Estimating (non)linear selection on reaction norms
A general framework for labile traits

Introductory tutorial for basic coding and implementation

Jordan S. Martin

Contents

Introduction

This series of tutorials demonstrates how to effectively code and interpret models of nonlinear selection on individual reaction norms (RNs), using the Stan statistical programming language (Carpenter et al. 2017) in R (R Core Team 2020). Stan is an open-source programming language for estimating complex probabilistic models using fully Bayesian inference with state-of-the-art Markov Chain Monte Carlo (MCMC) sampling techniques (Hoffman and Gelman 2014). Stan interfaces with R through the **RStan** and **CmdStan** packages, but you will first need to install Stan on your computer and ensure that it is appropriately configured with your C++ toolchain. Once you are able to effectively use Stan in R, you can begin creating the `.stan` files necessary for estimating the models introduced here. These files can be composed using RStudio or any text editor. A file can be also be composed directly in R with `write()`.

```
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 {...}
    model {...}
    generated quantities {...} ",
  "mod1.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. This is very useful for saving memory in a complex analysis.

Once an appropriate `.stan` file is prepared, it can be compiled in R for the C++ toolchain using the `stan_model()` function in RStan or `cmdstan_model()` function in CmdStan and subsequently estimated with an appropriate list of empirical data. The resulting posteriors of a model can then be accessed and manipulated for any further quantities or analyses of interest.

```
#load package
library(rstan) #or
library(cmdstanr)

#make data list
data = list(x = x, y = y, ...)

#compiles the model in C++ for MCMC estimation
mod1 = stan_model("mod1.stan") #or
mod1 = cmdstan_model(stan_file = "mod1.stan", stanc_options = list("O1"))

#samples posterior distribution of the model with default MCMC settings
results = sampling(object = mod1, data = data) #or
results = mod1$sample(data = data)

#extracts posterior estimates
samples =  extract(results) #or
samples = results$draws(format = "data.frame")
```

Generate data

For ease of introduction, this tutorial considers a simple model with a Gaussian phenotype and fitness measure. As is explained below, the code can be straightforwardly modified for non-Gaussian traits or fitness components.

Formal model

It's always helpful to write out the formal model we'd like to estimate in Stan before attempting to code it. There are a few reasons for this. Firstly, Stan is a probabilistic programming language and, as such, facilitates coding and estimation of probabilistic models through direct specification of the model parameters and likelihood functions. Therefore, some understanding of the formal structure of any model is necessary to flexibly code in Stan. Gaining a deeper understanding of formal statistical models can also be extremely valuable for building scientists' autonomy and ingenuity in data analysis, which opens up the door to developing novel models capturing the most salient features of one's specific empirical system and dataset, rather than being dependent on the assumptions and simplifications of prepackaged toolkits. Researchers unfamiliar with formal statistical models are encouraged to see McElreath (2020) for a detailed, accessible treatment in the context of Bayesian data analysis.

Notation here follows the main text (see **Eq. 1-2**) for the specific case of normally distributed measurements. The probability density function for measurement z_{jt} of individual j's phenotype z at time t is given by

$$
\begin{gathered}
z_{jt} \sim \text{Normal}\left(\mu_{jt}, \sigma_j\right) \\
\mu_{jt} = \mu_0 + \mu_{0j} + \left(\beta_x + \beta_{xj}\right) x_{jt} \\
\log\left(\sigma_j^2\right) = \sigma_0 + \sigma_{0j} \\
\left[\boldsymbol{\mu_0}^\intercal, \quad \boldsymbol{\beta_x}^\intercal, \quad \boldsymbol{\sigma_0}^\intercal\right]^\intercal \sim \text{MVNormal}\left(\mathbf{0}, \mathbf{P}\right)
\end{gathered}
$$

where $^\intercal$ indicates the transpose operator. It is important to note here that the (co)variance matrix $\boldsymbol{P}$ can be equivalently expressed as the product of a diagonal matrix of standard deviations $\boldsymbol{S}$ and a correlation matrix $\boldsymbol{R}$, such that $\boldsymbol{P} = \boldsymbol{SRS}$. We will exploit this identity below to speed up computation of the model in Stan. We are also interested in quantifying selection on the individual-level random intercept $\boldsymbol{\mu_0}$, slope $\boldsymbol{\beta_x}$, and residual $\boldsymbol{\sigma_0}$ parameters. To do so, the full multivariate/multi-response model simultaneously estimates a probability function for measurement W_{jt} of individual j's fitness component W at time t

$$
\begin{gathered}
W_{jt} \sim \text{Normal}\left(\theta_{jt}, \delta\right) \\
\theta_{jt} = W_0 + W_{0j} + b_1\mu_{0j} + b_2\beta_{xj} + b_3\sigma_{0j} \\
+q_1\mu_{0j}^2 + q_2\beta_{xj}^2 + q_3\sigma_{0j}^2 + q_4\mu_{0j}\beta_{xj} + q_5\mu_{0j}\sigma_{0j} + q_6\beta_{xj}\sigma_{0j} \\
\boldsymbol{W_0} \sim \text{Normal}\left(0, \text{sd}(\boldsymbol{W_0})\right)
\end{gathered}
$$

Here $\mathbf{b}$ are regression coefficients quantifying linear effects of the RN parameters on fitness and $\mathbf{q}$ are coefficients quantifying quadratic effects. Importantly, these coefficients are proportional to but *not* equivalent to $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ gradients used in phenotypic selection and quantitative genetic theory. However, we can easily calculate these values by further manipulating the posterior distributions of $\mathbf{b}$ and $\mathbf{q}$ after estimating the model. This is considered further below. When multiple fitness measures are available for the same subjects, the model should include individual-level random intercepts W_{0j} to quantify any patterns of repeatable among-individual differences in fitness that are not accounted for by the modeled phenotype(s) (i.e. unexplained selection). However, in the absence of repeated fitness measures, variation explained by $\text{sd}(\boldsymbol{W_0})$ cannot be partitioned from variation due to the fitness residual δ and should, therefore, be excluded from the model.

Simulate dataset

Here we simulate a dataset appropriate for estimating this formal model. To do so, the `NLS_RN_functions.R` file from the accompanying Github page should be saved locally, providing access to the `sim_RN_Gaus()` function. By default for this function, population parameter values are fixed so that $\mu_0 = \beta_x = 0$, $\sigma_0 = \log(2)$, $W_0 = \delta = 1$, and $\text{var}(\boldsymbol{\mu_0}) = \text{var}(\boldsymbol{\beta_x}) = \text{var}(\boldsymbol{\sigma_0}) = \text{var}(\boldsymbol{W_0}) = 1$, with correlations among RN parameters drawn from an $LKJ(10)$ to produce low to moderate correlations on average. These values can always be changed by adjusting the corresponding arguments (simply type `sim_RN_Gaus` in the console to reveal the full function). Using default settings, the user will need to input the range of effect sizes for the selection effects, which will be drawn from a uniform distribution ranging from `l_es` to `u_es`. For simplicity, these need to be positive values, but the signs of selection effects will be randomly flipped inside the function. The user also needs to provide the desired sample size `J`, number of repeated phenotypic measures `rep_z`, and repeated fitness measures `rep_W`. For the tutorial, we'll generate data for a sample of 1000 individuals with 3 repeated phenotype measures and 2 repeated fitness measures, with selection effects (regression coefficients) of small-to-moderate statistical effect size.

```
#custom sim function (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("NLS_RN_functions.R")

#simulate dataset
stan.dl = sim_RN_Gaus(J = 1000, rep_z = 3, rep_W = 2, l_es = 0.2, u_es = 0.3)
```

Looking over the structure of this list is useful for understanding how Stan expects data to be formatted.

