CRN tutorial: Gaussian quantitative genetic model

Jordan S. Martin

Contents

Introduction

This tutorial provides a gentle introduction into the application of a covariance reaction norm (CRN) model in R for quantitative genetic analysis. The first component of the tutorial focuses on the analysis of simulated data using a multivariate Gaussian model with single measures per subject. I then cover basics for preparing one's own data for a CRN analysis, as well as important extensions to the basic CRN model, such as for the inclusion of spatiotemporal heterogeneity via Gaussian processes, and for repeated measures data. The worked example on meerkat social behavior also provides additional code for dealing with the messy reality of real datasets. I intend to continually expand this tutorial as is relevant for the concerns or questions I receive from researchers applying CRNs. Therefore, if you'd like further input on an issue not covered here, please reach out to me by email (jordanscott.martin@eawag.ch) and I may include answers to your questions in subsequent updates.

General overview of Stan

CRN analyses can be conducted using the Stan statistical programming language. Stan is an open-source programming language for estimating complex probabilistic models, which can interface with multiple statistical environments such as R. Stan facilitates fully Bayesian inference using a state-of-the-art Markov Chain Monte Carlo (MCMC) sampling technique known as the No U-Turn Sampler (NUTS), which has been found to perform particularly well for quantitative genetic analysis. Stan is thus an ideal platform for flexibly estimating CRNs. Stan uses its own language for writing probabilistic models, including a variety of built-in functions designed to aid in efficient computation. The biggest conceptual hurdle for new users of Stan is likely to be the absence of an intuitive R-like syntax for specifying model formulas, such as formulas like `y ~ x + (1|z)` that can be used to quickly specify complex generalized linear mixed-effects models. These formulas facilitate highly efficient statistical modeling, but do so at the cost of limiting users' ability to specify atypical model structures. Instead, Stan provides the benefit of nearly unlimited flexibility in model specification, with the added cost of a steeper learning curve. In particular, models must be formally specified with mathematically appropriate likelihood functions, rather than this process being handled on the back-end through textual inputs from the user such as `family= poisson(link = "log")`. This may at first seem like a cumbersome task, but it provides a degree of independence and creativity for data analysis that is otherwise unavailable. It is this autonomy that makes it possible to unbiasedly estimate CRNs in Stan, which to the best of my knowledge cannot be accomplished with any other mainstream statistical software. Nonetheless, it is important to recognize that some practice and trial-and-error will be required to gain competency and comfortability with Stan. Therefore, I encourage those interested in CRN analysis to review the Stan Reference Manual, as well the extensive collection of Stan Case Studies, which will provide a more robust foundation for extending the basic models prevented here to more complex scenarios.

This tutorial will explain the basic structure of a Stan script in the process of introducing the CRN implementation. For the present purposes, it suffices to know that a `.stan` file for estimating a Bayesian model in Stan can be composed in any text editor (e.g. RStudio or using `write()` directly in R) and will always be composed of a series of programming blocks that are processed sequentially.

```r
write("// for Stan comments
    functions{...} // Stan models are composed of
    data {...} // multiple programming blocks
    transformed data {...} //only data, parameters, and model
    parameters {...} //blocks are necessary
    transformed parameters {...} //other blocks are optional
    model {...}
    generated quantities {...} ",
  "mod.stan")
```

The `transformed data{}`, `transformed parameters{}`, and `generated quantities{}` blocks are optional and can be used to create additional quantities of interest beyond the initial data provided for the Stan model in `data{}`, the essential model parameters estimated in `parameters{}`, and the likelihood function and priors specified in `model{}`. The utility of these optional blocks will be explored further below. Importantly, any quantities specified in `model{}` will not be saved in the output of the Stan model after estimation. As will become apparent in subsequent sections of the tutorial, this feature is very helpful for saving memory in a complex analysis such as a CRN model.

In most statistical software, empirical data are input with a single matrix or dataframe. For Stan, a list is instead required with data for each scalar (real or integer), vector, or matrix object declared in the `data{}` block of the corresponding `.stan` file. For example, we can make a list `stan.dl`

```r
stan.dl = list(N = nrow(df), #sample size
                 I = length(unique(df$id)), #number of subjects
                 P = ncol(X), #number of predictors
                 id = df$id, #index subjects per observation
```

```
                  X = X, #design matrix of predictors, incl. intercept
                  A = A, #relatedness matrix
                  z1 = df$z1, #response variable 1
                  z2 = df$z2 #response variable 2
                  )
```

It is essential that every quantity declared in `data{}` is present in the supplied data list, and there cannot be any NAs in the supplied list. This means that in some cases, it will be necessary to either drop cases or conduct missing data imputation manually prior to the analysis, or to input an integer that is used within the Stan model to differentiate missing values (e.g. -99 indicates NA). For general application, it also useful to supply environmental covariates as a design matrix `X` rather than as individually specified variables. This can be easily accomplished in R.

```
X = data.frame(model.matrix(response ~ pred1 + pred2 + pred3 * pred4, data = df))
```

The names of the variables in the data list `stan.dl` should be coded in `data{}` along with their expected dimensions, which ensures that inappropriate data structures or likelihood functions will throw errors. For instance, in the general case, the data block might look something like this. Note that Stan uses `//` rather than `#` for comments.

```
data {
  int<lower=1> N; //length of response vector/total observations
  int<lower=1> I; //number of individuals
  int<lower=1> P; //number of covariates

  int<lower=1> id[N]; //N length index linking observations of z to subject ID

  matrix[N,P] X; //design matrix of environmental covariates
  matrix[I,I] A; //relatedness matrix

  vector[N] z1; //vector of real valued responses (e.g. for Gaussian model)
  int z2[N]; //response variable of counts (e.g. for Poisson model)
}
```

This declarative approach requires that particular attention is given to the order of data input to the model, as values will need to be appropriately aligned and indexed throughout the model specification. This is because Stan will not take character strings for indexing subjects (e.g. “monkey1”, “subjectA015”, “juvenileB”, etc.) or other random effects such as group identity, season, time of day, etc. This can also be easily accomplished in R. For instance, if one is attempting to change the character string names in an R object `df$subj` for indexing in Stan

```
key.id = unique(df$subj) #list all unique subject IDs
new.id = seq(1:length(key.id)) #create numeric index of equal length
df$id = new.id[match(df$subj, key.id)] #create new id for Stan
```

Users will need to manually ensure that the integers used to index subjects are also appropriately aligned with the order of any other data structures corresponding to those subjects. The `A` matrix, for instance, should be arranged so that row 1 corresponds to the values expected for subject 1 and so on.

```
dimnames(A)[[1]] = new.id[match(dimnames(A)[[1]], key.id)]
dimnames(A)[[2]] = new.id[match(dimnames(A)[[2]], key.id)]
A = as.matrix(A[order(as.numeric(row.names(A))), order(as.numeric(colnames(A)))])
```

This may seem cumbersome at first, but it allows for many benefits unavailable in more standard statistical software, such as facilitating multivariate models with heterogeneous dimensions (differing N per response variable). See the meerkat data analysis files for a concrete example.

Once the model is written in a `.stan` file and the corresponding data list is prepared, the stan model can be compiled in C++ and estimated in R using either the rstan or cmdstanr packages. Details on the installation of these packages can be found elsewhere online (see hyperlinks). This tutorial will focus on the use of cmdstanr for model estimation, as it provides utility functions for speeding up MCMC chains that are currently unavailable in rstan. Optional packages such as shinystan can also be used to aid in inspection of model results and diagnostics.

```r
library(shinystan)
library(cmdstanr)

#directory for cmdstan installation
set_cmdstan_path("...")

#compile model
mod = cmdstan_model(stan_file = "model.stan", #set appropriate directory
                    stanc_options = list("O1")) #this speeds up estimation

#estimate model
est = mod$sample(
  data = stan.dl,
  iter_sampling = 500, #how many samples per chain after warmup
  iter_warmup = 500, #warmup for MCMC chains to 'burn in' and converge
  init = 0.01, #values for initializing chains
  chains = 4,  #how many chains to run
  parallel_chains = 4, #how many in parallel (check your machine)
  adapt_delta = 0.80, #default, >0.80 aids removing divergent transitions (essential)
  max_treedepth = 10, #default, 10+ to 12 can remove treedepth warnings
  refresh = 10) #how many iterations to update progress of MCMC chains

saveRDS(est, "CRN_fit.RDS") #save results
launch_shinystan(est) #GUI for investigating model fit
```

Worked example using simulated data

For this tutorial, we are going to consider the simplest case of multivariate Gaussian traits (3 response variables) with a single measure of each trait per subject. For estimating the CRN of these traits, we'll consider the additive effects of 3 continuous environmental predictors. As will generally be the case in field or laboratory research, it will be assumed that each measured level of these continuous environments, herein referred to as *contexts*, is experienced by multiple individuals. If no contexts were shared across subjects, we could not meaningfully estimate how the environment changes among-individual covariances. However, as shown in the main text, unbiased estimates can be obtained even with small numbers of individuals per context, and information can be pooled across contexts during model estimation even in cases where some contexts are only experienced by a single individual. In this example, we will assume there are 25 unique contexts across each continuous predictor experienced by 500 subjects, such that 20 individuals are measured per context assuming balanced sampling. For example, our field study might involve phenotyping 20 individuals per plot across 25 plots, with each plots' unique combination of local temperature, rainfall, predator density, resource availability, etc. serving as a multivariate environmental context for estimating the CRN.

