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 (see RStan link). Once you are able to effectively use Stan in R, you can begin creating the `.stan` files necessary for estimating models. 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. As will become apparent in subsequent sections of the tutorial, this feature is very helpful 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")
```

This series is currently under development and will continue to be extended in the coming months to cover a variety of additional modeling scenarios. For now, a full Gaussian model is presented to provide a general introduction to the proposed approach.

Generate data

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 pigeonholing things into prepackaged toolkits that may require undesirable assumptions or simplifications. Researchers unfamiliar with formal statistical models are encouraged to see McElreath (2020) for a detailed, accessible treatment in the context of Bayesian data analysis.

For this introductory tutorial, we'll consider Gaussian reaction norm and fitness models, following notation used in the main text (see **Eq. 1-2**). The probability density function for measurement z_{jt} of individual j's phenotype z at time t is given by

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

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 parameters $\boldsymbol{\sigma_0}$. To do so, the full multivariate/multi-response model simultaneously estimates a probability function for proxy measurement W_{jt} of individual j's fitness 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. We'll consider this 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 fitness residuals δ and should, therefore, be excluded from the model.

Simulate dataset

Now we want to simulate a dataset appropriate for estimating the formal model. This can be easily accomplished by saving the `NLS_RN_functions.R` file from the Github page for this paper and using the `sim_RN_Gaus()` function. By default for this function, population parameter values are fixed so that $\mu_0 = \beta_x = 0$, $\sigma_0 = \log(0.6)$, $W_0 = \delta = 1$, and $\text{var}(\boldsymbol{\mu_0}) = \text{var}(\boldsymbol{\beta_x}) = \text{var}(\boldsymbol{\sigma_0}) = \text{var}(\boldsymbol{W_0}) = 0.3$, 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`. The effect size values shuold be input as absolute (positive) values, and 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 this tutorial, we'll generate data for a sample of 1000 individuals with 4 repeated behavioral measures across the lifespan and 3 fitness proxy or component measures, with selection regression coefficients of moderate-to-large statistical effect size from 0.3-0.5.

```
#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("NLS_RN_functions.R")

#simulate dataset
stan.dl = sim_RN_Gaus(J = 1000, rep_z = 4, rep_W = 3, l_es = 0.3, u_es = 0.5)
```

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] 0.694 -1.33 -0.355 0.624 -0.835 ...
##  $ z      : num [1:3000] 0.282 -0.578 -0.449 -0.646 0.744 ...
##  $ W      : num [1:2000] 2.3 2.1 0.29 2.71 0.29 ...
##  $ true_b : num [1:3] 0.415 -0.453 -0.482
##  $ true_q : num [1:3] 0.374 -0.435 -0.411
##  $ true_qc: num [1:3] 0.404 -0.463 0.337
```

Note that 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. Firstly, 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). Secondly, no character strings can be supplied to Stan, including for indexing subject IDs (e.g. "monkey1," "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. This can also be easily accomplished in R for any character indices present in your original dataset. 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
```

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 it is important to realize that 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 ragged datasets with highly 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 may at first seem like a cumbersome task, but it affords a degree of flexibility and autonomy necessary for easily estimating nonlinear selection on RNs, which to the best of my knowledge cannot be accomplished with other mainstream statistical software. Nonetheless, it is important to recognize that some practice and trial-and-error will also be required to gain competency and comfortability with Stan. I therefore 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`. Alternatively, each selection effect could be written out as a separate effect, e.g. `real b_mu`, `real q_beta`, `qc_mu_sigma`, etc. aiding interpretation but decreasing the efficiency of model coding. 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. Note that

the fitness random effect parameters should be excluded here and below if only a single fitness measure is available per subject.

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)

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

scaled by the original SD

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

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 $\boldsymbol{V}_{\mathbf{std}}$ and a Cholesky decomposition of $\mathbf{P}$, so that

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

where

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

In this case, $\boldsymbol{V}_{\mathbf{std}}$ corresponds to `std_dev_RN` and the results of the function `diag_pre_multiply()` represent $\mathbf{P}_{\text{L}}$ 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

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

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.

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

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.