```
str(stan.dl)
```

```
## List of 11
##  $ J      : num 1000
##  $ N_z    : num 3000
##  $ N_W    : num 2000
##  $ ind_z  : int [1:3000] 1 1 1 2 2 2 3 3 3 4 ...
##  $ ind_W  : int [1:2000] 1 1 2 2 3 3 4 4 5 5 ...
##  $ x      : num [1:3000] 0.332 1.633 -0.801 1.207 -0.357 ...
##  $ z      : num [1:3000] -0.114 -3.786 -0.195 -1.555 3.838 ...
##  $ W      : num [1:2000] 0.448 -0.214 1.593 3.049 0.306 ...
##  $ true_b : num [1:3] 0.251 -0.298 -0.25
##  $ true_q : num [1:3] -0.225 -0.277 -0.259
##  $ true_qc: num [1:3] 0.217 0.241 -0.208
```

The data list not only includes the raw data variables `x` (measured environment), `z` (phenotype), and `W` (fitness), but also integers `N_z` and `N_W` describing the size of these variables and indices `ind_z` and `ind_W` for linking repeated observations of phenotype and fitness to individual observations. The integers `N_z` and `N_W` are used in the Stan model to declare the expected dimensions of `z` and `W`, which ensures that inappropriate data structures or likelihood functions will throw errors. This will become clearer in the subsequent model coding section. It is essential that every quantity declared in the `data{}` block of the corresponding Stan model is also present in this supplied data list. However, the list can also include values of interest that are not included in the Stan model, such as the true directional `true_b`, stabilizing/disruptive `true_q`, and correlational selection `true_qc` regression coefficients used for simulating the data. The variables in the data list that do also appear in `data{}` need to have the same name in both locations.

There are a few additional considerations when preparing your own data for using in Stan. 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). Missing data imputation can also be accomplished during model estimation (see the **Stan Reference Manual** for further details). Character strings can also not be supplied to Stan, including for indexing subject IDs (e.g. "subject1", "A015", etc.). Instead, these values will need to be converted to a numeric index. Particular attention needs to be given to the order of data input to the model, as these numeric values will need to be appropriately aligned and indexed throughout the model specification. For instance, if one is changing the character names in an R object `df$subj` for indexing in Stan

```
key.id = unique(df$subj) #all unique subject IDs
new.id = seq(1:length(key.id)) #create numeric index of equal length
df$id = new.id[match(df$subj, key.id)] #numeric id matching order in data frame
#or more simply: df$id = as.integer(factor(df$subj))
```

Users will also need to manually ensure that the integers used to index subjects are appropriately aligned with the order of any other data structures corresponding to those subjects. For instance, if one wants to extend the RN model for quantitative genetic analysis using an `A` relatedness matrix, the matrix 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)))])
```

These additional steps may seem cumbersome at first, but they also allow for many benefits unavailable in more standard statistical software. For example, when each variable can be declared manually with its own separate dimensions and indices, multivariate models can be straightforwardly estimated using data with heterogeneous dimensions (e.g. with differing numbers of subjects and repeated measures for each trait).

Code model in Stan

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, as noted above, 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 affords the degree of flexibility necessary for estimating nonlinear selection on RNs. It is important to recognize that some practice and trial-and-error will also be required to gain competency and comfortability with Stan. Therefore, I encourage researchers to review the **Stan Reference Manual**, as well the extensive collection of **Stan Case Studies**, which will provide a more robust foundation for estimating any model of interest in Stan.

As mentioned above, a basic Stan model consists of multiple programming blocks that together specify the data, parameters, likelihood, and quantities of interest for a model. Rather than tackling the model in a single step, we can consider the blocks in turn before putting them together in a single file.

Data

The first component of a Stan model is the data block, which as discussed above tells the model what to expect from our data list, as well as how to treat that data inside the model. Note that Stan uses `//` rather than `#` for comments.

```
data {
  int<lower=1> J; //total individuals
  int<lower=1> N_z; //total number of pheontype meausres (Z)
  int<lower=1> N_W; //total number of fitness measures (W)
  array[N_z] int<lower=1> ind_z; //index of individual measurements (z)
  array[N_w] int<lower=1> ind_w; //index of individual measurements (W)
  vector[N_z] x; //environmental covariate
  vector[N_z] z; //phenotype
  vector[N_W] W; //fitness
}
```

The variables in `stan.dl` are declared by their type (note that `int` = integer) with additional arguments regarding their dimensions. `<lower=1>` tells the model to expect sample sizes of at least 1 for the total number of individuals observed `J` and the total number of phenotype `N_z` and fitness measures `N_w`. Our indices `ind_z` and `ind_W`, which link repeated individual observations to the vectors of phenotype `z` and fitness `W` measures, are declared as integers in an array, i.e. multiple integers of length `N_z` and `N_w` that should never be of value less than 1.

Parameters and transformed parameters

The parameters block will take all of the basic parameters that are specified in the nonlinear selection model. Having the formal model above makes this part much easier to code. We can begin by considering the fixed population-level parameters in the model, although the order of specification in the parameters block is entirely arbitrary.

```
parameters {
  //fixed population effects
  real mu_0; //z population intercept
  real beta_x; //z population slope
  real sigma_0; //z population dispersion

  real W_0; //W population intercept
  vector[3] b; //direct selection
  vector[3] q; //stabilizing/disruptive selection
  vector[3] qc; //correlational selection
//...
```

`mu_0` is the population intercept μ_0 for the expectation of behavior `z`, `beta_x` is the population slope β_x, and `sigma_0` is the population intercept σ_0 of the dispersion parameter. For the fitness model, we specify `W_0` for the global intercept W_0, as well as vectors for the fixed effects describing directional selection `b` and quadratic selection due to stabilizing/disruptive selection `q` and correlational selection `qc`. In this case, because there are three RN parameters under selection, there are also 3 corresponding parameters for both types of quadratic selection. However, in the more general case (e.g. with multiple phenotypes), the vector `qc` will be of length $\frac{p(p-1)}{2}$ for each unique bivariate combination of p total parameters. Note that we could also write the stabilizing/disruptive and correlational effects together in a single vector e.g. `vector[3 + 3] q`, or even all selection effects in a single vector e.g. `vector[3+3+3] bq`. This choice is entirely arbitrary and based on what the user finds most intuitive for working with the model.

To increase computational efficiency while estimating the random effects, we are going to use a form that is mathematically equivalent to the formal model above but programmatically distinct. This form combines two computational tricks–a so-called *non-centered parameterization* of the random effects and a Cholesky factorization of the random effect correlations–that will help the model to fit much faster. First we'll look at each step in the code and then consider the maths behind it.

```
//...
  //random effects for z
  vector<lower=0>[3] sd_RN; //RN parameter sds
  matrix[J,3] std_dev_RN; //individual-level RN deviations
  cholesky_factor_corr[3] R_chol; //RN parameter correlations

  //random effects for W
  real<lower=0> sd_W0; //unexplained selection sd
  vector[J] std_dev_W; //individual-level selection deviations
  real<lower=0> delta; //W dispersion (SD of residuals)
}
```

We specify a vector `sd_RN` of length 3 for each of the SDs of the RN parameters and a real number `sd_W0` for the SD of the individual-level random fitness intercept. The argument `<lower=0>` is essential for letting the model know that these SDs cannot take on negative values. Given that a variance or SD is necessarily positive by definition, negative values are mathematically improper solutions and will lead to poor model convergence as a result. Note that unlike the fitness model, where `delta` is simply the SD of Gaussian fitness residuals, the dispersion of the phenotype z is specified by a linear predictor on the log scale $\log(\sigma_j^2) = \sigma_0 + \sigma_{0j}$, which is why `sigma_0` is not constrained to be zero (negative values on the log scale are always positive on the exponentiated scale). A matrix of dimension (I x 3) `std_dev_RN` is then specified for individuals' deviations from each of the RN population values (intercept, slope, and residual parameters), and a vector of length I `std_dev_W` is specified for individuals' deviations from the average fitness intercept. These can be thought of as standardized best linear unbiased predictors (BLUPs) or simply random effect z-scores.

