research 10, 181–196.

van Cranenburgh, S., Guevara, C.A., Chorus, C.G., 2015. New insights on random regret minimization models. Transportation Research Part A: Policy and Practice 74, 91–109. doi:10.1016/j.tra.2015.01.008.

Daly, A., 1987. Estimating Tree Logit models. Transportation Research Part B 21, 251–267.

Daly, A., Hess, S., de Jong, G., 2012a. Calculating errors for measures derived from choice modelling estimates. Transportation Research Part B 46, 333–341.

Daly, A., Zachary, S., 1978. Improved multiple choice models, in: Hensher, D.A., Dalvi, Q. (Eds.), Identifying and Measuring the Determinants of Mode Choice. Teakfields, London, pp. 335–357.

Daly, A.J., Hess, S., Patruni, B., Potoglou, D., Rohr, C., 2012b. Using ordered attitudinal indicators in a latent variable choice model: A study of the impact of security on rail travel behaviour. Transportation 39, 267–297.

Doornik, J.A., 2001. Ox: An Object-Oriented Matrix Language. Timberlake Consultants Press, London.

Dumont, J., Keller, J., 2019. RSGHB: Functions for Hierarchical Bayesian Estimation: A Flexible Approach. URL: https://CRAN.R-project.org/package=RSGHB. r package version 1.2.1.

Faure, H., Tezuka, S., 2000. Another random scrambling of digital (t,s)-sequences, in: Fang, K.T., Hickernell, F.J., Niederreiter, H. (Eds.), Monte Carlo and Quasi-Monte Carlo Methods. Springer, pp. 242–256.

Fletcher, R., 1970. A new approach to variable metric algorithms. The computer journal 13, 317–322.

Fosgerau, M., Mabit, S.L., 2013. Easy and flexible mixture distributions. Economics Letters 120, 206 – 210.

Geweke, J., 1992. Evaluating the accuracy of sampling-based approaches to the calculations of posterior moments. Bayesian statistics 4, 641–649.

Giergiczny, M., Dekker, T., Hess, S., Chintakayala, P., 2017. Testing the stability of utility parameters in repeated best, repeated best-worst and one-off best-worst studies. European Journal of Transport and Infrastructure Research 17, 457–476. URL: http://eprints.whiterose.ac.uk/118496/. © 2017, Author(s). Reproduced in accordance with the publisher's self-archiving policy.

Gilbert, P., Varadhan, R., 2016. numDeriv: Accurate Numerical Derivatives. URL: https://CRAN.R-project.org/package=numDeriv. r package version 2016.8-1.

Goldfarb, D., 1970. A family of variable metric updates derived by variational means, v. 24. Mathematics of Computation .

Greene, W.H., Hensher, D.A., 2013. Revealing additional dimensions of preference heterogeneity in a latent class mixed multinomial logit model. Applied Economics 45, 1897–1902.

Halton, J., 1960. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numerische Mathematik 2, 84–90.

Hancock, T.O., Hess, S., Choudhury, C.F., 2018. Decision field theory: Improvements to current methodology and comparisons with standard choice modelling techniques. Transportation Research Part B: Methodological 107, 18–40.

Hancock, T.O., Hess, S., Choudhury, C.F., 2019. An accumulation of preference: two alternative dynamic models for understanding transport choices. Submitted .

Henningsen, A., Toomet, O., 2011. maxlik: A package for maximum likelihood estimation in R. Computational Statistics 26, 443–458. URL: http://dx.doi.org/10.1007/s00180-010-0217-1, doi:10.1007/s00180-010-0217-1.

Hensher, D.A., Louviere, J.J., Swait, J., 1998. Combining sources of preference data. Journal of Econometrics 89, 197–221.

Hess, S., 2005. Advanced discrete choice models with applications to transport demand. Ph.D. thesis. Centre for Transport Studies, Imperial College London.

Hess, S., 2014. 14 latent class structures: taste heterogeneity and beyond, in: Handbook of choice modelling. Edward Elgar Publishing Cheltenham, pp. 311–329.

