Supplementary Materials 3: Examples using Stan and WinBUGS

Contents

NB this supplementary data is from: Richard M.A. Parker, George Leckie, Harvey Goldstein, Laura D. Howe, Jon Heron, Alun D. Hughes, David M. Phillippo, Kate Tilling. "Joint modelling of individual trajectories, within-individual variability and a later outcome: systolic blood pressure through childhood and left ventricular mass in early adulthood"

Introduction

In these supplementary materials, a series of models are fitted, of increasing complexity, to simulated data, concluding with a three-level joint model with a repeatedly-measured outcome and an individual-level outcome, and with complex within-individual variability.

The R script below fits models in Stan and WinBUGS. It uses 4 chains for each model, and otherwise the default number of total iterations and burnin/warmup iterations (and thinning) are used (see `?stan` and

`?bugs` for details). Dataset sample sizes and the number of chain iterations used can naturally be adjusted to facilitate better estimation, to run these example models more/less quickly, etc.

Stan is called via the R package `rstan`. To install `rstan`, please see https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started. Stan offers an efficient method of exploring complex posterior probabilities through its use of Hamiltonian Monte Carlo (HMC) and no-U-turn samplers (NUTS).

The Stan models use a non-centered parameterisation for the random effects, fitting them as independent standard Normals (N(0, 1)). Cholesky factorisation allows the covariance matrix for the random effects to be removed from the prior and recovered in the `transformed parameters` block. Since these optimisations change the shape of the posterior the HMC algorithm samples from, they can improve the efficiency with which multilevel models are estimated. Vectorisation (*cf.* the use of for loops, for example) also contributes to optimising the model.

(Cholesky factorisation can be used to derive the square-root of a symmatrical matrix, e.g., in R script:)

```
# symmetrical matrix, R:
R <- matrix(c(1, 0.5,
              0.5, 1), nrow = 2, ncol = 2)
# use Cholesky decomposition to derive lower triangular square root of R:
L <- t(chol(R))
# recover original symmetrical matrix:
L %*% t(L)
```

In the Stan model code below, `diag_pre_multiply(sigma_u, cholesky_corr_u)` calculates the Cholesky factor of the covariance matrix, multiplying `sigma_u` (as a diagonal matrix) with the Cholesky factor of the random effect correlation matrix. (Note `diag_pre_multiply(vector, matrix) = diag_matrix(vector) * matrix`, where `diag_matrix(vector)` would be (in R script), for example, `matrix(c(sigma_u[1], 0, 0, sigma_u[2]), nrow = 2, ncol = 2, byrow = FALSE)`).

When the resulting Cholesky factor of the random effect covariance matrix is multiplied by the independent (standard Normal) random effects, `z_u`, the correlated random effects `u` are recovered. (`z_u` is a matrix with `n_u` rows and `J` columns, where `n_u` is the number of random effects allowed to covary, and `J` denotes the number of individuals).

For further guidance and information see e.g. Stan Development Team (2018); McElreath (2016); Betancourt (2017); Sorensen, Hohenstein, and Vasishth (2016).

In addition to Stan, a model is also fitted to each simulated dataset using WinBUGS (Lunn et al. 2000), called via the R package `R2WinBUGS` (Sturtz, Ligges, and Gelman 2005).

A note on number of repeated measures

With regard to how many repeated measures are needed to estimate these models, then it can be helpful to examine what complexity of model could be fitted if the measures were taken at the same age for all individuals. For example, in the case of three measures per person, each at exactly the same ages, then three means and six variance/covariances could be estimated from the data. Such a dataset would be compatible with a model with individual-level random effects for mean and slope, plus random error, as this model estimates two means (intercept and slope) and four variance/covariances (variance for mean and slope, covariance between them, random error). However, we could not estimate a model including random terms for mean, slope and BPV, plus random error, since this model estimates seven variance/covariances (the variation for each of mean, slope, BPV and random error (four variances), plus the three covariances between mean, slope and within-individual variability). However, we could estimate the more elaborate model in the case of four measures per person (each at exactly the same ages), as ten variance/covariances could be estimated from these data.

Loading libraries; choosing options

```
# For Stan model fits:
library("rstan")
# For execution on a local, multicore CPU with excess RAM, rstan
# recommends calling:
options(mc.cores = parallel::detectCores())
## Check how many cores detected:
# getOption("mc.cores", 1L)
# To avoid recompilation of unchanged Stan programs, rstan
# recommends calling:
rstan_options(auto_write = TRUE)
# The following can improve execution time, but can also result in errors
# on some processors:
# Sys.setenv(LOCAL_CPPFLAGS = '-march=native')
library("shinystan") # further diagnostics

# For WinBUGS model fits:
library("R2WinBUGS") # fitting models in WinBUGS
library("MASS") # ginv()
bugs.directory <- "C:\\WinBUGS14\\"
```

2-level: random slope model

Equation

$$y_{1ij} = \beta_0 + \beta_1 \mathrm{x}_{1ij} + u_{0j} + u_{1j}\mathrm{x}_{1ij} + e_{ij}$$

$$\begin{pmatrix} u_{0j} \\ u_{1j} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \sigma_0^2 & \\ \sigma_{01} & \sigma_1^2 \end{pmatrix} \right]$$

$$e_{ij} \sim \mathrm{N}(0, \sigma_e)$$

Simulating data

```
# number of individuals:
J <- 1000
# number of repeated measures per individual:
n <- 10
# individual-level indicator at level 1 (observation level):
Ind_L1 <- rep(1:J, each = n)
# total number of observations:
N <- n * J

# design matrix for fixed part of mean function for y1;
# contains constant of ones and a covariate (for simplicity, randomly-drawn
# from a standard Normal distribution):
X_y1_mu <- cbind(rep(1, times = N),
                 rnorm(n = N, mean = 0, sd = 1))
```

```r
# design matrix for random part of mean function for y1 (same as above):
Z_y1_mu <- X_y1_mu

# number of fixed effects (betas) in mean function for y1:
n_b <- ncol(X_y1_mu)
# coefficient values for fixed effects in mean function for y1:
beta <- c(1, 1)

# number of individual-level random effects for y1:
n_u <- ncol(Z_y1_mu)
# number of unique covariances between random effects:
n_cov <- (n_u^2 - n_u) / 2

# SDs of random effects, and their correlation:
sigma_u <- c(0.7, 0.5)
rho <- 0.5
# correlation and covariance matrices for random effects:
corr_u <- matrix(c(1, rho, rho, 1), nrow = n_u)
cov_u <- diag(sigma_u) %*% corr_u %*% diag(sigma_u)

# generate random effects:
u <- MASS::mvrnorm(n = J,
                   mu = rep(0, times = n_u),
                   Sigma = cov_u)
# expand out to level 1:
u_long <- u[Ind_L1, ]

# generate fixed and random part (at level 2) of model for mean of y1:
fixpart_y1_mu <- as.vector(X_y1_mu %*% beta)
randpart_y1_mu <- rowSums(Z_y1_mu * u_long)

# generate level 1 residuals:
sigma_e <- 2.25
e <- rnorm(n = N, mean = 0, sd = sigma_e)

# generate repeatedly-measured outcome:
y1 <- fixpart_y1_mu + randpart_y1_mu + e
```

Stan

Specifying priors

To be adjusted as appropriate.

In this particular example, for the fixed effects in the mean function for y1, we use the equivalent as had the variables been standardised - of Normal(mean(y1), SD = 10) for intercept and Normal(0, SD = 2.5) for covariates (e.g. Gabry and Goodrich 2018).

```r
beta_prior_loc <- c(mean(y1), rep(0, times = n_b - 1))
beta_prior_scale_denom <- sd(X_y1_mu[, 2])
beta_prior_scale <- (2.5 * sd(y1)) / beta_prior_scale_denom
beta_prior_scale <- c(10 * sd(y1), beta_prior_scale)
```

```
# random effects:
# LKJ prior for correlation matrix:
LKJcorr_prior <- 2
# half-Cauchy for SDs:
sigma_u_prior_loc <- 0
sigma_u_prior_scale <- 10
sigma_e_prior_loc <- 0
sigma_e_prior_scale <- 10
```

Saving the data out

. . . in Stan-friendly format. NB: this saves a file out to the working directory.

```
stan_rdump(
  c("N",
    "J",
    "n_b",
    "n_u",
    "n_cov",
    "X_y1_mu",
    "Z_y1_mu",
    "y1",
    "Ind_L1",
    "beta_prior_loc",
    "beta_prior_scale",
    "LKJcorr_prior",
    "sigma_u_prior_loc",
    "sigma_u_prior_scale",
    "sigma_e_prior_loc",
    "sigma_e_prior_scale"
  ),
  file = "rand_slope_data.R")
```