Specifying `std_dev_` and `sd_` as separate parameters allows the model to estimate more efficiently, as the relative distances between subjects' random effects are separated from (not centered on) the absolute magnitude

of their variation. To see how and why this works, note that any normally distributed random variable

$$\boldsymbol{v} \sim \text{Normal}(0, \sigma_v)$$

can also be equivalently expressed as a standard normal variable (i.e. a z-score)

$$\boldsymbol{v_{\text{std}}} \sim \text{Normal}(0, 1)$$

scaled by the original SD

$$\boldsymbol{v} \equiv \boldsymbol{v_{\text{std}}} \sigma_v$$

We can use this identity to return the correctly scaled random intercepts $\boldsymbol{W_0}$ for the fitness model in the transformed parameters block, which will allow us to use these values directly while specifying the model likelihood.

```
transformed parameters {
  vector[J] W_0j = std_dev_W * sd_W0; //scaled random intercepts for fitness
//...
```

This code can also be placed in the model block to reduce file size if one isn't interested in saving the scaled values with the model output. To get the scaled random effects for the RN parameters, we also need to bring in the (co)variance between individuals' intercepts, slopes, and residual parameters. This is why we specified a parameter `R_chol` above in the parameters block for the *correlation* matrix $\boldsymbol{R}$ of the RNs. We do this, rather than specifying a single covariance matrix $\boldsymbol{P}$, because we are again going to speed up our computation by separating out the scale of RN parameter deviations (SDs, `sd_RN`) from their standardized associations (correlations), exploiting the identity $\boldsymbol{P} = \boldsymbol{SRS}$ explained with the formal model above. In addition, rather than using the Stan function `corr_matrix` for a full correlation matrix, we are also using a special function `Cholesky_factor_corr` to estimate a so-called *Cholesky factorization* of the full $\boldsymbol{R}$ matrix.

To understand why we do this, note that for any positive definite correlation matrix $\mathbf{R}$, a Cholesky decomposition can be defined such that

$$\mathbf{R} = \mathbf{R}_{\text{L}} \mathbf{R}_{\text{L}}^{\intercal}$$

where $\mathbf{R}_{\text{L}}$ is a lower-triangular matrix and $^{\intercal}$ indicates matrix transposition. This property means that we can always estimate the model using a smaller lower-triangular matrix $\mathbf{R}_{\text{L}}$ and subsequently recover the full positive-definitive matrix $\mathbf{R}$ by post-multiplying $\mathbf{R}_{\text{L}}$ with its transpose $\mathbf{R}_{\text{L}}^{\intercal}$. This trick is useful for making any Stan model sample more efficiently because it only requires estimating a reduced matrix of lower dimensionality, which lacks the redundant elements of the full symmetric correlation matrix.

With this basic understanding in place, we can now also specify the appropriately scaled random effects for the RN parameters in the transformed parameters block. This is accomplished as follows

```
//...
  matrix[J,3] RNj = std_dev_RN * diag_pre_multiply(sd_RN, R_chol)' ;
}
```

where the `*` operator now represents matrix multiplication. The function `diag_pre_multiply` creates a diagonal matrix with sd_RN, i.e. $\boldsymbol{S} = \text{diag}\left(\text{sd}(\boldsymbol{\mu_0}), \text{sd}(\boldsymbol{\beta_x}), \text{sd}(\boldsymbol{\sigma_0})\right)$ and then multiplies it with `R_chol`. The product of this multiplication is then transposed $^{\intercal}$ using the `'` operator in Stan. This specification gives the appropriate random effects for each individual, as described in the formal model. To see how this works, we can build on the univariate identity above, noting for the multivariate case that with a (J x p) matrix $\boldsymbol{V}$ of p phenotypes

$$\boldsymbol{V} \sim \text{MVNormal}(\mathbf{0}, \mathbf{P})$$

we can derive the appropriately scaled values with a matrix of standard normals $\boldsymbol{V_{\text{std}}}$ and a Cholesky decomposition of $\mathbf{P}$, so that

$$\boldsymbol{V} \equiv \boldsymbol{V_{\text{std}}} \mathbf{P}_{\text{L}}^{\intercal}$$

where

$$\mathbf{P}_{\mathrm{L}}^{\intercal} = \mathrm{Chol}(\mathbf{P})^{\intercal} = \mathrm{Chol}(\mathbf{SRS})^{\intercal} = (\mathbf{SR}_{\mathrm{L}})^{\intercal}$$

In this case, $\boldsymbol{V}_{\mathbf{std}}$ corresponds to `std_dev_RN` and the results of the function `diag_pre_multiply()` represent $\mathbf{P}_{\mathrm{L}}$ as just explained. This may seem like a lot of unnecessary work, but separating out the scale and associations of the random effects in this way will often lead to better model convergence and much more efficient model estimation. Therefore, these mathematically equivalent reparameterizations of the formal model are generally worth implementing although not always strictly necessary. Fortunately, one doesn't need to think much about the maths behind these steps to estimate the model correctly, and this code can generally be copied and applied to any random effect with little modification.

Model

Now that we've got our data and model parameters specified, it's time to write out the model we'd like to estimate. The model block contains the likelihood functions of the model, the priors for the basic parameters, as well as any data structures that one may want to create for pragmatic convenience in specifying the model but not save in the output (e.g. to reduce memory usage). We'll again work through each component of this block in turn.

```
model{
  //separate RN parameters
  vector[J] mu_0j = col(RNj,1); //intercepts
  vector[J] beta_xj = col(RNj,2); //slopes
  vector[J] sigma_0j = col(RNj,3); //residuals

  //initialize vectors for response models
  vector[N_z] mu; //linear predictor of phenotype expectation
  vector[N_z] sigma; //linear predictor of phenotype dispersion
  vector[N_W] theta; //linear predictor of fitness expectation
//...
```

In this first step, we specify a few new vectors to separate out individuals' values for each RN parameter from the matrix `RNj` created in the transformed parameters block. This is a purely pragmatic step that helps to reduce clutter in the model likelihood by avoiding repeated subsetting of the matrix for the respective columns `col(RNj,1)`, `col(RNj,2)`, and `col(RNj,3)`. We do this here rather than in the transformed parameters block because these vectors are redundant with the matrix and would thus be a waste of memory to save in the model output with `RNj`. It is important to realize that the order of indexing the columns in the random effect matrix is arbitrary at this stage. One could also treat column 3 as the intercepts column, for instance, with residuals in column 1. The important thing is that whatever order is used here reflects how the values are specified in the model likelihood. Wherever `mu_0j` and the other parameters are specified in the model likelihood will determine how the values from the respective columns of `RNj` are estimated. To further tidy up the model likelihood code without wasting memory, we create new vectors to temporarily hold the linear predictors of each phenotype and fitness measure. Note that there is no need to create a linear predictor for the dispersion of fitness, as nothing is predicting the residual SD of the fitness model, which is already taken care of by the `delta` parameter.

The next step is to fill in these vectors. For the response model of phenotype `z`

```
//...
  //RN model
  mu = mu_0 + mu_0j[ind_z] + (beta_x + beta_xj[ind_z]) .* x;
  sigma = sqrt(exp(sigma_0 + sigma_0j[ind_z]));
  z ~ normal(mu, sigma);
```

The index `ind_z` is used to appropriately repeat the random effect values of each RN parameter across repeated measures of the phenotype For example, if the first four observations are for individual 1, so that `ind_z`={1,1,1,1,2,...}, then `mu_0j[ind_z]` will repeat the first value of `mu_0j` for the first four observations in `z`. This is why it is essential to correctly match the order of the index and the response vectors. The final line tells Stan that the observed values `z` were generated by a Normal/Gaussian distribution with a likelihood function described by the expected means `z_mu` and standard deviations `z_sigma` of each observation. Note that `z_sigma` is calculated with the square root of the exponential function `sqrt(exp())` because the formal model is specified with a log link function on the variance. We apply the inverse link function to return estimates on the appropriate scale of SDs, i.e. if $\log(\sigma^2) = \sigma_0$ then $\text{sqrt}(\exp(\sigma_0)) = \sigma$. The operator `.*` indicates element-wise multiplication of vectors, which in this case multiplies the slopes `beta_x + beta_xj[ind_z]` by the observed environmental measures `x`. These three lines of code are therefore equivalent to

$$
\begin{aligned}
z_{jt} &\sim \text{Normal}\left(\mu_{jt}, \sigma_j\right) \\
\mu_{jt} &= \mu_0 + \mu_{0j} + \left(\beta_x + \beta_{xj}\right) x_{jt} \\
\log\left(\sigma_j^2\right) &= \sigma_0 + \sigma_{0j}
\end{aligned}
$$