Simulate multivariate Gaussian data

The helper function `sim_CRN_QG()` from the `CRN functions.R` file can be used to quickly simulate data for this scenario that is appropriately formatted for Stan. This function requires arguments `N` for the total sample size, `npred` for the number of continuous environmental predictors, `Nc` for the number of contexts measured for the continuous covariates, `ntrait` for the number of phenotypic traits, and `l_es` and `u_es` for the absolute-valued lower and upper bound of effect sizes for the CRN parameters. The simulated parameter values will be drawn from uniform distributions $U(l_{es}, u_{es})$, with signs randomly flipped to allow for both positive and negative effects. Additional optional arguments are available for returning only an empirical data frame (`standl = F`; see below) as well as for changing the informativeness of the relatedness matrix (`eta = 1`). As `eta` increases, the marginal relatedness coefficient among any two subjects will tend to become smaller (i.e. more sparse). To see how the expected distribution of coefficients scales, simply run `hist( 2 * rbeta(1e4, eta + ntrait/2, eta + ntrait/2) - 1)` in your console and replace `eta` and `ntrait` with your desired values. The default `eta = 1` generally produces a highly informative matrix that aids identification of model parameters with small sample sizes.

```r
#custom functions (make sure to set appropriate directory)
#the file should automatically install and load any necessary packages
#that are not found in your R library
source("CRN functions.R")

#simulate data
stan.dl = sim_CRN_QG(N = 500, Nc = 25, npred = 3, ntrait = 3, l_es = 0.3, u_es = 0.6)
```

From our Stan data list, we can see the simulated multivariate normal responses

```r
head(stan.dl$z, 5)
```

```
##            z1        z2         z3
## 1   -2.960651 -1.752683  0.5585328
## 1.1 -1.903928  2.566962 -2.3298299
## 1.2  2.559329  0.232979  0.8723139
## 1.3 -2.534577  1.672239 -1.6355481
## 1.4 -1.463970  2.912229  1.0770779
```

the design matrix containing the standardized, continuous environmental measures

```
head(stan.dl$X, 5)
```

```
##   X.Intercept.         X1         X2         X3
## 1            1  1.4798931 -1.2979887  1.0242250
## 2            1  1.1623586  0.1740553  0.3820382
## 3            1  1.5399547  0.1036221  0.3400949
## 4            1 -1.0097621  2.2361192 -1.8333117
## 5            1 -0.6390358  0.1710165 -1.2846423
```

the relatedness matrix among subjects (simply a random positive-definite correlation matrix)

```
stan.dl$A[1:5,1:5]
```

```
##              [,1]        [,2]         [,3]         [,4]         [,5]
## [1,]  1.000000000 -0.05849467 -0.005953564 -0.078991955 -0.041420702
## [2,] -0.058494670  1.00000000 -0.096756841  0.052342010  0.015465036
## [3,] -0.005953564 -0.09675684  1.000000000  0.001620696 -0.033395026
## [4,] -0.078991955  0.05234201  0.001620696  1.000000000 -0.002038257
## [5,] -0.041420702  0.01546504 -0.033395026 -0.002038257  1.000000000
```

the mean, variance, and canonical partial correlation RN parameters

```
stan.dl$B_m
```

```
##            [,1]       [,2]       [,3]
## [1,] -0.5510682 -0.3304345  0.5990958
## [2,] -0.5238086  0.4861180 -0.4757771
## [3,] -0.3611816 -0.3217273 -0.3215069
## [4,] -0.5030098 -0.5635266 -0.3295473
```

```
stan.dl$B_v
```

```
##           [,1]       [,2]       [,3]
## [1,] 0.4044814 -0.4858206 -0.5622857
## [2,] 0.5840890  0.3253103 -0.5429411
## [3,] 0.3203922 -0.4347445  0.3186363
## [4,] 0.5917739  0.5965077 -0.4768660
```

```
stan.dl$B_cpc
```

```
##            [,1]       [,2]       [,3]
## [1,]  0.5901370 -0.3505891 -0.5995460
## [2,] -0.5303605 -0.3171246 -0.3809773
## [3,] -0.3678121 -0.5484954 -0.4715446
## [4,]  0.3075629 -0.4135420 -0.5911267
```

and the resulting context-specific **G** matrices

```
stan.dl$Gcov[1:5]
```

```
## [[1]]
##            [,1]       [,2]       [,3]
## [1,]  4.3021500  1.9942155 -0.3247276
## [2,]  1.9942155  3.2247498 -0.5001433
## [3,] -0.3247276 -0.5001433  0.1035413
## 
## [[2]]
##             [,1]        [,2]       [,3]
## [1,]  3.91669640  0.05492583 -0.7669912
## [2,]  0.05492583  1.04553189 -0.3162549
## [3,] -0.76699122 -0.31625489  0.2671090
## 
## [[3]]
##            [,1]       [,2]       [,3]
## [1,]  4.6571844 -0.3735438 -0.7807001
## [2,] -0.3735438  1.1888216 -0.2195882
## [3,] -0.7807001 -0.2195882  0.2170663
## 
## [[4]]
##             [,1]        [,2]        [,3]
## [1,]  0.57477283 -0.04578692 -0.76746853
## [2,] -0.04578692  0.05613298 -0.02073951
## [3,] -0.76746853 -0.02073951  4.81968398
## 
## [[5]]
##           [,1]      [,2]      [,3]
## [1,] 0.5095650 0.1455216 0.2520410
## [2,] 0.1455216 0.2155951 0.2284703
## [3,] 0.2520410 0.2284703 1.5710864
```

among other simulated values and parameters necessary as input for the `CRN_tutorial_mod.stan` file. For example, the Stan model needs a matrix `cmat` indexing which subjects are observed in which contexts (rows)

```
head(stan.dl$cmat, 5)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
## [1,]    1    2    3    4    5    6    7    8    9    10    11    12    13    14
## [2,]   21   22   23   24   25   26   27   28   29    30    31    32    33    34
## [3,]   41   42   43   44   45   46   47   48   49    50    51    52    53    54
## [4,]   61   62   63   64   65   66   67   68   69    70    71    72    73    74
## [5,]   81   82   83   84   85   86   87   88   89    90    91    92    93    94
##      [,15] [,16] [,17] [,18] [,19] [,20]
## [1,]    15    16    17    18    19    20
## [2,]    35    36    37    38    39    40
## [3,]    55    56    57    58    59    60
## [4,]    75    76    77    78    79    80
## [5,]    95    96    97    98    99   100
```

indices linking observations to individuals `id`, contexts `c_d`, and individuals' positions in `cmat`

```r
head(stan.dl$id,5)
```

```
## [1] 1 2 3 4 5
```

```r
head(stan.dl$c_id, 5)
```

```
## [1] 1 1 1 1 1
```

```r
head(stan.dl$idc, 5)
```

```
## [1] 1 2 3 4 5
```

counts of the number of individuals per context `cn`, the max number number of individuals in any context `cm`, and the total number of context-specific individual values to be estimated `cnt`

```r
head(stan.dl$cn,5)
```

```
## [1] 20 20 20 20 20
```

```r
stan.dl$cm
```

```
## [1] 20
```

```r
stan.dl$cnt
```

```
## [1] 500
```

as well as the total number of observations `N` and subjects `I` (both equivalent to `cnt` in the case of balanced sampling and non-repeated measurements), the total number of contexts `C`, the total of traits/dimensions `D`, the total number of environmental predictors `P` (including a global intercept).

```r
str(stan.dl, list.len = 5)
```

```
## List of 21
##  $ N    : int 500
##  $ C    : int 25
##  $ I    : int 500
##  $ D    : num 3
##  $ P    : int 4
##   [list output truncated]
```

Inspecting the code inside this function (simply type `sim_CRN_QG` into the R console) will aid in generating these values manually and specifying the appropriate stan data list for your own analysis. See the meerkat data analysis as well for a worked example of data preparation using real measurements.