```
#load package
library(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)
```

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)
```

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 3000 iterations/chain used as posterior samples `iter = 3500` (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=0.90` argument reduces the risk of divergent transitions during sampling. If errors are thrown about divergence transitions.

```
#progress of MCMC chains can be tracked in the viewer pane of RStudio
results = sampling(object = mod1, data = stan.dl, warmup=500, iter = 3500, init = 0,
                          chains=4, cores=4, control=list(adapt_delta=0.90) )
#save model
saveRDS(results, "results_mod1.RDS")
```

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. 0.95 or 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 3500 to 4000 or 4500.

Investigate results

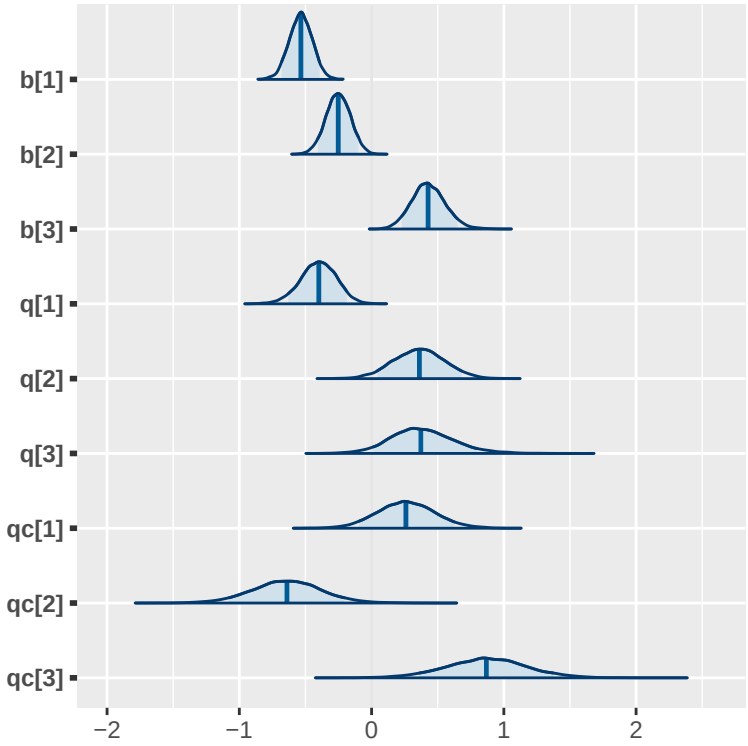
Assuming that the sampling procedure worked as intended, we can move ahead with extracting the posterior MCMC samples from the model.

