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

Stephane Hess* David Palma†

May 24, 2019

Abstract

The community of choice modellers has expanded substantially over recent years, covering many disciplines and encompassing users with very different levels of econometric and computational skills. This paper presents an introduction to *Apollo*, a powerful new freeware package for `R` that aims to provide a comprehensive set of modelling tools for both new and experienced users. *Apollo* also incorporates numerous post-estimation tools, allows for both classical and Bayesian estimation, and permits advanced users to develop their own routines for new model structures.

Keywords: Apollo; choice modelling; estimation; discrete continuous; software

1 Introduction

A brief explanation is needed as to our choice of the name *Apollo*. After failing to come up with a clever acronym, we turned 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 is already 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.

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 an ever 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. 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 have historically been computationally more powerful but may have more limitations in terms of available model structures or the possibility for customisation. On the other hand, freeware packages may have limitations in terms of performance and user friendliness but may benefit from more regular developments to accommodate new model structures.

*s.hess@leeds.ac.uk; Institute for Transport Studies and Choice Modelling Centre, University of Leeds (UK)
†D.Palma@leeds.ac.uk; Institute for Transport Studies and Choice Modelling Centre, University of Leeds (UK)

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.

Existing 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 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 either Windows or Linux, not both.

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[1] which does not rely on commercial statistical software as a host environment.

Big community: *Apollo* relies on `R`, a free software environment for statistical computing and graphics, which is very widely used across disciplines and works well across different operating systems (R Core Team, 2017)[2].

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

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

[2] In the remainder of this paper, 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). 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-3.5.1\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 in a file called `model.Rout`, which can be opened with any plain text editor. For the syntax shown in this paper, 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 and 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`). Secondly, for the predefined functions, the order of arguments passed to the function should be kept in the order specified in this paper.

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 the software permitting users to make mistakes and fixing them behind the scenes.

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 paper presents a brief introduction to the capabilities of *Apollo*. We focus on the case of a hybrid choice model so as to give an illustration of the functionalities of the package. We

illustrate this using both classical and Bayesian estimation and also explain a number of pre-estimation and post-estimation functions. Of course, in the context of an academic paper, we can only scrape the surface of the full level of detail, and furthermore, software packages change over time. For this reason, a more detailed manual (which also shows full details on function inputs) along with numerous examples (with data) and a user forum is available on the *Apollo* website (www.ApolloChoiceModelling.com). The syntax in the present paper is for *Apollo* version 0.0.8, but should remain forward compatible where not otherwise noted in the online manual. We strongly recommend prospective users to study the actual manual in detail rather than just relying on the short overview in the present paper.

This paper does not include any comparisons with other packages in terms of capabilities or speed, so as not to risk misrepresentations but also given the growing number of freeware tools, some of which we might not be aware of. 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.

The remainder of this paper is organised as follows. The following section briefly talks about installation. Section 3 discusses the econometric setup for our empirical example. Section 4 then presents the hybrid choice model application using classical estimation, with the Bayesian version covered in Section 5. A number of other functions are discussed in Section 6 before we present a summary in Section 7.

2 Installing *Apollo*

Apollo runs in R, with a minimum `R` version of 3.1.0. The easiest way to install *Apollo* is directly from CRAN using

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

This requires a working internet connection, but it has the benefit of installing all dependencies, i.e. other packages used by *Apollo*, automatically. Users of macOS (i.e. Apple computers) are advised to select the binary version of the package when prompted during installation. Alternatively, the source code of *Apollo* can be downloaded from www.ApolloChoiceModelling.com or from CRAN and *Apollo* can then be compiled from source.

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 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.

3 Empirical example setup

In this section, we describe the setup of the empirical example used in the remainder of this paper. The data file (`apollo_drugChoiceData.csv`) and the source files for the model using

classical (`hybrid_model_classical.r`) or Bayesian (`hybrid_model_bayesian.r`) estimation are available from the software website (`www.ApolloChoiceModelling.com`).

3.1 Data

We use a synthetic stated preference (SP) 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. A summary of the data is shown in Table 1.

3.2 Model specification

We next describe the structure of our hybrid choice model (see Abou-Zeid and Ben-Akiva, 2014, for a recent overview), where we look at the implementation of a model with a single latent variable but with additional random and deterministic heterogeneity in the utility functions[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. We additionally incorporate the same socio-demographic characteristics directly in the utility for the two branded alternatives, with brand-specific parameters, and also make the brand parameters for the Novum and Artemis brands random, in line with best practice for hybrid choice models (see theoretical discussions in Vij and Walker 2016 and an application in Kløjgaard and Hess 2014). We use an ordered logit model for the indicators, as discussed by Daly et al. (2012b).

Figure 1 presents the structure of the model. We use a subscript n to refer to individual decision makers (n=1,...,N; where N=1,000), t to refer to tasks (t=1,...,T; where T=10), and j to refer to alternatives (j=1,...,J; where J=4). We have individual specific responses to attitudinal questions (subscript n) and observation specific choice outcomes (subscript nt). Similarly, the error terms in the latent variable and the first two brand parameters are individual-specific. In Figure 1, we do not show the additional observation-specific error terms in the choice model and measurement models.

[3]Different specifications of the model would of course be possible, but this example is only included for the sake of illustration of the code.

Table 1: 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)

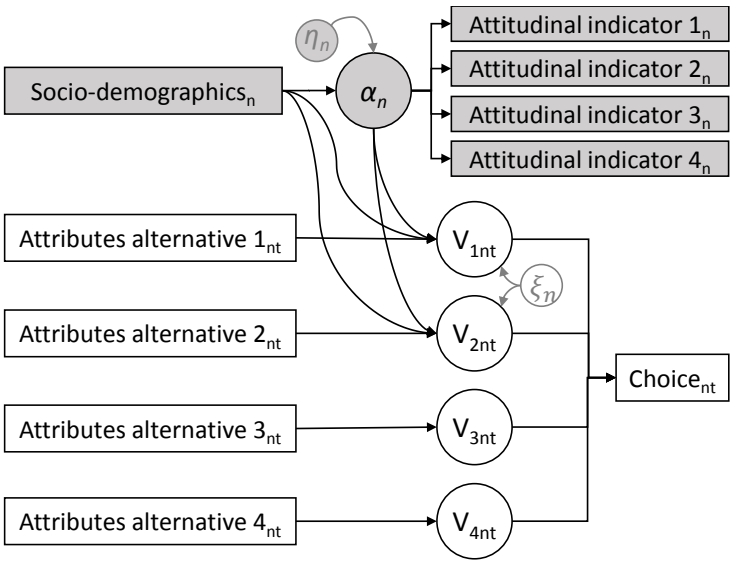

Figure 1: Structure of the hybrid choice model

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

$$\begin{aligned}\alpha_n = &\gamma_{\text{LV,regular user}} \cdot z_{n,\text{regular user}} \\ &+ \gamma_{\text{LV,univ. educ}} \cdot z_{n,\text{univ. educ}} \\ &+ \gamma_{\text{LV,over 50}} \cdot z_{n,\text{over 50}} \\ &+ \eta_n,\end{aligned} \tag{1}$$

where $z_{n,k}$ is a dummy variable which is equal to 1 if characteristic k applies to individual n, γ_{LV} 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 specification differs between the first two (branded) and last two (unbranded) alternatives. In particular, for the branded alternatives, where $j = 1, 2$, we have that the utility

(net of the extreme value term) in choice situation t for individual n is given by:

$$\begin{aligned}
V_{j,n,t} = {} & \beta_{n,\text{Artemis}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Artemis}\right) \\
& + \gamma_{\text{Artemis,regular user}} \cdot z_{n,\text{regular user}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Artemis}\right) \\
& + \gamma_{\text{Artemis,univ. educ}} \cdot z_{n,\text{univ. educ}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Artemis}\right) \\
& + \gamma_{\text{Artemis,over 50}} \cdot z_{n,\text{over 50}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Artemis}\right) \\
& + \beta_{n,\text{Novum}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Novum}\right) \\
& + \gamma_{\text{Novum,regular user}} \cdot z_{n,\text{regular user}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Novum}\right) \\
& + \gamma_{\text{Novum,univ. educ}} \cdot z_{n,\text{univ. educ}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Novum}\right) \\
& + \gamma_{\text{Novum,over 50}} \cdot z_{n,\text{over 50}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Novum}\right) \\
& + \beta_{\text{Switzerland}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Switzerland}\right) \\
& + \beta_{\text{Denmark}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Denmark}\right) \\
& + \beta_{\text{USA}} \cdot \left(x_{brand_{j,n,t}} == \texttt{USA}\right) \\
& + \beta_{\text{standard}} \cdot \left(x_{characteristic_{j,n,t}} == \texttt{standard}\right) \\
& + \beta_{\text{fast acting}} \cdot \left(x_{characteristic_{j,n,t}} == \texttt{fast acting}\right) \\
& + \beta_{\text{double strength}} \cdot \left(x_{characteristic_{j,n,t}} == \texttt{double strength}\right) \\
& + \beta_{side_effects} \cdot x_{side_effects_{j,n,t}} \\
& + \beta_{price} \cdot x_{price_{j,n,t}} \\
& + \lambda \cdot \alpha_n
\end{aligned} \tag{2}$$

while for $j = 3, 4$, we have:

$$\begin{aligned}
V_{j,n,t} = {} & \beta_{\text{BestValue}} \cdot \left(x_{brand_{j,n,t}} == \texttt{BestValue}\right) \\
& + \beta_{\text{Supermarket}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Supermarket}\right) \\
& + \beta_{\text{PainAway}} \cdot \left(x_{brand_{j,n,t}} == \texttt{PainAway}\right) \\
& + \beta_{\text{USA}} \cdot \left(x_{brand_{j,n,t}} == \texttt{USA}\right) \\
& + \beta_{\text{India}} \cdot \left(x_{brand_{j,n,t}} == \texttt{India}\right) \\
& + \beta_{\text{Russia}} \cdot \left(x_{brand_{j,n,t}} == \texttt{Russia}\right) \\
& + \beta_{\text{standard}} \cdot \left(x_{characteristic_{j,n,t}} == \texttt{standard}\right) \\
& + \beta_{\text{fast acting}} \cdot \left(x_{characteristic_{j,n,t}} == \texttt{fast acting}\right) \\
& + \beta_{side_effects} \cdot x_{side_effects_{j,n,t}} \\
& + \beta_{price} \cdot x_{price_{j,n,t}}
\end{aligned} \tag{3}$$

A number of differences arise between Equation 2 and 3. Some of these are a result of the data, namely in that the possible brands and countries differ between the first two and last two alternatives and the double strength characteristic is only possible for the first two brands.