Compose CRN model for Stan

Eq. 4-7 in the main text describe useful computational tricks that can be implemented in Stan for speeding up model estimation. Those interested in understanding why and how these tricks work can see the main text for further details. However, it is not essential that one fully appreciates the mechanics of these steps to carry out a CRN analysis. It is simply important to understand that some chunks of Stan code will need to be present in every CRN analysis to take advantage of these efficiency gains. As explained further below, custom functions in R can then be used to get more easily interpretable outputs from the model for summarizing and visualizing results.

Every CRN model will need to include two custom functions `sum_square_x` and `lkj_to_chol_corr` in the `functions{}` block. These functions are taken from prior work by Dan Schrage and facilitate using the environmental covariates to predict canonical partial correlations, from which effects on the genetic correlations can subsequently be derived.

```
functions {

//functions are used fom prior work by
//Dan Schrage (https://gitlab.com/dschrage/rcovreg)

  real sum_square_x(matrix x, int i, int j) {
    int j_prime;
    real sum_x = 0;
    if(j==1) return(sum_x);

    j_prime = 1;
    while(j_prime < j) {
      sum_x += x[i,j_prime]^2;
      j_prime += 1;
    }
    return(sum_x);
  }

  matrix lkj_to_chol_corr(row_vector constrained_reals, int ntrait) {
    int z_counter;
    matrix[ntrait,ntrait] x;

    z_counter = 1;
    x[1,1] = 1;
    for(j in 2:ntrait) {
      x[1,j] = 0;
    }
    // Fill the diagonal and lower triangle of x.
    for(i in 2:ntrait) {
      for(j in 1:ntrait) {
        if(i==j) {
          x[i,j] = sqrt(1 - sum_square_x(x, i, j));
        } else if(i > j) {
          x[i,j] = constrained_reals[z_counter]*sqrt(1 - sum_square_x(x, i, j));
          z_counter += 1;
        } else {
          x[i,j] = 0;
        }
      }
    }
    return(x);
  }
}
```

As noted above, `data{}` will then include statements for all of the objects in `stan.dl` that are necessary for programming the model. The data list may also include variables of interest that are not used in the Stan model, such as the context-specific **G** matrices used for simulation (`stan.dl$Gcov`).

```
data {
  int<lower=1> N; //total number of observations
  int<lower=1> C; //total number of environmental contexts
  int<lower=1> I; //total number of subjects
  int<lower=1> D; //total number of traits/dimensions
  int<lower=0> P; //total number of environmental predictors (+ intercept)

  int<lower=0> id[N]; //index linking observations to individuals
  int<lower=0> c_id[N]; //index linking observations to contexts
  int<lower=0> idc[N]; //index linking individuals to positions in cmat

  matrix[C,P] X; //environmental predictor matrix (+ intercept)
  matrix[I,I] A; //relatedness matrix

  int<lower=1> cm; //max number of individuals observed in a context
  int cmat[C, cm]; //matrix with all individuals observed in each context (row)
  int<lower=0> cn[C]; //count of individuals observed per context
  int<lower=1> cnt; //total number of individuals across contexts

  vector[D] z[N]; //multivariate normal response variables
}
```

In `transformed data{}`, any additional quantities or data objects of interested can be generated that are simply transformations of the values in `data{}`. For a CRN analysis, it will be necessary to include the following code for taking advantage of the Cholesky decomposition and QR decomposition (see Eq. 4-7 main text).

```
transformed data{
  matrix[I, I] LA = cholesky_decompose(A);
  int ncor = (D*(D-1))/2; //unique cov/cor parameters
  // Compute, thin, and then scale QR decomposition
  matrix[C, P] Q = qr_thin_Q(X) * sqrt(C-1);
  matrix[P, P] R = qr_thin_R(X) / sqrt(C-1);
  matrix[P, P] R_inv = inverse(R);
}
```

It suffices to say that we will now use `LA` in place of the `A` matrix and the `Q` matrix in place of the `X` matrix for computational efficiency. In `parameters{}`, we now put only the basic parameter values from which all other parameters of interest can be derived through transformation. For example, we do not directly specify all of the context-specific **G** matrices, but instead only specify the CRN parameters from which those matrices will be predicted in subsequent blocks. The `q` subscript is used here to indicate that these RN parameter values are defined with respect to the `Q` matrix from the QR decomposition. The RN parameters appropriately scaled to `X` can then be generated further below.

```
parameters {
  //fixed effects
  matrix[P, D] B_mq; //RN of means
  matrix[P, D] B_vq; //RN of variances
  matrix[P, ncor] B_cpcq; //RN of canonical partial correlations

  //random effects
  matrix[cnt, D] Z_G; //all context-specific additive genetic values
  cholesky_factor_corr[D] L_E; //Cholesky corr matrix for residuals
  vector<lower=0>[D] sd_E; //residual standard deviations
}
```

As is explained in Eq. 7 of the main text, these parameters need to be further transformed to derive appropriately scaled additive genetic values using a non-centered, matrix normal parametrization. This could be accomplished in the `transformed parameters{}` block. However, since we are not interested in saving posteriors for each of the individual-level predictions (population inferences are of primary concern), we can greatly reduce the resulting file size by doing these transformations in the `model{}` block instead. Any objects declared here can be used for specifying the likelihood and priors but will not be saved with the model output.

```
model {
  //predicted values from reaction norms
  //means
  matrix[C, D] mu = Q * B_mq;

  //variances
  matrix[C, D] sd_G = sqrt(exp(Q * B_vq));

  //correlations (expressed as canonical partial correlations)
  matrix[C, ncor] cpc_G = tanh(Q * B_cpcq);

  //scale context-specific multivariate additive genetic effects
  matrix[cnt, D] mat_G;
  int pos = 1; //keep track of position 1:cnt
  for(c in 1:C){
      mat_G[pos:(pos+cn[c]-1)] =
        LA[cmat[c,1:cn[c]],cmat[c,1:cn[c]]] * Z_G[pos:(pos+cn[c]-1)] *
                diag_pre_multiply(sd_G[c],lkj_to_chol_corr(cpc_G[c], D))';
      pos = pos + cn[c];
  }

//likelihood
  for(n in 1:N){
  row_vector[3] lin_pred = mu[c_id[n]] + mat_G[idc[n]];
  z[n] ~ multi_normal_cholesky(lin_pred,
                              //Chol factorized lower-tri residual cor matrix
                              diag_pre_multiply(sd_E, L_E));
  }

  //priors
  to_vector(B_mq) ~ normal(0,1);
  to_vector(B_vq) ~ normal(0,1);
  to_vector(B_cpcq) ~ normal(0,1);
  to_vector(Z_G) ~ std_normal();

  sd_E ~ exponential(2);
  L_E ~ lkj_corr_cholesky(2);
}
```

In the first steps, the model calculates the context-specific means `mu`, variances `sd_G`, and canonical partial correlations `cpc_G` using the transformed `Q` matrix of environmental covariates and accompanying RN parameters `B_mq`, `B_vq`, and `B_cpcq`. The `lkj_to_chol_corr` function is then used to return the context-specific (`c in 1:C`), lower-triangular Cholesky factorized genetic correlation matrices, which are calculated from the context-specific canonical partial correlations `cpc_G`. This correlation matrix is then scaled by the context-specific genetic standard deviations `sd_G` using the `diag_pre_multiply` function to return context-specific, lower-triangular Cholesky factorized genetic (co)variance matrices. Finally, the matrix normal parametrization is carried out for each context by moving sequentially along the matrix of standardized breeding values `Z_G` using the count of individuals per context `cn` and scaling these values by the transposed (`'`) context-specific genetic (co)variance matrix and the corresponding individuals' genetic relatedness, using the appropriate subset of `LA` indexed from `cmat`. With the appropriate additive genetic values now specified in `mat_G`, each individuals' (`n in 1:N`) trait values can be brought together into the linear predictors `lin_pred = mu[c_id[n]] + mat_G[idc[n]]` (i.e. context-specific means + breeding values) for their measured phenotypes `z[n]`. To further speed up computation, the residual (co)variance among these phenotypes is specified using the Cholesky decomposition and accompanying built-in Stan likelihood function `multi_normal_cholesky`.