The fitness model can also be specified accordingly

```
//...
  //fitness model
  theta = W_0 + W_0j[ind_W] +

          b[1] * mu_0j[ind_W] +
          b[2] * beta_xj[ind_W] +
          b[3] * sigma_0j[ind_W] +

          q[1] * (mu_0j[ind_W] .* mu_0j[ind_W]) +
          q[2] * (beta_xj[ind_W] .* beta_xj[ind_W]) +
          q[3] * (sigma_0j[ind_W] .* sigma_0j[ind_W]) +

          qc[1] * (mu_0j[ind_W] .* beta_xj[ind_W]) +
          qc[2] * (mu_0j[ind_W] .* sigma_0j[ind_W]) +
          qc[3] * (beta_xj[ind_W] .* sigma_0j[ind_W]) ;

  W ~ normal(theta, delta);
```

The `*` operator is used for multiplication with the scalar real values in `b`, `q`, and `qc`, while `.*` is necessary for element-wise multiplication of the individual values in each vector of RN parameters. This code is equivalent to our formal fitness model, where we've broken $\boldsymbol{q}$ into two separate vectors `q` and `qc` for convenience.

$$
\begin{aligned}
W_{jt} &\sim \text{Normal}\left(\theta_{jt}, \delta\right) \\
\theta_{jt} &= W_0 + W_{0j} + b_1\mu_{0j} + b_2\beta_{xj} + b_3\sigma_{0j} \\
&+ q_1\mu_{0j}^2 + q_2\beta_{xj}^2 + q_3\sigma_{0j}^2 + q_4\mu_{0j}\beta_{xj} + q_5\mu_{0j}\sigma_{0j} + q_6\beta_{xj}\sigma_{0j}
\end{aligned}
$$

The final necessary step is to introduce priors for all basic parameters listed in the parameters block. Given that we're using Bayesian inference for the analysis, specification of the formal model should also include priors for all estimated model parameters. We encourage using general-purpose, weakly regularizing priors on model parameters to promote more robust inference and enhance model identification (Lemoine 2019). The priors we specify for this example will be appropriate for general use with many datasets, but consideration should always be given to the scaling of covariates and intercepts, as well as the desired degree of regularization. Standardizing variables whenever possible helps to simplify this process.

$$\mu_0, \beta_x, \sigma_0, W_0, \boldsymbol{b}, \boldsymbol{q}, \boldsymbol{qc} \sim \text{Normal}(0, 1)$$

$$\text{sd}(\boldsymbol{\mu_0}), \text{sd}(\boldsymbol{\beta_x}), \text{sd}(\boldsymbol{\sigma_0}), \text{sd}(\boldsymbol{W_0}), \delta \sim \text{Exponential}(2)$$

$$\mathbf{R} \sim \text{LKJ}(2)$$

which translates into Stan code as

```
//...
  //model priors

  //fixed effects
  mu_0 ~ normal(0,1);
  beta_x ~ normal(0,1);
  sigma_0 ~ normal(0,1);
  W_0 ~ normal(0,1);

  b ~ normal(0,1);
  q ~ normal(0,1);
  qc ~ normal(0,1);

  //random effects
  sd_RN ~ exponential(2);
  R_chol ~ lkj_corr_cholesky(2);
  to_vector(std_dev_RN) ~ std_normal();

  sd_W0 ~ exponential(2);
  std_dev_W ~ std_normal();
  delta ~ exponential(2);
}
```

Note that for the vectors and matrices of random effect deviations `std_dev`, it is necessary to specify that all elements are described by a `std_normal()` distribution, which makes the non-centered parameterization introduced in the transformed parameters block above work. The prior `std_normal()` is equivalent to `normal(0,1)` as written for the other variables, but writing `std_normal()` is helpful to distinguish these essential priors from those than can be modified based on the goals of the analysis.

Generated quantities

The final programming block in our Stan model concerns the calculation of any quantities of interest which weren't directly estimated in earlier blocks. Here we can overcome the complexity of the reparameterizations used for increased efficiency by saving much easier to interpret full (co)variance $\mathbf{P}$ and correlation matrices $\mathbf{R}$, as well as the variances of the RN parameters.

```
generated quantities{
  matrix[3,3] R = R_chol * R_chol'; //RN correlation matrix
  matrix[3,3] S = diag_matrix(sd_RN); //RN SD matrix
  matrix[3,3] P = S*R*S; //RN covariance matrix
  vector<lower=0>[3] V_P = sd_RN .* sd_RN; //RN variances
}
```

Final model code

With each programming block coded, we can put them all together and write to a single `.stan` file in R. Doing this in RStudio using a formatted Stan file (`File > New File > Stan file`) will make coding and debugging (click `Check on save`) much easier.

```
write("
data {
  int<lower=1> J; //total individuals
  int<lower=1> N_z; //total number of pheontype meausres (Z)
  int<lower=1> N_W; //total number of fitness measures (W)
  array[N_z] int<lower=1> ind_z; //index of individual measurements (z)
  array[N_W] int<lower=1> ind_W; //index of individual measurements (W)
  vector[N_z] x; //environmental covariate
  vector[N_z] z; //phenotype
  vector[N_W] W; //fitness
}

parameters {
  //fixed population effects
  real mu_0; //z population intercept
  real beta_x; //z population slope
  real sigma_0; //z population dispersion

  real W_0; //W population intercept
  vector[3] b; //direct selection
  vector[3] q; //stabilizing/disruptive selection
  vector[3] qc; //correlational selection

  //random effects for z
  vector<lower=0>[3] sd_RN; //RN parameter sds
  matrix[J,3] std_dev_RN; //individual-level RN deviations
  cholesky_factor_corr[3] R_chol; //RN parameter correlations

  //random effects for W
  real<lower=0> sd_W0; //unexplained selection sd
  vector[J] std_dev_W; //individual-level selection deviations
  real<lower=0> delta; //W dispersion (SD of residuals)
}

transformed parameters {
  vector[J] W_0j = std_dev_W * sd_W0; //scaled random intercepts for fitness
  matrix[J,3] RNj = std_dev_RN * diag_pre_multiply(sd_RN, R_chol)' ;
}

model{
  //separate RN parameters
  vector[J] mu_0j = col(RNj,1); //intercepts
  vector[J] beta_xj = col(RNj,2); //slopes
  vector[J] sigma_0j = col(RNj,3); //residuals

  //initialize vectors for response models
  vector[N_z] mu; //linear predictor of phenotype expectation
  vector[N_z] sigma; //linear predictor of phenotype dispersion
```

```
  vector[N_W] theta; //linear predictor of fitness expectation

  //RN model
  mu = mu_0 + mu_0j[ind_z] + (beta_x + beta_xj[ind_z]) .* x;
  sigma = sqrt(exp(sigma_0 + sigma_0j[ind_z]));
  z ~ normal(mu, sigma);

  //fitness model
  theta = W_0 + W_0j[ind_W] +

          b[1] * mu_0j[ind_W] +
          b[2] * beta_xj[ind_W] +
          b[3] * sigma_0j[ind_W] +

          q[1] * (mu_0j[ind_W] .* mu_0j[ind_W]) +
          q[2] * (beta_xj[ind_W] .* beta_xj[ind_W]) +
          q[3] * (sigma_0j[ind_W] .* sigma_0j[ind_W]) +

          qc[1] * (mu_0j[ind_W] .* beta_xj[ind_W]) +
          qc[2] * (mu_0j[ind_W] .* sigma_0j[ind_W]) +
          qc[3] * (beta_xj[ind_W] .* sigma_0j[ind_W]) ;
  W ~ normal(theta, delta);

  //model priors

  //fixed effects
  mu_0 ~ normal(0,1);
  beta_x ~ normal(0,1);
  sigma_0 ~ normal(0,1);
  W_0 ~ normal(0,1);

  b ~ normal(0,1);
  q ~ normal(0,1);
  qc ~ normal(0,1);

  //random effects
  sd_RN ~ exponential(2);
  R_chol ~ lkj_corr_cholesky(2);
  to_vector(std_dev_RN) ~ std_normal();

  sd_W0 ~ exponential(2);
  std_dev_W ~ std_normal();
  delta ~ exponential(2);
}

generated quantities{
  matrix[3,3] R = R_chol * R_chol'; //RN correlation matrix
  matrix[3,3] S = diag_matrix(sd_RN); //RN SD matrix
  matrix[3,3] P = S*R*S; //RN covariance matrix
  vector<lower=0>[3] V_P = sd_RN .* sd_RN; //RN variances
}

", "mod1.stan")
```