In addition, we allow for socio-demographic shifts only in the sensitivities to the two branded products, and allow the utilities for Artemis and Novum to follow a Normal distribution across individuals, with $\beta_{n,\text{Artemis}} \sim N(\mu_{\text{Artemis}}, \sigma_{\text{Artemis}})$ and $\beta_{n,\text{Novum}} \sim N(\mu_{\text{Novum}}, \sigma_{\text{Novum}})$. This means that e.g. $\beta_{n,\text{Artemis}} = \mu_{\text{Artemis}} + \sigma_{\text{Artemis}} \cdot \xi_{n,\text{Artemis}}$, where $\xi_{n,\text{Artemis}}$ is a standard Normal error term, distributed across individuals. In terms of normalisation, we use dummy coding, setting β_{PainAway}, β_{USA} and β_{standard} to zero. Finally, the impact of the latent variable α_n is solely on the first two utilities, i.e. the branded products.

For ease of notation, we group together the parameters into vectors, such that β_n combines the various β terms from Equations 2 and 3, where this is person specific due to the random heterogeneity in $\beta_{n,\text{Artemis}}$ and $\beta_{n,\text{Novum}}$. We also define γ_V to group together the deterministic heterogeneity used in Equations 2 and 3. We thus get that the likelihood of the observed sequence of T_n choices for person n, is given by:

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

where $j^*_{n,t}$ is the alternative chosen as the best one by respondent n in task t, and where this probability is distributed across the random components in β_n and α_n.

The latent variable α_n is also used to explain the value of the four attitudinal questions, where, with the ordered logit model, we have that:

$$L_{I_n,ordered}(\tau, \zeta, \alpha_n) = \prod_{i=1}^{4} \left(\sum_{s=1}^{S} \delta_{(I_{n,i}=s)} \left[\frac{e^{\tau_{i,s} - \zeta_i \alpha_n}}{1 + e^{\tau_{i,s} - \zeta_i \alpha_n}} - \frac{e^{\tau_{i,s-1} - \zeta_i \alpha_n}}{1 + e^{\tau_{i,s-1} - \zeta_i \alpha_n}} \right] \right), \tag{5}$$

where ζ_i is an estimated parameter that measures the impact of α_n on the attitudinal indicator I_i, and $\tau_{i,\cdot}$ is a vector of threshold parameters for this indicator. Again, τ and ζ are defined to group together the various parameters used in the measurement model. Equation 5 is distributed across the random component in α_n.

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

$$LL(\Omega) = \sum_{n=1}^{N} log \int_{\beta_n} \int_{\alpha_n} L_{C_n}(\beta_n, \gamma_V, \alpha_n) \, L_{I_n,ordered}(\tau, \zeta, \alpha_n) \, f(\alpha_n) \, g(\beta_n) \, \mathrm{d}\alpha_n \mathrm{d}\beta_n, \tag{6}$$

where Ω combines all model parameters. This log-likelihood function requires integration over the random component in the latent variable, and the random component in β_n. In classical estimation this integral is calculated using Monte Carlo methods, instead, when Bayesian methods are used, the Metropolis-Hasting algorithm is employed.

4 Hybrid choice model example: classical estimation

In this section, we look at classical estimation of the hybrid choice model defined in Section 3. The structure of an *Apollo* model file varies across specifications, but a general overview is shown in Figure 2.

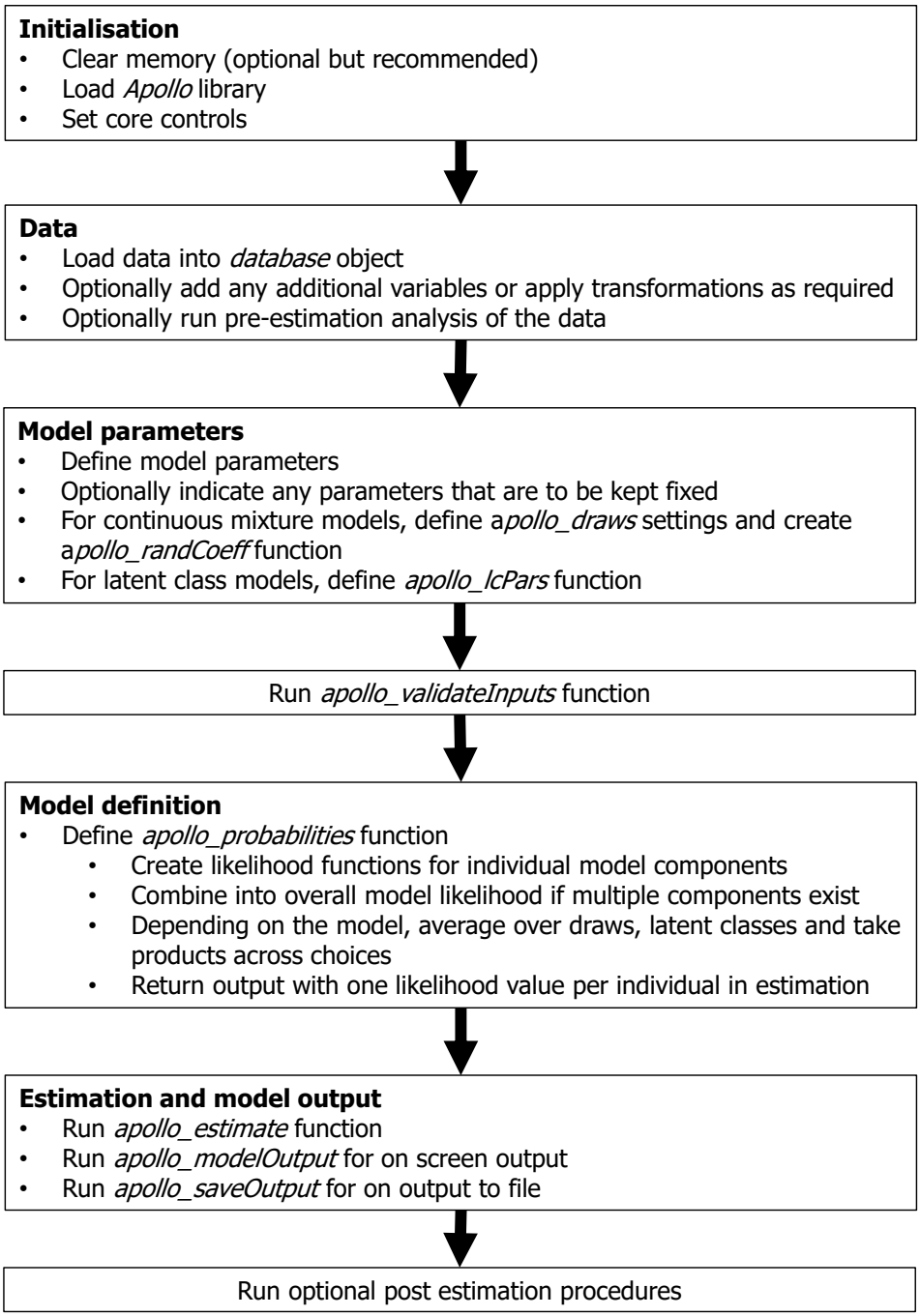

Figure 2: General structure of an *Apollo* model file

4.1 Code initialisation

The first step in every use of *Apollo* is to initialise the code. These steps are illustrated in Figure 3. In an optional step, we clear the memory/workspace by using `rm(list = ls())`. Every time users want to estimate a model, they should 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)
```

This is followed by calling the `apollo_initialise` function, which 'detaches' variables[4] and makes sure that output is directed to the console rather than a file. This 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 inside a list called `apollo_control`. In our case, we set the name of the model in `modelName` (where any output files will use this name too), give a brief description of the model in `modelDescr` (for use in the output) and provide in `indivID` the name (in quotes) of the column in the data which contains the identifier variable for individual decision makers. We also indicate that our model uses continuous random distributions by setting `mixing` to `TRUE`, and enable multi-core estimation by setting the number of cores in `nCores`. 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  = "hybrid_model_classical",
  modelDescr = "Hybrid choice model on drug choice data, classical estimation",
  indivID    = "ID",
  mixing     = TRUE,
  nCores     = 25
)
```

Figure 3: Code initialisation

Only 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 - details on these optional settings are given in the online manual.

4.2 Reading and processing the data

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

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

modellers refer to this as the *wide* format, as opposed to the *long* format, which would have one row per alternative.

The next step is to load the data into an object called `database`, in our case from a *csv* file, via the command:

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

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 (see R Core Team 2017 for more details). 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 4.5), all data needs to be combined in a single `R` data.frame called `database`. Thirdly, any new variables created by the user inside `R` need to be created in the database object rather than the global environment, and this needs to happen prior to validating the user inputs.

With labelled choice data (or even unlabelled data), 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 `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 and the corresponding value for the choice variable.
- `avail`: A list containing one element with availabilities per alternative.
- `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.

