The asp21bridge Package

1 The asp21bridge Package

This chapter introduces numerous facets of the `asp21bridge` package.

Section 1.1 explains how to use and combine the functions that are contained in the package most efficiently.

Section 1.2 focuses on the implementation of the Markov Chain Monte Carlo Sampler with Ridge Penalty, links the source code to the mathematical model from chapter **??** and explains how the main function `mcmc_ridge()` as well as internal functions implementing e.g. the Metropolis-Hastings algorithm are structured.

Finally, some additional components of the package development process and ideas, that the package is built upon, are discussed in section 1.3.

1.1 User Guide

This section aims to provide guidance for new users of the `asp21bridge` package. Although all exported functions are fully documented such that function arguments and brief examples can be looked up at the corresponding help page, the following tutorial far extends the documentation by illustrating a typical workflow of simulation via the penalized MCMC Sampler, extracting meaningful statistical quantities from the samples and visually analyzing the results.

The `asp21bridge` package inherits all functions from the `lmls` package and exports 9 additional functions, which can be grouped into three categories:

- **Sampling:**
  The whole sampling process is covered by the very flexible and robust `mcmc_ridge()` function that is explained in detail in section 1.2.1.
- **Graphical Analysis:**
  The `trace_plot()`, `density_plot()` and `acf_plot()` functions provide the three most common visualizations of a *single* chain's development over time, its distribution and the autocorrelation between the samples. Since all of these are *diagnostic* tools, they are combined in the high-level `diagnostic_plots()` function, which simply collects all three plots in a grid and is recommended to use for most applications.

  For visualizing *multiple* chains together, the `mult_plot()` function can be used. Several arguments for customizing the graphical output exist. In the most basic form, trace plots of all selected chains are displayed in the upper panel while the lower panel contains the corresponding density plots.
- **Statistical Analysis:**
  Here, the `summary_complete()` function represents the main tool and provides a more thorough statistical summary than the generic `summary()` function. In addition, the `burnin()` and `thinning()` functions are convenient in case the chains are highly autocorrelated.

1.1.1 Sampling with the `mcmc_ridge()` function

As explained in section 1.2.1, there are two valid input types for simulating with the `mcmc_ridge()` function. Most commonly, the sampling procedure will be built upon an existing model that was initialized by the `lmls()` function. In this case, only the name of the model object is required, although many further arguments can be specified such as the number of simulations `nsim`, which is set to 1000 by default.

The `mcmc_ridge()` function can, however, be used completely independent from the underlying `lmls` package. Therefore, the outcome vector $\mathbf{y}$ as well as the design matrices $\mathbf{X}$ and $\mathbf{Z}$, corresponding to the $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ coefficient vectors, respectively (as explained in chapter **??**), must be manually specified.

In order to illustrate both options, we construct two separate models:

- The first model uses the built-in `toy_data`, a data set which is specifically designed for tutorials, documentation and unit tests. It consists of a column `y` representing a vector of observed values and explanatory variables `x1`, `x2`, `z1` and `z2`.

  The data is simulated according to the correctly specified location-scale regression model from chapter **??**, where all explanatory variables predict the mean of $\mathbf{y}$ and only the latter two model the variance. This example will be used for model evaluation, since the true data generating values $\boldsymbol{\beta} = \begin{pmatrix} 0 & -2 & -1 & 1 & 2 \end{pmatrix}^T$ and $\boldsymbol{\gamma} = \begin{pmatrix} 0 & -1 & 1 \end{pmatrix}^T$ are known.

  For the simulations, we use the model object that is created by the `lmls()` function as input and use most of the `mcmc_ridge()` default settings, except for the number of simulations, which we increase to 10.000.

- The second model uses the more realistic `abdom` data set from the `lmls` package.

  In this case, we have to provide the data input as well as starting values for $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ explicitly. Note that the dimension of `beta_start` and `gamma_start` have to match the number of columns in `X` and `Z` in the model *input*. If necessary, the `mcmc_ridge()` functions adds intercept columns to both design matrices, such that the dimension of the coefficient vectors might have increased (as in the case illustrated here) in the model *output*.

  This example differs from the first one with regards to the conclusions that can be drawn: Since the true data generating values of $\boldsymbol{\beta} = \begin{pmatrix} \beta_0 & \beta_1 \end{pmatrix}^T$ and $\boldsymbol{\gamma} = \begin{pmatrix} \gamma_0 & \gamma_1 \end{pmatrix}^T$ are *not* known, we have to rely more heavily on the model estimates. The number of simulations is again increased to 10.000, such that statistically valid estimates of posterior characteristics are possible, even if preceding `burnin()` and `thinning()` steps might be required.