Analyze data

Estimate model

To estimate this model, we first pass it to Stan for C++ compilation. This step will be the same regardless of how the model was coded.

```
#load package
library(rstan)


#rstan: compiles the model in C++ for MCMC estimation
mod1 = stan_model("mod1.stan")

#basic settings for rstan
options(mc.cores = parallel::detectCores())
rstan_options(auto_write = TRUE)

#or in cmdstan
set_cmdstan_path(...) #... = directory of installation
mod1 = cmdstan_model(stan_file = "mod1.stan")
```

The compiled model in `mod1` is now ready to be sampled immediately using Stan's cutting-edge Markov Chain Monte Carlo (MCMC) algorithm, which is accomplished by passing it to the `sampling()` function from RStan. As noted above, the CmdStan package can also be used for more efficient computation. We'll use RStan here because it is less prone to complications during installation and provides more user-friendly functionality. For default MCMC settings in Stan, we could run

```
#sampling posterior dist of the model with default MCMC settings
results = sampling(object = mod1, data = stan.dl)

#or in cmdstan
results = mod1$sample(data = fishdl)
```

However, given that our model is somewhat complex, it is helpful to use custom settings for the sampler that will reduce the risk of poor performance. In particular, we can manually specify that the MCMC sampler should use 500 iterations per chain to converge on the target joint posterior distribution `warmup=500`, with the subsequent 2000 iterations/chain used as posterior samples `iter = 2500` (i.e. `iter` - `warmup` = number of MCMC samples per chain). For users familiar with Gibbs and Metropolis-Hastings MCMC algorithms, this will likely seem like an insufficient number of samples. However, the No U-Turn sampler implemented in Stan tends to be dramatically more efficient than these classical algorithms, requiring much fewer samples to reach effective sample sizes for accurately approximating posterior distributions. The `init = 0` argument can be used to initialize the samplers near null values, which is not necessary but can aid the speed of convergence for complex models. We'll use four MCMC chains to assess model convergence across independent random samplers `chains=4`, with one core assigned to each chain for parallel processing `cores=4`. The appropriate number of cores to use will be contingent on one's hardware. The `adapt_delta=0.80` argument reduces the risk of divergent transitions during sampling. If errors are thrown about divergence transitions, this value can be increased up to a maximum of 1, e.g. 0.90 or 0.95.

```
#progress of MCMC chains can be tracked in the viewer pane of RStudio
results = sampling(object = mod1, data = stan.dl, warmup=500, iter = 1000, init = 1e-4,
                           chains=4, cores=4, control=list(adapt_delta=0.90) )
```

```
#or in cmdstan (generally faster)
results = mod1$sample(data = stan.dl, iter_warmup = 500, iter_sampling = 1000, init = 1e-4,
                      chains = 4,  parallel_chains = 4, adapt_delta = 0.90)
```

Some readers may note that there is no argument specified for thinning the chain, which implicitly specifies the default argument `thin=1`. Although there are specific contexts where thinning is useful for MCMC sampling, it is generally unnecessary and computationally inefficient (Link and Eaton 2012).

If you estimate a model in Stan and receive a warning or error, it may indicate issues with the MCMC sampler, which should always be taken seriously. Further description of these and other warnings can be found in the **Stan Warning Guide**. Some warnings can be safely ignored in particular contexts, but efforts should always be taken to first remove the issue before interpreting or reporting results from the sampler. If you receive a warning regarding divergent transitions, a straightforward first step is to increase the `adapt_delta` value closer to 1, e.g. from 0.95 to 0.99. The higher this value, the slower the model will sample but the less likely that divergent iterations will occur. Similarly, if warnings of bulk or tail ESS are received, a first step is to simply let the chains sample for longer by increasing the `iter`, e.g. from 3000 to 3500 or 4000.

Investigate results

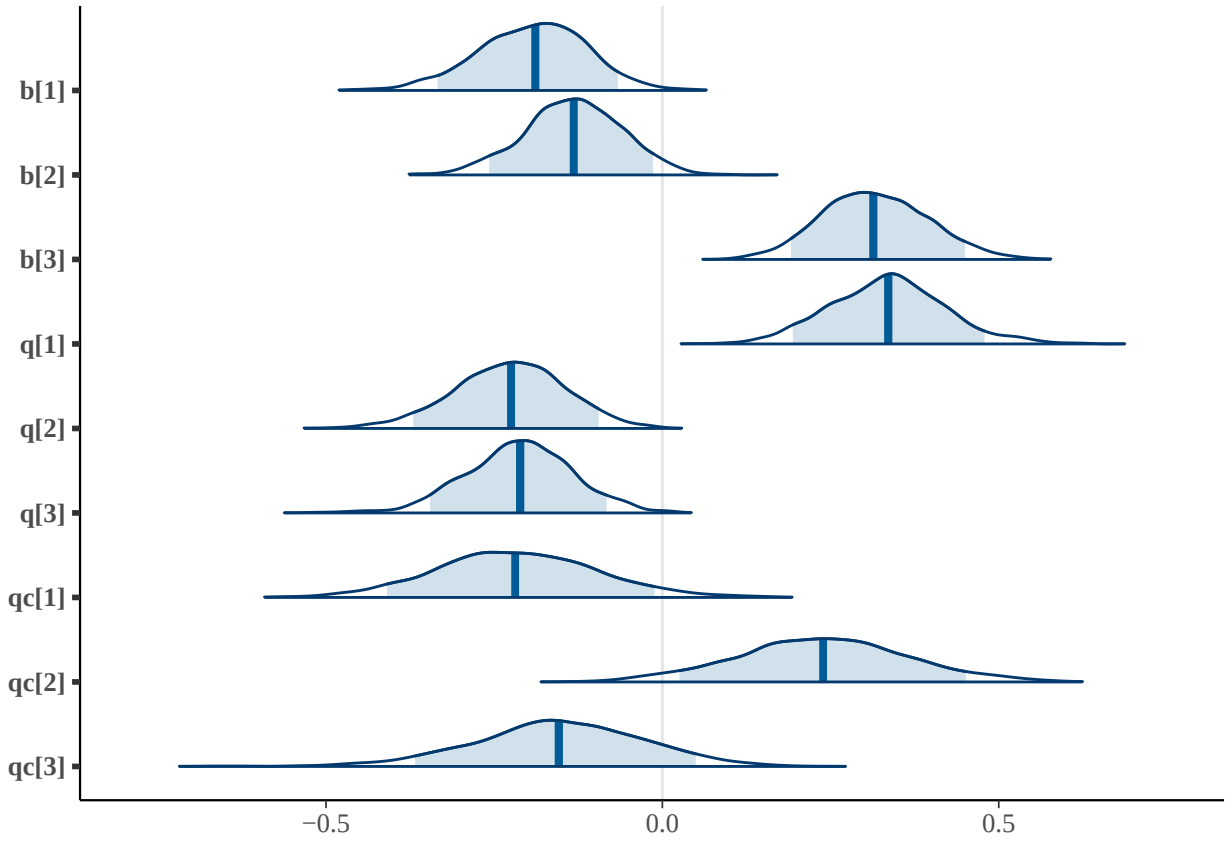
Assuming that the sampling procedure worked as intended, we can move ahead with extracting the posterior MCMC samples from the model.

```
#extracts posterior estimates
samples =  rstan::extract(results)

#or cmdstan (using custom function to mimic rstan)
samples = extract(results)

#quick glance of MCMC samples for linear selection effects
head(samples$b[,1:3])
```

```
##
## iterations       [,1]         [,2]      [,3]
##       [1,] -0.2214934 -0.252121794 0.3542963
##       [2,] -0.1978726 -0.111582286 0.3732336
##       [3,] -0.1841798 -0.001276365 0.3766329
##       [4,] -0.1821944 -0.090781450 0.5194939
##       [5,] -0.1951355 -0.064803276 0.3471871
##       [6,] -0.1893245 -0.185388096 0.2622879
```

Before hypothesis testing, it is useful to visualize the shapes and locations of the posterior distributions of model parameters. There are many ways this can be accomplished. For example, the `bayesplot` package can be used to generate a variety of useful plots.

