

Skewness and staging: Does the floor effect induce bias in multilevel AR(1) models?

Reproducible codes

MH Manuel Haqiqatkhah

Table of contents

1	Introduction	3
2	Core components	4
2.1	The <i>Simulation</i> component	4
2.1.1	Data-generating models specifications	4
2.1.1.1	The AR(1) model	5
2.1.1.1.1	Construction	5
2.1.1.1.2	Code	5
2.1.1.2	The χ^2 AR(1) model	6
2.1.1.2.1	Construction	6
2.1.1.2.2	Code	6
2.1.1.3	The BinAR(1) model	6
2.1.1.3.1	Construction	6
2.1.1.3.2	Code	7
2.1.1.4	The PoDAR(1) model	7
2.1.1.4.1	Construction	7
2.1.1.4.2	Code	7
2.1.2	General DGM wrappers	8
2.1.2.1	Parameter conversions	8
2.1.2.2	Wrapper around <code>dgm_*</code> functions	8
2.1.3	Dataset generation	8
2.1.3.1	Making individual datasets	9
2.1.3.2	Determining level-2 distribution parameters	9
2.1.3.3	Automate dataset generation	12
2.2	The <i>Analysis</i> component	12
2.2.0.1	Fixed residual variance	13
2.2.0.2	Random residual variance	13

2.2.0.3	Comparing different ANALYSIS parameters	13
2.3	The <i>Harvesting</i> component	13
2.4	The <i>Reporting</i> component	13
2.4.1	Cleaning the harvested dataset	14
2.4.2	Making distribution plots of simulated datasets	14
2.4.2.1	Individual histograms	14
2.4.2.2	Pairplots of summary statistics	14
2.4.2.3	Profile plots of simulated datasets	14
2.4.3	Making plots of estimation results	14
2.4.3.1	Plots for a given level-2 distribution	15
2.4.3.2	Combined plots of results	15
2.4.4	Making plots of parameter estimates coverage	15
2.4.5	Making plots for DGM time series	15
3	Pipeline	15
3.1	Book-keeping	16
3.1.0.1	Simulation file references	16
3.1.0.2	Analysis outputs file references	17
3.2	Pipeline functions	17
3.2.0.1	Simulation in parallel	17
3.2.0.2	Analysis in parallel	18
3.2.0.3	Harvesting in parallel	18
3.3	Pipeline code	18
3.3.0.1	Making references	19
3.3.0.2	Running the study	19
3.3.0.3	Harvesting and cleaning the results	19
4	Simulation results	19
4.1	Plots main results	20
4.1.0.1	Plots of a single outcome measure	20
4.1.0.2	Plots of two outcome measures combined	20
4.2	Plots of MCMC parameter estimates coverage	20
5	Additional figures and analyses	21
5.1	DGM time series plots	21
5.1.0.1	Combining DGM outputs in one dataframe	21
5.1.0.2	Making the plots	21
5.1.0.3	Saving the plots	21
5.2	Profiles of the simulated datasets	21
5.3	COGITO data analysis	21
5.3.0.1	Mplus input	21
5.3.0.2	Mplus output	21

1 Introduction

This document contains the reproducible code for the manuscript *Skewness and staging: Does the floor effect induce bias in multilevel AR(1) models?* by Haqiqatkhah, Ryan, and Hamaker (2022). Please cite this document with the following info:

Haqiqatkhah, M. M., Ryan, O., & Hamaker, E. L. (2022). *Skewness and staging: Does the floor effect induce bias in multilevel AR(1) models?* PsyArXiv. <https://doi.org/10.31234/osf.io/myuvr>

In this study, we simulated multilevel data from three data generating mechanisms (DGMs), namely, the AR(1), χ^2 AR(1), BinAR(1), and PoDAR(1) models with different parameter sets. The simulation was conducted using the following modular pipeline design, inspired by Bien's R package `simulator` (2016), consisting of the following **components**:




- A. *Simulation*: generating the datasets
- B. *Analysis*: modeling the data
- C. *Harvesting*: collecting the relevant parameter estimates
- D. *Reporting*: making tables and plots



And the components were placed in a **pipeline**, that managed:

- 1. Making the simulation design matrix that include all relevant conditions
- 2. Book-keeping data files belonging to each replication of each condition
- 3. Performing simulations in batch
- 4. Performing Analyses in batch
- 5. Collecting the data in batch

This document is structured as follows. In Section 2, we explain the four components and the functions used therein. Then, in Section 3 we explain the wrapper functions used in the pipeline, and show how the pipeline was—and can be—executed. Then, in Section 4, we discuss how the harvested data was used to make the figures used in the paper (and others that were not included). Finally, in Section 5 we discuss how supplementary plots were made and how the empirical data analysis [on COGITO data; Schmiedek, Lövdén, and Lindenberger (2010)] was done. Note that although the codes provided here are cleaned as much as possible, they are not necessarily succinctly written; some functions were written to accommodate the most general functionalities which turned out to be not necessary for the simulation study.

The [GitHub repository of this study](#) contains all of the code necessary to run the study (in the `scripts` folder). Furthermore, because running the whole study would take a huge amount of time (it took us more than 88 days on a 24-core server and produced 2.69 TB of data), the the raw and summarized datasets of the estimated parameters are provided in `simulation-files/harvests`. To give an impression of the simulated datasets and the analysis output files, the `.rds` files of the first replication of the study are also provided in

`simulation-files` folder , and the *M*plus files of the first replication of the $N = 100, T = 100$ condition is provided in the `Mplus-files` folder . Finally, all the figures that can be generated based on the study results are provided in the `figures` folder . The figures are also referenced individually in Section 4 and Section 5 of this document.