Specifying the model

NB: Assuming this Stan program is saved in its own file.

```
writeLines(readLines("rand_slope_model.stan"))
```

```
## data {
##   //num observations (level 1)
##   int<lower=0> N;
##   //num subjects (level 2)
##   int<lower=0> J;
##   //num FEs (betas) in mean fun. for y1
##   int<lower=1> n_b;
##   //num REs
##   int<lower=1> n_u;
##   //num unique covariances bw REs
##   int<lower=1> n_cov;
##   //design matrix:fixed part mean fun. for y1
##   matrix[N, n_b] X_y1_mu;
##   //design matrix:random part mean fun. for y1
```

```
##   matrix[N, n_u] Z_y1_mu;
##   //repeatedly-measured outcome
##   vector[N] y1;
##   //subject indicator (of N-length)
##   int<lower=1,upper=J> Ind_L1[N];
##   //location and scale of priors for betas
##   vector[n_b] beta_prior_loc;
##   vector<lower=0>[n_b] beta_prior_scale;
##   //eta for LKJcorr prior for RE correlations
##   real<lower=0> LKJcorr_prior;
##   //location and scale of priors for RE SDs
##   real sigma_u_prior_loc;
##   real<lower=0> sigma_u_prior_scale;
##   //location and scale of prior for residual SD (within-subject)
##   real sigma_e_prior_loc;
##   real<lower=0> sigma_e_prior_scale;
## }
##
## parameters {
##   //FE coeffiecents in mean function for y_repeat
##   vector[n_b] beta;
##   //Cholesky factor of random effect corr matrix
##   //(i.e. corr_u = cholesky_corr_u * cholesky_corr_u')
##   cholesky_factor_corr[n_u] cholesky_corr_u;
##   //random effect SDs (lower bound ensures half-Cauchy)
##   vector<lower=0>[n_u] sigma_u;
##   //residual SD at level 1 (lower bound ensures halfCauchy)
##   real<lower=0> sigma_e;
##   //unscaled random effects (N(0,1))
##   matrix[n_u, J] z_u;
## }
##
## transformed parameters {
##   //scaled random effects
##   matrix[J, n_u] u;
##   u = (diag_pre_multiply(sigma_u, cholesky_corr_u) * z_u)';
## }
##
## model {
##   //priors
##   beta ~ normal(beta_prior_loc, beta_prior_scale);
##   //normal() not applicable to matrices, hence to_vector
##   //(treats it like a vector but maintains matrix data type)
##   to_vector(z_u) ~ normal(0, 1);
##   cholesky_corr_u ~ lkj_corr_cholesky(LKJcorr_prior);
##   sigma_u ~ cauchy(sigma_u_prior_loc, sigma_u_prior_scale);
##   sigma_e ~ cauchy(sigma_e_prior_loc, sigma_e_prior_scale);
##   //likelihood; uses vectorisation and matrix multiplication
##   y1 ~ normal(X_y1_mu * beta + rows_dot_product(Z_y1_mu, u[Ind_L1]), sigma_e);
## }
##
## generated quantities {
##   corr_matrix[n_u] corr_u_complete;
##   vector<lower=-1, upper=1>[n_cov] corr_u;
```

```
##   vector[N] y1_pred;
##   // return correlation matrix (dropping redundant elements)
##   corr_u_complete = multiply_lower_tri_self_transpose(cholesky_corr_u);
##   corr_u[1] = corr_u_complete[1, 2];
##   // for posterior predictive checks
##   for (n in 1:N) {
##     y1_pred[n] = normal_rng(X_y1_mu[n] * beta
##                             + dot_product(Z_y1_mu[Ind_L1[n]], u[Ind_L1[n]]),
##                             sigma_e);
##   }
## }
```

Fitting the model

```
data <- read_rdump("rand_slope_data.R")

params_for_summary <- c("beta",
                        "sigma_u",
                        "corr_u",
                        "sigma_e")

params_for_post_pred <- "y1_pred"

chains <- 4

# (include u (etc.) in pars if wish to save e.g. residuals at that level)
stan_fit <- stan(file = "rand_slope_model.stan",
                 data = data,
                 pars = c(params_for_summary, params_for_post_pred),
                 chains = chains)
```

Inspecting results

```
print(stan_fit, pars = params_for_summary)

# Launching shinystan to check diagnostics
stan_fit_shiny <- as.shinystan(stan_fit, pars = c(params_for_summary,
                                                  params_for_post_pred))
launch_shinystan(stan_fit_shiny)

# # If wish to save out whole stanfit object (might be large!):
# saveRDS(stan_fit, "stan_fit.RDS")
# # Reading back in:
# stan_fit <- readRDS("stan_fit.RDS")

# # If wish to save out summary:
# fit_summary <- summary(stan_fit)$summary
# saveRDS(fit_summary, "fit_summary.RDS")
# # Reading back in:
# fit_summary <- readRDS("fit_summary.RDS")
```

WinBUGS

Specifying data

```
# covariate:
x1 <- X_y1_mu[, 2]
# for inverse Wishart prior for covariance matrix
# (specified with df equal to order of matrix, i.e. 'minimally informative'):
R2 <- 2 * cov_u

bugsDat <- list(N = N,
                J = J,
                x1 = x1,
                y1 = y1,
                Ind_L1 = Ind_L1,
                R2 = R2)
```

Initial values

Here, known parameter values are used (from the data-generating process for the simulated data), but in the case of real data, parameter estimates can be gleaned from simpler models, for example, to provide an informed guess at promising initial values.

```
# precision parameter for covariance matrix:
tau.u <- MASS::ginv(cov_u)

# precision parameter for level 1 residual variance:
tau <- 1 / sigma_e^2

# different initial values for each chain:
CV <- 0.1
inits_function <- function(){
  list(
    beta = rnorm(n = length(beta), mean = beta, sd = abs(CV * beta)),
    u = jitter(u, amount = 2),
    tau.u = jitter(tau.u),
    tau = jitter(tau)
  )
}

inits1 <- inits_function()
inits2 <- inits_function()
inits3 <- inits_function()
inits4 <- inits_function()

inits <- list(inits1,
              inits2,
              inits3,
              inits4)
```

Specifying the model

NB: Assuming this BUGS model is saved in its own file.

This model is adapted from BUGS output from a random slope model fitted in MLwiN (Browne 2018; Charlton et al. 2018).

```
writeLines(readLines("rand_slope_model.bugs"))
```

```
## model {
##   # Level 1 definition
##   for (i in 1:N) {
##     y1[i] ~ dnorm(mu[i], tau)
##     mu[i]<- beta[1] + beta[2] * x1[i] + u[Ind_L1[i], 1]  + u[Ind_L1[i], 2] * x1[i]
##   }
##   # Higher level definitions
##   for (j in 1:J) {
##     u[j, 1:2] ~ dmnorm(zero2[1:2], tau.u[1:2, 1:2])
##   }
##   # Priors for fixed effects
##   for (k in 1:2) {beta[k] ~ dflat()}
##   # Priors for random terms
##   tau ~ dgamma(0.001, 0.001)
##   sigma2 <- 1/tau
##   for (i in 1:2) {zero2[i] <- 0}
##   tau.u[1:2, 1:2] ~ dwish(R2[1:2, 1:2], 2)
##   sigma2.u[1:2, 1:2] <- inverse(tau.u[,])
## }
```

Fitting model

```
parameters.to.save <- c("beta", "sigma2.u", "sigma2")

chains <- 4

BUGS_fit <- bugs(
  data = bugsDat,
  inits = inits,
  parameters.to.save = parameters.to.save,
  model.file = "rand_slope_model.bugs",
  n.chains = chains,
  bugs.directory = bugs.directory,
  program = "WinBUGS",
  working.directory = NULL,
  clearWD = TRUE,
  save.history = FALSE)
```

Inspecting results

```
print(BUGS_fit, digits.summary = 2)

## estimate of SD for random effects:
```

```r
diag(sqrt(BUGS_fit$mean$sigma2.u))

## estimate of correlation between random effects:
cov2cor(BUGS_fit$mean$sigma2.u)[lower.tri(cov2cor(BUGS_fit$mean$sigma2.u))]

## estimate of residual SD at level 1:
sqrt(BUGS_fit$mean$sigma2)

# # If wish to save out bugs object (might be large!):
# saveRDS(BUGS_fit, "BUGS_fit.RDS")
# # Reading back in:
# BUGS_fit <- readRDS("BUGS_fit.RDS")
```

With complex within-individual variability, including random effect

Equation

$$y_{1ij} = \beta_0 + \beta_1 \mathrm{x}_{1ij} + u_{0j} + u_{1j}\mathrm{x}_{1ij} + e_{ij}$$

$$\begin{pmatrix} u_{0j} \\ u_{1j} \\ u_{2j} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \sigma_0^2 & & \\ \sigma_{01} & \sigma_1^2 & \\ \sigma_{02} & \sigma_{12} & \sigma_2^2 \end{pmatrix} \right]$$

$$e_{ij} \sim \mathrm{N}(0, \sigma_{eij}^2), \quad \ln(\sigma_{eij}^2) = \alpha_0 + \alpha_1 \mathrm{x}_{1ij} + u_{2j}$$

Simulating data

```r
# number of individuals:
J <- 1000
# number of repeated measures per individual:
n <- 10
# individual-level indicator at level 1 (observation level):
Ind_L1 <- rep(1:J, each = n)
# total number of observations:
N <- n * J

# design matrix for fixed part of mean function for y1;
# contains constant of ones and a covariate (for simplicity, randomly-drawn
# from a standard Normal distribution):
X_y1_mu <- cbind(rep(1, times = N),
                 rnorm(n = N, mean = 0, sd = 1))

# design matrix for random part of mean function for y1,
# and fixed part of within-individual-variance function for y1 (all same):
X_y1_wiv <- Z_y1_mu <- X_y1_mu
# design matrix for random part of within-individual variance function for y1
# (just constant):
Z_y1_wiv <- X_y1_mu[, 1]

# number of fixed effects (betas) in mean function for y1:
```

```r
n_b <- ncol(X_y1_mu)
# coefficient values for fixed effects in mean function for y1:
beta <- c(1, 1)

# number of fixed effects (alphas) in within-individual variance function for y1:
n_a <- ncol(X_y1_wiv)
# coefficient values for fixed effects in within-individual variance function
# for y1:
alpha <- c(1, 1)

# number of individual-level random effects in mean function for y1:
n_u_mu <- ncol(Z_y1_mu)
# max column number of these (assuming contiguous) in matrix of REs;
# for indexing in Stan model:
u_mu_colnum <- 2
# number of individual-level random effects in within-individual variance function for y1:
n_u_wiv <- 1
# column number of these (assuming one) in matrix of REs;
# for indexing in Stan model:
u_wiv_colnum <- 3
# number of REs in total:
n_u_total <- n_u_mu + n_u_wiv
# number of unique covariances between random effects:
n_cov <- (n_u_total^2 - n_u_total) / 2

# SDs of random effects, and their correlations:
sigma_u <- c(0.7, 0.6, 0.4)
rho_01 <- 0.5
rho_02 <- 0.3
rho_12 <- 0.4

# correlation and covariance matrices for random effects:
corr_u <- matrix(, nrow = n_u_total, ncol = n_u_total)
corr_u[lower.tri(corr_u, diag = FALSE)] <- c(rho_01, rho_02, rho_12)
diag(corr_u) <- rep(1, times = n_u_total)
corr_u[upper.tri(corr_u)] <- t(corr_u)[upper.tri(corr_u)]
cov_u <- diag(sigma_u) %*% corr_u %*% diag(sigma_u)

# generate random effects:
u <- MASS::mvrnorm(n = J,
                   mu = rep(0, times = n_u_total),
                   Sigma = cov_u)
# expand out to level 1:
u_long <- u[Ind_L1, ]

# generate fixed and random part (at level 2) of model for mean of y1:
fixpart_y1_mu <- as.vector(X_y1_mu %*% beta)
randpart_y1_mu <- rowSums(Z_y1_mu * u_long[, 1:u_mu_colnum])

# generate fixed and random part (at level 2) of model
# for within-individual variance of y1:
fixpart_y1_wiv <- as.vector(X_y1_wiv %*% alpha)
randpart_y1_wiv <- Z_y1_wiv * u_long[, u_wiv_colnum]
```

```
# generate level 1 residuals:
log_sigma2_e <- fixpart_y1_wiv + randpart_y1_wiv
sigma_e <- sqrt(exp(log_sigma2_e))
e <- rnorm(n = N, mean = 0, sd = sigma_e)

# generate response variable:
y1 <- fixpart_y1_mu + randpart_y1_mu + e
```

Stan

Specifying priors

To be adjusted as appropriate.