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

#quick glance of MCMC samples for linear selection coefficients
head(samples$b[,1:3]) #rows are MCMC samples, cols are b[1-3]
```

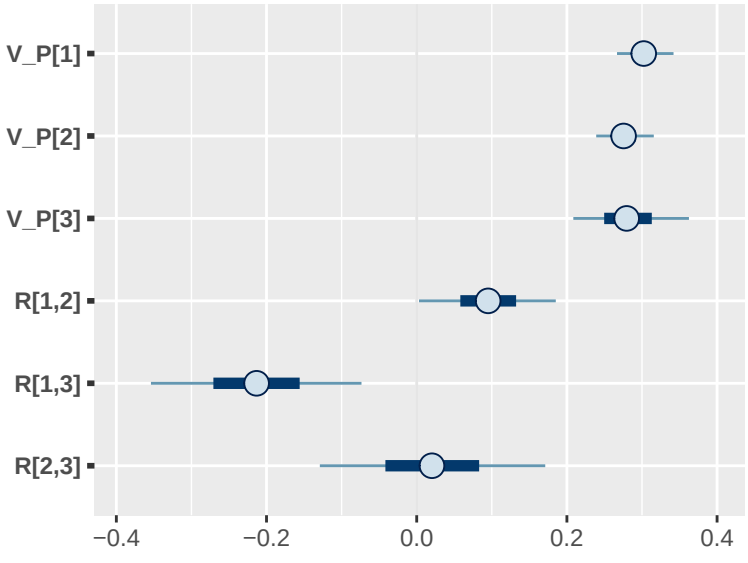
```
##
## iterations       [,1]       [,2]      [,3]
##       [1,] -0.6234310 -0.2226782 0.3421458
##       [2,] -0.5737457 -0.0848980 0.3960587
##       [3,] -0.6021217 -0.1549720 0.4902873
##       [4,] -0.6582333 -0.2700017 0.3059763
##       [5,] -0.6446493 -0.2307863 0.2538363
##       [6,] -0.3647965 -0.1683339 0.3793849
```

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(1:3),"]") ), prob = 0.9)
```




```
#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 for these posteriors can be quickly generated by summarizing parameters of interest in the model.

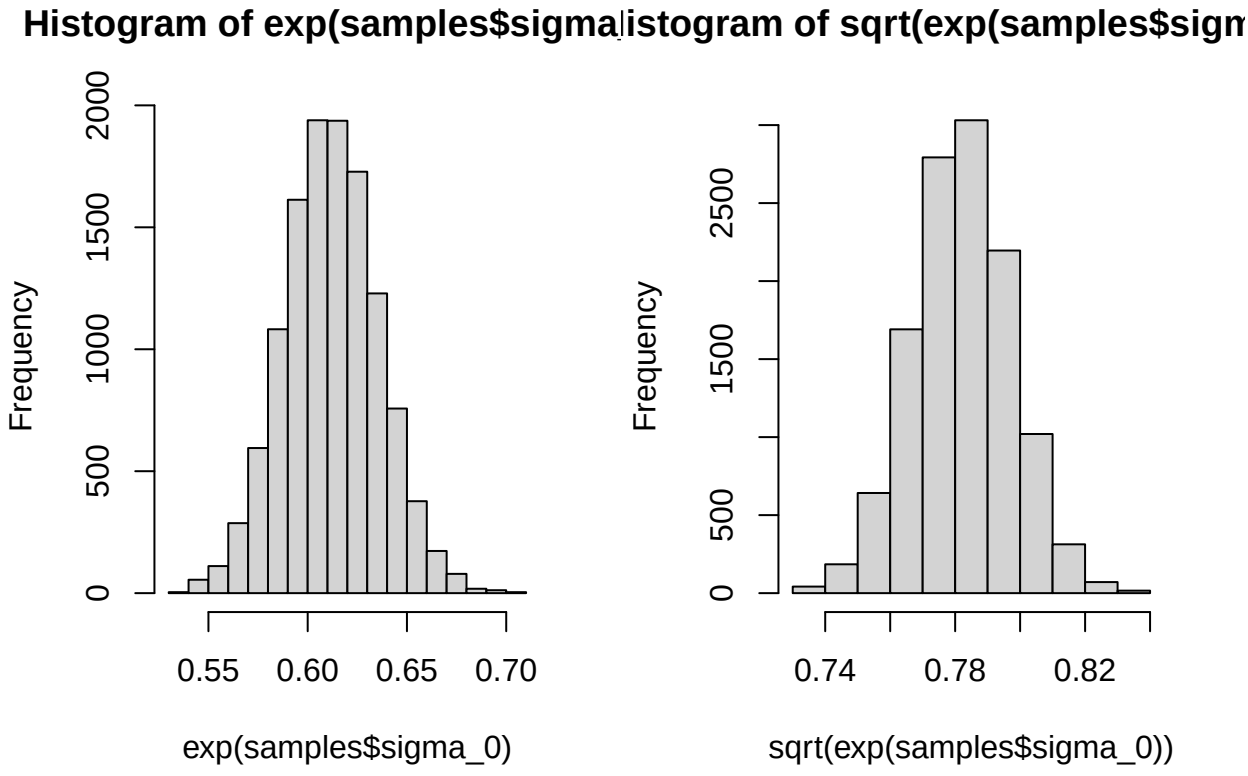
```
#only first 17 parameters, round to ease interpretation
round(summary(results)$summary[1:16,],2)
```