To replicate the study from the scratch, you should first either clone the repository (using `git clone https://github.com/psyguy/skewness-staging.git`) or [download the repository as a zip file](#)  and extract it on your machine. Then you can sequentially run the `.R` files you find in the `scripts` folder. Note that you would need to have a licensed version of *M*plus version 8.6 (Muthén and Muthén 2017) on your machine to run the whole study. Instead of running the scripts separately, if you have [Quarto installed](#) , you can also compile `Code documentations.qmd` located in the root directory using Quarto after setting the following variables to `TRUE`:

Finally, in case you want to change the scripts (e.g., to run a smaller portion of the simulation, or try other parameters, etc.), you should look up the `knitr` parameters called in each chunk (with `<<some_param>>`) and find the corresponding code (under `## @knitr some_param`) in the `scripts` folder.

2 Core components

2.1 The *Simulation* component

The Simulation component consists of three sets of functions:

- i. Functions that implement the DGMs and generate univariate ($N = 1$) time series of length T from the parameters given to them;
- ii. Wrappers that interface the DGM functions;
- iii. A wrapper to generate datasets (consisting on N time series of length T) with a given DGM

2.1.1 Data-generating models specifications

First we define functions for each data-generating models (DGMs) that can produce univariate, single-subject ($N = 1$) time series of desired length T (default: `T = 100`) with the two canonical parameters and a given random seed (default: `seed = 0`). All model(-implied) parameters are saved in a list (called `pa`).

For each model, the first observation (X_1) is randomly drawn from the model-implied marginal distribution, to eliminate the need for removing the burn-in window in the beginning of the data. After the data is generated, in case the argument `only.ts` is set to be `TRUE`, the raw data (as a vector of length `T`) is returned. Otherwise, the function calculates

empirical dynamic (ϕ) and marginal (μ , σ^2 , and γ) parameters based on the simulated data, and save it in a list (`Empirical.Parameters`). Furthermore, two \LaTeX -ready strings (`Model.Description` and `Model.Description.Short`) are made which include a summary of the model parameters (that can be used, e.g., in plots). Finally, in case `only.ts != TRUE`, the function returns a list consisting of the time series (stored in `x`), verbal description of the dataset (`Model.Description` and `Model.Description.Short`), theoretical (i.e., model-implied) parameters (`Model.Parameters`), and empirical (i.e., sample) estimated parameters (`Empirical.Parameters`).

2.1.1.1 The AR(1) model

The canonical parameters of the AR(1) model with normally distributed residuals (which we referred to as NAR(1) in the simulation) are the autoregressive parameter ϕ (default: `phi = 0.4`), mean μ (default: `Mean = 50`), and the marginal variance σ^2 (default: `var.marginal = 4`). Based on the marginal variance, the residual variance (`var.resid`) is calculated via $\sigma_\epsilon^2 = \sigma^2(1 - \phi^2)$.

2.1.1.1.1 Construction

The time series is constructed by first generating a zero-centered time series \tilde{X}_t (`x_cent`). To do so, first the initial observation in the time series (`x_cent[1]`) is sampled from normal distribution with mean zero and a variance equal to the marginal variance of the model:

$$\tilde{X}_1 \sim \mathcal{N}(0, \sigma^2)$$

Then, the remainder of the time series is generated using the definition of the AR(1) model (not that the here the residual variance is used in the normal distribution):

$$\begin{aligned}\tilde{X}_t &= \phi \tilde{X}_{t-1} + \epsilon_t \\ \epsilon_t &\sim \mathcal{N}(0, \sigma_\epsilon^2)\end{aligned}$$

Finally, the mean is added to the centered zero-centered time series to reach the final time series with mean μ :

$$X_t = \tilde{X}_t + \mu$$

2.1.1.1.2 Code

2.1.1.2 The $\chi^2\text{AR}(1)$ model

The canonical parameters of the $\chi^2\text{AR}(1)$ model (which we referred to as **ChiAR(1)** in the simulation) are the autoregressive parameter ϕ (default: `phi = 0.4`), and degrees of freedom ν (default: `nu = 3`). We set the intercept to zero (`c = 0`).¹

2.1.1.2.1 Construction

Similar to the $\text{AR}(1)$ model, we need to sample the first observation of the $\chi^2\text{AR}(1)$ model from its marginal distribution. However, since this model does not have a closed-form marginal distribution, as an approximation, we instead sample $\mathbf{x}[1]$ from a χ^2 distribution with ν degrees of freedom:

$$X_1 \sim \chi^2(\nu)$$

Then, we generate the remainder of the time series using the definition of the $\chi^2\text{AR}(1)$ model:

$$\begin{aligned} X_t &= c + \phi X_{t-1} + a_t \\ a_t &\sim \chi^2(\nu). \end{aligned}$$

2.1.1.2.2 Code

2.1.1.3 The $\text{BinAR}(1)$ model

The canonical parameters of the $\text{BinAR}(1)$ model (which we referred to as **BinAR(1)** in the simulation) are the survival probability α (default: `alpha = 0.5`) and the revival probability β (default: `beta = 0.4`). By default, the maximum value on scale k was set to `k = 10`.