In this particular example, for the fixed effects in the mean function for y1, we use the equivalent as had the variables been standardised - of Normal(mean(y1), SD = 10) for intercept and Normal(0, SD = 2.5) for covariates (e.g. Gabry and Goodrich 2018).

For the coefficients in the function for the within-individual variance, we also use Normal priors. For the location (mean) of the prior for the coefficient of the intercept we use an estimate of the within-individual SD gleaned from a simpler (random slopes) model, with zero for the location (mean) of the prior for the coefficient of the covariate. With regard to the scale of these priors, in this example we have chosen a value which would predict sigma_e of between c. 0.1 to 42 (i.e. very weakly informative, in this context) with +/- 2 SDs of a standard Normal predictor.

```
# priors (adjust as appropriate):
# fixed effects:
beta_prior_loc <- c(mean(y1), rep(0, times = n_b - 1))
beta_prior_scale_denom <- sd(X_y1_mu[, 2])
beta_prior_scale <- (2.5 * sd(y1)) / beta_prior_scale_denom
beta_prior_scale <- c(10 * sd(y1), beta_prior_scale)

within_ind_SD_estimate <- 2.1
alpha_prior_loc <- log(within_ind_SD_estimate^2)

alpha_prior_scale <- prior_sd <- 3
sqrt(exp(alpha_prior_loc - (2 * prior_sd)))
# [1] 0.1045528
sqrt(exp(alpha_prior_loc + (2 * prior_sd)))
# [1] 42.17963

alpha_prior_scale_denom <- sd(X_y1_wiv[, 2])
rescaled_prior_sd <- prior_sd / alpha_prior_scale_denom
rescaled_prior_sd
alpha_prior_scale <- c(alpha_prior_scale, rescaled_prior_sd)

alpha_prior_loc <- c(alpha_prior_loc, rep(0, times = n_a - 1))

# random effects:
# LKJ prior for correlation matrix:
LKJcorr_prior <- 2
# half-Cauchy for SDs:
sigma_u_prior_loc <- 0
```

```
sigma_u_prior_scale <- 10
```

Saving the data out

. . . in Stan-friendly format. NB: this saves a file out to the working directory.

```
stan_rdump(c("N",
             "J",
             "n_b",
             "n_a",
             "n_u_mu",
             "n_u_total",
             "u_mu_colnum",
             "u_wiv_colnum",
             "n_cov",
             "X_y1_mu",
             "Z_y1_mu",
             "X_y1_wiv",
             "y1",
             "Ind_L1",
             "beta_prior_loc",
             "beta_prior_scale",
             "alpha_prior_loc",
             "alpha_prior_scale",
             "LKJcorr_prior",
             "sigma_u_prior_loc",
             "sigma_u_prior_scale"
             ),
           file = "wiv_RE_data.R")
```

Specifying the model

NB: Assuming this Stan program is saved in its own file.

```
writeLines(readLines("wiv_RE.stan"))
```

```
## data {
##   //num observations (level 1)
##   int<lower=0> N;
##   //num subjects (level 2)
##   int<lower=0> J;
##   //num FEs (betas) in mean fun. for y1
##   int<lower=1> n_b;
##   //num FEs (alphas) in wiv fun. for y1
##   int<lower=1> n_a;
##   //num REs in mean fun. for y1
##   int<lower=1> n_u_mu;
##   //total num REs for y1
##   int<lower=1> n_u_total;
##   //max col num for REs in mean fun. (assumes contiguous) in u
##   int<lower=1> u_mu_colnum;
##   //col num for RE in wiv fun. (assumes only one) in u
##   int<lower=1> u_wiv_colnum;
```

```
##   //num unique covariances between REs
##   int<lower=1> n_cov;
##   //design matrix:fixed part mean fun. for y1
##   matrix[N, n_b] X_y1_mu;
##   //design matrix:random part mean fun. for y1
##   matrix[N, n_u_mu] Z_y1_mu;
##   //design matrix:fixed part wiv fun. for y1
##   matrix[N, n_a] X_y1_wiv;
##   //repeatedly-measured outcome
##   vector[N] y1;
##   //subject indicator (of N-length)
##   int<lower=1,upper=J> Ind_L1[N];
##   //location and scale of priors for betas
##   vector[n_b] beta_prior_loc;
##   vector<lower=0>[n_b] beta_prior_scale;
##   //location and scale of priors for alphas
##   vector[n_a] alpha_prior_loc;
##   vector<lower=0>[n_a] alpha_prior_scale;
##   //eta for LKJcorr prior for RE correlations
##   real<lower=0> LKJcorr_prior;
##   //location and scale of prior for RE SDs
##   real sigma_u_prior_loc;
##   real<lower=0> sigma_u_prior_scale;
## }
## parameters {
##   //FE coefficients in mean function for y_repeat
##   vector[n_b] beta;
##   //FE coefficients in within-individual variance function for y_repeat
##   vector[n_a] alpha;
##   //Cholesky factor of random effect corr matrix
##   //(i.e. corr_u = cholesky_corr_u * cholesky_corr_u')
##   cholesky_factor_corr[n_u_total] cholesky_corr_u;
##   //random effect SDs (lower bound ensures half-Cauchy)
##   vector<lower=0>[n_u_total] sigma_u;
##   //unscaled random effects (N(0,1))
##   matrix[n_u_total, J] z_u;
## }
## transformed parameters {
##   //scaled random effects
##   matrix[J, n_u_total] u;
##   u = (diag_pre_multiply(sigma_u, cholesky_corr_u) * z_u)';
## }
## model {
##   //priors
##   beta ~ normal(beta_prior_loc, beta_prior_scale);
##   alpha ~ normal(alpha_prior_loc, alpha_prior_scale);
##   //normal() not applicable to matrices, hence to_vector
##   //(treats it like a vector but maintains matrix data type)
##   to_vector(z_u) ~ normal(0, 1);
##   cholesky_corr_u ~ lkj_corr_cholesky(LKJcorr_prior);
##   sigma_u ~ cauchy(sigma_u_prior_loc, sigma_u_prior_scale);
##   //likelihood; uses vectorisation and matrix multiplication
##   y1 ~ normal(X_y1_mu * beta
##               + rows_dot_product(Z_y1_mu, u[Ind_L1, 1:u_mu_colnum]),
```

```
##                 sqrt(exp(X_y1_wiv * alpha + u[Ind_L1, u_wiv_colnum])));
## }
## generated quantities {
##   corr_matrix[n_u_total] corr_u_complete;
##   vector<lower=-1, upper=1>[n_cov] corr_u;
##   vector[N] y1_pred;
##   // return correlation matrix (dropping redundant elements)
##   corr_u_complete = multiply_lower_tri_self_transpose(cholesky_corr_u);
##   corr_u[1] = corr_u_complete[1, 2];
##   corr_u[2] = corr_u_complete[1, 3];
##   corr_u[3] = corr_u_complete[2, 3];
##   // for posterior predictive checks
##   for (n in 1:N) {
##     y1_pred[n] = normal_rng(X_y1_mu[n] * beta +
##                             dot_product(Z_y1_mu[Ind_L1[n]],
##                                         u[Ind_L1[n], 1:u_mu_colnum]),
##                             sqrt(exp(X_y1_wiv[n] * alpha +
##                                      u[Ind_L1[n], u_wiv_colnum])));
##   }
## }
```

Fitting the model

```
data <- read_rdump("wiv_RE_data.R")

params_for_summary <- c("beta",
                        "alpha",
                        "sigma_u",
                        "corr_u")

params_for_post_pred <- "y1_pred"

chains <- 4

# (include u (etc.) in pars if wish to save e.g. residuals at that level)
stan_fit <- stan(file = "wiv_RE.stan",
                 data = data,
                 pars = c(params_for_summary,
                          params_for_post_pred),
                 chains = chains)
```

Inspecting results

```
print(stan_fit, pars = params_for_summary)

# Launching shinystan to check diagnostics
stan_fit_shiny <- as.shinystan(stan_fit, pars = c(params_for_summary,
                                                  params_for_post_pred))
launch_shinystan(stan_fit_shiny)

# # If wish to save out whole stanfit object (might be large!):
```

```r
# saveRDS(stan_fit, "stan_fit.RDS")
# # Reading back in:
# stan_fit <- readRDS("stan_fit.RDS")

# # If wish to save out summary:
# fit_summary <- summary(stan_fit)$summary
# saveRDS(fit_summary, "fit_summary.RDS")
# # Reading back in:
# fit_summary <- readRDS("fit_summary.RDS")
```

WinBUGS

Specifying data

```r
# covariate:
x1 <- X_y1_mu[, 2]
# for inverse Wishart prior for covariance matrix
# (specified with df equal to order of matrix, i.e. 'minimally informative'):
R2 <- 2 * cov_u

bugsDat <- list(N = N,
                J = J,
                x1 = x1,
                y1 = y1,
                Ind_L1 = Ind_L1,
                R2 = R2)
```

Initial values

Here, known parameter values are used (from the data-generating process for the simulated data), but in the case of real data, parameter estimates can be gleaned from simpler models, for example, to provide an informed guess at promising initial values.

```r
# precision parameter for covariance matrix:
tau.u <- MASS::ginv(cov_u)

# different initial values for each chain:
CV <- 0.1
inits_function <- function(){
  list(
    alpha = rnorm(n = length(alpha), mean = alpha, sd = abs(CV * alpha)),
    beta = rnorm(n = length(beta), mean = beta, sd = abs(CV * beta)),
    u = jitter(u, amount = 2),
    tau.u = jitter(tau.u)
    )
}

inits1 <- inits_function()
inits2 <- inits_function()
inits3 <- inits_function()
inits4 <- inits_function()
```

```
inits <- list(inits1,
              inits2,
              inits3,
              inits4)
```