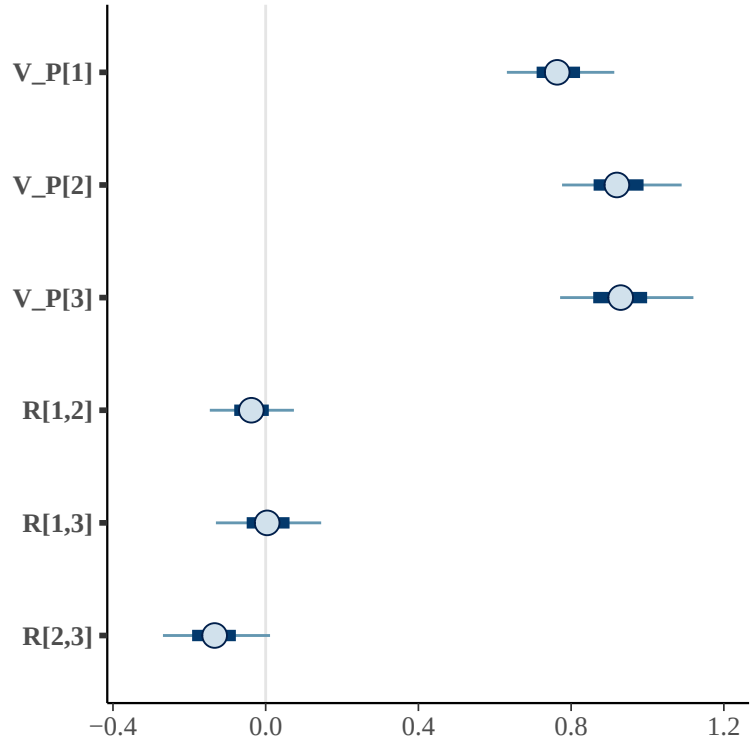
```
library(bayesplot)

#selection coefficients, with shaded central tendencies and 90% CIs
mcmc_areas(results, pars = c( paste0("b[",seq(1:3),"]"), paste0("q[",seq(1:3),"]"), paste0("qc[",seq(1:
```



340

341

```
#variance & corrs of RN parameters, mean and 50% CIs (dark line) and 90% CIs (light line)
mcmc_intervals(results, pars = c( paste0("V_P[",seq(1:3),"]"),"R[1,2]","R[1,3]","R[2,3]" ) )
```



Point estimates summarizing these posteriors can be quickly generated by summarizing the model.

```
#only first 17 parameters, round to ease interpretation
round(summary(results)$summary[1:16,],2)
```

```
##           mean se_mean   sd  2.5%   25%   50%   75% 97.5%  n_eff Rhat
## mu_0      0.00    0.00 0.04 -0.09 -0.03  0.00  0.03  0.09 890.59 1.00
## beta_x   -0.01    0.00 0.05 -0.11 -0.04 -0.01  0.03  0.09 900.31 1.00
## sigma_0   0.71    0.00 0.06  0.59  0.67  0.71  0.75  0.81 473.27 1.01
## W_0       0.97    0.01 0.11  0.76  0.90  0.96  1.04  1.20 222.47 1.03
## b[1]     -0.19    0.00 0.08 -0.36 -0.25 -0.19 -0.14 -0.04 343.58 1.01
## b[2]     -0.13    0.00 0.07 -0.28 -0.18 -0.13 -0.08  0.01 318.98 1.02
## b[3]      0.32    0.01 0.08  0.17  0.26  0.31  0.37  0.47 152.39 1.05
## q[1]      0.34    0.01 0.09  0.17  0.28  0.34  0.39  0.52 213.23 1.01
## q[2]     -0.23    0.01 0.08 -0.40 -0.28 -0.22 -0.17 -0.07 199.15 1.01
## q[3]     -0.21    0.01 0.08 -0.37 -0.26 -0.21 -0.16 -0.06  99.43 1.06
## qc[1]    -0.22    0.01 0.12 -0.45 -0.30 -0.22 -0.14  0.03 148.90 1.01
## qc[2]     0.24    0.01 0.13 -0.01  0.15  0.24  0.32  0.50 170.02 1.02
## qc[3]    -0.15    0.01 0.13 -0.42 -0.23 -0.15 -0.07  0.08 103.61 1.04
## sd_RN[1]  0.87    0.00 0.05  0.78  0.84  0.87  0.91  0.97 447.50 1.01
## sd_RN[2]  0.96    0.00 0.05  0.87  0.93  0.96  0.99  1.06 573.02 1.00
## sd_RN[3]  0.96    0.00 0.05  0.86  0.93  0.96  1.00  1.07 481.84 1.01
```

The extracted posterior samples can also be manually plotted and summarized using base R functions. The **shinystan** package also provides a very helpful graphical user interface for looking at all aspects of model

fit and estimation. Running this code will open a new window in your internet browser for looking at the model in greater detail.

```
library(shinystan)
launch_shinystan(results)
```

Hypothesis testing

MCMC not only facilitates sampling complex Bayesian models but also conducting straightforward and direct forms of hypothesis testing. For example, if we want to know how much support there is for positive linear and nonlinear selection effects p_+, we simply need to calculate the proportion of the MCMC samples for these parameters with positive magnitude, which approximates the area under the posterior distribution providing support for positive effects.

```
#for each column, calculate probability of positive effect
apply(samples$b, 2, FUN = function(x) sum(x>0)/length(x) )
```

```
## [1] 0.0075 0.0300 1.0000
```

```
apply(samples$q, 2, FUN = function(x) sum(x>0)/length(x) )
```

```
## [1] 1.0000 0.0015 0.0045
```

```
apply(samples$qc, 2, FUN = function(x) sum(x>0)/length(x) )
```

```
## [1] 0.040 0.967 0.109
```

Strong support for positive selection effects will approach 1, while strong support for negative selection effects will approach 0. If the model is uncertain about the direction, the value will instead move toward 0.5. A more general way to write the function is to test whatever direction the median effect size is in (so p_+ or p_- depending on the location of the posterior).

```
#for each column, calculate probability of positive effect
apply(samples$b, 2, FUN = function(x)   sum(sign(x) == sign(median(x)))/length(x) )
```

```
## [1] 0.9925 0.9700 1.0000
```

```
apply(samples$q, 2, FUN = function(x) sum(sign(x) == sign(median(x)))/length(x) )
```

```
## [1] 1.0000 0.9985 0.9955
```

```
apply(samples$qc, 2, FUN = function(x) sum(sign(x) == sign(median(x)))/length(x) )
```

```
## [1] 0.960 0.967 0.891
```

We can also calculate quantile-based Bayesian credible intervals (CI). 90% CI are particularly useful for interpretation. If the 90% CI excludes 0, then there is at least 0.95 probability in support of a directional effect (i.e. ≤ 0.5 for an effect in the opposite direction).

```r
#for each column, calculate quantile based CI
apply(samples$b, 2, FUN = function(x) quantile(x, c(0.05, 0.95)) ) #90% CI
```

```
##
##              [,1]       [,2]      [,3]
##   5%  -0.33416102 -0.2575130 0.1912458
##   95% -0.06632102 -0.0137624 0.4498332
```

```r
apply(samples$q, 2, FUN = function(x) quantile(x, c(0.05, 0.95)) ) #90% CI
```

```
##
##             [,1]        [,2]        [,3]
##   5%  0.1939239 -0.37042841 -0.34518754
##   95% 0.4790374 -0.09467571 -0.08289907
```

```r
apply(samples$qc, 2, FUN = function(x) quantile(x, c(0.05, 0.95)) ) #90% CI
```

```
##
##              [,1]       [,2]       [,3]
##   5%  -0.40934611 0.02526734 -0.3676424
##   95% -0.01150826 0.45137351  0.0496357
```

It is important to keep in mind that the size of CI interval is semi-arbitrary and so should not in itself be used as a discrete threshold for determining whether effects are 'significant' (McElreath (2020)). Although 0.95 is a useful heuristic for designating clear evidence of an effect, discretizing this information into "significant" or "non-significant" is generally a waste of information. Put another way, these Bayesian hypothesis tests provides a continuous measure of evidence that should also be interpreted continuously. Much as the difference between a significant and non-significant result is itself often not statistically significant (see McShane et al. 2019 for discussion), so too is the difference between e.g. a posterior probability of 0.93 and 0.97 not necessarily indicative of crossing a biologically or mathematically meaningful threshold. Thus, one should eschew the notion that a posterior probability <0.95 indicates "no evidence of an effect", and instead get comfortable describing varying degrees of support (weak, moderate, and strong) for or against hypothesized effects. Any probability greater than 0.50 provides some (perhaps extremely weak) support for an effect, but most researchers would be uncomfortable to confidently assert empirical claims without much greater empirical support in their favor, e.g. evidence approaching only a 1/20 chance of an effect in the opposite direction (i.e. a posterior probability of 0.95). Thus, one should qualify the strength of evidence and interpet accordingly, rather than simply dichotomizing effects into two bins. Encouraging this Bayesian attitude by taking a continuous approach to statistical inference encourages researchers to put greater emphasis on effect sizes and multiple other metrics that collectively increase or decrease the overall biological significance of an empirical finding (McShane et al. 2019).