We illustrate this process for our example in Figure 4. In most cases, this function would be used for labelled data, where a given alternative refers to a specific product. In our dataset, we have five different brands which are associated with different alternatives in different choice tasks. Some transformations are needed before running our analysis. We first define these five brands as being the *pseudo-alternatives* for our analysis of choices. We then create availabilities for these alternatives, where the `Artemis` alternative is for example available if the brand in the first or second alternative corresponds to `Artemis` (remembering that this brand is only possible for the first two alternatives in the data). We then similarly create a *pseudo choice variable*, where e.g. the new alternative 11, which corresponds to `Artemis`, is chosen if either the first or second alternative in the data is given as the highest ranked (i.e. best) and if its brand is equal

```
choiceAnalysis_settings <- list(
  alternatives = c(Artemis=11, Novum=12, BestValue=21, Supermarket=22, PainAway=23),
  avail        = with(database,list(
                      Artemis=(brand_1=="Artemis")|(brand_2=="Artemis"),
                      Novum=(brand_1=="Novum")|(brand_2=="Novum"),
                      BestValue=(brand_3=="BestValue")|(brand_4=="BestValue"),
                      Supermarket=(brand_3=="Supermarket")|(brand_4=="Supermarket"),
                      PainAway=(brand_3=="PainAway")|(brand_4=="PainAway"))),
  choiceVar    = with(database,
                   (11*((best==1)*(brand_1=="Artemis")+(best==2)*(brand_2=="Artemis"))
                   +12*((best==1)*(brand_1=="Novum")+(best==2)*(brand_2=="Novum"))
                   +21*((best==3)*(brand_3=="BestValue")+(best==4)*(brand_4=="BestValue"))
                   +22*((best==3)*(brand_3=="Supermarket")+(best==4)*(brand_4=="Supermarket"))
                   +23*((best==3)*(brand_3=="PainAway")+(best==4)*(brand_4=="PainAway")))),
  explanators  = database[,c("regular_user","university_educated","over_50")]
)

apollo_choiceAnalysis(choiceAnalysis_settings, apollo_control, database)
```

	Mean for regular_user if chosen	Mean for regular_user if not chosen	t-test for difference
Artemis	0.315	0.3877	6.94
Novum	0.3199	0.368	4.48
BestValue	0.4329	0.3364	-5.8
Supermarket	0.4417	0.3345	-6.82
PainAway	0.4387	0.334	-5.43

Figure 4: Running `apollo_choiceAnalysis` (syntax and excerpt of output)

to `Artemis`. Finally, we define three explanatory variables for our analysis, which are the three socio-demographic characteristics[5].

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.). The results of this process are shown in the bottom part of Figure 4 (for the first explanator only), clearly highlighting differences across brands in terms of the share of choices made by regular users. 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)`.

4.3 Model parameters

The user next needs to define the parameters and their starting values, and also indicate whether any of the parameters are to be kept fixed at their starting values. This process is illustrated in Figure 5. 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

[5]The syntax used in this function is standard `R` syntax, where the use of `with` allows us to omit `database$` when referring to individual attributes.

values. Typically users will use one line per parameter, we have placed two on each line simply for formatting reasons. In our case, we then keep some of these parameters 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.

```
apollo_beta = c(mu_brand_Artemis          = 0, sig_brand_Artemis          = 0,
                gamma_Artemis_reg_user    = 0, gamma_Artemis_university   = 0,
                gamma_Artemis_age_50      = 0, mu_brand_Novum             = 0,
                sig_brand_Novum           = 0, gamma_Novum_reg_user       = 0,
                gamma_Novum_university    = 0, gamma_Novum_age_50         = 0,
                b_brand_BestValue         = 0, b_brand_Supermarket        = 0,
                b_brand_PainAway          = 0, b_country_CH               = 0,
                b_country_DK              = 0, b_country_USA              = 0,
                b_country_IND             = 0, b_country_RUS              = 0,
                b_country_BRA             = 0, b_char_standard            = 0,
                b_char_fast               = 0, b_char_double              = 0,
                b_risk                    = 0, b_price                    = 0,
                gamma_LV_reg_user         = 0, gamma_LV_university        = 0,
                gamma_LV_age_50           = 0, lambda                     = 1,
                zeta_quality              = 1, zeta_ingredient            = 1,
                zeta_patent               = 1, zeta_dominance             = 1,
                tau_quality_1             =-2, tau_quality_2              =-1,
                tau_quality_3             = 1, tau_quality_4              = 2,
                tau_ingredients_1         =-2, tau_ingredients_2          =-1,
                tau_ingredients_3         = 1, tau_ingredients_4          = 2,
                tau_patent_1              =-2, tau_patent_2               =-1,
                tau_patent_3              = 1, tau_patent_4               = 2,
                tau_dominance_1           =-2, tau_dominance_2            =-1,
                tau_dominance_3           = 1, tau_dominance_4            = 2)

apollo_fixed = c("b_brand_PainAway", "b_country_USA", "b_char_standard")
```

Figure 5: Setting names and starting values for model parameters, and fixing some parameters to their starting values

4.4 Draws and random parameters

The next step concerns the generation of draws for random distributions. In our case, we need to produce normally distributed inter-individual draws for $\beta_{n,\text{Artemis}}$, $\beta_{n,\text{Novum}}$ and the random component η_n in the latent variable α_n. Draws are generated by *Apollo* whenever `mixing==TRUE` in `apollo_control`, using the settings defined in a list called `apollo_draws`.

We first set the type of draws in `interDrawsType`, where different pre-defined types of draws are available in *Apollo*, including (but not limited to) `pmc` for pseudo-Monte Carlo draws, `halton` for Halton draws Halton (1960) and `mlhs` for MLHS draws (Hess et al., 2006). The number of draws per individual is set in `interNDraws` for inter-individual draws. 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` or `interUnifDraws`, respectively. These two distributions (standard Normal and Uniform between 0 and 1) can later be transformed to any other distribution by the user inside `apollo_randCoeff`.

The process used for this is illustrated in the first part of Figure 6. In our example, we create three sets of draws, each time with 500 draws per individual, where these are transformed into standard Normal distributions. The settings that are not used in our example, namely any intra-individual draws (cf. Section 6.3.1) and uniformly distributed inter-individual draws, are omitted (or alternatively could be left empty, or set to zero).

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 with uniformly distributed draws in `apollo_draws$interDrawsType` instead of providing one of the specific types of draws listed above. Using the example from Figure 6, the user would need 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 user also still needs to specify `interNDraws`.

```
apollo_draws = list(
  interDrawsType="halton",
  interNDraws=500,
  interNormDraws=c("eta","xi_Artemis","xi_Novum"))

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

  randcoeff[["LV"]] = gamma_LV_reg_user*regular_user + gamma_LV_university*university_educated +
      ↪ gamma_LV_age_50*over_50 + eta
  randcoeff[["b_brand_Artemis"]] = mu_brand_Artemis + sig_brand_Artemis * xi_Artemis +
      ↪ gamma_Artemis_reg_user*regular_user + gamma_Artemis_university*university_educated +
      ↪ gamma_Artemis_age_50*over_50
  randcoeff[["b_brand_Novum"]] = mu_brand_Novum + sig_brand_Novum * xi_Novum + gamma_Novum_reg_user*
      ↪ regular_user + gamma_Novum_university*university_educated + gamma_Novum_age_50*over_50

  return(randcoeff)
}
```

Figure 6: Defining draws and random parameters

After defining the draws, the next step concerns the actual definition of those coefficients in the model that follow a random distribution, in our case the latent variable and the two random brand coefficients. For this, the user creates an additional function, namely `apollo_randCoeff`. Just as with `apollo_probabilities`, which we will see below, this is a function that the user does not call but which the user defines. 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 is illustrated in the second part of Figure 6, where the correspondence with e.g. Equation 1 for the `LV` component 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.

In the example used here, the three random coefficients all follow Normal distributions. If the user for example wants to use a negative Lognormal distribution for β_1, a symmetrical Triangular distribution for β_2 and a Johnson S_B distribution for β_3, then this could be specified as:

$$\texttt{randcoeff[["beta1"]]} = -\texttt{exp(log\_b1\_mu} + \texttt{log\_b1\_sig} * \texttt{draws1)}$$
$$\texttt{randcoeff[["beta2"]]} = \texttt{b2\_a} + \texttt{b2\_b} * (\texttt{draws2a} + \texttt{draws2b})$$
$$\texttt{randcoeff[["beta3"]]} = \texttt{b3\_a} + \texttt{b3\_b} * 1/(1 + \texttt{exp}(-(\texttt{log\_b3\_mu} + \texttt{log\_b3\_sig} * \texttt{draws3})))$$

where `draws1` and `draws3` need to be defined in `interNormDraws` and `draws2a` and `draws2b` in `interUnifDraws`. The other terms are parameters to estimate, namely means (e.g. `log_b1_mu`), standard deviations means (`log_b1_sig`), bounds (`b2_a`) and ranges (`b2_b`).

4.5 Validation and preparing user inputs

The final step in preparing the code and data for model estimation or application is to call `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. The function also looks for a number of optional objects and sets default values for any missing ones as illustrated in Figure 7.

Before returning the list of model inputs, `apollo_validateInputs` runs a number of validation tests on the `apollo_control` settings and the `database`. It also sorts the data by ID and adds an extra column called `apollo_sequence` which is a running index of observations for each individual in the data. The list that is returned, `apollo_inputs`, contains the validated versions of the various objects mentioned above, e.g. `database`.

```
> apollo_inputs = apollo_validateInputs()
Missing setting for workInLogs, set to default of FALSE
Missing setting for seed, set to default of 13
Missing setting for HB, set to default of FALSE
Several observations per individual detected based on the value of ID.
  Setting panelData set to TRUE.
All checks on apollo_control completed.
All checks on data completed.
Generating inter draws ... Done
```

Figure 7: Running `apollo_validateInputs`

4.6 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 hybrid choice model in Figure 8. As with `apollo_randCoeff`, this is a function 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. 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 6.2. Additionally, the user can define his/her own models. Finally, this part of the code can contain either a single model or multiple individual model components, as shown in our example.

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 4.9.2. The value used depends on which function makes the call to `apollo_probabilities` and is controlled internally. The function returns probabilities, where the specific format depends on `functionality`.

In the following three subsections, we look at the individual components of the code shown in Figure 8.