In the general case, the code generating `mu`, `sd_G`, `cpc_G`, and `mat_G` can simply be copy-pasted into any CRN model irrespective of the complexity or specifics of the analysis. Researchers may instead need to include additional terms in the linear predictors `lin_pred` (e.g. other random effects, adjusted factors, offsets, etc.), declare distinct distributions (e.g. Poisson, binomial, gamma, beta, etc.) for the response variables, or use more complex indexing to bring the appropriate context-specific values into the likelihood function (e.g. if response variables differ in length due to repeated measurements). See the meerkat data analysis scripts for an example including each of these extensions. For phenotypic analysis in the absence of genetic data on relatedness, the `LA` component of the code can simply be removed to indicate the implicit assumption of independent random effects. Also note that priors specified with `std_normal()` should not be changed, while others are at the discretion of the researcher applying the model. The priors shown above are general purpose weakly regularizing priors that I recommend for the standard case.

Finally, the `generated quantities{}` block can be used to return appropriate parameter values by undoing the Cholesky and QR decompositions used for speeding up model estimation.

```
generated quantities{
  matrix[D,D] E = L_E * L_E'; //residual correlations
  matrix[P,D] B_m; //mean RN parameters for X
  matrix[P,D] B_v; //variance RN parameters for X
  matrix[P,ncor] B_cpc; //partial correlation RN parameters for X

  for(d in 1:D){
    B_m[,d]= R_inv * B_mq[,d];
    B_v[,d]= R_inv * B_vq[,d];
    }

  for(d in 1:ncor){
    B_cpc[,d]= R_inv * B_cpcq[,d];
    }
}
```

The complete model combining these code blocks can be found in the accompanying `CRN_tutorial_mod.stan` file. Any CRN model may begin by modifying this Stan file to meet specific needs of a particular analysis.

Estimate the CRN model in Stan

Let's now compile the model and estimate it with cmdstanr, using the basic approach outlined above. After the MCMC chains finish running, posterior samples are extracted into the rstan .csv format and can be nicely visualized and checked for appropriate convergence in shinystan. If the CRN model returns warnings about reaching the max treedepth during sampling, the model can be run again with a slightly higher `max_treedepth` value (e.g. 11 or 12 instead of 10). This will make the MCMC chains take longer to finish running. Warnings about divergent transitions indicate the model should be estimated again with a slightly higher `adapt_delta` value (e.g. 0.90 or 0.95 instead of the default 0.80). Further information about checking diagnostics and the convergence of the MCMC chains can be found elsewhere online.

```r
library(shinystan)
library(cmdstanr)
library(rstan)

#directory for cmdstan installation
set_cmdstan_path("...")

#compile model
CRN_mod = cmdstan_model(stan_file = "CRN_tutorial_mod.stan",
                    stanc_options = list("O1"))

#estimate model
est = CRN_mod$sample(
    data = stan.dl,
    iter_sampling = 500,
    iter_warmup = 500,
    init = 0.01,
    chains = 4,
    parallel_chains = 4,
    adapt_delta = 0.80,
    max_treedepth = 10,
    refresh = 10)

saveRDS(est, "CRN_fit.RDS")
launch_shinystan(est)
```

Summarize results

Once the model is estimated and its diagnostics are appropriately checked to ensure reliable results, we are ready to investigate our findings. We'll again use custom functions supplied in the accompanying `CRN functions.R` file to summarize the results of the CRN analysis, as well as to generate beta coefficients on the scale of genetic correlations rather than canonical partial correlations used for the Stan model. Greater details on each function and their arguments can be found in the R file. Given that Stan uses numeric rather than character labels for indexing vectors and matrices across response variables, the custom functions take an input string of trait names to output more easily interpreted results. We'll also need to pull out the posterior estimates from our model using the rstan function `extract()`.

```r
#custom functions for getting results from Stan model
source("CRN functions.R")

#name of traits in order of stan.dl$z columns (1, 2, ..., D)
#(e.g. "count_egg", "limb.length", "BD1", etc.)
traits = paste0("Z",seq(1:stan.dl$D)) #z1-z3

#get posterior MCMC samples
post = extract(CRN_fit)
```

First we summarize the RN parameters for genetic variances using the `v_beta_f()` function. This function returns a dataframe with each row corresponding to a posterior sample of the beta coefficients for each trait listed in the column `element`. Keep in mind that these effect sizes are on the log scale due to the use of the log link.

```r
#get posteriors of RN parameters in long format
v_b = v_beta_f(x = stan.dl$X, traits = traits, v_b = post$B_v)
head(v_b, 3)
```

```
##   element X.Intercept.         X1         X2         X3
## 1      Z1     0.385764  0.343493  0.385977  0.658845
## 5      Z2    -0.567767  0.401308 -0.584942  0.825341
## 9      Z3    -0.184124 -0.975593  0.389174 -0.280420
```

We'll summarize each effect by its posterior median (a more robust point estimate than the mean), the posterior probability in support of a positive or negative effect (in the direction of the median), and its 90% Bayesian credible interval (CI).

```r
#posterior median
aggregate(.~ element, data = v_b, FUN = function(x)
  round(median(x),2))
```

```
##   element X.Intercept.    X1    X2    X3
## 1      Z1        -0.21  0.71  0.29  0.67
## 2      Z2        -0.45  0.26 -0.62  0.63
## 3      Z3        -0.43 -0.65  0.24 -0.82
```

```r
#posterior probability in direction of posterior median
aggregate(.~ element, data = v_b, FUN = function(x)
  sum(sign(median(x))==sign(x))/length(x))
```

```
##   element X.Intercept.      X1    X2      X3
## 1      Z1      0.66650 0.99750 0.909 0.99850
## 2      Z2      0.86925 0.77825 0.982 0.99600
## 3      Z3      0.95300 0.99825 0.922 0.99975
```

```r
#posterior 90% CIs
aggregate(.~ element, data = v_b, FUN = function(x)
  round(quantile(x, c(0.05,0.95)),2), simplify = F)
```

```
##   element X.Intercept.          X1          X2          X3
## 1      Z1  -1.10, 0.38   0.31, 1.22  -0.07, 0.70   0.28, 1.20
## 2      Z2  -1.38, 0.17  -0.35, 0.82 -1.25, -0.12   0.24, 1.11
## 3      Z3 -0.96, -0.01 -1.06, -0.26  -0.04, 0.56 -1.24, -0.43
```

Results indicate that the first continuous covariate x_1 on average increased the genetic variance of z_1 ($\beta_{\sigma^2_a} =$ 0.71, $p_+ = 0.9975$) and decreased the genetic variance of z_3 ($\beta_{\sigma^2_a} =$ -0.65, $p_- = 0.99825$), with very weak statistical support for a positive effect on the genetic variance of z_2 ($\beta_{\sigma^2_a} = 0.26$, $p_- = 0.77825$); the second continuous covariate x_2 showed modest evidence for increasing the genetic variance of z_1 ($\beta_{\sigma^2_a} = 0.29$, $p_+ =$ 0.909) and z_3 ($\beta_{\sigma^2_a} = 0.24$, $p_+ = 0.922$), with much stronger evidence that x_2 decreased the genetic variance of z_2 ($\beta_{\sigma^2_a} =$ -0.62, $p_- = 0.982$); and the third genetic covariate x_3 increased the genetic variance of z_1 ($\beta_{\sigma^2_a} =$ 0.67, $p_+ = 0.9985$) and z_2 ($\beta_{\sigma^2_a} = 0.63$, $p_+ = 0.996$) but decreased the genetic variance of z_3 ($\beta_{\sigma^2_a} =$ -0.82, $p_+ = 0.99975$).