A variety of other hypotheses could also be easily tested for any parameter in the model. For instance, we could ask whether there is support for directional selection on intercepts being greater than directional selection on slopes among individuals.

```r
sum(samples$b[,1] > samples$b[,2])/length(samples$b[,1])
```

```
## [1] 0.2915
```

We can also quantify further pieces of information about the difference of these effect sizes to inform our inferences

```r
diff = samples$b[,1] - samples$b[,3] #posterior of the difference in coefs
median(diff) #central tendency of difference
```

```
## [1] -0.5001592
```

```r
mad(diff) #median absolute deviation (robust SD) of expected difference
```

```
## [1] 0.1195088
```

```r
quantile(diff, c(0.05,0.95)) #90% CI of expected difference
```

```
##         5%        95%
## -0.7108768 -0.3191017
```

If one is so inclined, more robust null hypothesis tests can also be conducted within a Bayesian framework by specifying a range of biologically trivial effect sizes. For example, on a standardized scale, regression coefficients $-0.10 < r < 0.10$ are extremely small and usually negligible effects. We might, therefore, think of these as "trivial hypothesis" tests rather than null hypothesis tests per se. Comparing these trivial hypothesis tests with directional hypothesis tests can provide further pieces of information. Consider the RN parameter correlations

```r
R=samples$R[,,] #3d array, 1 dim = samples, 2 dim = rows, 3 dim = columns

#directional hypothesis tests
sum(R[,1,2]<0)/length(R[,1,2]) #cor(intercept, slope)
```

```
## [1] 0.71
```

```r
sum(R[,1,3]>0)/length(R[,1,3]) #cor(intercept, residual)
```

```
## [1] 0.515
```

```r
sum(R[,2,3]>0)/length(R[,2,3]) #cor(slope, residual)
```

```
## [1] 0.0645
```

```r
#trivial hypothesis tests
sum(-0.1< R[,1,2] & R[,1,2]  <0.1)/length(R[,1,2])
```

```
## [1] 0.8005
```

```r
sum(-0.1< R[,1,3] & R[,1,3]  <0.1)/length(R[,1,3])
```

```
## [1] 0.7665
```

```r
sum(-0.1< R[,2,3] & R[,2,3]  <0.1)/length(R[,2,3])
```

```
## [1] 0.3375
```

The directional and trivial hypothesis tests work together to inform our understanding of the direction and magnitude of the estimated correlations. While strong supported is provided for non-trivial correlations among intercepts and slopes and residuals, the correlation among intercepts and residuals is both highly uncertain in its direction and largely overlapping the trivial effect size range, providing little evidence for an association of biologically biologically meaningful magnitude

Calculate selection gradients

Analytic approach (Gaussian case)

We now want to calculate selection gradients from the estimated model. For the simple Gaussian case considered here, this is straightforward to accomplish analytically. Since we didn't mean-scale fitness prior to the analysis, we need to divide all gradients by mean fitness. For Gaussian measures, one can of course also mean-scale fitness prior to analysis. However, the sample mean will often be confounded by measurement error (Dingemanse, Araya-Ajoy, and Westneat 2021), motivating use of the estimated fitness intercept `W_0`. In cases where adjusted effects are included, it is important to consider whether the intercept is equal to average fitness or instead conditional of a specific factor or covariate value. The quadratic selection effects for the squared trait values (i.e. not correlational selection effects) then need to be multiplied by 2 (Stinchcombe et al. 2008). This step could be circumevented by including $0.5*q$ in the Stan model formula above. However, we have not done so here in keeping with the general model structure presented in the main text, which applies irrespective of the trait distribution. It is important that these arithmetic operations are done across each posterior sample to ensure that posterior uncertainty is retained across stages of analysis (Stinchcombe, Simonsen, and Blows 2014).

```r
b = samples$b[,1:3]
q = samples$q[,1:3]
qc = samples$qc[,1:3]

beta = apply(b, 2, function(x) x/ samples$W_0 )
colnames(beta) =  c("mu","beta","sigma")
data.frame(median = apply(beta, 2, median),
           robust.SD = apply(beta, 2, mad),
           pd = apply(beta, 2, function(x)
             sum(sign(x) == sign(median(x)))/length(x)))
```

```
##           median  robust.SD     pd
## mu    -0.1964688 0.08650714 0.9925
## beta  -0.1352696 0.07923651 0.9700
## sigma  0.3258650 0.08071981 1.0000
```

```r
#doubling values (unless explicitly putting 0.5 * q in likelihood)
quad_diag = apply(q, 2, function(x) 2 * x / samples$W_0 )
colnames(quad_diag) = c("mu.mu","beta.beta","sigma.sigma")
data.frame(median = apply(quad_diag, 2, median),
           robust.SD = apply(quad_diag, 2, mad),
           pd = apply(quad_diag, 2, function(x)
             sum(sign(x) == sign(median(x)))/length(x)))
```

```
##                  median robust.SD     pd
## mu.mu         0.7014768 0.2142443 1.0000
## beta.beta    -0.4634167 0.1499078 0.9985
## sigma.sigma -0.4320118 0.1460994 0.9955
```

```
#correlational selection
quad_corr = apply(qc, 2, function(x) x / samples$W_0 )
colnames(quad_corr) = c("mu.beta","mu.sigma","beta.sigma")
data.frame(median = apply(quad_corr, 2, median),
           robust.SD = apply(quad_corr, 2, mad),
           pd = apply(quad_corr, 2, function(x)
             sum(sign(x) == sign(median(x)))/length(x)))
```

```
##                 median robust.SD    pd
## mu.beta    -0.2264383 0.1256402 0.960
## mu.sigma    0.2459766 0.1311841 0.967
## beta.sigma -0.1572348 0.1243206 0.891
```

Standardized gradients can also be calculated with the associated RN parameter variances and standard deviations (or population means).

```
#function for correlational gradients
pwSD = function(M) combn(ncol(M), 2, \(ij) as.matrix(M)[, ij[1]] * as.matrix(M)[, ij[2]])

sd.beta = beta * samples$sd_RN
sd.quad_diag = quad_diag * samples$sd_RN^2
sd.quad_corr = quad_corr * pwSD(samples$sd_RN)
```

The same techniques for summarizing and visualizing shown above can be used to plot and report these final results.

Numeric approach (general case)

In many cases, fitness residuals will not be well described by a Gaussian distribution, warranting an alternative numeric approach. We demonstrate this here with our Gaussian dataset to show the compatibility of these approaches. See (Martin et al. 2025) for a more in-depth worked example of a complex model combining Gaussian, binomial, and cumulative-logit (ordinal) fitness models.

The first step codes the fitness function `Wbar` in R with the estimated parameters so that we can return the expected mean fitness ($\bar{\theta}$). We want the function to capture all posterior uncertainty, and so it is run over each iteration `i` of MCMC samples. The RN parameter values are denoted by a vector `p` with as many elements as there are parameters under selection (so in this case, 3). Adapting this code to non-Gaussian distributions simply requires using the right link functions, e.g. `theta = logistic(...)` for a Binomial model with logit link or `theta = exp(...)` for a Poisson model with log link. The structure of the function should exactly match the likelihood function in the Stan model.