Hess, S., Bierlaire, M., Polak, J.W., 2007a. A systematic comparison of continuous and discrete mixture models. European Transport 36, 35–61.

Hess, S., Daly, A., 2014. Handbook of Choice Modelling. Edward Elgar publishers, Cheltenham.

Hess, S., Daly, A., Dekker, T., Cabral, M.O., Batley, R., 2017. A framework for capturing heterogeneity, heteroskedasticity, non-linearity, reference dependence and design artefacts in value of time research. Transportation Research Part B: Methodological 96, 126 – 149.

Hess, S., Polak, J.W., Daly, A., Hyman, G., 2007b. Flexible Substitution Patterns in Models of Mode and Time of Day Choice: New evidence from the UK and the Netherlands. Transportation 34, 213–238.

Hess, S., Rose, J.M., Hensher, D.A., 2008. Asymmetric preference formation in willingness to pay estimates in discrete choice models. Transportation Research Part E 44, 847–863.

Hess, S., Stathopoulos, A., Daly, A.J., 2012. Allowing for heterogeneous decision rules in discrete choice models: an approach and four case studies. Transportation 39, 565–591.

Hess, S., Train, K., 2011. Recovery of inter- and intra-personal heterogeneity using mixed logit models. Transportation Research Part B 45, 973–990.

Hess, S., Train, K., 2017. Correlation and scale in mixed logit models. Journal of Choice Modelling 23, 1–8.

Hess, S., Train, K., Polak, J.W., 2006. On the use of a Modified Latin Hypercube Sampling (MLHS) method in the estimation of a Mixed Logit model for vehicle choice. Transportation Research Part B 40, 147–163.

Hotaling, J.M., Busemeyer, J.R., Li, J., 2010. Theoretical developments in decision field theory: comment on Tsetsos, Usher, and Chater (2010). Psychological Review 117, 1294–1298.

Huber, P., 1967. The behavior of maximum likelihood estimation under nonstandard conditions, in: LeCam, L., Neyman, J. (Eds.), Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability. University of California Press, pp. 221–233.

Koppelman, F.S., Wen, C.H., 1998. Alternative Nested Logit Models: structure, properties and estimation. Transportation Research Part B 32, 289–298.

Krinsky, I., Robb, A., 1986. On approximating the statistical properties of elasticities. Review of Economics and Statistics 68, 715–719.

Lancsar, E., Louviere, J., Donaldson, C., Currie, G., Burgess, L., 2013. Best worst discrete choice experiments in health: Methods and an application. Social Science & Medicine 76, 74 – 82. URL: http://www.sciencedirect.com/science/article/pii/S0277953612007290, doi:https://doi.org/10.1016/j.socscimed.2012.10.007.

Lenk, P., 2014. Bayesian estimation of random utility models, in: Handbook of Choice Modelling. Edward Elgar Publishing. Chapters. chapter 20, pp. 457–497. URL: https://ideas.repec.org/h/elg/eechap/14820_20.html.

Louviere, J.J., Woodworth, G., 1983. Design and analysis of simulated consumer choice and allocation experiments: A method based on aggregate data. Journal of Marketing Research 20, 350–367.

Luce, R., 1959. Individual choice behavior: a theoretical analysis. J.Wiley and Sons, New York.

McFadden, D., 1974. Conditional logit analysis of qualitative choice behaviour, in: Zarembka, P. (Ed.), Frontiers in Econometrics. Academic Press, New York, pp. 105–142.

McFadden, D., 1978. Modelling the choice of residential location, in: Karlqvist, A., Lundqvist, L., Snickars, F., Weibull, J.W. (Eds.), Spatial Interaction Theory and Planning Models. North Holland, Amsterdam. chapter 25, pp. 75–96.

McFadden, D., 2000. Economic Choices. Nobel Prize Lecture. URL: https://www.nobelprize.org/uploads/2018/06/mcfadden-lecture.pdf.