Specifying the model

NB: Assuming this BUGS model is saved in its own file.

```
writeLines(readLines("wiv_RE.bugs"))
```

```
## model {
##   # Level 1 definition
##   for (i in 1:N) {
##     y1[i] ~ dnorm(mu[i], tau[i])
##     mu[i]<- beta[1] + beta[2] * x1[i] + u[Ind_L1[i], 1]  + u[Ind_L1[i], 2] * x1[i]
##     tau[i] <- 1 / exp(alpha[1] + alpha[2] * x1[i] + u[Ind_L1[i], 3])
##   }
##   # Higher level definitions
##   for (j in 1:J) {
##     u[j, 1:3] ~ dmnorm(zero2[1:3],tau.u[1:3, 1:3])
##   }
##   # Priors for fixed effects
##   for (k in 1:2) {
##     beta[k] ~ dflat()
##     alpha[k] ~ dflat()
##   }
##   # Priors for random terms
##   for (i in 1:3) {zero2[i] <- 0}
##   tau.u[1:3, 1:3] ~ dwish(R2[1:3, 1:3], 3)
##   sigma2.u[1:3, 1:3] <- inverse(tau.u[,])
## }
```

Fitting model

```
parameters.to.save <- c("beta", "sigma2.u", "alpha")

chains <- 4

BUGS_fit <- bugs(
  data = bugsDat,
  inits = inits,
  parameters.to.save = parameters.to.save,
  model.file = "wiv_RE.bugs",
  n.chains = chains,
  bugs.directory = bugs.directory,
  program = "WinBUGS",
  working.directory = NULL,
  clearWD = TRUE,
  save.history = FALSE)
```

Inspecting results

```
print(BUGS_fit, digits.summary = 2)

## estimate of SD for random effects:
diag(sqrt(BUGS_fit$mean$sigma2.u))

## estimate of correlation between random effects:
cov2cor(BUGS_fit$mean$sigma2.u)[lower.tri(cov2cor(BUGS_fit$mean$sigma2.u))]

# # If wish to save out bugs object (might be large!):
# saveRDS(BUGS_fit, "BUGS_fit.RDS")
# # Reading back in:
# BUGS_fit <- readRDS("BUGS_fit.RDS")
```

Adding a individual-level outcome: joint model

Equation

$$y_{1ij} = \beta_0 + \beta_1 \mathrm{x}_{1ij} + u_{0j} + u_{1j}\mathrm{x}_{1ij} + e_{ij}$$

$$y_{2j} = \gamma_0 + \gamma_1 \mathrm{x}_{2j} + \gamma_2 u_{0j} + \gamma_3 u_{1j} + \gamma_4 u_{2j} + u_{3j}$$

$$\begin{pmatrix} u_{0j} \\ u_{1j} \\ u_{2j} \\ u_{3j} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \sigma_0^2 & & & \\ \sigma_{01} & \sigma_1^2 & & \\ \sigma_{02} & \sigma_{12} & \sigma_2^2 & \\ 0 & 0 & 0 & \sigma_3^2 \end{pmatrix} \right]$$

$$e_{ij} \sim \mathrm{N}(0, \sigma_{eij}^2), \quad \ln(\sigma_{eij}^2) = \alpha_0 + \alpha_1 \mathrm{x}_{1ij} + u_{2j}$$

Simulating data

```
# number of individuals:
J <- 1000
# number of repeated measures per individual:
n <- 10
# individual-level indicator at level 1 (observation level):
Ind_L1 <- rep(1:J, each = n)
# total number of observations:
N <- n * J

# design matrix for fixed part of mean function for y1;
# contains constant of ones and a covariate (for simplicity, randomly-drawn
# from a standard Normal distribution):
X_y1_mu <- cbind(rep(1, times = N),
                 rnorm(n = N, mean = 0, sd = 1))

# design matrix for random part of mean function for y1,
# and fixed part of within-individual variance function for y1 (all same):
X_y1_wiv <- Z_y1_mu <- X_y1_mu
```

```r
# design matrix for random part of within-individual variance function
# for y1 (just a constant):
Z_y1_wiv <- X_y1_mu[, 1]

# number of fixed effects (betas) in mean function for y1:
n_b <- ncol(X_y1_mu)
# coefficient values for fixed effects in mean function for y1:
beta <- c(1, 1)

# number of fixed effects (alphas) in within-individual variance function for y1:
n_a <- ncol(X_y1_wiv)
# coefficient values for fixed effects in within-individual variance function
# for y1:
alpha <- c(1, 1)

# number of individual-level random effects in mean function for y1:
n_u_mu <- ncol(Z_y1_mu)
# max column number of these (assuming contiguous) in matrix of REs;
# for indexing in Stan model:
u_mu_colnum <- 2
# number of individual-level random effects in within-individual variance
# function for y1:
n_u_wiv <- 1
# column number of these (assuming one) in matrix of REs;
# for indexing in Stan model:
u_wiv_colnum <- 3
# number of REs in total:
n_u_total <- n_u_mu + n_u_wiv
# number of unique covariances between random effects:
n_cov <- (n_u_total^2 - n_u_total) / 2

# SDs of random effects, and their correlations:
sigma_u <- c(0.7, 0.6, 0.4)
rho_01 <- 0.5
rho_02 <- 0.3
rho_12 <- 0.4

# correlation and covariance matrices for random effects:
corr_u <- matrix(, nrow = n_u_total, ncol = n_u_total)
corr_u[lower.tri(corr_u, diag = FALSE)] <- c(rho_01, rho_02, rho_12)
diag(corr_u) <- rep(1, times = n_u_total)
corr_u[upper.tri(corr_u)] <- t(corr_u)[upper.tri(corr_u)]
cov_u <- diag(sigma_u) %*% corr_u %*% diag(sigma_u)

# generate random effects:
u <- MASS::mvrnorm(n = J,
                   mu = rep(0, times = n_u_total),
                   Sigma = cov_u)
# expand out to level 1:
u_long <- u[Ind_L1, ]

# generate fixed and random part (at level 2) of model for mean of y1:
fixpart_y1_mu <- as.vector(X_y1_mu %*% beta)
```

```r
randpart_y1_mu <- rowSums(Z_y1_mu * u_long[, 1:u_mu_colnum])

# generate fixed and random part (at level 2) of model for within-individual
# variance of y1:
fixpart_y1_wiv <- as.vector(X_y1_wiv %*% alpha)
randpart_y1_wiv <- Z_y1_wiv * u_long[, u_wiv_colnum]

# generate level 1 residuals:
log_sigma2_e <- fixpart_y1_wiv + randpart_y1_wiv
sigma_e <- sqrt(exp(log_sigma2_e))
e <- rnorm(n = N, mean = 0, sd = sigma_e)

## generate repeatedly-measured outcome
y1 <- fixpart_y1_mu + randpart_y1_mu + e

# design matrix for observed covariates in mean function for y2:
X_y2_obs <- cbind(rep(1, times = J),
                  rnorm(n = J, mean = 0, sd = 1))
# number of fixed effects (gammas) for observed covariates
# in mean function for y2:
n_g_obs <- ncol(X_y2_obs)
# number of fixed effects (gammas) for REs fitted as exposures
# in mean function for y2:
n_g_RE <- ncol(u)
# design matrix for mean function for y2, incl. REs as exposures:
X_y2 <- cbind(X_y2_obs, u)
# total number of exposures in mean function for y2:
n_g_total <- n_g_obs + n_g_RE

# coefficient values for fixed effects in mean function for y2:
gamma <- c(0, 1.2, 0.2, 0.02, 0.4)

# residual SD for individual-level outcome, y2
sigma_y2 <- 0.5

# generate fixed and random part of model for mean of y2:
fixpart_y2 <- as.vector(X_y2 %*% gamma)
randpart_y2 <- rnorm(n = J, mean = 0, sd = sigma_y2)

# generate individual-level outcome:
y2 <- fixpart_y2 + randpart_y2
```

Stan

Specifying priors

To be adjusted as appropriate.

In this particular example, for the fixed effects in the mean function for y1, we use the equivalent as had the variables been standardised - of Normal(mean(y1), SD = 10) for intercept and Normal(0, SD = 2.5) for covariates (e.g. Gabry and Goodrich 2018). We do the same for the coefficients in the equation for the individual-level outcome, y2, as well - we use the random effects from the dataset we simulated to get an idea

of their SDs, but can of course use estimates of the SD of the random effects from simpler (e.g. univariate outcome) models fitted to actual data for same purpose.