```
##           mean se_mean   sd  2.5%   25%   50%   75% 97.5%   n_eff Rhat
```

```
## mu_0      0.01    0.00 0.02 -0.04 -0.01  0.01  0.02  0.05 9336.90 1.00
## beta_x    0.03    0.00 0.02 -0.01  0.02  0.03  0.05  0.08 9048.31 1.00
## sigma_0  -0.49    0.00 0.04 -0.57 -0.52 -0.49 -0.47 -0.42 4200.64 1.00
## W_0       1.04    0.00 0.07  0.89  0.99  1.04  1.09  1.18 2048.77 1.00
## b[1]     -0.54    0.00 0.09 -0.71 -0.59 -0.53 -0.48 -0.36 3014.83 1.00
## b[2]     -0.25    0.00 0.09 -0.44 -0.31 -0.25 -0.19 -0.07 3390.86 1.00
## b[3]      0.43    0.00 0.13  0.19  0.35  0.43  0.51  0.70 1750.09 1.00
## q[1]     -0.40    0.00 0.14 -0.68 -0.49 -0.40 -0.31 -0.14 3136.02 1.00
## q[2]      0.36    0.01 0.19 -0.02  0.23  0.36  0.49  0.74 1229.96 1.00
## q[3]      0.38    0.01 0.24 -0.06  0.22  0.37  0.53  0.90  971.12 1.01
## qc[1]     0.26    0.00 0.22 -0.16  0.12  0.26  0.41  0.70 1988.75 1.00
## qc[2]    -0.64    0.01 0.27 -1.17 -0.81 -0.64 -0.47 -0.11 1333.81 1.01
## qc[3]     0.87    0.01 0.31  0.26  0.67  0.87  1.07  1.48  829.36 1.01
## sd_RN[1]  0.55    0.00 0.02  0.51  0.54  0.55  0.56  0.59 4659.85 1.00
## sd_RN[2]  0.53    0.00 0.02  0.48  0.51  0.52  0.54  0.57 4283.61 1.00
## sd_RN[3]  0.53    0.00 0.04  0.44  0.50  0.53  0.56  0.62 3182.37 1.00
```

The extracted posterior samples can also be manually plotted and summarized using base R functions. For example, we can look at the population average residual variance or SD of z on the original data scale by manually applying the inverse link function `exp()` to the log-scale SD `theta_0`.

```
par(mfrow=c(1,2))
#discrete approximation of posterior dist
hist(exp(samples$sigma_0)) #residual var
hist(sqrt(exp(samples$sigma_0))) #residual SD
```



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 of 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/negative linear and nonlinear selection effects, we simply need to calculate the proportion of the MCMC samples for these parameters with positive/negative magnitude, which approximates the area under the posterior distribution providing support for positive effects. Changing the `dimnames` attributes of the parameters to match how `b` and `q` were specified in the Stan model will ease interpretation.

```
#change dimnames
dimnames(samples$b)[[2]] = c("mu","beta","sigma")
dimnames(samples$q)[[2]] = c("mu_mu","beta_beta","sigma_sigma")
dimnames(samples$qc)[[2]] = c("mu_beta","mu_sigma","beta_sigma")

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

```
##        mu      beta     sigma
## 0.0000000 0.0030000 0.9998333
```

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

```
##       mu_mu   beta_beta sigma_sigma
## 0.001166667 0.966833333 0.956083333
```

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

```
##    mu_beta   mu_sigma beta_sigma
## 0.88575000 0.01116667 0.99675000
```