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

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

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

  ### Likelihood of choices
  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[['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 )

  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
  )

  P[["choice"]] = apollo_mnl(mnl_settings, functionality)

  ### Likelihood of indicators
  ol_settings1 = list(outcomeOrdered=attitude_quality,
                      V=zeta_quality*LV,
                      tau=c(tau_quality_1, tau_quality_2, tau_quality_3, tau_quality_4),
                      rows=(task==1))

  ...

  ol_settings4 = list(outcomeOrdered=attitude_dominance,
                      V=zeta_dominance*LV,
                      tau=c(tau_dominance_1, tau_dominance_2, tau_dominance_3, tau_dominance_4),
                      rows=(task==1))

  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 the whole model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  P = apollo_panelProd(P, apollo_inputs, functionality)
  P = apollo_avgInterDraws(P, apollo_inputs, functionality)
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 8: The `apollo_probabilities` function for a hybrid choice model using classical estimation (excerpt)

4.6.1 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 `brand_1` instead of `database$brand_1`. 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))
```

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.

4.6.2 Model definition

We create a list `P` which will in the end have five components, namely the probabilities from the choice model and the probabilities of the four measurement models. Conditional on the random parameters, the choice model component of the hybrid model is of the MNL type.

For a model with just a single component, 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. In our case, our model contains several individual model components, and we thus use `P[["choice"]]` for this component instead of `P[["model"]]`. The function takes as its core input a list called `mnl_settings` which has four compulsory inputs and one optional input. We will now look at these in turn.

`alternatives:` A vector containing for each alternative its name and the value associated with it in the dependent variable in the data.

`avail:` A list containing one element per alternative, using the same names as in `alternatives`, with each entry being a vector of values of the same length as the number of observations (i.e. a column from the data) or a scalar of 1 if an alternative is always available. A user can also set `avail=1` which implies full availability for all alternatives.

`choiceVar:` A vector of length equal to the number of observations, containing the chosen alternative for each observation.

`V:` A list 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.

rows: An optional vector 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, i.e. they will not contribute to model estimation.

As seen in Figure 8, each alternative is given a name, and in the setting `alternatives` inside `mnl_settings`, we associate each name with the corresponding value for the choice variable in the data (going from 1 to 4). Our four alternatives always have full availability, and we use the variable `best` for `choiceVar`. 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 is that they are then copied into a component called `V` inside `mnl_settings`. We define the utilities in line with Equations 2 and 3, where we can directly incorporate the random components `LV`, `b_brand_Artemis` and `b_brand_Novum` defined in `apollo_randCoeff`. After creating the settings for the MNL component, we make the call to `apollo_mnl` to compute the choice probabilities.

In the example shown here (and the rest of 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 Figure 9 for a simple example, where we have 100 alternatives, and where the utility includes two attributes (`x1` and `x2`). Values for these exist in the data for each alternative, with for example `x1_1` being the value for the first attribute for the first alternative, and there is also a vector of availabilities for each alternative, e.g. `av1` for the first alternative.

```
J = 100
V = list()
for(j in 1:J) V[[paste0("alt",j)]] = b1*database[,paste0("x1_",j)] + b2*database[,paste0("x2_",j)]

mnl_settings = list(
  alternatives  = setNames(1:J, names(V)),
  avail         = setNames(database[,paste0("av",1:J)], names(V)),
  choiceVar     = choice,
  V             = V
)
```

Figure 9: Defining utilities for large choicesets

We next turn to the measurement model component of our hybrid model. We have four indicators in our model, and use an ordered logit model for each. In *Apollo*, the `apollo_ol` function for a model with just one component is called as follows:

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

In our case, we have four indicators, so use for example `P[["indic_quality"]]`. The contents of `ol_settings` are a little different from MNL. In particular, we have:

outcomeOrdered: A vector indicating the level selected for the ordinal variable in each observation. This will usually be a column (variable) inside `database`.

V: A numeric vector containing the explanatory variable used in the ordered logit model, i.e. the utility.

`tau`: A vector containing the names of the threshold parameters that are used in the model. These should have one fewer element than the number of possible values for the dependent variable, as extreme thresholds are assumed to be $-\infty$ and $+\infty$.

`coding`: An optional argument of numeric or character vector type which is only required as an input if the dependent variable is in string format or does not use an incremental coding from 1 to a value equal to the number of possible values for the dependent variable `outcomeOrdered`.

`rows`: The optional `rows` argument already described for MNL.

In Figure 8, we first create the four lists of settings for each individual ordered logit model, before making the four calls to `apollo_ol`. The utility is given by $\zeta\alpha_n$, and we estimate four thresholds for each indicator. As the answers to each attitudinal question are given once per respondent but are repeated in each line in the data for that respondent, we include `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.

The list `P` now contains five individual components, and the call to `apollo_combineModels` combines these into a joint model. This is called as follows:

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

This function takes the list `P` which contains several individual model components and produces a combined model, 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.

4.6.3 Function output

Three further final steps are required. We first call `apollo_panelProd` which multiplies the probabilities across individual choice observations for the same individual, thus recognising the repeated choice nature of our data, in our case the product across choice tasks in Equation 4 (noting that for the indicators, only the first row is used for each person). This function is only to be used in the presence of multiple observations per individual, and is called as:

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

We next need to average across the inter-individual draws. The function `apollo_avgInterDraws` is called as:

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

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[6]. 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`.

4.7 Model estimation

4.7.1 Deciding on number of cores and draws

Apollo allows for multi-threaded estimation for classical estimation[7], leading to significant estimation speed improvements for some models (especially those using few iterations that each take a long time). 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 number of draws and cores to try, as well as the number of times `apollo_probabilities` is calculated

[6]The user has the possibility of using weights by including the setting `weights` in `apollo_control`. Weights are only used in estimation, and if the user wants to use weights, then in addition to including the setting in `apollo_control`, the function `apollo_weighting` needs to be called prior to `apollo_prepareProb`. This is called as `P = apollo_weighting(P, apollo_inputs, functionality)`.

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

to ensure stable results. We illustrate the use of this function in Figure 10, which shows major benefits by moving away from using a single core. We again create the list of settings prior to the call to `apollo_speedTest` just for ease of coding. 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 (*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.

4.7.2 Actual model 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 5. The function is called via:

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

where we have already covered the first four arguments. The final argument, `estimate_settings`, is optional and default values will be used when it is not provided. As discussed in the online manual, this allows the user to change the optimisation and Hessian routines, set the maximum number of iterations, disable writing of intermediate results to file and control the level of detail for on screen printing. The writing of iteration-level estimates into a file in the working directory happens by default when using BFGS for estimation, and allows the user to monitor progress during estimation, which is useful especially for complex models.

Users can also impose constraints in estimation and apply scaling to individual model parameters. 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 - they are scaled by *Apollo* prior to estimation and then returned to their original scale after convergence. For Bayesian estimation, the scales are applied to the posterior parameter chains. Figure 11 illustrates what happens when running `apollo_estimate` on our model, where only part of the output is shown. After some initial checks (not shown here), *Apollo* splits the data across the cores. This is followed by the main estimation process and finally the calculation of the

```
> speedTest_settings=list(nDrawsTry = c(250, 500, 1000),nCoresTry = c(1,5,10,15,20),nRep = 10)

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

         ___Draws___                    sec/
nCores  inter  intra   progress    LLCall RAM(MB)
     1    250      0 ..........      0.91   222.6
     5    250      0 ..........      0.38   447.1
    10    250      0 ..........      0.24   613.6
    15    250      0 ..........      0.21   778.8
    20    250      0 ..........      0.19   944.6
     1    500      0 ..........      1.89   279.8
     5    500      0 ..........      0.61   561.8
    10    500      0 ..........      0.35   727.8
    15    500      0 ..........      0.28   894.1
    20    500      0 ..........      0.26  1059.8
     1   1000      0 ..........      4.11   394.2
     5   1000      0 ..........      1.05   790.6
    10   1000      0 ..........      0.64   956.2
    15   1000      0 ..........      0.46  1122.1
    20   1000      0 ..........      0.42  1288.2

Summary of results (sec. per call to LL function)
        draws250 draws500 draws1000
cores1    0.9093   1.8869    4.1108
cores5    0.3830   0.6115    1.0508
cores10   0.2370   0.3524    0.6428
cores15   0.2061   0.2836    0.4600
cores20   0.1938   0.2582    0.4180
```

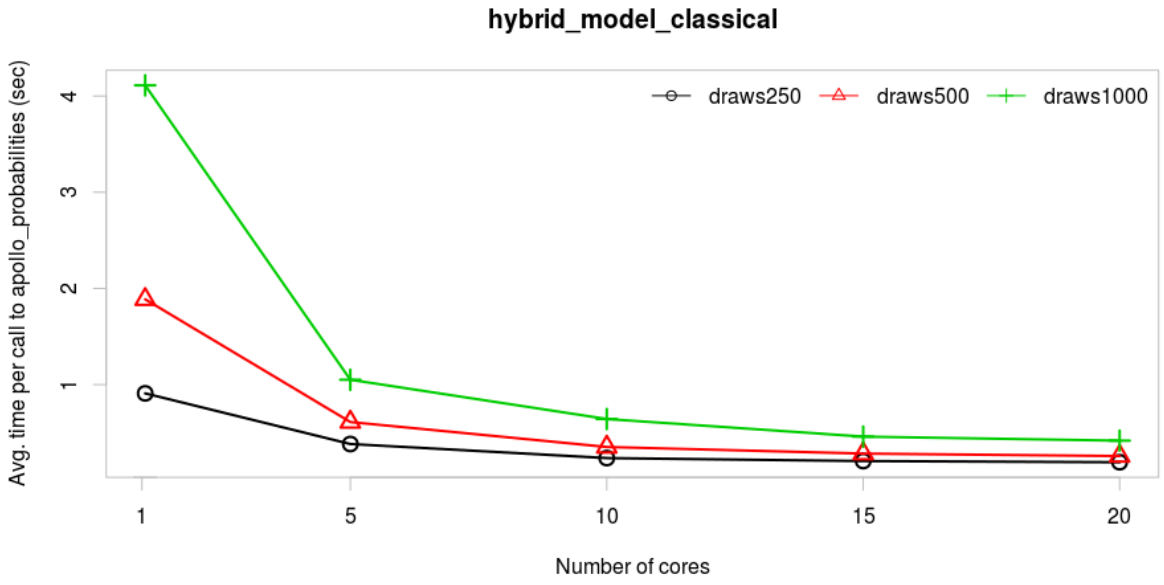

Figure 10: Running `apollo_speedTest`

Hessian. Prior to that step, which can take a long time in complex models, the code also prints out the final estimates[8]. We can see from Figure 11 that the estimation uses minimisation of the negative of the log-likelihood, which is of course equivalent to maximisation of the log-likelihood itself.

[8] At the time of writing, *Apollo* relies on numerical gradients and Hessians only.

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`).

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

...

Attempting to split data into 25 pieces.
Number of observations per worker (thread):
 worker_1  worker_2  worker_3  worker_4  worker_5  worker_6  worker_7  worker_8  worker_9 worker_10
      400       400       400       400       400       400       400       400       400       400
worker_11 worker_12 worker_13 worker_14 worker_15 worker_16 worker_17 worker_18 worker_19 worker_20
      400       400       400       400       400       400       400       400       400       400
worker_21 worker_22 worker_23 worker_24 worker_25
      400       400       400       400       400
 169.2MB of RAM in use before splitting.
Splitting draws..... Done. 283.8MB of RAM in use.
Splitting database..... Done. 285.3MB of RAM in use.
Creating workers and loading libraries... Done. 1111.4MB of RAM in use.
Copying data to workers... Done. 1112.8MB of RAM in use (max was 1233.5MB)

Starting main estimation
Initial function value: -19734.5
Initial gradient value:
        mu_brand_Artemis         sig_brand_Artemis    gamma_Artemis_reg_user gamma_Artemis_university
            8.416564e+02             -1.777880e-01             2.067306e+02             2.926585e+02
...
         tau_dominance_4
            4.056565e+00
initial  value 19734.497781
iter   2 value 19647.274815
iter   3 value 18787.804991
...
iter  58 value 16537.665650
final  value 16537.665650
converged

Estimated values:
                          [,1]
mu_brand_Artemis        1.4613
sig_brand_Artemis       0.0034
...
tau_dominance_4         2.1254

Computing covariance matrix using numDeriv package.
 (this may take a while)
0%....25%....50%....75%....100%
Hessian calculated with numDeriv will be used.
Calculating LL(0)... -20300.7
Updating inputs... Done.
Calculating LL of each model component...Done.
```