For the coefficients in the function for the within-individual variance, we also use Normal priors. For the location (mean) of the prior for the coefficient of the intercept we use an estimate of the within-individual SD gleaned from a simpler (random slopes) model, with zero for the location (mean) of the prior for the coefficient of the covariate. With regard to the scale of these priors, in this example we have chosen a value which would predict sigma_e of between c. 0.1 to 42 (i.e. very weakly informative, in this context) with +/- 2 SDs of a standard Normal predictor.

```
# priors (adjust as appropriate):
# fixed effects:
beta_prior_loc <- c(mean(y1), rep(0, times = n_b - 1))
beta_prior_scale_denom <- sd(X_y1_mu[, 2])
beta_prior_scale <- (2.5 * sd(y1)) / beta_prior_scale_denom
beta_prior_scale<- c(10 * sd(y1), beta_prior_scale)

within_ind_SD_estimate <- 2.1
alpha_prior_loc <- log(within_ind_SD_estimate^2)

alpha_prior_scale <- prior_sd <- 3
sqrt(exp(alpha_prior_loc - (2 * prior_sd)))
# [1] 0.1045528
sqrt(exp(alpha_prior_loc + (2 * prior_sd)))
# [1] 42.17963

alpha_prior_scale_denom <- sd(X_y1_wiv[, 2])
rescaled_prior_sd <- prior_sd / alpha_prior_scale_denom
rescaled_prior_sd
alpha_prior_scale <- c(alpha_prior_scale, rescaled_prior_sd)

alpha_prior_loc <- c(alpha_prior_loc, rep(0, times = n_a - 1))

# random effects:
# LKJ prior for correlation matrix:
LKJcorr_prior <- 2
# half-Cauchy for SDs:
sigma_u_prior_loc <- 0
sigma_u_prior_scale <- 10
sigma_y2_prior_loc <- 0
sigma_y2_prior_scale <- 10

gamma_obs_prior_loc <- c(mean(y2), rep(0, times = n_g_obs - 1))
gamma_obs_prior_scale_denom <- sd(X_y2_obs[, n_g_obs])
gamma_obs_prior_scale <- (2.5 * sd(y2)) / gamma_obs_prior_scale_denom
gamma_obs_prior_scale <- c(10 * sd(y2), gamma_obs_prior_scale)

gamma_RE_prior_scale_denom <- apply(u, MARGIN = 2, sd)
gamma_RE_prior_scale <- (2.5 * sd(y2)) / gamma_RE_prior_scale_denom
gamma_RE_prior_loc <- rep(0, times = n_g_RE)
```

Saving the data out

. . . in Stan-friendly format. NB: this saves a file out to the working directory.

```
stan_rdump(
  c("N",
    "J",
    "n_b",
    "n_a",
    "n_g_obs",
    "n_g_RE",
    "n_u_mu",
    "n_u_total",
    "u_mu_colnum",
    "u_wiv_colnum",
    "n_cov",
    "X_y1_mu",
    "Z_y1_mu",
    "X_y1_wiv",
    "X_y2_obs",
    "y1",
    "y2",
    "Ind_L1",
    "beta_prior_loc",
    "beta_prior_scale",
    "alpha_prior_loc",
    "alpha_prior_scale",
    "gamma_obs_prior_loc",
    "gamma_obs_prior_scale",
    "gamma_RE_prior_loc",
    "gamma_RE_prior_scale",
    "LKJcorr_prior",
    "sigma_u_prior_loc",
    "sigma_u_prior_scale",
    "sigma_y2_prior_loc",
    "sigma_y2_prior_scale"
  ),
  file = "joint_L2_data.R")
```

Specifying the model

NB: Assuming this Stan program is saved in its own file.

```
writeLines(readLines("joint_L2.stan"))
```

```
## data {
##   //num observations (level 1)
##   int<lower=0> N;
##   //num subjects (level 2)
##   int<lower=0> J;
##   //num FEs (betas) in mean fun. for y1
##   int<lower=1> n_b;
##   //num FEs (alphas) in wiv fun. for y1
##   int<lower=1> n_a;
##   //num observed exposures for y2
##   int<lower=1> n_g_obs;
##   //num REs fitted as exposures for y2
```

```
##   int<lower=1> n_g_RE;
##   //num REs in mean fun. for y1
##   int<lower=1> n_u_mu;
##   //total num REs for y1
##   int<lower=1> n_u_total;
##   //max col num for REs in mean fun. (assumes contiguous) in u
##   int<lower=1> u_mu_colnum;
##   //col num for RE in wiv fun. (assumes only one) in u
##   int<lower=1> u_wiv_colnum;
##   //num unique covariances between REs
##   int<lower=1> n_cov;
##   //design matrix:fixed part mean fun. for y1
##   matrix[N, n_b] X_y1_mu;
##   //design matrix:random part mean fun. for y1
##   matrix[N, n_u_mu] Z_y1_mu;
##   //design matrix:fixed part wiv fun. for y1
##   matrix[N, n_a] X_y1_wiv;
##   //design matrix:observed exposures for y2
##   matrix[J, n_g_obs] X_y2_obs;
##   //repeatedly-measured outcome
##   vector[N] y1;
##   //individual-level outcome
##   vector[J] y2;
##   //subject indicator (of N-length)
##   int<lower=1,upper=J> Ind_L1[N];
##   //location and scale of priors for betas
##   vector[n_b] beta_prior_loc;
##   vector<lower=0>[n_b] beta_prior_scale;
##   //location and scale of priors for alphas
##   vector[n_a] alpha_prior_loc;
##   vector<lower=0>[n_a] alpha_prior_scale;
##   //location and scale of priors for observed exposures for y2
##   vector[n_g_obs] gamma_obs_prior_loc;
##   vector<lower=0>[n_g_obs] gamma_obs_prior_scale;
##   //location and scale of priors for RE exposures for y2
##   vector[n_g_RE] gamma_RE_prior_loc;
##   vector<lower=0>[n_g_RE] gamma_RE_prior_scale;
##   //eta for LKJcorr prior for RE correlations
##   real<lower=0> LKJcorr_prior;
##   //location and scale of priors for RE SDs
##   real sigma_u_prior_loc;
##   real<lower=0> sigma_u_prior_scale;
##   //location and scale of prior for observed exposures for y2
##   real sigma_y2_prior_loc;
##   real<lower=0> sigma_y2_prior_scale;
## }
## parameters {
##   //FE coefficients in mean function for y_repeat
##   vector[n_b] beta;
##   //FE coefficients in within-individual variance function for y_repeat
##   vector[n_a] alpha;
##   //FE coefficients of observed exposures for y2
##   vector[n_g_obs] gamma_obs;
##   //FE coefficients of RE exposures for y2
```

```
##   vector[n_g_RE] gamma_RE;
##   //Cholesky factor of random effect corr matrix
##   //(i.e. corr_u = cholesky_corr_u * cholesky_corr_u')
##   cholesky_factor_corr[n_u_total] cholesky_corr_u;
##   //random effect SDs (lower bound ensures half-Cauchy)
##   vector<lower=0>[n_u_total] sigma_u;
##   //unscaled random effects (N(0,1))
##   matrix[n_u_total, J] z_u;
##   //residual SD for y2 (lower bound ensures half-Cauchy)
##   real<lower=0> sigma_y2;
## }
## transformed parameters {
##   //scaled random effects
##   matrix[J, n_u_total] u;
##   u = (diag_pre_multiply(sigma_u, cholesky_corr_u) * z_u)';
## }
## model {
##   //priors
##   beta ~ normal(beta_prior_loc, beta_prior_scale);
##   alpha ~ normal(alpha_prior_loc, alpha_prior_scale);
##   gamma_obs ~ normal(gamma_obs_prior_loc, gamma_obs_prior_scale);
##   gamma_RE ~ normal(gamma_RE_prior_loc, gamma_RE_prior_scale);
##   //normal() not applicable to matrices, hence to_vector
##   //(treats it like a vector but maintains matrix data type)
##   to_vector(z_u) ~ normal(0, 1);
##   cholesky_corr_u ~ lkj_corr_cholesky(LKJcorr_prior);
##   sigma_u ~ cauchy(sigma_u_prior_loc, sigma_u_prior_scale);
##   sigma_y2 ~ cauchy(sigma_y2_prior_loc, sigma_y2_prior_scale);
##   //likelihood; uses vectorisation and matrix multiplication
##   y1 ~ normal(X_y1_mu * beta
##               + rows_dot_product(Z_y1_mu, u[Ind_L1, 1:u_mu_colnum]),
##               sqrt(exp(X_y1_wiv * alpha + u[Ind_L1, u_wiv_colnum])));
##   y2 ~ normal(X_y2_obs * gamma_obs + u * gamma_RE, sigma_y2);
## }
## generated quantities {
##   corr_matrix[n_u_total] corr_u_complete;
##   vector<lower=-1, upper=1>[n_cov] corr_u;
##   vector[N] y1_pred;
##   vector[J] y2_pred;
##   // return correlation matrix (dropping redundant elements)
##   corr_u_complete = multiply_lower_tri_self_transpose(cholesky_corr_u);
##   corr_u[1] = corr_u_complete[1, 2];
##   corr_u[2] = corr_u_complete[1, 3];
##   corr_u[3] = corr_u_complete[2, 3];
##   // for posterior predictive checks
##   for (n in 1:N) {
##     y1_pred[n] = normal_rng(X_y1_mu[n] * beta +
##                             dot_product(Z_y1_mu[Ind_L1[n]],
##                                         u[Ind_L1[n], 1:u_mu_colnum]),
##                             sqrt(exp(X_y1_wiv[n] * alpha +
##                                      u[Ind_L1[n], u_wiv_colnum])));
##   }
##   for (j in 1:J) {
##     y2_pred[j] = normal_rng(X_y2_obs[j] * gamma_obs + u[j] * gamma_RE,
```

```
##                                 sigma_y2);
##   }
## }
```

Fitting the model

```
data <- read_rdump("joint_L2_data.R")

params_for_summary <- c("beta",
                        "alpha",
                        "gamma_obs",
                        "gamma_RE",
                        "sigma_y2",
                        "sigma_u",
                        "corr_u")

params_for_post_pred <- c("y1_pred",
                          "y2_pred")

chains <- 4

# (include u (etc.) in pars if wish to save e.g. residuals at that level)
stan_fit <- stan(file = "joint_L2.stan",
                 data = data,
                 pars = c(params_for_summary,
                          params_for_post_pred),
                 chains = chains)
```

Inspecting results

```
print(stan_fit, pars = params_for_summary)

# Launching shinystan to check diagnostics
stan_fit_shiny <- as.shinystan(stan_fit, pars = c(params_for_summary,
                                                  params_for_post_pred))
launch_shinystan(stan_fit_shiny)

# # If wish to save out whole stanfit object (might be large!):
# saveRDS(stan_fit, "stan_fit.RDS")
# # Reading back in:
# stan_fit <- readRDS("stan_fit.RDS")

# # If wish to save out summary:
# fit_summary <- summary(stan_fit)$summary
# saveRDS(fit_summary, "fit_summary.RDS")
# # Reading back in:
# fit_summary <- readRDS("fit_summary.RDS")
```

WinBUGS

Specifying data

```
# covariates:
x1 <- X_y1_mu[, 2]
x2 <- X_y2[, 2]
# for inverse Wishart prior for covariance matrix
# (specified with df equal to order of matrix, i.e. 'minimally informative'):
R2 <- 2 * cov_u

bugsDat <- list(N = N,
                J = J,
                x1 = x1,
                y1 = y1,
                x2 = x2,
                y2 = y2,
                Ind_L1 = Ind_L1,
                R2 = R2)
```