McFadden, D., Train, K., 2000. Mixed MNL Models for discrete response. Journal of Applied Econometrics 15, 447–470.

Owen, A.B., 1995. Randomly permuted (t,m,s)-nets and (t,s)-sequences, in: Niederreiter, H., Shiue, J.S. (Eds.), Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing. Springer, New York, pp. 351–368.

Palma, D., 2016. Modelling wine consumer preferences using hybrid choice models: inclusion of intrinsic and extrinsic attributes. Ph.D. thesis. School of Engineering, Pontificia Universidad Católica de Chile.

Pinjari, A.R., Bhat, C., 2010a. A multiple discrete–continuous nested extreme value (mdcnev) model: formulation and application to non-worker activity time-use and timing behavior on weekdays. Transportation Research Part B: Methodological 44, 562–583.

Pinjari, A.R., Bhat, C.R., 2010b. An efficient forecasting procedure for kuhn-tucker consumer demand model systems: application to residential energy consumption analysis. Technical paper, Department of Civil and Environmental Engineering, University of South Florida , 263–285.

R Core Team, 2017. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Vienna, Austria. URL: https://www.R-project.org/.

Roe, R.M., Busemeyer, J.R., Townsend, J.T., 2001. Multialternative decision field theory: A dynamic connectionist model of decision making. Psychological Review 108, 370.

RStudio Team, 2015. Rstudio: Integrated development for r. RStudio, Inc., Boston, MA URL http://www.rstudio.com/ .

Shanno, D.F., 1970. Conditioning of quasi-newton methods for function minimization. Mathematics of computation 24, 647–656.

Sobol', I.M., 1967. On the distribution of points in a cube and the approximate evaluation of integrals. Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki 7, 784–802.

Train, K., 2009. Discrete Choice Methods with Simulation. second edition ed., Cambridge University Press, Cambridge, MA.

Train, K., Weeks, M., 2005. Discrete choice models in preference space and willingness-to-pay space, in: Scarpa, R., Alberini, A. (Eds.), Application of simulation methods in environmental and resource economics. Springer, Dordrecht. chapter 1, pp. 1–16.

Vovsha, P., 1997. Application of a Cross-Nested Logit model to mode choice in Tel Aviv, Israel, Metropolitan Area. Transportation Research Record 1607, 6–15.

Walker, J.L., Ben-Akiva, M., Bolduc, D., 2007. Identification of parameters in normal error component logit-mixture (neclm) models. Journal of Applied Econometrics 22, 1095–1125.

Weisberg, S., 2005. Applied Linear Regression. Third ed., Wiley, Hoboken NJ. URL: http://www.stat.umn.edu/alr.

Wen, C.H., Koppelman, F.S., 2001. The Generalized Nested Logit Model. Transportation Research Part B 35, 627–641.

Williams, H.C.W.L., 1977. On the Formulation of Travel Demand Models and Economic Evaluation Measures of User Benefit. Environment & Planning A 9, 285–344.

Yáñez, M.F., Cherchi, E., Heydecker, B., Ortúzar, J. de D., 2011. On the treatment of repeated observations in panel data: Efficiency of mixed logit parameter estimates. Networks and Spatial Economics 11, 393–418.

Index: *Apollo* syntax

This section provides an index covering each *Apollo* function and key lists/elements. Functions are differentiated by being followed by `()`. Elements of lists are preceded by `$`. For each function or list, a link to a syntax example is given where available. We also provide links to the use of the four datasets.