Figure 11: Running `apollo_estimate` on hybrid choice model

4.8 Model outputs

Two separate functions are used for outputting results, 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. The functions are called via:

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

and

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

The two lists `modelOutput_settings` and `saveOutput_settings` are optional and give the user the ability to control the level of detail produced in the output, for example enabling or disabling the reporting of classical standard errors (in addition to robust ones), p-values, t-ratios against 1 (in addition to against 0) and covariance and correlation matrices. The user also has control over what output files are produced when using `apollo_saveOutput`. An example of the on screen output is shown in Figure 12. For `apollo_saveOutput`, a text file containing output using the above settings will be produced, using a filename corresponding to `apollo_control$modelName`. This is in addition to files contains estimates, covariance and correlation matrices, unless instructed not to do so. The default settings imply a more verbose output for the log file as opposed to the on screen output.

4.9 Processing of results

We now look at some of the additional functions that are provided in *Apollo* to allow the user to analyse model results after estimation.

4.9.1 Delta method

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 sraightforwardly and accurately with the Delta method, as discussed by Daly et al. (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`, `diff` and `ratio` for the sum, difference and ratio of two parameters; `exp` for the exponential of a single parameter; `logistic` for a logistic transform with either one or two parameters, and `lognormal` for the mean and standard deviation for a Lognormal distribution on the basis of the mean and standard deviation for the logarithm of the coefficient.

`parName1`: A character object giving the name of the first parameter.

`parName2`: Like `parName1`, but for the second parameter.

```
apollo_modelOutput(model)
Model run using Apollo for R, version 0.0.8
www.ApolloChoiceModelling.com

Model name                          : hybrid_model_classical
Model description                   : Hybrid choice model on drug choice data, classical estimation
Model run at                        : 2019-04-15 08:58:54
Estimation method                   : bfgs
Model diagnosis                     : successful convergence
Number of individuals               : 1000
Number of observations              : 10000

Number of inter-person draws        : 500 (halton)
Number of cores used                : 25

LL(start)                           : -19734.5
LL(0)                               : -20300.7
LL(final, whole model)              : -16537.67
  LL(choice)                        : -10879.29
  LL(indic_quality)                 : -1453.893
  LL(indic_ingredients)             : -1486.283
  LL(indic_patent)                  : -1485.031
  LL(indic_dominance)               : -1425.619
Rho-square (0)                      :  0.1854
Adj.Rho-square (0)                  :  0.1831
AIC                                 :  33165.33
BIC                                 :  33489.8
Estimated parameters                :  45
Time taken (hh:mm:ss)               :  01:04:14.05
Iterations                          :  60

Estimates:
                          Estimate  Std.err.  t.ratio(0)  Rob.std.err.  Rob.t.ratio(0)
mu_brand_Artemis            1.4613   0.0834       17.53        0.0815           17.92
sig_brand_Artemis           0.0034   0.2629        0.01        0.0395            0.09
gamma_Artemis_reg_user     -0.0849   0.0825       -1.03        0.0845           -1.00
gamma_Artemis_university   -0.0769   0.0744       -1.03        0.0737           -1.04
gamma_Artemis_age_50        0.0882   0.0740        1.19        0.0743            1.19
mu_brand_Novum              1.1628   0.0851       13.67        0.0834           13.94
sig_brand_Novum             0.2939   0.0659        4.46        0.0643            4.57
gamma_Novum_reg_user        0.0409   0.0862        0.47        0.0883            0.46
gamma_Novum_university     -0.0989   0.0779       -1.27        0.0795           -1.24
gamma_Novum_age_50          0.0369   0.0776        0.47        0.0784            0.47
b_brand_BestValue           0.6724   0.0550       12.22        0.0549           12.26
b_brand_Supermarket         0.9856   0.0552       17.85        0.0539           18.27
b_brand_PainAway            0.0000       NA          NA            NA              NA
b_country_CH                0.6742   0.0401       16.83        0.0391           17.25
b_country_DK                0.3392   0.0385        8.82        0.0377            8.99
b_country_USA               0.0000       NA          NA            NA              NA
b_country_IND              -0.2970   0.0574       -5.17        0.0581           -5.11
b_country_RUS              -0.8934   0.0618      -14.46        0.0612          -14.60
b_country_BRA              -0.6589   0.0603      -10.93        0.0619          -10.64
b_char_standard             0.0000       NA          NA            NA              NA
b_char_fast                 0.7720   0.0293       26.32        0.0291           26.56
b_char_double               1.2187   0.0381       31.95        0.0369           33.02
b_risk                     -0.0016   0.0001      -26.99        0.0001          -26.47
b_price                    -0.7280   0.0183      -39.83        0.0174          -41.89
gamma_LV_reg_user          -0.9522   0.1049       -9.08        0.1066           -8.93
gamma_LV_university        -0.5461   0.0961       -5.68        0.0966           -5.65
gamma_LV_age_50             0.4344   0.0952        4.56        0.0972            4.47
lambda                      0.6187   0.0361       17.14        0.0357           17.32
zeta_quality                0.9566   0.0875       10.93        0.0889           10.77
zeta_ingredient            -0.8720   0.0819      -10.65        0.0832          -10.48
zeta_patent                 1.0775   0.0938       11.48        0.0940           11.46
zeta_dominance             -0.6782   0.0723       -9.38        0.0709           -9.57
tau_quality_1              -1.9310   0.1251      -15.43        0.1287          -15.01
tau_quality_2              -1.0241   0.1083       -9.46        0.1088           -9.41
tau_quality_3               1.1215   0.1061       10.57        0.1064           10.54
tau_quality_4               2.2524   0.1335       16.88        0.1329           16.95
tau_ingredients_1          -2.0797   0.1234      -16.85        0.1237          -16.81
tau_ingredients_2          -0.9898   0.0997       -9.92        0.1008           -9.82
tau_ingredients_3           0.9205   0.1022        9.01        0.1038            8.87
tau_ingredients_4           1.8713   0.1195       15.66        0.1201           15.59
tau_patent_1               -2.1004   0.1386      -15.15        0.1366          -15.38
tau_patent_2               -1.0301   0.1168       -8.82        0.1171           -8.79
tau_patent_3                0.9398   0.1104        8.51        0.1123            8.37
tau_patent_4                1.8827   0.1274       14.78        0.1296           14.53
tau_dominance_1            -2.2097   0.1209      -18.28        0.1204          -18.35
tau_dominance_2            -1.1089   0.0922      -12.03        0.0932          -11.89
tau_dominance_3             1.0678   0.0942       11.34        0.0928           11.51
tau_dominance_4             2.1254   0.1158       18.36        0.1143           18.59
```

Figure 12: On screen output (partial) obtained using `apollo_modelOutput`

multPar1: An optional numerical value used to multiply the first parameter, set to 1 if omitted.
multPar2: Like multPar1, but for the second parameter.

An example application of this function is illustrated in Figure 13, where we calculate the WTP for a reduction in the risk of side effects, and also look at the significance of the difference between the means for the two branded products. For the first calculation (i.e. the WTP), we also apply a multiplication by 1,000 to the numerator, thus looking at the WTP for a change by one percentage point.

```
> deltaMethod_settings=list(operation="ratio", parName1="b_risk", parName2="b_price", multPar1 = 1000)
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                                                    Value Robust s.e. Rob t-ratio (0)
Ratio of b_risk (multiplied by 1000) and b_price:  2.1647       0.091          23.79

> deltaMethod_settings=list(operation="diff", parName1="mu_brand_Artemis", parName2="mu_brand_Novum")
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                                                      Value Robust s.e. Rob t-ratio (0)
Difference between mu_brand_Artemis and mu_brand_Novum:  0.2985       0.0513          5.82
```

Figure 13: 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.

4.9.2 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 follows:

```
forecast = apollo_prediction(model,
                             apollo_probabilities,
                             apollo_inputs,
                             modelComponent)
```

The majority of these arguments have been discussed already. The only additional new argument is modelComponent, where this is 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.

The application of this function is illustrated in Figure 14, where we look at what happens to the mean market share for the two branded products after a 50% increase in their cost. Of course, it is overly simplistic to make predictions on SC data alone, and the manual discusses in

detail the steps for rescaling to RP elasticities. Model predictions in *Apollo* always use `database` as an input, whether applying the model to the base data or a forecast scenario. This means that for looking at the impact of changes to explanatory variables, these changes need to be made in `database`, and can then of course be reversed after applying `apollo_prediction`.

```
> base_forecast <- apollo_prediction(model, apollo_probabilities, apollo_inputs, modelComponent="choice")
Updating inputs...Done.
Running predictions from model... Done.
> mean(base_forecast[,1]+base_forecast[,2])
[1] 0.694406
> database$price_1=1.5*database$price_1
> database$price_2=1.5*database$price_2

> change_forecast <- apollo_prediction(model, apollo_probabilities, apollo_inputs, modelComponent="choice")
Updating inputs...Done.
Running predictions from model... Done.

> mean(change_forecast[,1]+change_forecast[,2])
[1] 0.4959083
> database$price_1=1/1.5*database$price_1
> database$price_2=1/1.5*database$price_2
```

Figure 14: Running `apollo_prediction`

Model predictions are also implemented for MDCEV and MDCNEV, where instead of probabilities, the predictions will instead return expected values of consumption for each alternative at the observation level. Predictions are not implemented for exploded logit and Normal density models.