Note that $P(r > 0) = 1 - P(r < 0)$, so that we can always infer the probability of an effect in the opposite direction. For instance, the posterior probability of $b_1 > 0$ is 0, which means that the posterior probability of $b_1 < 0$ is 1. In other words, the model provides very strong support for negative selection on RN intercepts, consistent with the true selection effect used for simulating this data `stan.dl$true_b[1]`= -0.42. We can also extend the custom function to assist with this interpretation, by flipping the sign of the hypothesis test toward the direction with greater relative posterior probability.

```
pp.fun = function(x){
  y = sum(x>0) / length(x)
  if(y > 0.5){ paste0("P(+) = ",round(y,3)) }
  else{ paste0("P(-) = ",round(1-y,3)) }
  }

#for each column, calculate probability of positive effect
apply(samples$b, 2, FUN = pp.fun)
```

```
##             mu           beta          sigma
##     "P(-) = 1" "P(-) = 0.997"     "P(+) = 1"
```

```
apply(samples$q, 2, FUN = pp.fun)
```

```
##          mu_mu      beta_beta    sigma_sigma
## "P(-) = 0.999" "P(+) = 0.967" "P(+) = 0.956"
```

```
apply(samples$qc, 2, FUN = pp.fun)
```

```
##        mu_beta       mu_sigma     beta_sigma
## "P(+) = 0.886" "P(-) = 0.989" "P(+) = 0.997"
```

Overall, the model provides clear consistent support for the direction of linear and nonlinear selection effects across RN parameters, as indicated by the large posterior probabilities for either positive or negative selection across all RN parameters (note that PP(+) or PP(-) near 0.50 indicates complete uncertainty / weakest possible evidence for a +/- selection effect). Another way to think about these probabilities is in relation to Bayesian credible intervals (CIs). In particular, we expect that if there is at least 0.95 probability of a directional effect, the 90% Bayesian CI will exclude zero.

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

```
##
##                mu       beta     sigma
##    5%  -0.6795427 -0.4065146 0.2290906
##    95% -0.3936332 -0.1002295 0.6497718
```

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

```
##
##             mu_mu  beta_beta sigma_sigma
##    5%  -0.6297641 0.04356094  0.01321682
##    95% -0.1876975 0.68016174  0.79885930
```

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

```
##
##           mu_beta   mu_sigma beta_sigma
##    5%  -0.09474155 -1.0781684  0.3603278
##    95%  0.62602159 -0.2042479  1.3749717
```

It is important to emphasize that 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 0.89 and 0.96 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 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. only a 1/20 chance of an effect in the opposite direction (i.e. a posterior probability of 0.95). Therefore, the posterior probability of 0.89 for positive correlational selection on RN intercepts and slopes `q_c[1]` / `mu_beta` indicates that our data provides evidence for a positive selection effect, but also that this evidence is uncertain, with an approximately 1/10 chance of the effect actually being negative (1 - 0.89). This warrants cautious description and interpretation, neither overselling the strength of evidence for this effect nor stating that "no effect was found." Encouraging this Bayesian attitude toward evidence within evolutionary ecology is an important tool for promoting the goals of open science, as it helps to dampen issues such as file-drawer effects and reduce the risk of *P*-hacking. A continuous approach to statistical inference also encourages researchers to put greater emphasis on effect sizes, credible intervals, and additional metrics which can collectively increase or decrease the overall "significance" of an empirical finding (McShane et al. 2019).

Uncertainty for claims about the relative strength of selection on different RN parameters can also be directly quantified. For example, we may be interested in asking whether there is overall stronger directional selection on RN intercepts than on RN slopes.

```
median(abs(samples$b[,1]) - abs(samples$b[,2]) )
```

```
## [1] 0.2835862
```

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

```
## [1] 0.9766667
```

The model provides strong support for the absolute value of selection on RN intercepts being greater than on RN slopes ($P(|b_{\mu_0} > b_{\beta_x}|) = 0.98$), with an expected difference of 0.28.

If one is so inclined, robust null hypothesis tests can also be conducted within a Bayesian framework by specifying a range of biologically trivial effect sizes, which might thought of as "trivial hypothesis" tests rather than null hypothesis tests per se. For example, on a standardized scale such as a correlation coefficient, values $-0.10 < b < 0.10$ will generally be considered extremely small and not worthy of strong biological interpretation (explaining less than 1% of variation). Comparing these trivial hypothesis tests with directional hypothesis tests can provide distinct pieces of information, and will generally be more conservative than standard null hypothesis tests focuses solely on the assumption of the effect size being exactly 0. Consider the RN parameter correlation between RN intercepts and the RN slope and residual parameters.