`apollo_attach()` 26
  syntax example 25
`apollo_avgInterDraws()` 73
  bayesian estimation 97
  syntax example 72
`apollo_avgIntraDraws()` 73
  bayesian estimation 97
  syntax example 72
`apollo_basTest()` 105
  syntax example 106
`apollo_beta` 22
  syntax example 23
`apollo_bootstrap()` 129
  `$bootstrap_settings` 130
    `$nRep` 130
    `$samples` 130
    `$seed` 130
  syntax example 131
  interruption 130
`apollo_choiceAnalysis()` 101
  `$choiceAnalysis_settings` 101
    `$alternatives` 101
    `$avail` 101
    `$choiceVar` 101
    `$explanators` 102
    `$rows` 102
  syntax example 102
`apollo_cnl()` 42
  `$cnl_settings` 42
    `$alternatives` 42
    `$avail` 42
    `$choiceVar` 42
    `$cnlNests` 42
    `$cnlStructure` 42
    `$componentName` 42
    `$rows` 42
    `$V` 42
  syntax example 43
`apollo_combineModels()` 85
  P in hybrid 85
  syntax example 87
`apollo_combineResults()` 119
  `$combineResults_settings` 119
    `$estimateDigits` 119
    `$modelNames` 119
    `$pDigits` 119
    `$printClassical` 119
    `$printPVal` 119
    `$printT1` 119
    `$sortByDate` 119
    `$tDigits` 119
`apollo_conditionals()` 115
  syntax example 116
`apollo_control` 20

$HB 20
$mixing 20, 69
$nCores 20, 69
$noDiagnostics 20, 88
$noValidation 20
$panelData 20
$seed 20
$weights 21
$workInLogs 20, 124, 145, 146
syntax example 20
apollo_deltaMethod() 112
$deltaMethod_settings 112
$multPar1 112
$multPar2 112
$operation 112
$parName1 112
$parName2 112
syntax example 113
apollo_detach() 26
syntax example 25
apollo_dft() 48
$dft_settings 48
$altStart 48
$alternatives 48
$attrScalings 48
$attrValues 48
$attrWeights 48
$avail 48
$choiceVars 48
$componentName 49
$procPars 48
$rows 49
syntax example 50
apollo_draws 69
$interDrawsType 69
$interNDraws 70
$interNormDraws 70
$interUnifDraws 70
$intraDrawsType 69
$intraNDraws 70
$intraNormDraws 70
$intraUnifDraws 70
Halton draws 69, 70
mlhs draws 70
pmc draws 69
Sobol draws 70
Sobol-Faure-Tezuka draws 70
Sobol-Owen draws 70
Sobol-Owen-Faure-Tezuka draws 70
syntax example 70
user-generated draws 71
apollo_drugChoiceData
example application 53, 90, 93, 122
apollo_el() 51, 88
$el_settings 51
$alternatives 51
$avail 51
$choiceVars 51
$componentName 51
$rows 51
$scales 51
$V 51
syntax example 53
apollo_estimate() 29
$estimate_settings 29
$bootstrapSE 31, 147
$bootstrapSeed 31
$constraints 30
$estimationRoutine 29
$hessianRoutine 30
$maxIterations 30, 146
$numDeriv_settings 31
$printLevel 30
$scaling 30, 147
$silent 31
$writeIter 30
syntax example 32
apollo_firstRow() 77, 116
syntax example 77, 116, 118
apollo_fitsTest() 111
$fitsTest_settings 111
$subsamples 111
syntax example 111
apollo_fixed 22