2.1.1.3.1 Construction

We first calculate the θ parameter, which characterizes the marginal distribution of the $\text{BinAR}(1)$ model:

$$\theta = \frac{k\beta}{1 - (\alpha - \beta)}$$

Then we draw X_1 ($\mathbf{x}[1]$) from the marginal distribution of the model:

¹The $\chi^2\text{AR}(1)$, in a more general form, can have an intercept ($X_t = c + \phi X_{t-1} + a_t$, $a_t \sim \chi^2(\nu)$). Since the intercept was set to zero in the simulation study, we discussed a zero-intercept version of this model ($c = 0$) in the paper. See the Supplemental Materials for more details.

$$X_1 \sim Binom(k, \theta)$$

The rest of time series is generated sequentially, for each time point t , by drawing values for the number of survived ($\mathbf{S_t[t]}$) and revived ($\mathbf{R_t[t]}$) elements of the BinAR(1) model based on the previous observations (X_{t-1}), and then adding them:

$$\begin{aligned} S_t &\sim Binom(X_{t-1}, \alpha) \\ R_t &\sim Binom(k - X_{t-1}, \beta) \\ X_t &= S_t + R_t \end{aligned}$$

2.1.1.3.2 Code

2.1.1.4 The PoDAR(1) model

The canonical parameters of the PoDAR(1) model (which we referred to as PoDAR(1) in the simulation) are the persistence probability τ (default: `tau = 0.7`) and the average rate λ (default: `lambda = 0.5`).

2.1.1.4.1 Construction

To generate the time series, we first draw the first observation X_1 (`x[1]`) from a Poisson distribution with rate λ :

$$X_1 \sim Poisson(\lambda)$$

And generate the rest of the time series by first drawing Z_t from a Poisson distribution with rate λ and P_t from a binomial distribution with size probability of success τ (that is equivalent to a Bernoulli distribution with probability τ). Then, we calculate X_t based on the previous observation (`x[t-1]`) and values of Z_t (`Z_t[t]`) and P_t (`P_t[t]`), using the definition of the PoDAR(1) model:

$$\begin{aligned} Z_t &\sim Poisson(\lambda) \\ P_t &\sim Binom(1, \tau) \\ X_t &= P_t X_{t-1} + (1 - P_t) Z_t \end{aligned}$$

2.1.1.4.2 Code

2.1.2 General DGM wrappers

Given that, in each model, two canonical parameters characterize the dynamic and marginal features of the generated time series, and given that we have analytic formulas that link the canonical parameters to the model-implied ϕ , μ , σ^2 , and γ , we use a function (`dgm_parameterizer`) to calculate canonical parameters from two given parameters, and make a complete list of parameters (called `pa`). This list also includes non-parameter variables, importantly, the time series length T (saved in `pa$T`) and the random seed used in the `dgm_*` functions (saved in `pa$seed`). A wrapper function (`dgm_generator`) is used as an interface to all `dgm_*` functions, which first makes sure the given parameters are sufficient for data generation, makes a complete parameter list `pa` with the help of `dgm_parameterizer`, and passes `pa` to the respective DGM generating function.

2.1.2.1 Parameter conversions

The function `dgm_parameterizer` calculates canonical/model-implied parameters of a given DGM (specified using the `Model` argument) based on the parameters given to it as arguments, and saves them in a list of parameters (`pa`), which is returned by the function. The function makes sure that the set of parameters provided are sufficient to characterize the dynamic parameter of the model (i.e., the autoregression ϕ) and at least one of the marginal parameters (importantly, the mean μ) but giving default values to some parameters.

2.1.2.2 Wrapper around `dgm_*` functions

The function `dgm_generator` gets a set of parameters (either as separate arguments, or a list of parameters, like the one returned by `dgm_parameterizer`), saves them in a list called `pa`. It checks whether ϕ is included in the list (if not, sets the default value `pa$phi = 0.2`), and checks if at least one other parameter (which, together with ϕ , is required to characterize the marginal properties of the DGMs) is calculated for it (if not, it sets the default value `pa$Mean = 5` for μ). Furthermore, if the DGM name, time series length, and the random seed are not provided, it gives them default values (respectively: `Model = "ChiAR(1)"`, `T = 100`, and `seed = 0`) and adds them to `pa`.

Then, it passes the `pa` list to `dgm_parameterizer` to do the necessary conversions to complete the list of canonical and model-implied parameters. Finally, given the model name, it checks if non-canonical parameters k and c are set (otherwise assigns appropriate defaults to them), and passes the complete parameter list to the respective DGM function.

2.1.3 Dataset generation

The machinery described above can be used to generate individual ($N = 1$) time series. However, for the simulation study, we need datasets comprising of multiple ($N = 25, 50, 100$)

individuals. As we discussed in the paper, in our study, all individuals in a dataset of a DGM share the same autoregressive parameter ($\phi_i = 0.4$) and the individual differences are only in the individual means ($\mu = [\mu_1, \mu_2, \dots, \mu_N]$). Thus, we write a function (`dgm_make.sample`) that can generate, for each DGM, a dataset of N individuals based on an N -dimensional vector of individual means, all with the same ϕ_i . We then need to find the appropriate parameters for the level-2 distributions (Gaussian and χ^2 distributions) for each DGM, such that we get a considerable proportion of individuals with considerably high skewness while respecting the lower and upper bounds of values supported by each model. Finally, with a wrapper function (`make_datasets`), we facilitate making dataset by automatically generating the means vector suitable for each DGM.