Initial values

Here, known parameter values are used (from the data-generating process for the simulated data), but in the case of real data, parameter estimates can be gleaned from simpler models, for example, to provide an informed guess at promising initial values.

```
# precision parameter for covariance matrix:
tau.u <- MASS::ginv(cov_u)

# precision parameter for residual variance in y2:
tau.y2 <- 1 / sigma_y2^2

# different initial values for each chain:
CV <- 0.1
inits_function <- function(){
  list(
    alpha = rnorm(n = length(alpha), mean = alpha, sd = abs(CV * alpha)),
    beta = rnorm(n = length(beta), mean = beta, sd = abs(CV * beta)),
    gamma = rnorm(n = length(gamma), mean = gamma, sd = abs(CV * gamma)),
    u = jitter(u, amount = 2),
    tau.u = jitter(tau.u),
    tau.y2 = jitter(tau.y2)
  )
}

inits1 <- inits_function()
inits2 <- inits_function()
inits3 <- inits_function()
inits4 <- inits_function()

inits <- list(inits1,
              inits2,
              inits3,
```

```
              inits4)
```

Specifying the model

NB: Assuming this BUGS model is saved in its own file.

```
writeLines(readLines("joint_L2.bugs"))
```

```
## model {
##   # Level 1 definition
##   for (i in 1:N) {
##     y1[i] ~ dnorm(mu[i], tau[i])
##     mu[i] <- beta[1] + beta[2] * x1[i] +
##  u[Ind_L1[i], 1] + u[Ind_L1[i], 2] * x1[i]
##     tau[i] <- 1 / exp(alpha[1] + alpha[2] * x1[i] + u[Ind_L1[i], 3])
##   }
##   # Higher level definitions
##   for (j in 1:J) {
##     y2[j] ~ dnorm(mu.y2[j], tau.y2)
##     mu.y2[j] <- gamma[1] + gamma[2] * x2[j] + gamma[3] * u[j, 1] +
##     gamma[4] * u[j, 2] + gamma[5] * u[j, 3]
##     u[j, 1:3] ~ dmnorm(zero2[1:3], tau.u[1:3, 1:3])
##   }
##   # Priors for fixed effects
##   for (k in 1:2) {
##     beta[k] ~ dflat()
##     alpha[k] ~ dflat()
##   }
##   for (k in 1:5) {
##     gamma[k] ~ dflat()
##   }
##   # Priors for random terms
##   for (i in 1:3) {zero2[i] <- 0}
##   tau.u[1:3, 1:3] ~ dwish(R2[1:3, 1:3], 3)
##   sigma2.u[1:3, 1:3] <- inverse(tau.u[,])
##   tau.y2 ~ dgamma(0.001, 0.001)
##   sigma2.y2 <- 1 / tau.y2
## }
```

Fitting model

```
parameters.to.save <- c("beta",
                        "sigma2.u",
                        "alpha",
                        "gamma",
                        "sigma2.y2")

chains <- 4

BUGS_fit <- bugs(
  data = bugsDat,
  inits = inits,
```

```
  parameters.to.save = parameters.to.save,
  model.file = "joint_L2.bugs",
  n.chains = chains,
  bugs.directory = bugs.directory,
  program = "WinBUGS",
  working.directory = NULL,
  clearWD = TRUE,
  save.history = FALSE)
```

Inspecting results

```
print(BUGS_fit, digits.summary = 2)

## estimate of SD for random effects:
diag(sqrt(BUGS_fit$mean$sigma2.u))

## estimate of correlation between random effects:
cov2cor(BUGS_fit$mean$sigma2.u)[lower.tri(cov2cor(BUGS_fit$mean$sigma2.u))]

## estimate of residual SD for y2:
sqrt(BUGS_fit$mean$sigma2.y2)

# # If wish to save out bugs object (might be large!):
# saveRDS(BUGS_fit, "BUGS_fit.RDS")
# # Reading back in:
# BUGS_fit <- readRDS("BUGS_fit.RDS")
```

Introducing an additional, lower level

Equation

$$y_{1hij} = \beta_0 + \beta_1 \mathrm{x}_{1ij} + u_{0j} + u_{1j}\mathrm{x}_{1ij} + e_{ij} + \epsilon_{hij}$$

$$y_{2j} = \gamma_0 + \gamma_1 \mathrm{x}_{2j} + \gamma_2 u_{0j} + \gamma_3 u_{1j} + \gamma_4 u_{2j} + u_{3j}$$

$$\begin{pmatrix} u_{0j} \\ u_{1j} \\ u_{2j} \\ u_{3j} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \sigma^2_{u0} & & & \\ \sigma_{u01} & \sigma^2_{u1} & & \\ \sigma_{u02} & \sigma_{u12} & \sigma^2_{u2} & \\ 0 & 0 & 0 & \sigma^2_{u3} \end{pmatrix} \right]$$

$$e_{ij} \sim \mathrm{N}(0, \sigma^2_{eij}), \quad \ln(\sigma^2_{eij}) = \alpha_0 + \alpha_1 \mathrm{x}_{1ij} + u_{2j}$$

$$\epsilon_{hij} \sim \mathrm{N}(0, \sigma^2_{\epsilon})$$

Simulating data

```
# number of individuals:
J <- 1000
# number of clincs attended per individual:
```

```r
num_clinics_per_J <- 10
# number of measurements taken per clinic
n <- 2
# total number of clinic sessions attended:
total_num_clinics <- num_clinics_per_J * J
# total number of observations:
N <- n * num_clinics_per_J * J

# individual-level indicator at level 2 (clinic level):
Ind_L2 <- rep(1:J, each = num_clinics_per_J)
# individual-level indicator at level 1 (observation level):
Ind_L1 <- rep(1:J, each = n * num_clinics_per_J)
# clinic_level indicator at level 1 (observation level)
clinic_L1 <- rep(1:total_num_clinics, each = n)

# design matrix at level 2 (clinic level) for fixed part of mean function for
# y1; contains constant of ones and a covariate (for simplicity, randomly-drawn
# from a standard Normal distribution):
X_y1_mu_L2 <- cbind(rep(1, times = total_num_clinics),
                    rnorm(n = total_num_clinics, mean = 0, sd = 1))

# design matrix at level 2 (clinic level) for random part of mean function for
# y1 and fixed part of within-individual variance function for y1 (all same):
X_y1_wiv_L2 <- Z_y1_mu_L2 <- X_y1_mu_L2
# design matrix at level 2 (clinic level) for random part of within-individual
# variance function for y1 (just constant):
Z_y1_wiv_L2 <- X_y1_mu_L2[, 1]

# expanding out to create design matrices at level 1 (observation level):
X_y1_mu_L1 <- cbind(rep(1, times = N),
                    rep(X_y1_mu_L2[, 2], each = n))
X_y1_wiv_L1 <- Z_y1_mu_L1 <- X_y1_mu_L1
Z_y1_wiv_L1 <- X_y1_mu_L1[, 1]

# number of fixed effects (betas) in mean function for y1:
n_b <- ncol(X_y1_mu_L2)
# coefficient values for fixed effects in mean function for y1:
beta <- c(1, 1)

# number of fixed effects (alphas) in within-individual variance function for y1:
n_a <- ncol(X_y1_wiv_L2)
# coefficient values for fixed effects in within-individual variance function
# for y1:
alpha <- c(1, 1)

# number of individual-level random effects in mean function for y1:
n_u_mu <- ncol(Z_y1_mu_L2)
# max column number of these (assuming contiguous) in matrix of REs;
# for indexing in Stan model:
u_mu_colnum <- 2
# number of individual-level random effects in within-individual variance
# function for y1:
n_u_wiv <- 1
```

```r
# column number of these (assuming one) in matrix of REs;
# for indexing in Stan model:
u_wiv_colnum <- 3
# number of REs in total:
n_u_total <- n_u_mu + n_u_wiv
# number of unique covariances between random effects:
n_cov <- (n_u_total^2 - n_u_total) / 2

# SDs of random effects, and their correlations:
sigma_u <- c(0.7, 0.6, 0.4)
rho_01 <- 0.5
rho_02 <- 0.3
rho_12 <- 0.4

# correlation and covariance matrices for random effects:
corr_u <- matrix(, nrow = n_u_total, ncol = n_u_total)
corr_u[lower.tri(corr_u, diag = FALSE)] <- c(rho_01, rho_02, rho_12)
diag(corr_u) <- rep(1, times = n_u_total)
corr_u[upper.tri(corr_u)] <- t(corr_u)[upper.tri(corr_u)]
cov_u <- diag(sigma_u) %*% corr_u %*% diag(sigma_u)

# generate random effects:
u <- MASS::mvrnorm(n = J,
                   mu = rep(0, times = n_u_total),
                   Sigma = cov_u)
# expand out to level 2:
u_long <- u[Ind_L2, ]

# generate fixed and random part (at level 2) of model for mean of y1:
fixpart_y1_mu <- as.vector(X_y1_mu_L2 %*% beta)
randpart_y1_mu <- rowSums(Z_y1_mu_L2 * u_long[, 1:u_mu_colnum])

# generate fixed and random part (at level 2) of model for within-individual
# variance of y1:
fixpart_y1_wiv <- as.vector(X_y1_wiv_L2 %*% alpha)
randpart_y1_wiv <- Z_y1_wiv_L2 * u_long[, u_wiv_colnum]

# generate random effects at the clinic level
log_sigma2_e <- fixpart_y1_wiv + randpart_y1_wiv
sigma_e <- sqrt(exp(log_sigma2_e))
e <- rnorm(n = total_num_clinics, mean = 0, sd = sigma_e)

# generate preliminary repeatedly-measured outcome at level 2 (clinic level):
y1_L2 <- fixpart_y1_mu + randpart_y1_mu + e

# expand out to 1 (observation level), adding variation as do so:
y1_L2_expanded <- rep(y1_L2, each = n)
sigma_epsilon <- 0.5
y1 <- rnorm(n = length(y1_L2_expanded), mean = y1_L2_expanded, sd = sigma_epsilon)

# design matrix for observed covariates in mean function for y2:
X_y2_obs <- cbind(rep(1, times = J),
                  rnorm(n = J, mean = 0, sd = 1))
```

```r
# number of fixed effects (gammas) for observed covariates
# in mean function for y2:
n_g_obs <- ncol(X_y2_obs)
# number of fixed effects (gammas) for REs fitted as exposures
# in mean function for y2:
n_g_RE <- ncol(u)
# design matrix for mean function for y2, incl. REs as exposures:
X_y2 <- cbind(X_y2_obs, u)
# total number of exposures in mean function for y2:
n_g_total <- n_g_obs + n_g_RE

# coefficient values for fixed effects in mean function for y2:
gamma <- c(0, 1.2, 0.2, 0.02, 0.4)

# residual SD for individual-level outcome, y2
sigma_y2 <- 0.5

# generate fixed and random part of model for mean of y2:
fixpart_y2 <- as.vector(X_y2 %*% gamma)
randpart_y2 <- rnorm(n = J, mean = 0, sd = sigma_y2)

# generate individual-level outcome:
y2 <- fixpart_y2 + randpart_y2
```

Stan

Specifying priors

To be adjusted as appropriate.