We then use the `numDeriv` package to calculate the selection gradients by simply taking the first and second partial derivatives of the function with respect to expected RN parameters across the population, which are by definition 0 due to mean centering. Lists are used to collect the results for each iteration before reorganizing.

```
library(numDeriv)

beta_num = list()
gamma_num = list()

for(i in 1:length(samples$W_0)){ #total posterior samples
  #Wbar = returns mean value of fitness function based on model likelihood
  #p[1]-[3] are placeholders for reaction norm intercepts, slopes, and residuals
  Wbar = function(p,i) {
    theta =  (
              samples$W_0[i] + samples$W_0j[i, stan.dl$ind_z] +

              b[i,1] * p[1] +
              b[i,2] * p[2] +
              b[i,3] * p[3] +

              q[i,1] * p[1]^2 +
              q[i,2] * p[2]^2 +
              q[i,3] * p[3]^2 +

              qc[i,1] * p[1] * p[2] +
              qc[i,2] * p[1] * p[3] +
              qc[i,3] * p[2] * p[3]
              )
    return(mean(theta))  }

  p = rep(0,3)
  first.derivatives = grad(func = Wbar, x = p, i = i)
  second.derivatives = hessian(func = Wbar, x = p, i = i)
  denom = Wbar(p = p, i = i)
  beta_num[[i]] = first.derivatives/denom
  gamma_num[[i]] = second.derivatives/denom
}

#organize results
pars =  c("mu", "beta", "sigma")
beta2 = do.call(rbind.data.frame, beta_num)
colnames(beta2) = pars

gamma_num2 = do.call(rbind.data.frame, lapply(gamma_num, reshape2::melt) )
gamma_num2 = gamma_num2[gamma_num2$Var1 <= gamma_num2$Var2,]
gamma_num2$Var1 = factor(pars[gamma_num2$Var1], levels = pars)
gamma_num2$Var2 = factor(pars[gamma_num2$Var2], levels = pars)
gamma_num2$vars = factor(paste0(gamma_num2$Var1,".", gamma_num2$Var2),
                         levels = c("mu.mu", "beta.beta", "sigma.sigma",
                                    "mu.beta", "mu.sigma", "beta.sigma"))
quad_diag2 = droplevels(gamma_num2[gamma_num2$Var1==gamma_num2$Var2, ])
quad_diag2 = as.data.frame(split(quad_diag2$value, quad_diag2$vars))
quad_corr2 = droplevels(gamma_num2[gamma_num2$Var1!=gamma_num2$Var2, ])
quad_corr2 = as.data.frame(split(quad_corr2$value, quad_corr2$vars))

#summarize
apply(beta2, 2, median)
```

```
##          mu       beta      sigma
## -0.1956786 -0.1347033  0.3262571
```

```
apply(quad_diag2, 2, median)
```

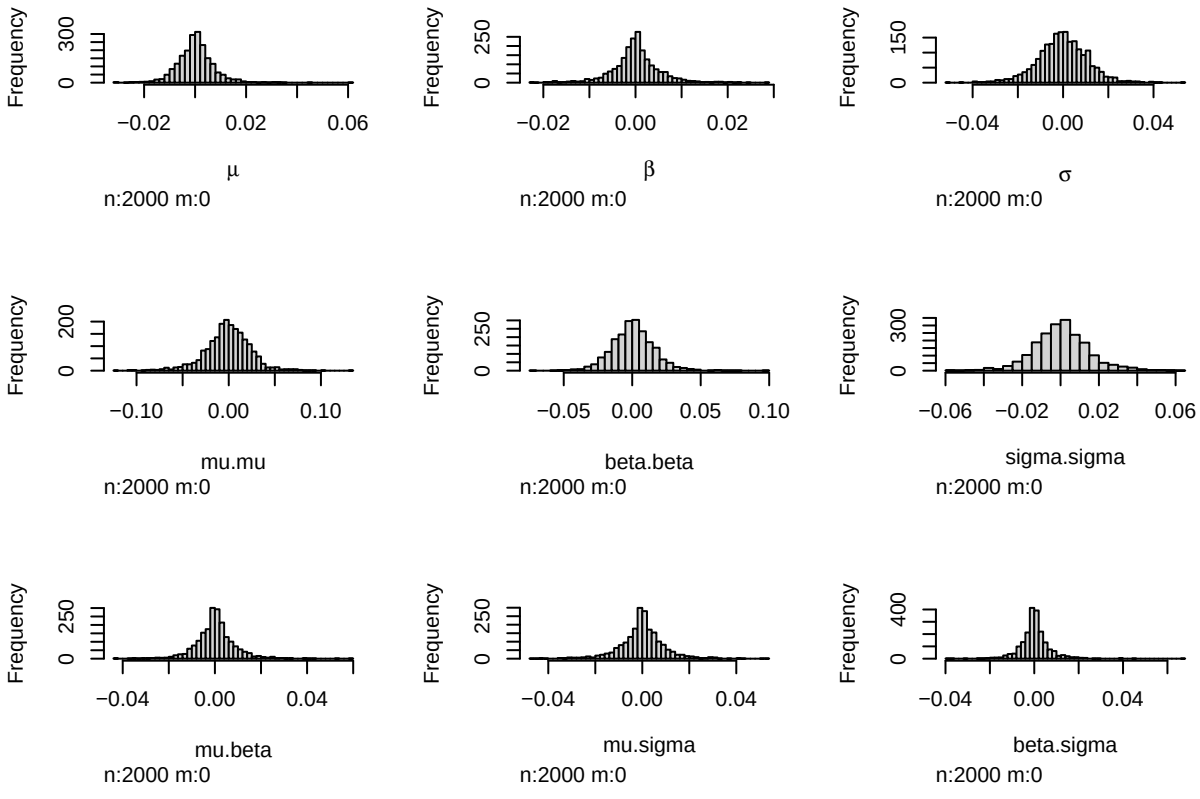
```
##       mu.mu   beta.beta sigma.sigma
##   0.7019211  -0.4624542  -0.4317222
```

```
apply(quad_corr2, 2, median)
```

```
##    mu.beta   mu.sigma beta.sigma
## -0.2277783  0.2463497 -0.1574348
```

Very small, stochastic differences are expected between the analytic and numeric approaches, but the expected difference is centered on 0, indicating convergence between methods.

```
library(Hmisc)
#compare
hist(cbind(beta2 - beta, quad_diag2 - quad_diag, quad_corr2 - quad_corr))
```



References

Carpenter, B., A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, and... A. Riddell. 2017. "Stan: A Probabilistic Programming Language." *Journal of Statistical Software* 74. https://www.jstatsoft.org/article/view/v076i01.

Dingemanse, N. J., Y. G. Araya-Ajoy, and D. F. Westneat. 2021. "Most Published Selection Gradients Are Underestimated: Why This Is and How to Fix It." *Evolution* Early View.

Hoffman, M. D., and A. Gelman. 2014. "The No-u-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–623.

Lemoine, N. P. 2019. "Moving Beyond Noninformative Priors: Why and How to Choose Weakly Informative Priors in Bayesian Analyses." *Oikos* 128. https://onlinelibrary.wiley.com/doi/full/10.1111/oik.05985.

Link, W. A., and M. J. Eaton. 2012. "On Thinning of Chains in MCMC." *Methods in Ecology and Evolution* 3: 112–15.

Martin, J. S., D. F. Westneat, S. Nakagawa, J. Schroeder, and T. Burke. 2025. "Measuring Selection on Reaction Norms: Lack's Principle and Plasticity in Clutch Size." *Evolution* qpaf168.

McElreath, R. 2020. *Statistical Rethinking: A Bayesian Course with Examples in r and Stan*. 2nd ed. CRC Press. https://xcelab.net/rm/statistical-rethinking/.

McShane, B. B., D. Gal, A. Gelman, C. Robert, and J. L. Tackett. 2019. "Abandon Statistical Significance." *The American Naturalist* 73: 235–45.

R Core Team. 2020. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org.

Stinchcombe, J. R., A. F. Agrawal, P. A. Hohenlohe, S. J. Arnold, and M. W. Blows. 2008. "Estimating Nonlinear Selection Gradients Using Quadratic Regression Coefficients: Double or Nothing?" *Evolution* 68. https://onlinelibrary.wiley.com/doi/full/10.1111/evo.12321.

Stinchcombe, J. R., A. K. Simonsen, and M. W. Blows. 2014. "Estimating Uncertainty in Multivariate Responses to Selection." *Evolution* 68. https://onlinelibrary.wiley.com/doi/full/10.1111/evo.12321.