2.1.3.1 Making individual datasets

The function `dgm_make.sample` generates a dataset of time series of length T with the autoregressive parameter `phi` from a desired DGM (determined by the `Model` argument) given a vector of means (passed as the argument `Means`). The length of `Means` determine the number of individuals in the dataset (`N <- length(Means)`). If `Means` is not provided, a randomly generated vector of $N = 100$ is used as default. Since each individual time series is generated with a random seed, we need a vector of N unique seeds, which can be provided using the `seeds` argument. In case `seeds` is not provided, it is generated based on the provided means (`seeds.from.means`), and if it is a scalar, the seeds vector is created by adding the scalar to the `seeds.from.means` (which would allow generating different datasets with the same mean distributions).

2.1.3.2 Determining level-2 distribution parameters

For each alternative DGM—the χ^2 AR(1), BinAR(1), and PoDAR(1) models—we should determine appropriate parameters for the level-2 distribution of means such that we have enough skewness in the generated datasets. To do so, we make a function (`Mean.vs.Skewness`) to help us experiment with different values for μ and σ^2 (of the Gaussian level-2 distribution) and ν (of the χ^2 level-2 distribution) for each alternative DGM. Note that we start by generating more than enough samples for each distribution ($10 \times N$) and subsample N values after applying the model-specific lower and upper bounds.

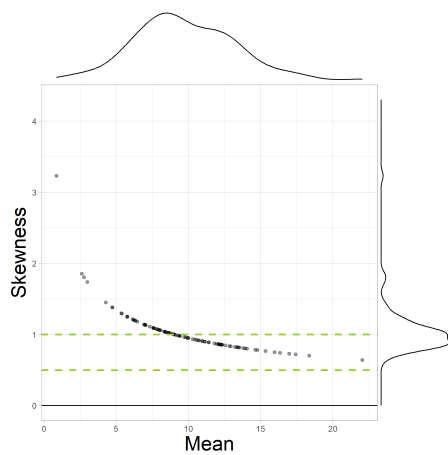
We notice that we get the desired distribution of skewness with the following parameters:

Table 1: Parameters of level-2 distribution of means

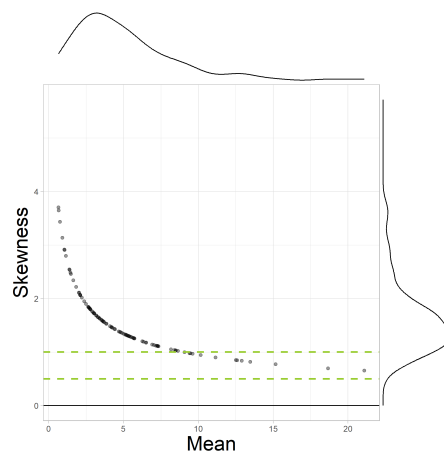
Model	μ	σ^2	ν
χ^2 AR(1)	10	10	5
BinAR(1)	2	1	2.9
PoDAR(1)	4	4	1.5

Giving us the following distributions:

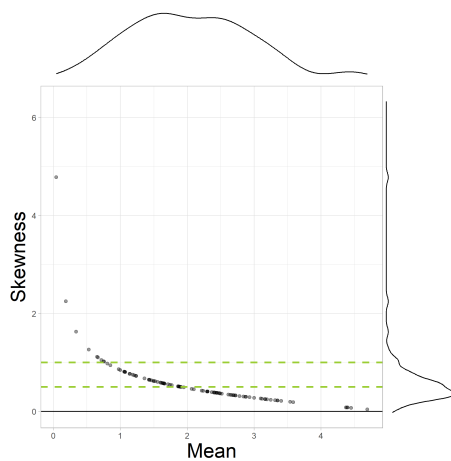
$\chi^2\text{AR}(1)$ with $\mu_i \sim N(10, 10)$



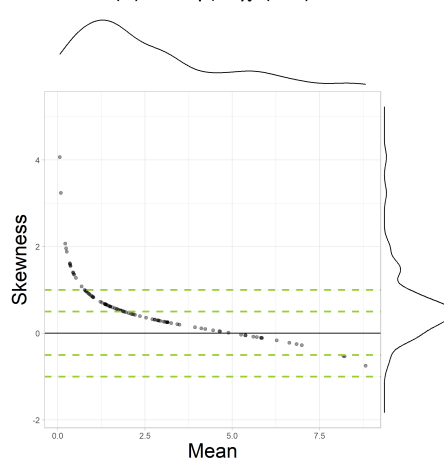
$\chi^2\text{AR}(1)$ with $\mu_i \sim \chi^2(5)$



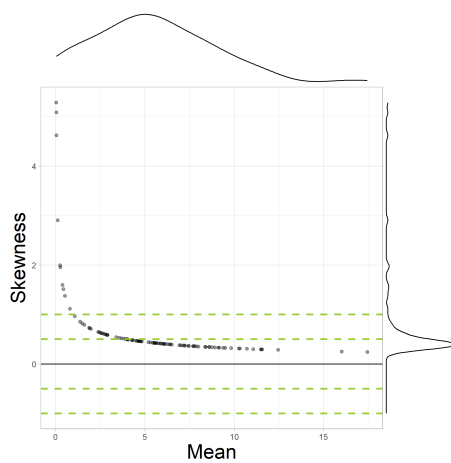
$\text{BinAR}(1)$ with $\mu_i \sim N(2, 1)$