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

#point estimate
median(R[,1,2]) #cor(mu_0, beta_x)
```

```
## [1] 0.09512301
```

```
median(R[,1,3]) #cor(mu_0, sigma_0)
```

```
## [1] -0.2132523
```

```
#directional hypothesis test
sum(R[,1,2]>0)/length(R[,1,2])
```

```
## [1] 0.9546667
```

```
sum(R[,1,3]<0)/length(R[,1,3])
```

```
## [1] 0.99475
```

```
#trivial hypothesis test
sum(-0.1< R[,1,2] & R[,1,2]  <0.1)/length(R[,1,3])
```

```
## [1] 0.5353333
```

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

```
## [1] 0.0905
```

The directional and trivial hypothesis tests work together to inform our understanding of the direction and magnitude of the estimated correlations. For $\text{cor}(\boldsymbol{\mu_0}, \boldsymbol{\beta_x})$, there is clear support for a positive correlation $P(\text{cor}(\boldsymbol{\mu_0}, \boldsymbol{\beta_x}) > 0) = 0.95$. However, consistent with the median estimate of `0.10`. the trivial hypothesis test shows that there is little to not support for this positive correlation being of biologically meaningful magnitude $P(0.10 > \text{cor}(\boldsymbol{\mu_0}, \boldsymbol{\beta_x}) > -0.10) = 0.53$. In other words, while there is a 0.95 probability that a positive correlation exists between RN intercepts and slopes, there is only a 0.47 probability that this correlation is of greater magnitude than $|0.1|$ / captures more than 1% of the variation observed in intercepts and slopes. These parameters are, therefore, effectively independent, despite our large sample size allowing us to detect a non-zero correlation with a low degree of uncertainty. For $\text{cor}(\boldsymbol{\mu_0}, \boldsymbol{\sigma_0})$, there is clear evidence of a negative correlation $P(\text{cor}(\boldsymbol{\mu_0}, \boldsymbol{\sigma_0}) < 0) = 0.99$ and for this correlation being of non-trivial effect size $P(0.10 > \text{cor}(\boldsymbol{\mu_0}, \boldsymbol{\sigma_0}) > -0.10) = 0.09$. In other words, there is a 0.99 probability of a negative correlation between RN intercepts and slopes, and a 0.91 probability that this value is larger than $|0.1|$.

Calculate selection gradients

We now want to transform our posteriors of the `b`, `q`, and `qc` selection effects into appropriately scaled directional $\boldsymbol{\beta}$ and quadratic $\boldsymbol{\gamma}$ selection gradients. As described in **Eq 6.1** of the main text, these values can always be manually calculated using partial derivative functions in R. Fortunately for the Gaussian case, things are much simpler and only require us to perform a few straightforward arithmetic operations. In particular, because we did not mean-scale fitness prior to the analysis, we need to divide all the selection coefficients through by the population fitness intercept `W_0`. We'll then need to double the quadratic gradients in `q` (proportional to the diagonals of the γ matrix), as is explained in further detail by Stinchcombe et al. (2008). Importantly, we do these calculations over the entire posterior distributions of the selection effects, rather than on point estimates of the posteriors, to ensure that statistical uncertainty is pooled across stages of the analysis (Stinchcombe, Simonsen, and Blows 2014).

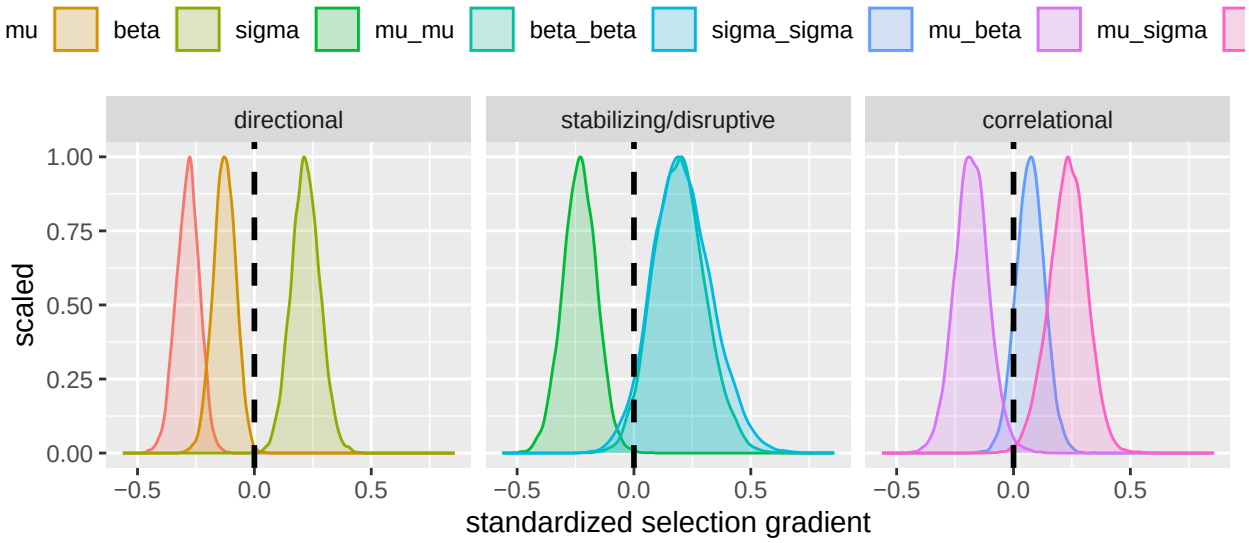
```
betas = apply(samples$b, 2, function(x) x / samples$W_0) #directional
gamma_diag = apply(samples$q, 2, function(x) (x / samples$W_0) * 2) #stabilizing/disruptive
gamma_cor = apply(samples$qc, 2, function(x) x / samples$W_0) #correlational
```

For effect size comparison, the gradients can also be standardized following **Eq. 6.2** in the main text.