The same approach can be taken to summarize RN parameters for genetic correlations, after appropriately transforming the RN parameters for canonical partial correlations from the Stan model using the custom `cor_beta_f()` function.

```r
#generate beta coefficients for genetic correlations
cor_b = cor_beta_f(x = stan.dl$X, traits = traits, cpc_b = post$B_cpc)
head(cor_b, 3)
```

```
##   element X.Intercept.       X1         X2       X3
## 1   Z1_Z2      1.40334 -1.50431  0.0653383 0.955857
## 2   Z1_Z2      0.80565 -1.02439 -0.2303300 0.671019
## 3   Z1_Z2      1.29514 -1.14050  0.2597010 0.621774
```

```r
#posterior median
aggregate(.~ element, data = cor_b, FUN = function(x)
  round(median(x),2))
```

```
##   element X.Intercept.    X1    X2    X3
## 1   Z1_Z2         0.93 -0.92 -0.18  0.80
## 2   Z1_Z3        -0.71 -0.42 -0.33 -0.27
## 3   Z2_Z3        -0.63  0.37 -0.16 -0.59
```

```r
#posterior probability in direction of posterior median
aggregate(.~ element, data = cor_b, FUN = function(x)
  sum(sign(median(x))==sign(x))/length(x))
```

```
##   element X.Intercept.      X1      X2      X3
## 1   Z1_Z2      0.97100 0.99625 0.75950 0.99050
## 2   Z1_Z3      0.97425 0.89125 0.85975 0.73375
## 3   Z2_Z3      0.93650 0.77950 0.64550 0.82975
```

```r
#posterior 90% CIs
aggregate(.~ element, data = cor_b, FUN = function(x)
  round(quantile(x, c(0.05,0.95)),2), simplify = F)
```

```
##   element X.Intercept.          X1          X2          X3
## 1   Z1_Z2   0.14, 1.81 -1.57, -0.37 -0.63, 0.26  0.21, 1.68
## 2   Z1_Z3 -1.65, -0.09  -1.33, 0.11 -1.08, 0.23 -0.93, 0.62
## 3   Z2_Z3  -1.44, 0.06  -0.50, 1.08 -1.10, 0.63 -1.67, 0.43
```

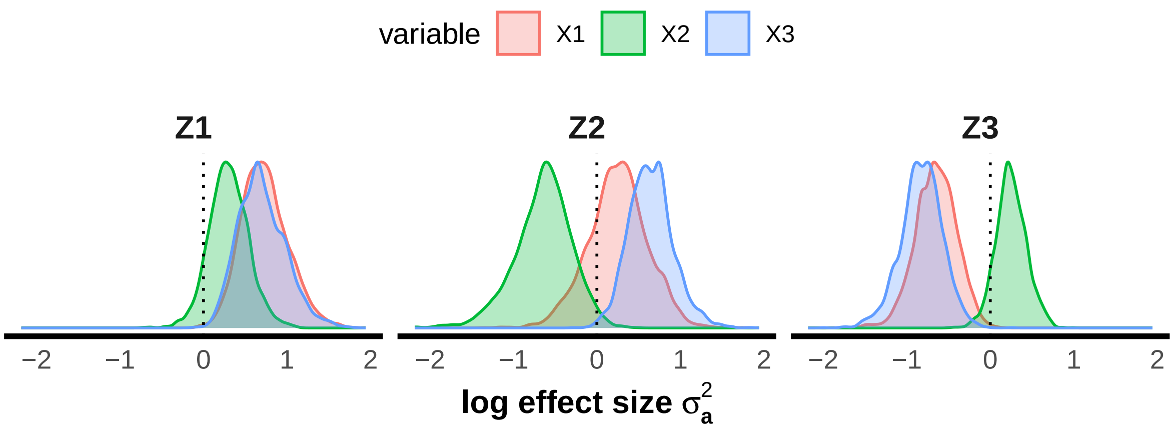
These effects are on the atanh scale. The first continuous covariate x_1 on average decreased the genetic correlation between z_1 and z_2 (β_{r_a} = -0.92, $p_- = 0.99625$) with weaker evidence for decreasing the correlation between z_1 and z_3 (β_{r_a} = -0.42, $p_- = 0.89125$) and increasing the correlation between z_2 and z_3 ($\beta_{r_a} = 0.37$, $p_+ = 0.7795$); the second continuous covariate x_2 showed modest to weak support for decreasing the genetic correlation among all behaviors ($z_1 \sim z_2$: β_{r_a} = -0.18, $p_- = 0.7595$; $z_1 \sim z_3$: β_{r_a} = -0.33, $p_- = 0.85975$; $z_2 \sim z_3$: β_{r_a} = -0.16, $p_- = 0.6455$); and the third covariate x_3 increased the genetic correlation among z_1 and z_2 ($\beta_{r_a} = 0.8$, $p_+ = 0.9905$), with weak evidence for decreasing the genetic correlation among z_1 and z_3 (β_{r_a} = -0.27, $p_- = 0.73375$) and z_2 and z_3 (β_{r_a} = -0.59, $p_- = 0.82975$). The uncertainty of these effects is unsurprising, given that our simulation considers moderately sized effects of 3 environmental factors on 3 traits across 25 contexts. Increasing the number of contexts observed will increase power to detect multiple effects across contexts, as with any multivariate regression analysis.

Plot results

There are multiple ways to visualize the results of the analysis. One way is to simply plot the full posterior estimates of the $\beta_{\sigma_a^2}$ and β_{r_a} values. This can be easily accomplished with the outputs generated above using the `geom_density()` function from the ggplot2 package.

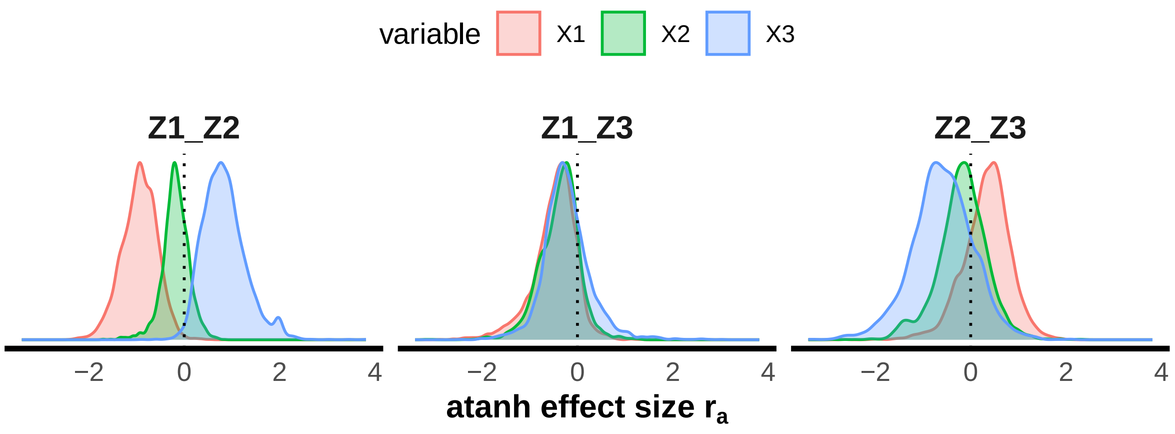
```r
#long format
v_bl = melt(v_b)
v_bl2 = v_bl[v_bl$variable!="X.Intercept.", ] #excl intercepts

#plot posteriors of variance RNs
ggplot(v_bl2, aes(x = value, color = variable, fill = variable))+
  geom_density(aes(y = after_stat(scaled)), alpha = 0.30)+
  geom_vline(xintercept = 0, lty = "dotted")+
  facet_wrap(.~element)+
  labs(x=bquote(bold(paste("log effect size ", sigma[a]^2))))+
  theme(
    legend.position = "top",
    axis.ticks.x=element_blank(),
    axis.ticks.y=element_blank(),
    axis.title.x=element_text(size=12,face="bold"),
    axis.title.y = element_blank(),
    axis.text.x=element_text(size=10),
    axis.text.y=element_blank(),
    axis.line.x = element_line(linewidth = 1),
    axis.line.y = element_blank(),
    strip.text = element_text(size = 12, face = "bold"),
    strip.background = element_blank(),
    panel.background= element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    plot.margin = unit(c(0.1,0.5,0.5,0.5), "cm"))+
    guides()
```



```
cor_bl = melt(cor_b)
cor_bl2 = cor_bl[cor_bl$variable!="X.Intercept.", ] #excl intercepts

#plot posteriors of variance RNs
ggplot(cor_bl2, aes(x = value, color = variable, fill = variable))+
  geom_density(aes(y = after_stat(scaled)), alpha = 0.30)+
  geom_vline(xintercept = 0, lty = "dotted")+
  facet_wrap(.~element)+
  labs(x=bquote(bold(paste("atanh effect size ", r[a]))))+
  theme(
    legend.position = "top",
    axis.ticks.x=element_blank(),
    axis.ticks.y=element_blank(),
    axis.title.x=element_text(size=12,face="bold"),
    axis.title.y = element_blank(),
    axis.text.x=element_text(size=10),
    axis.text.y=element_blank(),
    axis.line.x = element_line(linewidth = 1),
    axis.line.y = element_blank(),
    strip.text = element_text(size = 12, face = "bold"),
    strip.background = element_blank(),
    panel.background= element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    plot.margin = unit(c(0.1,0.5,0.5,0.5), "cm"))+
    guides()
```