syntax example 23
apollo_HB 95
$constraintNorm 97
$fixedA 97
$fixedD 97
$gINFOSKIP 96
$gNCREP 96
$gNEREP 96
$hbDist 95
excluded arguments 97
syntax example 96
apollo_initialise() 18
syntax example 20
apollo_lc() 79
$lc_settings 79
$classProb 79
$inClassProb 79
P 79
P in hybrid 85
syntax example 78
apollo_lcConditionals() 117
limitations 117
syntax example 118
apollo_lcEM() 131
$lcEM_settings 132
$EMmaxIterations 132, 136
$postEM 132, 136
$silent 132, 136
$stoppingCriterion 132, 136
syntax example 134, 135, 137
apollo_lcPars() 76
lcPars 77
$beta 77
$pi_values 77, 79
syntax example 77
apollo_lcUnconditionals() 114
syntax example 118
apollo_llCalc() 103
syntax example 103
apollo_loadModel() 103
apollo_lrTest() 104
syntax example 104
apollo_mdcev() 57
$mdcev_settings 57
$V 58
$alpha 58
$alternatives 57
$avail 57
$budget 58
$componentName 58
$continuousChoice 57
$cost 58
$gamma 58
$minConsumption 58
$outside 58
$rows 58
$sigma 58
syntax example 59
apollo_mdcnev() 60
$mdcnev_settings 60
$mdcnevNests 60
$mdcnevStructure 60
syntax example 62
apollo_mixEM() 136
$mixEM_settings 136
$transforms 136
syntax example 138
apollo_mnl() 27
$mnl_settings 27
$alternatives 27
$avail 27
$choiceVar 27
$componentName 27
$V 27
rows 27
syntax example .. 25, 72, 77, 81, 87, 90, 93, 139
apollo_modeChoiceData
example application . 25, 40, 43, 45, 50, 87, 96, 102, 108, 110, 111
apollo_modelOutput() 33
$modelOutput_settings 34
$printChange 35
$printClassical 34

$printCorr 35
$printCovar 34
$printDataReport 34
$printFixed 34
$printFunctions 35
$printModelStructure 34
$printOutliers 35
$printPVal 34
$printT1 34
syntax example 36
apollo_nl() 40
$nl_settings 40
$alternatives 40
$avail 40
$choiceVar 40
$nlNests 40
$nlStructure 40
$V 40
syntax example 40
apollo_normalDensity() 56
$normalDensity_settings 56
$componentName 56
$mu 56
$outcomeNormal 56
$sigma 56
$xNormal 56
$rows 56
syntax example 94
apollo_ol() 54
$ol_settings 54
$coding 54
$componentName 54
$outcomeOrdered 54
$rows 54, 92
$tau 54
$V 54
syntax example 93
apollo_op() 55
$op_settings 55
$coding 55
$componentName 55
$outcomeOrdered 55
$rows 55
$tau 55
$V 55
apollo_outOfSample() 127
$outOfSample_settings 128
$nRep 128
$samples 128
$validationSize 128
syntax example 129
apollo_panelProd() 28, 142
bayesian estimation 97
syntax example 25
apollo_prediction() 105
$prediction_settings 106
$modelComponent 106
$nRep 106
$runs 106
output 107
prediction variability 107
syntax example 108
apollo_prepareProb() 28
syntax example 25
apollo_probabilities() 24
closure 28
debugging 122
functionality 63
initialisation 26
model definition 26
syntax example 25
apollo_randCoeff() 71
syntax example 71
apollo_readBeta() 22
inputModelName 23
overwriteFixed 23
syntax example 24
apollo_saveOutput() 33
$saveOutput_settings 34
$saveCorr 35
$saveCov 35
$saveEst 35
$saveModelObject 35
$writeF12 35

bayesian estimation 97
apollo_searchStart() 125, 145
$searchStart_settings 126
$apolloBetaMax 126
$apolloBetaMin 126
$bfgsIter 126
$dTest 126
$gTest 126
$llTest 126
$maxStages 126
$nCandidates 126
$smartStart 126
syntax example 127
apollo_sharesTest() 109
$sharesTest_settings 109
$alternatives 109
$choiceVar 109
$modelComponent 110
$newAltsOnly 110
$newAlts 110
$subsamples 110
syntax example 110
apollo_speedTest() 83
$speedTest_settings 83
$nCoresTry 83
$nDrawsTry 83
$nRep 83
syntax example 84
apollo_swissRouteChoiceData
example application ... 72, 78, 116, 118, 127, 129, 131, 134, 137
apollo_timeUseData
example application 59, 61, 62
apollo_unconditionals() 113
syntax example 116
apollo_validateInputs() 23
syntax example 24
apollo_weighting() 21, 29
functionality
estimate 63
output 63
raw 77
validate 63
zero_LL 63
model 32
$estimates 32
$robvarcov 33
$varcov 33
syntax example 32
P 26–28
joint models 85
syntax example 25