4.9.3 Summary of random heterogeneity

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).

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 coefficients using inter-individual draws (one row per individual, one column per draw), and a 3-dimensional array (i.e. a cube) for coefficients using inter and intra-individual draws (with one row per observation, and draws in the second and third dimensions). The outputs from this function can then readily be used for summary statistics or to produce plots.

4.9.4 Posterior distributions

There is also 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.

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)
```

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 there will also be included in the calculation for the conditionals.

Figure 15 illustrates the calculation of unconditionals and conditionals for the latent variable in our example. We also show how the results can be contrasted for different subgroups of the data. To do this, we make use of the function `apollo_firstRow`, which produces a subset of the input data it receives, only retaining the first row for each individual[9]. This is required here as the data has one row per observation, but the conditionals and unconditionals only have one row per person. It is then also possible to use the conditional means for example in regression analysis against characteristics of the individual, as discussed by Train (2009, chapter 11), and as illustrated in the online manual.

5 Extension to 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. To explain the process, we now look at Bayesian estimation of the same model implemented in Section 4. We only discuss those steps where the code differs from the classical estimation case.

The first steps in the model definition are shown in Figure 16. We first set `apollo_control$HB = TRUE` and we now also omit the setting `apollo_control$mixing = TRUE` as this relates to classical estimation. We then define the individual coefficients and their starting values in `apollo_beta` as before, with the key difference that for any random coefficients, we now define a single parameter, not a mean and a standard deviation. The definition of `apollo_fixed` is the same as before.

We next create a list called `apollo_HB` which can contain any of the settings used in `RSGHB` (Dumont and Keller, 2019), where we only use a small subset here. One further difference arises.

[9] 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 3-dimensional array.

```
> unconditionals <- apollo_unconditionals(model,apollo_probabilities,apollo_inputs)
Updating inputs...Done.
Unconditional distributions computed

> conditionals <- apollo_conditionals(model,apollo_probabilities,apollo_inputs)
Updating inputs...Done.
For information:
 This function is meant for use only with continuous mixture models, i.e. no latent class.

Calculating conditionals...Done.


> mean(unconditionals[["LV"]])
[1] -0.3842727
> sd(unconditionals[["LV"]])
[1] 1.151479

> summary(conditionals[["LV"]])
       ID            post. mean          post. sd
 Min.   :   1.0   Min.   :-3.5027   Min.   :0.5606
 1st Qu.: 250.8   1st Qu.:-1.0512   1st Qu.:0.6014
 Median : 500.5   Median :-0.3214   Median :0.6164
 Mean   : 500.5   Mean   :-0.3848   Mean   :0.6206
 3rd Qu.: 750.2   3rd Qu.: 0.3137   3rd Qu.:0.6354
 Max.   :1000.0   Max.   : 2.1166   Max.   :0.8103

> regular_user_n=apollo_firstRow(database$regular_user, apollo_inputs)

> mean(subset(unconditionals[["LV"]],regular_user_n==0))
[1] -0.04717565
> mean(subset(unconditionals[["LV"]],regular_user_n==1))
[1] -0.9994272
> summary(subset(conditionals[["LV"]],regular_user_n==0))
       ID            post. mean          post. sd
 Min.   :   2.0   Min.   :-2.26740   Min.   :0.5630
 1st Qu.:246.2   1st Qu.:-0.57602   1st Qu.:0.6041
 Median :499.5   Median :-0.01971   Median :0.6195
 Mean   :500.9   Mean   :-0.04802   Mean   :0.6229
 3rd Qu.:755.5   3rd Qu.: 0.50459   3rd Qu.:0.6392
 Max.   :999.0   Max.   : 2.11656   Max.   :0.7213
> summary(subset(conditionals[["LV"]],regular_user_n==1))
       ID            post. mean          post. sd
 Min.   :   1.0   Min.   :-3.5027   Min.   :0.5606
 1st Qu.: 258.2   1st Qu.:-1.6053   1st Qu.:0.5982
 Median : 501.0   Median :-0.9868   Median :0.6126
 Mean   : 499.8   Mean   :-0.9993   Mean   :0.6164
 3rd Qu.: 747.5   3rd Qu.:-0.3826   3rd Qu.:0.6306
 Max.   :1000.0   Max.   : 1.5828   Max.   :0.8103
```

Figure 15: Running `apollo_unconditionals` and `apollo_conditionals`

`RSGHB` requires the user to create an element called `gdist` with a numeric coding for distributions. *Apollo* instead requires the user to create a named character vector inside `apollo_HB` that is called `hbDist` and which contains one entry for each of the parameters in `apollo_beta`, setting the distribution to use, with the following definitions:

- `"F"`: non-random (fixed) parameters[10];
- `"N"`: normally distributed random parameters;
- `"LN-"`: negative lognormally distributed random parameters;
- `"LN+"`: positive lognormally distributed random parameters;
- `"CN-"`: normally distributed random parameters, bounded above at 0;
- `"CN+"`: normally distributed random parameters, bounded below at 0; and
- `"JSB"`: Johnson SB distributed random parameters.

[10]This is also the obvious choice for parameters that are to be kept fixed at their starting values.

```
apollo_control = list(
  modelName  = "hybrid_model_bayesian",
  modelDescr = "Hybrid choice model on drug choice data, bayesian estimation",
  indivID    = "ID",
  HB         = TRUE
)

apollo_beta = c(b_brand_Artemis           = 0, gamma_Artemis_reg_user    = 0,
                gamma_Artemis_university  = 0, gamma_Artemis_age_50      = 0,
                b_brand_Novum             = 0, gamma_Novum_reg_user      = 0,
                gamma_Novum_university    = 0, gamma_Novum_age_50        = 0,
                b_brand_BestValue         = 0, b_brand_Supermarket       = 0,
                b_brand_PainAway          = 0, b_country_CH              = 0,
                b_country_DK              = 0, b_country_USA             = 0,
                b_country_IND             = 0, b_country_RUS             = 0,
                b_country_BRA             = 0, b_char_standard           = 0,
                b_char_fast               = 0, b_char_double             = 0,
                b_risk                    = 0, b_price                   = 0,
                gamma_LV_reg_user         = 0, gamma_LV_university       = 0,
                gamma_LV_age_50           = 0, lambda                    = 1,
                zeta_quality              = 1, zeta_ingredient           = 1,
                zeta_patent               = 1, zeta_dominance            = 1,
                tau_quality_1             =-2, tau_quality_2             =-1,
                tau_quality_3             = 1, tau_quality_4             = 2,
                tau_ingredients_1         =-2, tau_ingredients_2         =-1,
                tau_ingredients_3         = 1, tau_ingredients_4         = 2,
                tau_patent_1              =-2, tau_patent_2              =-1,
                tau_patent_3              = 1, tau_patent_4              = 2,
                tau_dominance_1           =-2, tau_dominance_2           =-1,
                tau_dominance_3           = 1, tau_dominance_4           = 2,
                eta                       = 0)

apollo_fixed = c("b_brand_PainAway", "b_country_USA", "b_char_standard")

apollo_HB = list(
  hbDist      = c(b_brand_Artemis           ="N", gamma_Artemis_reg_user    ="F",
                  gamma_Artemis_university  ="F", gamma_Artemis_age_50      ="F",
                  b_brand_Novum             ="N", gamma_Novum_reg_user      ="F",
                  gamma_Novum_university    ="F", gamma_Novum_age_50        ="F",
                  b_brand_BestValue         ="F", b_brand_Supermarket       ="F",
                  b_brand_PainAway          ="F", b_country_CH              ="F",
                  b_country_DK              ="F", b_country_USA             ="F",
                  b_country_IND             ="F", b_country_RUS             ="F",
                  b_country_BRA             ="F", b_char_standard           ="F",
                  b_char_fast               ="F", b_char_double             ="F",
                  b_risk                    ="F", b_price                   ="F",
                  gamma_LV_reg_user         ="F", gamma_LV_university       ="F",
                  gamma_LV_age_50           ="F", lambda                    ="F",
                  zeta_quality              ="F", zeta_ingredient           ="F",
                  zeta_patent               ="F", zeta_dominance            ="F",
                  tau_quality_1             ="F", tau_quality_2             ="F",
                  tau_quality_3             ="F", tau_quality_4             ="F",
                  tau_ingredients_1         ="F", tau_ingredients_2         ="F",
                  tau_ingredients_3         ="F", tau_ingredients_4         ="F",
                  tau_patent_1              ="F", tau_patent_2              ="F",
                  tau_patent_3              ="F", tau_patent_4              ="F",
                  tau_dominance_1           ="F", tau_dominance_2           ="F",
                  tau_dominance_3           ="F", tau_dominance_4           ="F",
                  eta                       ="N"),
  gNCREP      = 1000,
  gNEREP      = 1000,
  gINFOSKIP   = 500,
  fixedA      = c(NA,NA,0),
  fixedD      = c(NA,NA,1),
  gFULLCV     = FALSE
  )
```

Figure 16: Bayesian estimation in *Apollo*: model settings

The entry `hbDist` is the only compulsory setting when using Bayesian estimation in *Apollo*. In our example, we also define six additional settings, namely:

`gNCREP`: number of burn-in iterations to use prior to convergence (default=100000);
`gNEREP`: number of iterations to keep for averaging after convergence (default=100000);

`gINFOSKIP`: number of iterations between printing/plotting information (default=250);
`fixedA`: a vector of the same length as the number of random parameters, containing the value that the mean of each parameter should be kept fixed to, with NA for freely estimated random parameters (all NA by default);
`fixedD`: a vector of the same length as the number of random parameters, containing the value that the variance of each parameter should be kept fixed to, with NA for freely estimated random parameters (all NA by default); and
`gFULLCV`: a boolean variable indicating whether the full covariance matrix should be estimated between all random terms (default=TRUE).

In our example, we use these final three settings to constrain the mean and standard deviation of η, the random component in the latent variable, to be 0 and 1, respectively.

In Bayesian estimation, we no longer use *Apollo* to generate draws, and `apollo_draws` and `apollo_randCoeff` are thus not used. The `apollo_probabilities` function for a model estimated using Bayesian techniques uses the same approach as for a model without random heterogeneity, where `RSGHB` produces individual-specific values to be used for each parameter at each iteration.

Figure 17 shows the `apollo_probabilities` function for our example, where we only show those parts that have changes. As we no longer create random coefficients using `apollo_randCoeff`, we now compute the values for the two first brand parameters and the latent variable inside `apollo_probabilities`, where the random variation in these is introduced by `RSGHB`. The calculation of probabilities remains exactly the same as before. When using Bayesian estimation, the use of `apollo_avgInterDraws` and `apollo_avgIntraDraws` does not apply even in the presence of random coefficients. In addition, the call to `apollo_panelProd` is ignored as `RSGHB` automatically groups together observations for the same individual. The inclusion of any of these three commands however does no harm.

The call to `apollo_estimate` is made in exactly the same way as with classical estimation. In our example, we use scaling for those parameters that we saw obtain very small values in classical estimation. The estimation process is illustrated in Figure 18 for the text output and Figure 19 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.

The post-estimation output from a model using Bayesian estimation is substantially different from that with classical estimation, and is summarised in Figure 20. The early information on model name, description, and so forth 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`