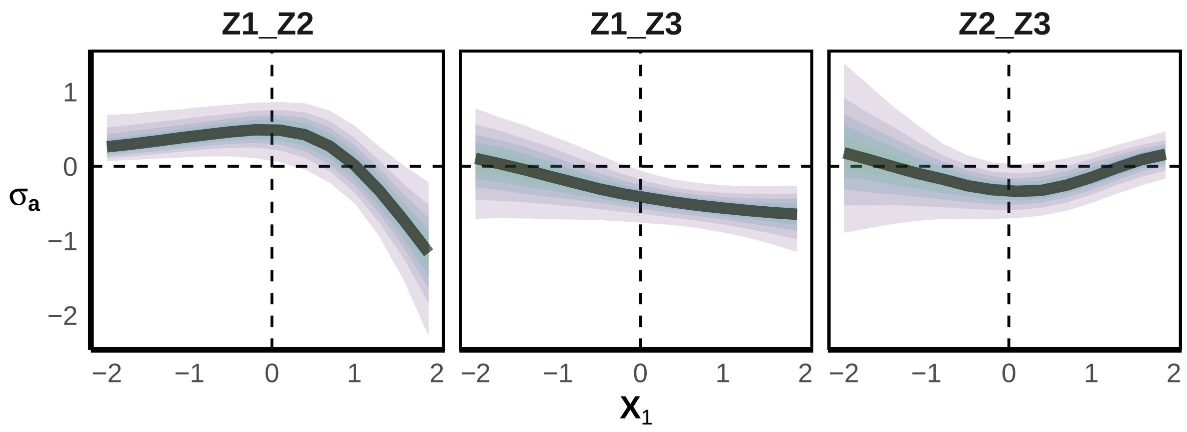


The posterior RNs for genetic variances and correlations can also be integrated together to predict and visualize how environmental contexts shape genetic (co)variances. In this case, we'll consider how genetic (co)variances are expected to change across the range [-2,+2] due to the marginal additive effect of environmental covariate x_1, holding x_2 and x_3 constant at their average values (0). The `stat_lineribbon()` function from the tidybayes package is used to visualize these posterior predictions as a CRN, with the dark line for the posterior median indicating the most probable curve and the accompanying ribbons of 10-90% credible intervals indicating statistical uncertainty around the median.

```r
#predictor matrix for generating predictions from model
#X_pred columns must follow order of original design matrix
#colnames(stan.df$X), but the colnames of X_pred are arbitrary
seq = seq(-2,2, by = 0.3) #standardized values from -2 to +2
X_pred = data.frame(int = 1, X1 = seq, X2 = 0, X3 = 0)

#functions for predicting genetic correlations and covariances
mv_cpc = cpc_f(x = X_pred, cpc_b = post$B_cpc)
mv_cor = cor_f(x = X_pred, traits = traits,  cpc = mv_cpc)
mv_cov = cov_f(x = X_pred, traits = traits, v_b = post$B_v, cpc = mv_cpc)

#visualize change in covariance
levels = 6 #number of bands for Bayesian CIs, try ppoints(levels)
ggplot(mv_cov,
      aes(x = X1, y = value))+
      geom_hline(yintercept = 0, lty = "dashed")+
      geom_vline(xintercept = 0, lty = "dashed")+
      stat_lineribbon(size=2, .width = ppoints(levels), alpha=0.8/levels)+
      labs(x=bquote(bold(paste(X[1]))),
          y=bquote(bold(paste(sigma[a]))))+
      facet_wrap(.~ element)+
      theme(plot.title =element_text(size=12, face="bold",hjust=0.5),
        axis.ticks.y=element_blank(),
        axis.ticks.x=element_blank(),
        axis.title.x=element_text(size=12,face="bold"),
        axis.title.y=element_text(size=12,face="bold", angle = 0, vjust = 0.5),
        axis.text.x=element_text(size=10),
        axis.text.y=element_text(size=10),
        axis.line = element_line(linewidth = 1),
        panel.border=element_rect(fill=NA,color="black", linewidth=1,
                                  linetype="solid"),
        strip.text = element_text(size = 12, face = "bold"),
        strip.background = element_blank(),
        panel.background= element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        plot.margin = unit(c(0.1,0.5,0.5,0.5), "cm"))+
        guides(fill="none", color ="none")
```



Preparing your own data for a CRN analysis

In the previous section, we used simulated data that was preformatted for immediate use in Stan. Now we'll consider how to format data from a real study. To do so, we'll use the same simulated data function `sim_CRN_QG()`, but set `standl = FALSE` (i.e. not formatted as a list for Stan) so that it returns the kind of data frame typical in empirical research. We'll otherwise use the same simulation settings as above.

```
#simulate data
stan.df = sim_CRN_QG(N = 500, Nc = 25, npred = 3, ntrait = 3, l_es = 0.3, u_es = 0.6, standl = FALSE)
head(stan.df)
```

```
##   id         z1          z2         z3           X1          X2         X3
## 1  1 -1.4549986 -0.46986697  0.4865679  0.522041302 -0.21825196  0.3665135
## 2  2 -0.6421054  1.26457552  0.7268089  0.471092869 -0.02189907  0.2395832
## 3  3  0.1361168 -0.83326092 -3.2384834  0.378782473  1.33445177  0.3758605
## 4  4  0.2007080  1.76456192 -0.9520012 -0.445808060  0.27312408 -1.0837493
## 5  5  3.8283733  1.18049329  0.6990489  0.002784033  1.14014892  0.7580011
## 6  6 -0.8805482  0.03529682  1.2029749  0.373376576 -1.15949104 -0.3171307
```

As is usually the case, we have a data frame where subject IDs are listed (`id`), followed by three trait measures (`z1 - z3`), and three environmental measures (`X1 - X3`). For Stan, as explained above, we'll need a list that separates these elements, quantifies their sizes, and also provides accompanying objects to appropriate index them in the likelihood function of the CRN model. Every CRN analysis will require the following elements, as can be seen from the data block of the `CRN_tutorial_mod.stan` file.

- N (total number of observations)
- C (total number of environmental contexts)
- I (total number of subjects)
- D (total number of traits/dimensions)
- P (total number of environmental predictors + intercept)
- X (a C x P matrix of environmental predictors)
- A (an I x I relatedness matrix, or identity matrix for phenotypic analysis)
- z (an N x D object of multivariate normal response variables
- cm (the max number of individuals observed in a context)
- cmat (a C x cm object with all individuals observed in each context [row])
- cn (a C length object counting the number of individuals per context)
- cnt (the total number of individuals across contexts)
- id (an N sized object linking observations to individuals)
- c_id (an N sized object linking observations to contexts)
- idc (an N sized object linking individuals to positions in cmat)

Many of these can already be gleaned from `stan.df`, so it's clear that the bulk of additional information needed concerns the environmental contexts, which are not yet structured in our data. We'll start by clarifying how many unique contexts there are (i.e. the total number of unique combinations of environmental values).

```
contexts = unique(stan.df[,c("X1","X2","X3")])
nrow(contexts)
```

```
## [1] 25
```

As should be the case, there are more subjects than unique environmental contexts (I > C), meaning that the values of `X1 - X3` repeat across rows for subjects that experienced the same multivariate environment. We can use `contexts` to make the environmental predictor matrix (`X`).

```r
X = data.frame(model.matrix(1:nrow(contexts) ~ X1 + X2 + X3, data = contexts))
rownames(X) = 1:nrow(X)
```

Now we need a way of telling the model which contexts correspond to the rows of our dataframe.

```r
c_id = list()
  for(c in 1:nrow(contexts)){
    c_id[[c]] = as.integer(rownames(match_df(stan.df[,c("X1","X2","X3")], contexts[c,])))
  }
c_id  = data.frame(context=rep(seq_along(c_id),lengths(c_id)), row=unlist(c_id))
stan.df$c_id = c_id[order(c_id$row),"context"]
```

In the first stage, a list is made where rows are matched between each unique context and the original dataframe. These values are then organized in long format and assigned back to `stan.df` as a unique identifier (`c_id`). This allows us to easily pull out the appropriate row from `X` for the linear predictors in the Stan model.

Now we also want to make a matrix providing much the same information. This matrix organizes all the IDs together for subjects who share the same environmental context. This is crucial for efficiently structuring subjects' additive genetic values (or repeatable phenotypic trait values) in the Stan model.

```r
c_list = list()
  for(c in 1:nrow(X)){
    c_list[[c]] = unique(stan.df[stan.df$c_id==c,"id"])
  }
  cmat = matrix(t(sapply(c_list, function(x) x[1:max(lengths(c_list))])),
              ncol = max(lengths(c_list)))
  cmat = t(apply(cmat, 1, function(x) `length<-`(na.omit(x), length(x))))
  cmat_n = apply(cmat, 1, FUN = function(x) sum(!is.na(x)) ) #N/context
  cmat[is.na(cmat)] = 0 #remove NAs
```

The first stage makes a list where each element contains the IDs for subjects who experienced that context (1 to C). This list is then transformed into the matrix we'll use in Stan (`cmat`), the number of subjects per context is counted (`cmat_n`), and then `NA` are replaced with 0s as Stan does not accept `NA` values. This final step is unnecessary for the simulated dataset where sampling is balanced across contexts, but it is essential for real data where the number of subjects per context will likely vary.