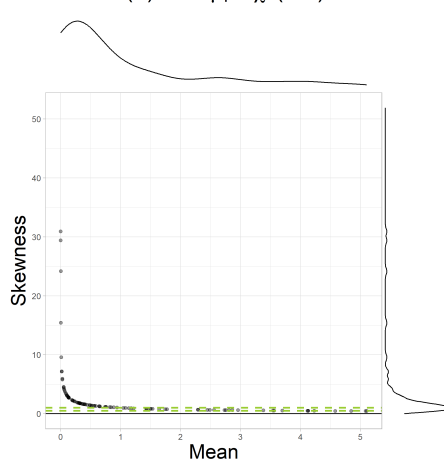
$\text{BinAR}(1)$ with $\mu_i \sim \chi^2(2.9)$



$\text{PoDAR}(1)$ with $\mu_i \sim N(4, 20)$



$\text{PoDAR}(1)$ with $\mu_i \sim \chi^2(1.5)$



2.1.3.3 Automate dataset generation

We then use a wrapper (`make_datasets`) around `dgm_make.sample` that generates datasets for all four DGMs with the appropriate level-2 parameters specified above. Note that here we first generate more than enough (i.e., $2 \times N$) samples of means to make sure we end up with N samples after applying the upper and lower bounds. The datasets are then saved, with some additional variables in separate `.rds` files, using wrapper functions described in Section 3.2.

2.2 The *Analysis* component

We analyzed each dataset with AR(1) models with fixed and random residual variance using *Mplus* v. 8.1 (Muthén and Muthén 2017). To interface *Mplus* from R, we used the package *MplusAutomation* (Hallquist and Wiley 2018) and wrote a function (`run_MplusAutomation`) that for each iteration of each condition would save the dataset as a `.dat` file, generate the `.inp` input script for the desired analysis type, and run the model for that dataset. *Mplus* then saves the output files (`.out` and `.gh5`). After the analysis, `run_MplusAutomation` extracts parameter estimates and returns them in an R object, which, with some additional variables, are saved in separate `.rds` files using wrapper functions described in Section 3.2.

In each analysis, we simulated two MCMC chains (`CHAINS = 2`), and to reduce autocorrelation in the estimated parameters, by defining `THIN = 5` we asked *Mplus* to save every 5th sample. By setting `BITERATIONS = 5000(2000)`, we made sure to have between 2000 to 5000 samples (after thinning) for each parameter from each chain. *Mplus* considers the first half of each chain as burn-in samples and discards them, thus, in total, we got at least 2000 “independent” samples from both chains combined. Finally, with `FACTORS = ALL (500)` we asked *Mplus* to draw 500 samples for each individuals when estimating level-1 parameters. We visually inspected the traceplots and autocorrelation plots of parameter estimates and of a sample of analyzed datasets and good convergence was observed. Furthermore, to make sure the number of iterations and thinning used in the analyses provide sufficiently accurate estimates, we re-analyzed two replications of each alternative DGM with Gaussian and χ^2 -distributed means with $N = 100$ and $T = 100$ with `BITERATIONS = 12500(5000)` and `THIN = 20`, which led to estimates of the parameters of interest (`unstd X.WITH.PHI`, `unstd Variances.PHI`, and `stdyx X.WITH.PHI`) almost identical (up to the third decimal) to those estimated with `BITERATIONS = 5000(2000)` and `THIN = 5` (see below).

The generated input files looked like the following. Note that the `TITLE` and `DATA` strings in the `.inp` files are unique to the dataset being analyzed and included the unique dataset seed `uSeed` (passed to `make_datasets` to generate datasets), the number of individuals in the dataset `N`, the length of the time series `T`, the model types used (`resid.fixed` or `resid.random`), and the replication number `Rep` (see Section 3.1 for further details).

2.2.0.1 Fixed residual variance

By estimating the covariances between mean and autoregression (`x phi WITH x phi` under the `BETWEEN%` command), the following *Mplus* script runs an AR(1) with random effect mean and autoregressive parameter but with fixed effect residual variance.

2.2.0.2 Random residual variance

By estimating the logarithm of the residual variance at level 1 (by `logv | x` under the `%WITHIN%` command) and estimating the covariances between the level-2 mean, autoregression, and residual variance (`x phi logv WITH x phi logv` under the `%BETWEEN%` command), the following *Mplus* script runs an AR(1) model with random effect mean, autoregressive parameter, and residual variance.

2.2.0.3 Comparing different ANALYSIS parameters

The table below shows the estimated parameters of two replications of with `BITERATIONS = 5000(2000)` and `THIN = 5` (specified in the table with `2k x 5`) and `BITERATIONS = 12500(5000)` and `THIN = 20` (specified in the table with `5k x 20`). The relevant parameters (level-2 correlations `X.WITH.PHI unstd`, covariances `X.WITH.PHI stdyx`, and variance `Variances.PHI unstd`) are highlighted with orange.

2.3 The *Harvesting* component

Extracting parameter estimates of individual analyses

The function `fit_extract` gets an `.rds` file generated by `do_fit_doFuture` (see Section 3.2)—which includes book-keeping information and *Mplus* output object that was generated by `run_MplusAutomation` (see Section 2.2)—and extracts the `unstandardized` and `stdyx.standardized` *Mplus* parameter estimates and stores them, along with the additional book-keeping information, in a dataframe.