```
betas_std = betas * samples$sd_RN
gamma_diag_std = gamma_diag * samples$V_P
gamma_cor_std = gamma_cor * (samples$sd_RN[,c(1,1,2)] * samples$sd_RN[,c(2,3,3)])
```

We can plot these results in `ggplot2` for a quick comparison of standardized effect sizes.

```
#combine wide to long format
std_gradients = rbind(melt(betas_std), melt(gamma_diag_std), melt(gamma_cor_std))
#index different gradients
std_gradients$type = factor(ifelse(std_gradients$Var2 %in% c("mu","beta","sigma"), "directional",
                            ifelse(std_gradients$Var2 %in% c("mu_mu","beta_beta","sigma_sigma"),
                                   "stabilizing/disruptive", "correlational")),
                            levels = c("directional","stabilizing/disruptive","correlational"))

library(ggplot2)
ggplot(std_gradients, aes(x = value, group = Var2, fill = Var2, color = Var2))+
  geom_density(aes(y = after_stat(scaled)), alpha = 0.2)+
  geom_vline(xintercept = 0, linetype = "dashed", linewidth = 1)+
  facet_wrap(.~ type)+
  labs(x = "standardized selection gradient")+
  theme(legend.position = "top", legend.title=element_blank())+
  guides(fill = guide_legend(nrow = 1))
```



The same basic approach used above for summarizing and conducting hypothesis tests on the `b` and `q` coefficients can be used for quantifying uncertainty and reporting on these β and γ posteriors.

Forthcoming tutorials

Further tutorials are currently under construction covering the estimation and calculation of selection gradients for non-Gaussian models, for coding the more complex nonlinear RNs and fitness functions discussed in the supplementary appendix of the manuscript, as well as for using the `brms` package to construct and subsequently edit a basic model for nonlinear selection analysis. In the meantime, please write me at jordan.martin@uzh.ch if you have a questions or would like assistance with coding such scenarios.

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.

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.

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.