The final step now is to index the position of each subject will respect to `cmat`, which is used to increase the efficiency of sampling genetic values in Stan (see `CRN_tutorial_mod.stan` for details).

```r
  #new index corresponding to subject position in cmat
  temp = t(cmat)
  corder = data.frame(id = temp[temp>0], c = rep(seq(1:nrow(cmat)), times = cmat_n))
  stan.df$idc = match(paste(stan.df$id, stan.df$c_id, sep="."), paste(corder$id, corder$c, sep="."))
```

With `idc`, we can now make our data list for Stan. We put `D=3` because we know there are three traits being analyzed. Note that the relatedness matrix (`A`) is assumed to already be saved as an additional object in the R environment. The order is arbitrary so long as the list contains all the necessary information for the data block of the corresponding Stan model.

```r
stan.dl = list(N = nrow(stan.df), C = nrow(X), I = max(stan.df$id), D = 3, P = ncol(X),
               X = X, A = A, z = stan.df[,c("z1","z2","z3")],
               cmat = cmat, cm = ncol(cmat),cn = cmat_n, cnt = sum(cmat_n),
               id = stan.df$id, c_id = stan.df$c_id, idc = stan.df$idc)
```

Now the same steps taken above for the simulated data can be carried out. Of course, one will want to ensure that the `CRN_tutorial_mod.stan` is appropriate for their data structure. Otherwise, the model will need to be revised accordingly. Some key extensions for a CRN analysis are considered in the next section.

```r
library(shinystan)
library(cmdstanr)

#directory for cmdstan installation
set_cmdstan_path("...")

#compile model
CRN_mod = cmdstan_model(stan_file = "CRN_tutorial_mod.stan",
                    stanc_options = list("O1"))

#estimate model
est = CRN_mod$sample(
    data = stan.dl,
    iter_sampling = 500,
    iter_warmup = 500,
    init = 0.01,
    chains = 4,
    parallel_chains = 4,
    adapt_delta = 0.80,
    max_treedepth = 10,
    refresh = 10)

saveRDS(est, "CRN_fit.RDS")
launch_shinystan(est)
post = extract(est)
```

Pertinent extensions

As emphasized in the main text, the CRN is simply a method for extending a multilevel model designed for quantitative genetic analysis (i.e. an ‘animal model’) to predict trait variances and correlations. Thus, any model of any complexity can be conceptualized as a CRN model, and researchers should not think of the tutorial model presented above as the only available likelihood structure for their data. This is precisely why I have coded the CRN in Stan, because this probabilistic programming language facilitates refreshingly open-ended and flexible statistical modeling that is rarely available for ecological and evolutionary research. The space of possible extensions for the CRN is, therefore, rather vast, as with any multilevel model more generally, and it would not be possible nor an economical use of time to elucidate them all here. Instead, I will cover the basics of some key extensions, giving empiricists a solid starting ground to further calibrate the basic CRN model presented here to their specific needs. The accompanying R and Stan files for the meerkat analysis provide a more advanced example showcasing how Stan can handle data structures with multiple complexities, such as traits with heterogeneous sampling (distinct N for each trait), which is not covered here.

Non-Gaussian distributions

Empirical data often comes in the form of counts, durations, proportions, reaction times, and other measures that tend to produce non-Gaussian errors. As an extension of a generalized multilevel model, the CRN can of course handle this issue with ease. The various distributions available in Stan for modeling traits can be found easily online and through use of the Stan Functions Reference. Compared to the multivariate normal distribution used in the examples, two changes will typically need to be made to the model. Firstly, the input will tend to require a different structure. For the multivariate Gaussian, we fed in all traits at once as a vector `z` with `D` values for `N` individuals.

```
data {
  //...same as above
  vector[D] z[N]; //multivariate normal response variables
}
```

Imagine that instead of three Gaussian traits, we’ve now got two trait measures better described by Poisson (`p_trait`), and binomial (`b_trait`) distributions. We’ll declare these data as arrays of integers (counts).

```
data {
  //...
  array[N] int p_t; //poisson
  array[N] int b_t; //binomial, 1s
  array[N] int b_trials; //binomial, total 'trials'
}
```

Now we just need to modify the parameters to remove the multivariate Gaussian residuals

```
parameters {
  //...
  //commenting out to remove from the model
  //cholesky_factor_corr[D] L_E;
  //vector<lower=0>[D] sd_E;
}
```

as well as their corresponding priors in the model block.

```
model {
  //...
  //sd_E ~ exponential(2);
  //L_E ~ lkj_corr_cholesky(2);
}
```

We could instead add a matrix of standardized observation-level random effects `matrix[N, D] Z_E` and, following the same approach taken for the genetic effects above, use these to model additive overdispersion for the Poisson and binomial responses. Overdispersion is common in real-world datasets, so this will generally be a good choice. I leave the coding involved as an exercise for the reader.

We'll also need to change the likelihood statements in the model block to separate out the predictors of the Poisson and binomial responses and specify the desired distributions. Of course, the call for the multivariate Gaussian likelihood is removed (not shown). The rest of the model stays the same, which is the beauty of generalized linear modeling.

```
model {
  //...
  for(n in 1:N){
  //separate out predictions for each trait by matrix column
  //the order is arbitrary (e.g. column 2 could be the Poisson trait)
  //but it needs to stay consistent across model statements
  real lin_pred_p = col(mu,1)[c_id[n]] + col(mat_G,1)[idc[n]];
  real lin_pred_b = col(mu,2)[c_id[n]] + col(mat_G,2)[idc[n]];
  p_t[n] ~ poisson(exp(lin_pred_p));
  b_t[n] ~ binomial(b_trials, inv_logit(linear_pred_b))
  }
  //....
}
```

There are a few different ways of calling non-Gaussian distributions in Stan, some of which do not require explicitly specifying link functions, but I will leave the task of learning about these to the interested reader. I personally prefer to make the link functions explicit in the model code. Importantly, because our predictions are made on the transformed scale, we use the inverse link functions `exp()` for the log link and `inv_logit()` (i.e. the logistic function) for the logit link when specifying distributional parameters. This is the same approach taken for specifying the linear predictors of the trait variances and partial correlations as shown above. This tells Stan the linear predictors are on the transformed scale and back transforms these predictions to the appropriate scale for the distributions. The Poisson needs a $\lambda = exp(...)$ rate parameter and the binomial needs the trials n (`b_trials`) and a $\tau = logit^{-1}(...)$ probability parameter. Keep in mind that effect sizes for the CRN parameters now need to be understood both with respect to the link functions on the G matrix components as well as the link functions on the trait means (where the additive genetic values come into the linear predictors).

Additional random effects

The additive genetic effects for each subject are realizations of multivariate normal random variables and are thus "random effects", while the CRN parameters predicting the (co)variances of these values are "fixed effects". However, it may just as well be pertinent to include random effects as components of the CRN. For example, the environmental context of interest may be non-randomly distributed across space or time, resulting in stochastic fluctuations in covariances due to unmeasured environmental variation. Let's use the concrete example of the same population measured under different climate regimes across multiple years to work through the code. The climate predictors are in the **X** matrix for the CRN fixed effects, so our goal will be to add random effects to the linear predictors accounting for the year of observation.

The supplementary material and tutorial above have already explained how random effects can be effectively programmed in Stan. The most efficient parameterization involves separating out the scale of random effects (i.e. their standard deviations) from the standardized distances among subjects' values (i.e. their z-scores). In other words, for a univariate vector $\mathbf{u}$ of random effects with standard deviation σ

$$\mathbf{u} = \mathbf{z}\sigma$$

$$\mathbf{z} \sim N(0,1)$$

Without needing to understand why in a deeper sense, it suffices to say that this so-called "non-centered" parameterization generally enhances the efficiency of MCMC sampling. For a multivariate matrix $\mathbf{X}$ of independent random effects, using the upper triangle Cholesky decomposition of the covariance matrix $\mathbf{\Sigma}$

$$\mathbf{U} = \mathbf{Z}\mathbf{L}_{\mathbf{\Sigma}}{}^{\mathrm{T}}$$

$$\mathbf{Z} \sim MVN(0, \mathbf{I})$$

See Eq. S5 in the supplement for the full multivariate generalization of this relationship. When we assume random effects are independent across measures, as we will do below, this simplifies to

$$\mathbf{U} = \mathbf{Z}\mathrm{diag}(\sigma_1, ..., \sigma_D)$$

where the $diag()$ function creates a matrix with SDs on the diagonals and 0s elsewhere.