```
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 random components
  LV = gamma_LV_reg_user*regular_user + gamma_LV_university*university_educated + gamma_LV_age_50*
      ↪ over_50 + eta
  b_brand_Artemis = b_brand_Artemis  + gamma_Artemis_reg_user*regular_user + gamma_Artemis_university*
      ↪ university_educated + gamma_Artemis_age_50*over_50
  b_brand_Novum = b_brand_Novum  + gamma_Novum_reg_user*regular_user + gamma_Novum_university*
      ↪ university_educated + gamma_Novum_age_50*over_50

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

...

  ### 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)

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

Figure 17: The `apollo_probabilities` function for a hybrid choice model using bayesian estimation

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). As highlighted in Figure 20, 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 thus not convenient. We thus report the scaled outputs only for the fixed parameters, the random parameters after transformation to the actual distributions used, and the posterior means.

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. As shown in Figure 21, the outcomes in prediction are in line with what we saw in classical estimation in Figure 14.

6 Additional functionalities

Apollo provides many additional functionalities beyond those covered in this paper. A full overview is provided in the online manual and only some brief highlights are presented here.

```
estimate_settings=list(scaling=c(gamma_Artemis_reg_user    = 0.1,
                                 gamma_Artemis_university  = 0.1,
                                 gamma_Artemis_age_50      = 0.1,
                                 gamma_Novum_reg_user      = 0.1,
                                 gamma_Novum_university    = 0.1,
                                 gamma_Novum_age_50        = 0.1,
                                 b_risk                    = 0.1))

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

------------------------------------------------------------

                        Number of Individuals:          1000
                      Number of Observations:          10000
                               Prior variance:              2
                  Target Acceptance (Fixed):            0.3
                Target Acceptance (Normal):            0.3
                           Degrees of Freedom:              5
 Avg. Number of Observations per Individual:             10
                      Initial Log-Likelihood: -19973.99029

------------------------------------------------------------

          Fixed Parameters Start
   gamma_Artemis_reg_user     0
 gamma_Artemis_university     0
...
          tau_dominance_4     2

------------------------------------------------------------

 Random Parameters Start Dist.
   b_brand_Artemis     0     N
     b_brand_Novum     0     N
               eta     0     N

------------------------------------------------------------

...

------------------------------------------------------------
Iteration:      250000
------------------------------------------------------------

              RHO (Fixed): 4.007008066e-05
  Acceptance Rate (Fixed):             2.76
             RHO (Normal):      1.186027669
 Acceptance Rate (Normal):            0.296
            Parameter RMS:      1.284610686
            Avg. Variance:      1.016972827
           Log-Likelihood:     -15994.70055
                      RLH:     0.2097300907

------------------------------------------------------------

...

 Random Parameters Estimate
  b_brand_Artemis: 1.477496
    b_brand_Novum: 1.244831
              eta: 0.000000

------------------------------------------------------------
Time per iteration: 0.0525 secs
Time to completion: 0 mins
------------------------------------------------------------
Estimation complete.
```

Figure 18: Bayesian estimation in *Apollo*: estimation process

6.1 Additional pre-estimation tools

Three additional functions provide some flexibility to a user before starting model estimation.

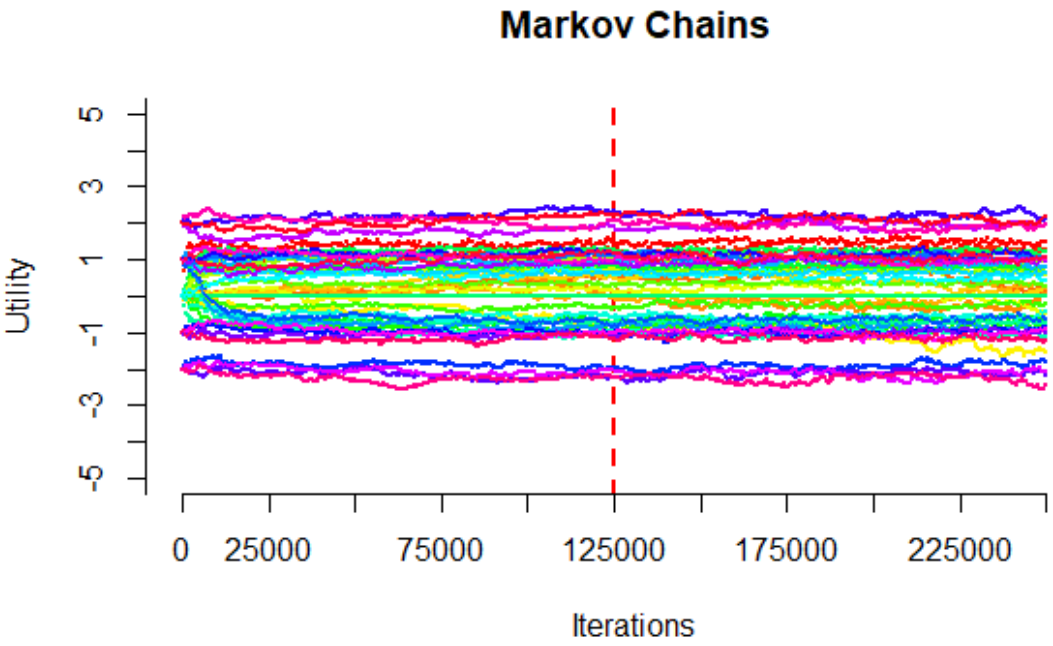

Figure 19: Bayesian estimation in *Apollo*: estimation process (parameter chains)

`apollo_readBeta` allows a user to read in estimates from a previous model to use as starting values.

`apollo_searchStart` implements a simplified version of the algorithm proposed by Bierlaire et al. (2010) to find better starting values for a model with the hope of reducing the risk of convergence to a poor local optimum.

`apollo_llCalc` allows the user to calculate the log-likelihood of the model (and subcomponents) for given parameter values, before or after estimation.

6.2 Additional model structures

Apollo provides ready made functions for many other model components beyond the MNL and ordered logit models discussed in this paper. While additional models will be added over time, at the time of writing this paper, *Apollo* version 0.0.8 included functions for seven additional model structures, as follows:

`apollo_cnl` provides an implementation of the cross-nested logit (CNL) model (Vovsha, 1997), where 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

```
> apollo_modelOutput(model)

Model run using Apollo for R, version 0.0.8
www.apollochoicemodelling.com

LL(start)                          : -19973.99
LL(0)                              : -20300.7
Average post. LL post burn-in      : -15965.7
Average post. RLH post burn-in     : 0.2101


Chain convergence report

Fixed (non random) parameters
  gamma_Artemis_reg_user gamma_Artemis_university
                 -0.2443                   5.8471
...
         tau_dominance_4
                  6.1194

Random parameters
b_brand_Artemis    b_brand_Novum              eta
        -5.6693          -8.5749              NaN

Covariances of random parameters
b_brand_Artemis_b_brand_Artemis    b_brand_Novum_b_brand_Artemis
                         1.1550                              NaN
...

Summary of parameter chains

Non-random coefficients
These outputs have had the scaling used in estimation applied to them

                            Mean     SD
gamma_Artemis_reg_user     0.0543 0.0214
gamma_Artemis_university  -0.0244 0.0129
...
tau_dominance_4            2.1014 0.1065

Upper level model results for mean parameters for underlying Normals
These outputs have NOT had the scaling used in estimation applied to them
                  Mean     SD
b_brand_Artemis 1.4716 0.0627
b_brand_Novum   1.2149 0.0652
eta             0.0000 0.0000

Upper level model results for covariance matrix for underlying Normals (means across iterations)
These outputs have NOT had the scaling used in estimation applied to them
               b_brand_Artemis b_brand_Novum eta
b_brand_Artemis          0.0877         0.0000   0
b_brand_Novum            0.0000         0.1236   0
eta                      0.0000         0.0000   1

Upper level model results for covariance matrix for underlying Normals (SD across iterations)
These outputs have NOT had the scaling used in estimation applied to them
               b_brand_Artemis b_brand_Novum eta
b_brand_Artemis          0.0218         0.0000   0
b_brand_Novum            0.0000         0.0311   0
eta                      0.0000         0.0000   0

Summary of distributions of random coeffients (after distributional transforms)
These outputs have had the scaling used in estimation applied to them
                  Mean     SD
b_brand_Artemis 1.4749 0.2949
b_brand_Novum   1.2120 0.3517
eta             0.0008 0.9915

Results for posterior means for random coefficients
These outputs have had the scaling used in estimation applied to them
                  [,1]   [,2]
b_brand_Artemis 1.4715 0.0899
b_brand_Novum   1.2149 0.1257
eta             0.0061 0.7823
```

Figure 20: Bayesian estimation in *Apollo*: output (extracts)

```
> base_forecast <- apollo_prediction(model, apollo_probabilities, apollo_inputs, modelComponent="choice")
Updating inputs...Done.
Running predictions from model... Done.
> mean(base_forecast[,1]+base_forecast[,2])
[1] 0.6984594
> databaseprice_1 = 1.5 * databaseprice_1
> databaseprice_2 = 1.5 * databaseprice_2

> change_forecast <- apollo_prediction(model, apollo_probabilities, apollo_inputs, modelComponent="choice")
Updating inputs...Done.
Running predictions from model... Done.

> mean(change_forecast[,1]+change_forecast[,2])
[1] 0.4937169
> databaseprice_1 = 1/1.5 * databaseprice_1
> databaseprice_2 = 1/1.5 * databaseprice_2
```

Figure 21: Running `apollo_prediction` after Bayesian estimation

(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.