2.4 The *Reporting* component

This component includes a function that clean the harvested dataset (and extract the relevant information) and functions to generate dataset profile plots, results figures, and parameter estimates coverage plots.

Given that the figures contain customized typefaces (from the CMU `Serif` and `Merriweather` font families), we need to make load the fonts:

2.4.1 Cleaning the harvested dataset

Given that the output of `fit_extract` has too much information in it, the function `harvest_cleanup` extracts the relevant parameter estimates—the point estimates and the upper and lower 2.5 credibility intervals of level-2 covariance (`unstd X.WITH.PHI`), level-2 correlation (`stdyx X.WITH.PHI`), and fixed-effect autoregressive parameter at level 2 (`unstd Means.PHI`)—cleans the datasets and makes it *tidy*, and calculates the outcomes of interest, namely, the (absolute) average estimates, mean absolute and squared error, root mean squared error, estimation bias and variance, number and percentage of non-converged datasets, and the percentages of significantly positive or negative or non-significant parameter estimates.

2.4.2 Making distribution plots of simulated datasets

These functions visualize datasets (simulated or empirical) by generating individual histograms and pairplots of summary statistics of datasets, and a function to generate (and combine) these two plots for any given dataset simulated by `do_sim_parallel` in the pipeline (see Section 3.2).

2.4.2.1 Individual histograms

The function `plot_histograms` makes person histograms of the individuals in a dataset (stored as dataframes) and arranges them based on individual means.

2.4.2.2 Pairplots of summary statistics

The function `plot_pairplots` calculates the mean, variance, and skewness of each individual in a dataset (stored as dataframes) and generates pairplots of their joint distribution.

2.4.2.3 Profile plots of simulated datasets

The function `plot_dataset.profile` gets a simulated object (generated by `do_sim_parallel`) and combines individual histograms and pairplots in a single figure.

We also write the function `save_dataset_profile` to put together two dataset profile plots to get to plots similar to those in Figures S1, S2, S4, and S5 in the Supplemental Materials, and save the final figure as a PDF file.

2.4.3 Making plots of estimation results

The functions here make the main figures of the paper, which include aspects of model fit (e.g., estimation bias and RMSE, and Type-I error rates, etc.).

2.4.3.1 Plots for a given level-2 distribution

The function `plot_Model.x.Resid` makes a figure with two columns (for the models with fixed or random residual variance), that show the outcomes of a fit measure (e.g., bias) for all the four DGMs with a given level-2 distribution of means for different N s and T s.

2.4.3.2 Combined plots of results

The function `plot_quadrants` makes a larger figure that includes four *quadrants*, each generated with `plot_Model.x.Resid.`, that show the main results of the paper.

2.4.4 Making plots of parameter estimates coverage

The function `plot_caterpillar`, for a given condition and a given parameter, makes plots of 95% confidence intervals (as vertical bars) of all converged replications within that condition. The lines are ordered based on the point estimates and are colored based on whether they crossed zero (the correct estimate) or whether the whole 95% interval was below or above zero (significantly negative or positive estimates).

2.4.5 Making plots for DGM time series

A function was also made to make profiles of individual time series, that include the time series itself, its marginal distribution, and its sample ACF.

3 Pipeline

We implemented each of these tasks in separate functions that were essentially wrapper functions (with parallel-computing implementation) around the modular components. Using these wrapper functions, each replication of each simulated condition was saved in a separate `.rds` file. These data files were fed to the analysis wrapper function whose output was saved in separate `.rds` data files. To collect relevant parameter estimates, another wrapper function was used to read the data files and save the desired parameters in a dataframe, which then used in reporting.

3.1 Book-keeping

The outcomes of the components are saved in separate `.rds` files and indexed by unique, descriptive names, and each replication is given a unique numeric identifier that is also used as the random seed used to generate the dataset within each replication. The file names and address are stored in two dataframes along with model parameters used to generate each dataset, and these dataframes are used when reading and writing data files in other components.

3.1.0.1 Simulation file references

The function `make_sim_refs` makes a table consisting of all possible combinations of each conditions (that are provided as a list `conditions`), and replicates it `Reps` times (number of replications, $R = 100$). Then, to make sure that each simulated dataset is produced with a unique random seed, the function generates a unique `uSeed` based on the number (and values) of conditions and the replication number. Furthermore, an initial simulation seed (`simSeed`) is included in generation of `uSeed` which makes it possible to have different *batches* of simulations (for instance, if one wants to run the simulation for another 1000 times).

For each condition, a unique seed is generated by weighting different conditions (that are indexed in a vector `d.integer`) by prime numbers (that are not among the prime factors of the number of conditions) and summing them up (which is done with dot-producting `d.integer %*% primes.seq`). Then, to make unique seeds per replications, the replication number is concatenated to the left side of `d.integer %*% primes.seq`, and everything is put into a dataframe (called `d.headers`, that is eventually returned by the function) with unique file names for individual `.rds` files (that include `uSeed` as well as the replication number and the value of condition used in data generation, e.g., `sim_uSeed-13293_12.dist-Chi2_Model-BinAR_N-100_phi-0.4_T-100_sim.Seed-0_Rep-1`) and the address of the folder in which the `.rds` files were stored. Finally, if desired, the simulation reference table `d.headers` is saved as `.csv` and `.rds`.