For this example, the random effects we'll be adding concern the year in which sampling took place. We're assuming for now that year effects on the trait means and $\mathbf{G}$ matrix are independent once we've adjusted for the measured environmental covariates. We could attempt to estimate correlations among these random effects on the trait (co)variances, but this would be a very data hungry model that may sampling poorly and not necessarily gain much in terms of biological insight. Our goal is to simply quantify among year variability, so we focus on this here. We could, of course, reduce the number of parameter further by, for example, removing some of these random effects with minimal variation in a stepwise procedure. A more elegant solution would be to assume a common latent year variable with covarying effects across traits, reducing free parameters while still retaining quantifying correlations among taits. See Merkle and colleagues' detailed treatment of latent variable modeling in Stan for further insight on how one can code this.

```
parameters {
  //...

  //z-scores (standardized random effects)
  matrix[C,D] Z_year_m;
  vector[C,D] Z_year_v;
  vector[C,ncor] Z_year_cpc;

  //scales for random effects (sigmas / standard deviations)
  vector<lower=0>[D] sd_year_m; //random annual variation in means
  vector<lower=0>[D] sd_year_v; //...in genetic variances
  vector<lower=0>[ncor] sd_year_cpc; //...in genetic correlations (partial corrs)

  //no corr matrices because we assume effect are independent during sampling
  //note that posterior correlations can still be manually estimated among effects
}
```

Now we scale the z-score matrices for each component by their corresponding scales (SDs), as just described for independent multivariate normal random effects. Note that this code for transforming the random effects could also be placed in a `transformed parameters` block before the `model` block, which we have not used thus far in the tutorial (see the meerkat code for an example). This would save the community-specific scaled random effects with the model output, but would as a consequence increase the memory required for storing and saving the model.

```
model {
  //scaled year random effects
  matrix[C, D] U_m = diag_post_multiply(Z_year_m, sd_year_m);
  matrix[C, D] U_v = diag_post_multiply(Z_year_v, sd_year_v);
  matrix[C, ncor] U_cpc = diag_post_multiply(Z_year_cpc, sd_year_cpc);
  //....
```

We can now bring these random effects into the linear predictors of the different trait components, simply by extending the code we already have for fixed effects predictors. We'll also add priors for the new random effect parameters.

```
//....
//predicted values from reaction norms
  //means
  matrix[C, D] mu = Q * B_mq + U_m;

  //variances
  matrix[C, D] sd_G = sqrt(exp(Q * B_vq + U_v));

  //correlations (expressed as canonical partial correlations)
  matrix[C, ncor] cpc_G = tanh(Q * B_cpcq + U_cpc);


//....

  to_vector(Z_year_m) ~ std_normal();
  to_vector(Z_year_v) ~ std_normal();
  to_vector(Z_year_cpc) ~ std_normal();
  sd_year_m ~ exponential(2);
  sd_year_v ~ exponential(2);
  sd_year_cpc ~ exponential(2);
}
```

Gaussian processes for autocorrelation and nonlinearity

The previous subsection discussed typical random effects, where we assume that values for each context are independently distributed. In other words, the expected deviation for a given year is not expected to be correlated with the value of the previous year. Of course in reality, temporal and spatial variation, whether stochastic or directional, tends to exhibit autocorrelation, with adjacent patches or annual periods tending to have more similar values. The overall pattern of variation as well as the slope by which this similarity decays across space or time will determine the complexity of the model required to capture it. When this functional relationship is known a priori, such as for the decay of genetic similarity across kin classes expected under Mendelian inheritance, we can simply calculate a matrix of weights to scale the expected similarity of random effects, as we do above when multiplying additive genetic values by the **A** matrix. However, we are often not in a position to know these relationships beforehand. We might expect that there is temporal or spatial autocorrelation, and we might expect that there are complex nonlinear directional trends, but these remain open empirical questions. Gaussian Processes (GP) are one extremely useful, albeit technically advanced, tool for accomplishing this task, allowing for complex nonlinearities and patterns of autocorrelation to emerge in our data. The goal of this tutorial is not to provide a serious introduction to GPs, which is a task left to the interested reader. Instead, I will here briefly show how GPs can be introduced to a CRN analysis.

We will focus on implementing a so-called exponentiated quadratic function GP in Stan, which predicts similarity among values i and j of predictor $\mathbf{x}$, independently of residual noise, by

$$\alpha^2 \exp\left(-\frac{1}{2\rho^2}\sum(x_i - x_j)^2\right)$$

where ρ is a length scale reflecting the frequency of the nonlinear function and α is the marginal standard deviation of the effects. Fundamentally, the function is scaling how the similarity of values changes in virtue of their distance, i.e. $x_i - x_j$. This means that, when applicable, we can also replace this difference with a distance matrix (see Nicholas Clark's helpful tutorial for further details).

Let's consider the nonlinear effect of a single predictor across environmental contexts like the year of observation. Note that the same approach shown here can be applied to multivariate predictors. See the corresponding Stan User's Guide entry for GPs for further details. We might expect nonlinearity in the effect of year on trait covariances, in addition to standard linear directional trends, due to unmeasured ecological dynamics clustered among years throughout the study period. We might also expect that the effect of year is partially shared and thus covarying across traits, due to its capturing a variety of unmeasuring environmental factors with similar effects on the measured phenotypes. In the previous subsection, we ignored these correlations across traits to reduce model complexity for pragmatic purposes. Here we will code a full model to show what is possible when a large dataset is available. The GP will help us to quantify these covarying and nonlinear deviations that would otherwise be missed by standard fixed linear regression coefficients.

To include a GP in the CRN analysis, we're going to use an in-build Stan function called `gp_exp_quad_cov()` to calculate a covariance matrix `K` capturing the expected similarity among years (`year`) as a function of the estimated GP parameters (`rho`, `alpha`), whose elements will be predicted by the exponentiated quadratic function above. We'll then take the Cholesky of this matrix (`L_K`) for statistical efficiency, as has been discussed repeatedly throughout the tutorial and in the supplementary material. As shown above for genetic random effects, this matrix will then be premultiplied by a matrix of z-score random effects to produce the appropriately scaled temporally autocorrelated random effects `U_nl`. This matrix will also be premultiplied by a Cholesky factorized correlation matrix (`L_Omega`) capturing correlations among these year effects across traits. For purely pragmatic purposes, we add a very small `delta` value to the diagonals of `K`, which is simply a programmatic trick to aid model identification. Prior choice for the parameters of the GP is very important and sensitive to the particular scale of the data being analyzed. Please see the Stan entry linked above for further details.

```
data {
//...
array[C] int year; //year can still be included in X to partition linear and nl effects
}

transformed data{
//...
real tot_D = D * 2 + ncor; //total dimensions for GP (mean + var + pcor)
real delta = 1e-9;
}

parameters {
//...
  real<lower=0> rho; //frequency of function (length scale)
  real<lower=0>[tot_D] alpha; //marginal SDs of trait-specific GP deviations
  cholesky_factor_corr[tot_D] L_Omega; //among-trait GP correlations
  vector[C, tot_D] Z_year; //GP deviations per environmental context
}

model {
```

```
//...
  matrix[C, tot_D] U_nl;
  matrix[C, C] K = gp_exp_quad_cov(x, 1.0, rho);
  matrix[C, C] L_K;

  // diagonal elements
  for (c in 1:C) {
    K[c, c] = K[c, c] + delta;
    }

  L_K = cholesky_decompose(K);
  U_nl = L_K * Z_year * diag_pre_multiply(alpha, L_Omega)';

  //predicted values from reaction norms
  //means
  matrix[C, D] mu = Q * B_mq + U_m + U_nl[,1:D];

  //variances
  matrix[C, D] sd_G = sqrt(exp(Q * B_vq + U_v) + U_nl[,(D+1):(2 * D)]);

  //correlations (expressed as canonical partial correlations)
  matrix[C, ncor] cpc_G = tanh(Q * B_cpcq + U_cpc + U_nl[,(2 * D):(tot_D)]);

//....
  to_vector(Z_year) ~ std_normal();
  alpha ~ std_normal();
  L_Omega ~ lkj_corr_cholesky(3);
  rho ~ inv_gamma(5, 5);
}
```

Conclusion

This concludes the tutorial. I hope that you have found it helpful. Please contact me by email (jordanscott.martin@eawag.ch or jordanscottmartin@gmail.com) if you have any questions or would like some assistance in further adapting the provided code to your dataset.