`apollo_dft` allows the user to estimate decision field theory (DFT) models, a popular dynamic preference accumulation structure in mathematical psychology (Busemeyer and Townsend, 1992, 1993) which has recently made the transition into mainstream choice modelling (Hancock et al., 2019).

`apollo_el` provides an implementation of an exploded logit model for ranking data (or best-worst data, see Lancsar et al. 2013) where the user can allow for scale differences across stages.

`apollo_mdcev` allows the user to estimate the Multiple Discrete Continuous Extreme Value (MDCEV) model (Bhat, 2008), where no restriction are imposed on the profile to be used and where the model can be used with or without an outside good.

`apollo_mdcnev` provides an implementation of the Multiple Discrete Continuous Nested Extreme Value (MDCNEV) model of Pinjari and Bhat (2010). 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.

`apollo_nl` allows the estimation of nested logit (NL) models (Daly and Zachary, 1978; McFadden, 1978; Williams, 1977) without any constraints on the number of layers in the nesting structure. 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).

`apollo_normalDensity` is an implementation of the Normal probability density function for continuous dependent variables (or ordinal dependent variables that are treated as continuous).

In general, all models implemented in *Apollo* allow for prediction as well as estimation (except for `apollo_normalDensity` and `apollo_el`) and also allow for the inclusion of random heterogeneity in model parameters, with very few restrictions. In particular, only the α parameters in CNL, the σ parameters in MDCEV and the θ parameters in MDCNEV need to be kept non-random.

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 estima-

tion. Users can either create new functions in `R` that are defined outside `apollo_probabilities` much in the same way as for example `apollo_mnl` or simply code the probabilities for a model inside `apollo_probabilities`. An illustration of the latter approach is shown in the expectation-maximisation example in the online manual, which also discusses the requirements for user provided functions.

6.3 Other heterogeneity

6.3.1 Intra-individual heterogeneity

We have discussed in detail the implementation of continuous random heterogeneity at the level of individual decision makers. *Apollo* allows for a very general use of continuous random coefficients and works for models allowing for intra-individual mixing (i.e. heterogeneity at the level of individual choices), inter-individual mixing (i.e. heterogeneity at the level of individual people), as well as a mixture of the two.

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, with the code largely remaining the same. In terms of internal implementation, 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 at the level of individual people, with multiple choices 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 at the level of individual observations, the likelihood becomes a 3-dimensional array (i.e. a cube) of dimensions OxR_1xR_2, where in this third dimension, different draws are used across different choices 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.

The inclusion of intra-individual heterogeneity in *Apollo* simply requires the definition of a different type of draws, which are distributed across and within individuals (using `intraDrawsType`, `intraNDraws`, `intraUnifDraws` and `intraNormDraws` in `apollo_draws`) and the averaging across intra-individual draws (using `apollo_avgIntraDraws`) prior to taking the product across choices for the same individual (`apollo_panelProd`).

6.3.2 Discrete mixtures and latent class

Apollo offers the same degree of flexibility with latent class and discrete mixture models as with continuous mixture models. Full details of the implementation of latent class models are given in the online manual. The implementation follows an approach very similar to that for continuous random heterogeneity. The user implements a function called `apollo_lcPars`, which performs

a role analagous to `apollo_randCoeff` for continuous mixtures. 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 outputs from this function can then be used inside `apollo_probabilities`, typically with a loop over classes. Functions are also available to compute conditional and unconditional heterogeneity from latent class models, using `apollo_lcConditionals` and `apollo_lcUnconditionals`, respectively. *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.

6.4 Additional post-estimation tools

A number of additional post-estimation tools are included in *Apollo*, namely:

`apollo_bootstrap` allows estimation of the variance of parameters through a simple block bootstrap. Given a number of repetitions, this function generates as many new samples as requested, by sampling individuals (i.e. blocks of observations) *with replacement* from the original dataset. Then parameters are 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.

`apollo_combineResults` to produce an output file combining the estimates from multiple models run on the same data.

`apollo_fitsTest` compares the performance of the estimated model to predict the chosen alternative for different subsets of the data

`apollo_loadModel` allows the user to load into memory a `model` object previously saved using `apollo_saveOutput`, thus permitting the use of all post-estimation functions.

`apollo_lrTest` provides an implementation of the likelihood ratio test.

`apollo_outOfSample` estimates models on multiple subsets of the data and then compares the per observation log-likelihood to that in the remaining (hold-out) sample.

`apollo_sharesTest` provides an implementation of a routine like *apply* tables in ALogit (ALogit, 2016), looking at the recovery of market shares in given subset of the data.

7 Summary

In this paper, we have given a brief overview of the capabilities of *Apollo*. We have illustrated how a popular class of models, namely hybrid choice structures, can be easily implemented in *Apollo* and estimated using either classical or Bayesian estimation. Numerous functions are then available for processing of the results. Of course, a user can also make use of the many tabulation and plotting functions available in `R` for further analysis and formatting of model results.

Throughout the paper, we made reference to the online manual which contains more extensive details on the available functions, as well as numerous examples showing the implementation of different models in *Apollo*. We again strongly recommend that any prospective user of *Apollo* studies the online manual in depth after the general introduction provided in this paper.

In the years to come, it is our hope to add many exciting new capabilities to *Apollo*. This includes functions for new model structures, further computational improvements by coding more of the underlying calculations in `C++` (Eddelbuettel and François, 2011), and making further refinements to the implementation of expectation-maximisation (EM) routines (cf. Train, 2009, ch. 14).

8 Acknowledgments

While the *Apollo* package is the results 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. Finally, we would like to thank Michiel Bliemer and an anonymous referee for substantial comments on an earlier draft of this paper.

References

Abou-Zeid, M., Ben-Akiva, M., 2014. Hybrid choice models. In: Hess, S., Daly, A. (Eds.), Handbook of Choice Modelling. Edward Elgar.

ALogit, 2016. ALOGIT 4.3. ALOGIT Software & Analysis Ltd.
URL www.alogit.com

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 (3), 274–303.

Bierlaire, M., 2003. BIOGEME: a free package for the estimation of discrete choice models. Proceedings of the 3^{rd} 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 (1), 59–70.

Busemeyer, J. R., Townsend, J. T., 1992. Fundamental derivations from decision field theory. Mathematical Social Sciences 23 (3), 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 (3), 432.

Daly, A., 1987. Estimating Tree Logit models. Transportation Research Part B 21 (4), 251–267.

Daly, A., Hess, S., de Jong, G., 2012a. Calculating errors for measures derived from choice modelling estimates. Transportation Research Part B 46 (2), 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.

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 (2), 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. R package version 1.2.1.
URL https://CRAN.R-project.org/package=RSGHB

Eddelbuettel, D., François, R., 2011. Rcpp: Seamless R and C++ integration. Journal of Statistical Software 40 (8), 1–18.
URL http://www.jstatsoft.org/v40/i08/

Fosgerau, M., Mabit, S. L., 2013. Easy and flexible mixture distributions. Economics Letters 120 (2), 206 – 210.

Geweke, J., 1992. Evaluating the accuracy of sampling-based approaches to the calculations of posterior moments. Bayesian statistics 4, 641–649.

Greene, W. H., Hensher, D. A., 2013. Revealing additional dimensions of preference heterogeneity in a latent class mixed multinomial logit model. Applied Economics 45 (14), 1897–1902.

Halton, J., 1960. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numerische Mathematik 2 (1), 84–90.

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 (3), 443–458.
URL http://dx.doi.org/10.1007/s00180-010-0217-1

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., Daly, A., 2014. Handbook of Choice Modelling. Edward Elgar publishers, Cheltenham.

Hess, S., Train, K., 2011. Recovery of inter- and intra-personal heterogeneity using mixed logit models. Transportation Research Part B 45 (7), 973–990.

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 (2), 147–163.

Kløjgaard, M. E., Hess, S., 2014. Understanding the formation and influence of attitudes in patients' treatment choices for lower back pain: Testing the benefits of a hybrid choice model approach. Social Science & Medicine 114, 138 – 150.
URL http://www.sciencedirect.com/science/article/pii/S0277953614003591

Koppelman, F. S., Wen, C.-H., 1998. Alternative Nested Logit Models: structure, properties and estimation. Transportation Research Part B 32 (5), 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

Lenk, P., February 2014. Bayesian estimation of random utility models. In: Handbook of Choice Modelling. Chapters. Edward Elgar Publishing, Ch. 20, pp. 457–497.
URL https://ideas.repec.org/h/elg/eechap/14820_20.html

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, Ch. 25, pp. 75–96.

McFadden, D., 2000. Economic Choices. Nobel Prize Lecture.
URL https://www.nobelprize.org/uploads/2018/06/mcfadden-lecture.pdf

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., 2010. 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 (4), 562–583.

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/

RStudio Team, 2015. Rstudio: Integrated development for r. RStudio, Inc., Boston, MA URL http://www.rstudio.com/.

Train, K., 2009. Discrete Choice Methods with Simulation, second edition Edition. Cambridge University Press, Cambridge, MA.

Vij, A., Walker, J. L., 2016. How, when and why integrated choice and latent variable models are latently useful. Transportation Research Part B: Methodological 90, 192 – 217.
URL http://www.sciencedirect.com/science/article/pii/S019126151630234X

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.

Weisberg, S., 2005. Applied Linear Regression, 3rd Edition. 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 (7), 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 (3), 285–344.