Note that initially, we started by having two values for the length of the time series ($T = 30$ corresponding to one month of measurements, and $T = 100$ for a reasonably long time series) and one sample size ($N = 100$, and subsample the datasets for $N = 25$ and $N = 50$), and we decided to include five DGMs in the simulations: the $AR(1)$, $\chi^2 AR(1)$, $BinAR(1)$, $PoDAR(1)$ models, and a $DAR(1)$ model with binomial marginal distribution). Eventually (and midway through the simulations), we decided to omit the $T = 30$ condition and the $DAR(1)$ model from our simulation. However, as omitting these would have led to change in the calculations of `uSeed`, we decided to keep it the simulation reference table intact and omit `Model == "DAR"` and `T == 30` later on (see Section 3.3 for details).

Further note that we only generated datasets with $N = 100$ and $T = 100$, and sub-sampled other sample sizes and time series lengths ($N = 25, 50$ and $T = 25, 50$) from them. Although

this had computational benefits (less simulation time, and less storage required), the main motivation behind this decision was to be able to mimic empirical data collection: Sub-sampling the large datasets would be equated with collecting less data than the ideal case in which we have many participants with many measurements (so sub-sampling what *could have been* collected). Furthermore, comparing the datasets with the same DGM and level-2 distribution would be more meaningful: All datasets with same `Rep` value within the same DGM (say, PoDAR(1) model) and with the same level-2 distribution (say, χ^2 distribution) have the same `uSeed`, thus they can be matched. Consequently, the differences we observe for different N s and T s within each cell of the figures reported in the paper can mimic the effect of “non-ideal” sampling.

Changing the conditions (omitting `Model == "DAR"` and `T == 30` from the simulation reference dataset, and adding additional conditions for N and T) is done when running the pipeline (see Section 3.3) and the sub-sampling is implemented when running the analysis in `do_fit_doFuture` (see Section 3.2).

3.1.0.2 Analysis outputs file references

Another dataframe was generated to include the references of analyses results per simulation condition and analysis condition (or *hyperparameters*), and file names and directory of individual `.rds` analysis outputs. The hyperparameters included the analysis type (with fixed vs. random residual variance) and the number of iterations and thinning in the MCMC algorithm, which were then used as arguments of `run_MplusAutomation`.

3.2 Pipeline functions

We implemented the components described in Section 2 using the following functions.

3.2.0.1 Simulation in parallel

The function `do_sim_parallel` is a wrapper around `make_datasets` (see Section 2.1.3) and uses the `clusterApplyLB` function from the package `snow` (Tierney et al. 2021) for parallelization. This wrapper gets the dataframe of simulation file references (`sim_refs`, which is the output of `make_sim_refs`), and for each of its rows, simulates an $N \times T$ dataframe and stores it with other information in the same row—and also additional info about the time the simulating the dataset started and ended—in a list, and saves each list as a separate `.rds` file in the target folder (specified with `sim.Path`). Having the additional data in the saved files allow using them independent from the table of references. Note that to prevent the overload of the hard drive with many parallel write requests (to save the simulated datasets), for the first `nClust` rounds, using the `Sys.sleep` function, the data generation is started after a varying amount of delay. Given that every dataset simulation in the first round may take different

amount of times, after the first round is complete, the simulations on different cores fall out of sync and there is no more need for adding delays for the next rounds.

3.2.0.2 Analysis in parallel

The function `do_fit_doFuture` is a wrapper around `run_MplusAutomation` (see Section 2.2) and uses the package `doFuture` (Bengtsson 2020) as a parallelization backend to `plyr::a_ply`. The function gets a dataframe containing references of analyses results (`fit_refs`, which is the output of `make_fit_refs`), and row by row reads the simulated datasets, sub-samples the data if $N < 100$ or $T < 100$ (see the second note under `make_sim_refs` in Section 3.1) and runs the analysis based on the hyperparameters specified in `fit_refs`. It then saves the outcome of the analysis (the *Mplus* model object produced by `MplusAutomation::mplusModeler` and returned by `run_MplusAutomation`), which includes *Mplus* outputs and parameter estimates) with additional information (other values in the same row of `fit_refs` as well as timings) in separate `.rds` files in the target folder. Note that, again, to prevent the overload of the hard drive with many parallel read and write requests (to read the simulated datasets and save the *Mplus* outputs), for the first `nClust` rounds, using the `Sys.sleep` function, the data analysis is started after a varying amount of delay. Given that every analysis instance in the first round take different amount of times, after the first round is complete the analyses on cores fall out of sync and there is no more need for adding delays for the next rounds.

3.2.0.3 Harvesting in parallel

The function `do_harvest_doFuture` is a wrapper around `fit_extract` (see Section 2.3) and uses the package `doFuture` as a parallelization backend to `foreach`. It gets a list of `.rds` files (that can be taken from `fit_refs`, or by using `list.files` function in the directory in which the analysis results are saved) and returns a large dataframe including parameter estimates (and additional information) extracted by `fit_extract`. Again, like in `do_sim_parallel` and `do_fit_doFuture`, the first `nClust` files are read with varying amount of delay to prevent the overload of the hard drive by multiple read requests.

3.3 Pipeline code

The following is the code we used to run the whole pipeline, which consists of three chunk that are responsible of the following:

1. Creating the book-keeping reference files (using functions described in Section 3.1).
2. Running the whole simulation study (using `do_sim_parallel` and `do_fit_doFuture` described in Section 3.2). The codes in this chunk can be interrupted (deliberately or otherwise) and resumed by running it again.