In this particular example, for the fixed effects in the mean function for y1, we use the equivalent as had the variables been standardised - of Normal(mean(y1), SD = 10) for intercept and Normal(0, SD = 2.5) for covariates (e.g. Gabry and Goodrich 2018). We do the same for the coefficients in the equation for the individual-level outcome, y2, as well - we use the random effects from the dataset we simulated to get an idea of their SDs, but can of course use random effects from simpler models fitted to actual data for same purpose.

For the coefficients in the function for the within-individual variance, we also use Normal priors. In this example with simulated data we use the known residual SD for the location (mean) of the prior for the coefficient of the intercept, but with real data an estimate of the residual SD could be taken from a simpler model (e.g. random slope), for example, for the same purpose. Otherwise, zero is used for the location (mean) of the prior for the coefficient of the covariate. With regard to the scale of these priors, in this example we have chosen a value which would predict sigma_e of between c. 0.1 to 45 (i.e. very weakly informative, in this context) with +/- 2 SDs of a standard Normal predictor.

```r
# priors (adjust as appropriate):
# fixed effects:
beta_prior_loc <- c(mean(y1), rep(0, times = n_b - 1))
beta_prior_scale_denom <- sd(X_y1_mu_L1[, 2])
beta_prior_scale <- (2.5 * sd(y1)) / beta_prior_scale_denom
beta_prior_scale<- c(10 * sd(y1), beta_prior_scale)

within_ind_SD_estimate <- 2.1
alpha_prior_loc <- log(within_ind_SD_estimate^2)
```

```
alpha_prior_scale <- prior_sd <- 3
sqrt(exp(alpha_prior_loc - (2 * prior_sd)))
# [1] 0.1045528
sqrt(exp(alpha_prior_loc + (2 * prior_sd)))
# [1] 42.17963

alpha_prior_scale_denom <- sd(X_y1_wiv_L2[, 2])
rescaled_prior_sd <- prior_sd / alpha_prior_scale_denom
rescaled_prior_sd
alpha_prior_scale <- c(alpha_prior_scale, rescaled_prior_sd)
alpha_prior_loc <- c(alpha_prior_loc, rep(0, times = n_a - 1))

# random effects:
# LKJ prior for correlation matrix:
LKJcorr_prior <- 2
# half-Cauchy for SDs:
sigma_u_prior_loc <- 0
sigma_u_prior_scale <- 10
sigma_y2_prior_loc <- 0
sigma_y2_prior_scale <- 10
sigma_epsilon_prior_loc <- 0
sigma_epsilon_prior_scale <- 10

gamma_obs_prior_loc <- c(mean(y2), rep(0, times = n_g_obs - 1))
gamma_obs_prior_scale_denom <- sd(X_y2_obs[, n_g_obs])
gamma_obs_prior_scale <- (2.5 * sd(y2)) / gamma_obs_prior_scale_denom
gamma_obs_prior_scale <- c(10 * sd(y2), gamma_obs_prior_scale)

gamma_RE_prior_scale_denom <- apply(u, MARGIN = 2, sd)
gamma_RE_prior_scale <- (2.5 * sd(y2)) / gamma_RE_prior_scale_denom
gamma_RE_prior_loc <- rep(0, times = n_g_RE)
```

Saving the data out

. . . in Stan-friendly format. NB: this saves a file out to the working directory.

```
stan_rdump(
  c("N",
    "total_num_clinics",
    "J",
    "n_b",
    "n_a",
    "n_g_obs",
    "n_g_RE",
    "n_u_mu",
    "n_u_total",
    "u_mu_colnum",
    "u_wiv_colnum",
    "n_cov",
    "X_y1_mu_L1",
    "Z_y1_mu_L1",
    "X_y1_wiv_L2",
    "Z_y1_wiv_L2",
```

```
    "X_y2_obs",
    "y1",
    "y2",
    "Ind_L1",
    "Ind_L2",
    "clinic_L1",
    "beta_prior_loc",
    "beta_prior_scale",
    "alpha_prior_loc",
    "alpha_prior_scale",
    "gamma_obs_prior_loc",
    "gamma_obs_prior_scale",
    "gamma_RE_prior_loc",
    "gamma_RE_prior_scale",
    "LKJcorr_prior",
    "sigma_u_prior_loc",
    "sigma_u_prior_scale",
    "sigma_y2_prior_loc",
    "sigma_y2_prior_scale",
    "sigma_epsilon_prior_loc",
    "sigma_epsilon_prior_scale"
    ),
  file = "joint_L3_data.R")
```

Specifying the model

NB: Assuming this Stan program is saved in its own file.

```
writeLines(readLines("joint_L3.stan"))
```

```
## data {
##   //num observations (level 1)
##   int<lower=0> N;
##   //num clinic sessions attended (level 2)
##   int<lower=0> total_num_clinics;
##   //num subjects (level 3)
##   int<lower=0> J;
##   //num FEs (betas) in mean fun. for y1
##   int<lower=1> n_b;
##   //num FEs (alphas) in wiv fun. for y1
##   int<lower=1> n_a;
##   //num observed exposures for y2
##   int<lower=1> n_g_obs;
##   //num REs fitted as exposures for y2
##   int<lower=1> n_g_RE;
##   //num REs in mean fun. for y1
##   int<lower=1> n_u_mu;
##   //total num REs for y1
##   int<lower=1> n_u_total;
##   //max col num for REs in mean fun. (assumes contiguous) in u
##   int<lower=1> u_mu_colnum;
##   //col num for RE in wiv fun. (assumes only one) in u
##   int<lower=1> u_wiv_colnum;
```

```
##   //num unique covariances between REs
##   int<lower=1> n_cov;
##   //design matrix (at level 1):fixed part mean fun. for y1
##   matrix[N, n_b] X_y1_mu_L1;
##   //design matrix (at level 1):random part mean fun. for y1
##   matrix[N, n_u_mu] Z_y1_mu_L1;
##   //design matrix (at level 2):fixed part wiv fun. for y1
##   matrix[total_num_clinics, n_a] X_y1_wiv_L2;
##   //design matrix:observed exposures for y2
##   matrix[J, n_g_obs] X_y2_obs;
##   //repeatedly-measured outcome
##   vector[N] y1;
##   //individual-level outcome
##   vector[J] y2;
##   //subject indicator (of N-length)
##   int<lower=1,upper=J> Ind_L1[N];
##   //subject indicator (of total_num_clinics-length)
##   int<lower=1,upper=J> Ind_L2[total_num_clinics];
##   //clinic indicator (of N-length)
##   int<lower=1,upper=total_num_clinics> clinic_L1[N];
##   //location and scale of priors for betas
##   vector[n_b] beta_prior_loc;
##   vector<lower=0>[n_b] beta_prior_scale;
##   //location and scale of priors for alphas
##   vector[n_a] alpha_prior_loc;
##   vector<lower=0>[n_a] alpha_prior_scale;
##   //location and scale of priors for observed exposures for y2
##   vector[n_g_obs] gamma_obs_prior_loc;
##   vector<lower=0>[n_g_obs] gamma_obs_prior_scale;
##   //location and scale of priors for RE exposures for y2
##   vector[n_g_RE] gamma_RE_prior_loc;
##   vector<lower=0>[n_g_RE] gamma_RE_prior_scale;
##   //eta for LKJcorr prior for RE correlations
##   real<lower=0> LKJcorr_prior;
##   //location and scale of priors for RE SDs
##   real sigma_u_prior_loc;
##   real<lower=0> sigma_u_prior_scale;
##   //location and scale of prior for observed exposures for y2
##   real sigma_y2_prior_loc;
##   real<lower=0> sigma_y2_prior_scale;
##   //location and scale of prior for residual SD (at level 1) for y1
##   real sigma_epsilon_prior_loc;
##   real<lower=0> sigma_epsilon_prior_scale;
##   }
## parameters {
##   //FE coefficients in mean function for y1
##   vector[n_b] beta;
##   //FE coefficients in within-individual variance function for y1
##   vector[n_a] alpha;
##   //FE coefficients of observed exposures for y2
##   vector[n_g_obs] gamma_obs;
##   //FE coefficients of RE exposures for y2
##   vector[n_g_RE] gamma_RE;
##   //Cholesky factor of random effect corr matrix
```

```
##   //(i.e. corr_u = cholesky_corr_u * cholesky_corr_u')
##   cholesky_factor_corr[n_u_total] cholesky_corr_u;
##   //random effect SDs (lower bound ensures half-Cauchy)
##   vector<lower=0>[n_u_total] sigma_u;
##   //unscaled random effects (N(0,1))
##   matrix[n_u_total, J] z_u;
##   //REs at clinic level
##   vector[total_num_clinics] e;
##   //residual SD for y2 (lower bound ensures half-Cauchy)
##   real<lower = 0> sigma_y2;
##   //residual SD for y1 at level 1 (lower bound ensures half-Cauchy)
##   real<lower = 0> sigma_epsilon;
## }
## transformed parameters {
##   //scaled random effects
##   matrix[J, n_u_total] u;
##   u = (diag_pre_multiply(sigma_u, cholesky_corr_u) * z_u)';
## }
## model {
##   //priors
##   beta ~ normal(beta_prior_loc, beta_prior_scale);
##   alpha ~ normal(alpha_prior_loc, alpha_prior_scale);
##   gamma_obs ~ normal(gamma_obs_prior_loc, gamma_obs_prior_scale);
##   gamma_RE ~ normal(gamma_RE_prior_loc, gamma_RE_prior_scale);
##   //normal() not applicable to matrices, hence to_vector
##   //(treats it like a vector but maintains matrix data type)
##   to_vector(z_u) ~ normal(0, 1);
##   cholesky_corr_u ~ lkj_corr_cholesky(LKJcorr_prior);
##   sigma_u ~ cauchy(sigma_u_prior_loc, sigma_u_prior_scale);
##   sigma_y2 ~ cauchy(sigma_y2_prior_loc, sigma_y2_prior_scale);
##   sigma_epsilon ~ cauchy(sigma_epsilon_prior_loc, sigma_epsilon_prior_scale);
##
##   //likelihood
##   y1 ~ normal(X_y1_mu_L1 * beta
##               + rows_dot_product(Z_y1_mu_L1, u[Ind_L1, 1:u_mu_colnum])
##               + e[clinic_L1],
##               sigma_epsilon);
##   e ~ normal(0, sqrt(exp(X_y1_wiv_L2 * alpha + u[Ind_L2, u_wiv_colnum])));
##   y2 ~ normal(X_y2_obs * gamma_obs + u * gamma_RE, sigma_y2);
## }
## generated quantities {
##   corr_matrix[n_u_total] corr_u_complete;
##   vector<lower=-1, upper=1>[n_cov] corr_u;
##   vector[N] y1_pred;
##   vector[J] y2_pred;
##   // return correlation matrix (dropping redundant elements)
##   corr_u_complete = multiply_lower_tri_self_transpose(cholesky_corr_u);
##   corr_u[1] = corr_u_complete[1, 2];
##   corr_u[2] = corr_u_complete[1, 3];
##   corr_u[3] = corr_u_complete[2, 3];
##   // for posterior predictive checks
##   for (n in 1:N) {
##     y1_pred[n] = normal_rng(X_y1_mu_L1[n] * beta
##                             + dot_product(Z_y1_mu_L1[Ind_L1[n]],
```

```
##                                             u[Ind_L1[n], 1:u_mu_colnum])
##                               + e[clinic_L1[n]],
##                               sigma_epsilon);
##   }
##   for (j in 1:J) {
##     y2_pred[j] = normal_rng(X_y2_obs[j] * gamma_obs + u[j] * gamma_RE,
##                               sigma_y2);
##   }
## }
```

Fitting the model

```
data <- read_rdump("joint_L3_data.R")

params_for_summary <- c("beta",
                        "alpha",
                        "gamma_obs",
                        "gamma_RE",
                        "sigma_y2",
                        "sigma_u",
                        "corr_u",
                        "sigma_epsilon")

params_for_post_pred <- c("y1_pred",
                          "y2_pred")

chains <- 4

# (include u (etc.) in pars if wish to save e.g. residuals at that level)
stan_fit <- stan(file = "joint_L3.stan",
                 data = data,
                 pars = c(params_for_summary,
                          params_for_post_pred),
                 chains = chains)
```

Inspecting results

```
print(stan_fit, pars = params_for_summary)

# Launching shinystan to check diagnostics
stan_fit_shiny <- as.shinystan(stan_fit, pars = c(params_for_summary,
                                                  params_for_post_pred))

launch_shinystan(stan_fit_shiny)

# # If wish to save out whole stanfit object (might be large!):
# saveRDS(stan_fit, "stan_fit.RDS")
# # Reading back in:
# stan_fit <- readRDS("stan_fit.RDS")

# # If wish to save out summary:
```

```
# fit_summary <- summary(stan_fit)$summary
# saveRDS(fit_summary, "fit_summary.RDS")
# # Reading back in:
# fit_summary <- readRDS("fit_summary.RDS")
```

WinBUGS

Specifying data

```
# covariates:
x1_L1 <- X_y1_mu_L1[, 2]
x1_L2 <- X_y1_mu_L2[, 2]
x2 <- X_y2[, 2]

# for inverse Wishart prior for covariance matrix
# (specified with df equal to order of matrix, i.e. 'minimally informative'):
R2 <- 2 * cov_u

bugsDat <- list(N = N,
                total_num_clinics = total_num_clinics,
                J = J,
                x1_L1 = x1_L1,
                x1_L2 = x1_L2,
                y1 = y1,
                x2 = x2,
                y2 = y2,
                Ind_L1 = Ind_L1,
                Ind_L2 = Ind_L2,
                clinic_L1 = clinic_L1,
                R2 = R2)
```

Initial values

Here, known parameter values are used (from the data-generating process for the simulated data), but in the case of real data, parameter estimates can be gleaned from simpler models, for example, to provide an informed guess at promising initial values.

```
# precision parameter for covariance matrix:
tau.u <- MASS::ginv(cov_u)

# precision parameter for level 1 variance in y1:
tau <- 1 / sigma_epsilon^2

# precision parameter for residual variance in y2:
tau.y2 <- 1 / sigma_y2^2

# different initial values for each chain:
CV <- 0.1
inits_function <- function(){
  list(
    alpha = rnorm(n = length(alpha), mean = alpha, sd = abs(CV * alpha)),
```

```
    beta = rnorm(n = length(beta), mean = beta, sd = abs(CV * beta)),
    gamma = rnorm(n = length(gamma), mean = gamma, sd = abs(CV * gamma)),
    u = jitter(u, amount = 2),
    tau.u = jitter(tau.u),
    tau = jitter(tau),
    tau.y2 = jitter(tau.y2)
  )
}

inits1 <- inits_function()
inits2 <- inits_function()
inits3 <- inits_function()
inits4 <- inits_function()

inits <- list(inits1,
              inits2,
              inits3,
              inits4)
```

Specifying the model

NB: Assuming this BUGS model is saved in its own file.

```
writeLines(readLines("joint_L3.bugs"))
```

```
## model {
##   ## Level 1 definition
##   for (i in 1:N) {
##     y1[i] ~ dnorm(mu[i], tau)
##     mu[i] <- beta[1] + beta[2] * x1_L1[i] +
##         u[Ind_L1[i], 1] + u[Ind_L1[i], 2] * x1_L1[i] + e[clinic_L1[i]]
##   }
##   ## Higher level definitions
##   for (j in 1:J) {
##     y2[j] ~ dnorm(mu.y2[j], tau.y2)
##     mu.y2[j] <- gamma[1] + gamma[2] * x2[j] + gamma[3] * u[j, 1] +
##         gamma[4] * u[j, 2] + gamma[5] * u[j, 3]
##     u[j, 1:3] ~ dmnorm(zero2[1:3], tau.u[1:3, 1:3])
##   }
##   for (i in 1:total_num_clinics) {
##     e[i] ~ dnorm(0, tau.e[i])
##     tau.e[i] <- 1 / exp(alpha[1] + alpha[2] * x1_L2[i] + u[Ind_L2[i], 3])
##   }
##   ## Priors for fixed effects
##   for (k in 1:2) {
##     beta[k] ~ dflat()
##     alpha[k] ~ dflat()
##   }
##   for (k in 1:5) {
##     gamma[k] ~ dflat()
##   }
##   ## Priors for random terms
##   tau ~ dgamma(0.001,0.001)
```

```
##   sigma2 <- 1/tau
##   tau.y2 ~ dgamma(0.001, 0.001)
##   sigma2.y2 <- 1 / tau.y2
##   for (i in 1:3) {zero2[i] <- 0}
##   tau.u[1:3, 1:3] ~ dwish(R2[1:3, 1:3], 3)
##   sigma2.u[1:3, 1:3] <- inverse(tau.u[,])
## }
```

Fitting model

```
parameters.to.save <- c("beta",
                        "sigma2.u",
                        "alpha",
                        "sigma2",
                        "gamma",
                        "sigma2.y2")

chains <- 4

BUGS_fit <- bugs(
  data = bugsDat,
  inits = inits,
  parameters.to.save = parameters.to.save,
  model.file = "joint_L3.bugs",
  n.chains = chains,
  bugs.directory = bugs.directory,
  program = "WinBUGS",
  working.directory = NULL,
  clearWD = TRUE,
  save.history = FALSE)
```

Inspecting results

```
print(BUGS_fit, digits.summary = 3)

## estimate of SD for random effects:
diag(sqrt(BUGS_fit$mean$sigma2.u))

## estimate of correlation between random effects:
cov2cor(BUGS_fit$mean$sigma2.u)[lower.tri(cov2cor(BUGS_fit$mean$sigma2.u))]

## estimate of residual SD at level 1:
sqrt(BUGS_fit$mean$sigma2)

## estimate of residual SD for y2:
sqrt(BUGS_fit$mean$sigma2.y2)

# # If wish to save out bugs object (might be large!):
# saveRDS(BUGS_fit, "BUGS_fit.RDS")
# # Reading back in:
# BUGS_fit <- readRDS("BUGS_fit.RDS")
```

References

Betancourt, M. 2017. "Diagnosing Biased Inference with Divergences." https://mc-stan.org/users/documentation/case-studies/divergences_and_bias.html.

Browne, W.J. 2018. *MCMC Estimation in Mlwin V3.02*. Centre for Multilevel Modelling, University of Bristol.

Charlton, C., J. Rasbash, W.J. Browne, M. Healy, and B. Cameron. 2018. "MLwiN Version 3.02." Centre for Multilevel Modelling, University of Bristol.

Gabry, J., and B. Goodrich. 2018. "Prior Distributions for Rstanarm Models." http://mc-stan.org/rstanarm/articles/priors.

Lunn, D.J., A. Thomas, N. Best, and D. Spiegelhalter. 2000. "WinBUGS — a Bayesian Modelling Framework: Concepts, Structure, and Extensibility." *Statistics and Computing* 10: 325–37.

McElreath, R. 2016. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan.* Boca Raton, FL: CRC Press.

Sorensen, T., S. Hohenstein, and S. Vasishth. 2016. "Bayesian Linear Mixed Models Using Stan: A Tutorial for Psychologists, Linguists, and Cognitive Scientists." *Quantitative Methods for Psychology* 12 (3): 175–200.

Stan Development Team. 2018. *Stan Modeling Language Users Guide and Reference Manual, Version 2.18.0.* http://mc-stan.org.

Sturtz, Sibylle, Uwe Ligges, and Andrew Gelman. 2005. "R2WinBUGS: A Package for Running Winbugs from R." *Journal of Statistical Software* 12 (3): 1–16. http://www.jstatsoft.org.