3. Harvesting the parameter estimates and storing them in a single dataframe (using `do_harvest_doFuture` of Section 3.2). The resulting dataframe is then used to make the plots using in Section 4)

But first we need to set the directories in which the results of each chunk is saved:

3.3.0.1 Making references

The following code makes `sim_refs` and `fit_refs` tables and make backups of them. In line 17 the rows with `T != 100` and `Model == "DAR"` are omitted, and with the codes in lines 22-31 we add other values of *N*s and *T*s (see the notes under `make_sim_refs` in Section 3.1).

3.3.0.2 Running the study

Here we reads the reference tables `sim_refs` and `fit_refs`. We then make a list of already simulated files (that are located in the `sim.Path` directory) and a list of analysis output files (that are located in the `fit.Path` directory). Then, by comparing `sim_refs` and `fit_refs` with the the lists of simulated and analyzed files, we only simulate/analyze the datasets that have not been yet simulated/analyzed. This allows us resume the simulation study after an interruption to the R session.

3.3.0.3 Harvesting and cleaning the results

Finally, the analyzed datasets are read using `do_harvest_doFuture` in parallel and stored in an `.rds` file suffixed with date and time (to prevent over-writing the previously saved harvest files). Furthermore, using the function `harvest_cleanup` (of Section 2.4.1), an abridged version (`d_abridged`, including all results of all replications for the parameters of interest) and a more summarized version (`d_important`, containing the outcome measures of interest per condition) of the raw harvest dataframe are made and saved in two `.rds` files (again suffixed with date and time).

4 Simulation results

We read the abridged harvested data file (`harvest-important_***.rds`) and make the outcome plots using `plot_Model.x.Resid` and `plot_quadrants` functions (of Section 2.4.3) and the MCMC estimates coverage plots using the function `plot_caterpillar` (of Section 2.4.4). To ensure small file size and good resolutions, all plots are saved in separate PDF files (which can easily be included in a \LaTeX script) in a specific directory. Before going any further, we should read the abridged harvest dataframe and set the plot output folder.

4.1 Plots main results

4.1.0.1 Plots of a single outcome measure













The left and right panels (including results for Gaussian and χ^2 distributed means) are made for each outcome measure of interest using `plot_Model.x.Resid`, combined using the package `patchwork`, and saved as PDF files.

4.1.0.2 Plots of two outcome measures combined

The plots containing pairs of outcome measures (respectively including results for Gaussian and χ^2 distributed means) are made for each outcome measure of interest, combined using the package `patchwork`, and saved as PDF files.

The generated plots are located in `figures` folder and can be accessed via the following tables:

Table 2: Combined plots of two outcome measures

Outcome measures	Correlations	Covariances
Bias and Variance		
Bias and RMSE		
Bias and MAE		
Positive and negative Type-I error		
Positive and total Type-I error		
Negative and total Type-I error		

4.2 Plots of MCMC parameter estimates coverage

It should be noted that `plot_caterpillar` requires the abridged dataset of harvested results (`d_abridged`) that contain estimates of all replications within each condition.

5 Additional figures and analyses

5.1 DGM time series plots

We make plots including time series plots, marginal distributions, and sample ACFs for three time series generated by each DGMs.

5.1.0.1 Combining DGM outputs in one dataframe

5.1.0.2 Making the plots

5.1.0.3 Saving the plots

5.2 Profiles of the simulated datasets

We read the first replication of simulated datasets with $N = 100, T = 100$ and plot using the function `plot_dataset.profile` (from Section 2.4.2).

5.3 COGITO data analysis

In the paper, we reported an analysis of distress time series in of the COGITO dataset Schmiedek, Lövdén, and Lindenberg (2010). Because we are not allowed to make the dataset public, here we provide the *Mplus* input script and the analysis output.

5.3.0.1 Mplus input

5.3.0.2 Mplus output

Bengtsson, Henrik. 2020. “A Unifying Framework for Parallel and Distributed Processing in *r* Using Futures,” August. <https://arxiv.org/abs/2008.00553>.

Bien, Jacob. 2016. “The Simulator: An Engine to Streamline Simulations.” *arXiv:1607.00021 [Stat]*, June. <http://arxiv.org/abs/1607.00021>.

Hallquist, Michael N., and Joshua F. Wiley. 2018. “*MplusAutomation* : An R Package for Facilitating Large-Scale Latent Variable Analyses in *M Plus*.” *Structural Equation Modeling: A Multidisciplinary Journal* 25 (4): 621–38. <https://doi.org/10.1080/10705511.2017.1402334>.

Haqiqatkhah, MH Manuel, Oisín Ryan, and Ellen L. Hamaker. 2022. “Skewness and Staging: Does the Floor Effect Induce Bias in Multilevel AR(1) Models?” <https://doi.org/10.31234/osf.io/myuvr>.

- Muthén, Linda K., and Bengt O. Muthén. 2017. *Mplus User's Guide*. Eighth Edition. Los Angeles, CA: Muthén & Muthén.
- Schmiedek, Florian, Martin Lövdén, and Ulman Lindenberger. 2010. "Hundred Days of Cognitive Training Enhance Broad Cognitive Abilities in Adulthood: Findings from the COGITO Study." *Frontiers in Aging Neuroscience* 2. <https://doi.org/10.3389/fnagi.2010.00027>.
- Tierney, Luke, A. J. Rossini, Na Li, and H. Sevcikova. 2021. *Snow: Simple Network of Workstations*. <https://CRAN.R-project.org/package=snow>.