  Note, that we first standardize the covariates in this case, which is strongly recommended when working with penalized methods.

```
library(asp21bridge)
set.seed(1234)

toy_fit <- lmls(
  location = y ~ x1 + x2 + z1 + z2, scale = ~ z1 + z2,
  data = toy_data, light = FALSE
) %>%
  mcmc_ridge(num_sim = 10000)

standardize <- function(x) (x - mean(x)) / sd(x)
y <- abdom$y
X <- as.matrix(standardize(abdom$x))
Z <- X

abdom_fit <- mcmc_ridge(
  y = y, X = X, Z = Z, beta_start = 1, gamma_start = 1, num_sim = 10000
)
```

The different types of data input cause different structures in the resulting model objects: `toy_fit` inherits the model structure as well as the S3 Class `lmls` from the `lmls()` function and adds a list entry `mcmc_ridge` containing the sampling results (refer to section 1.2.1 for details).

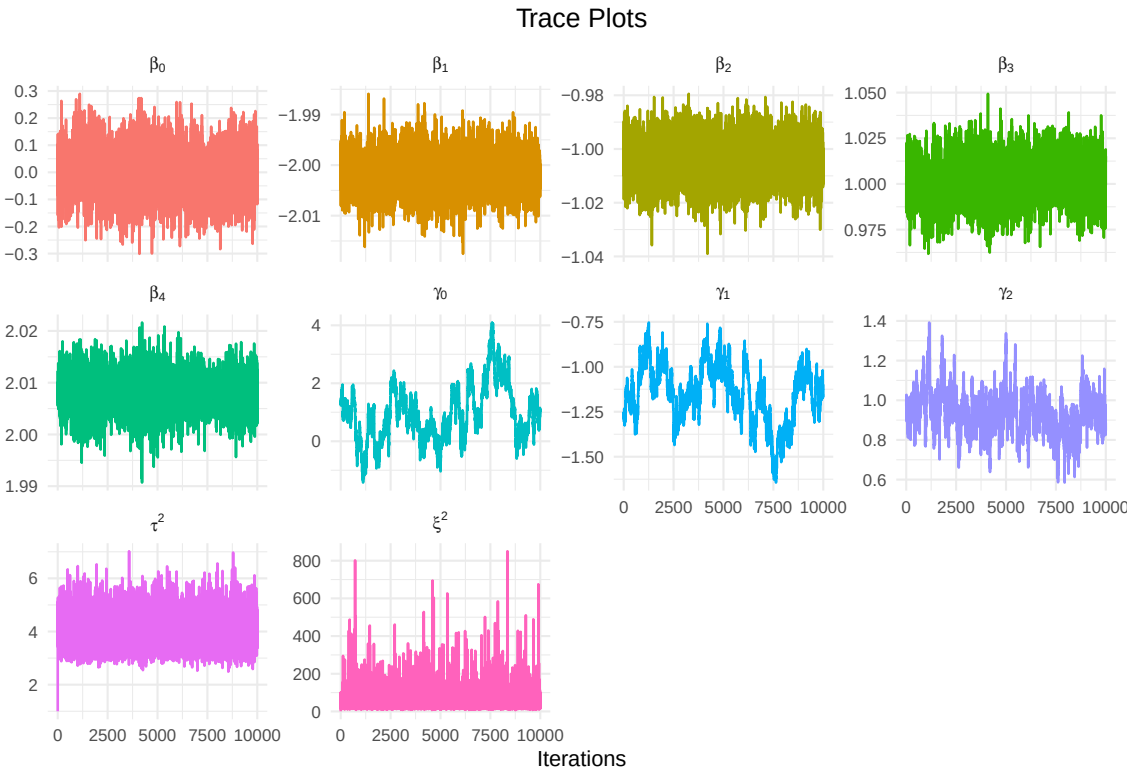

Figure 1: Trace Plots for all Coefficients, data: toy_data

In contrast, `abdom_fit` only contains matrices filled with the simulations and the acceptance probability of the Metropolis-Hastings step for sampling $\boldsymbol{\gamma}$:

```
str(abdom_fit, max.level = 1)
## List of 2
##  $ sampling_matrices:List of 4
##  $ acceptance_rate  : num 0.305
```

1.1.2 Graphical Analysis

Most statistical analyses start with an exploratory phase. To obtain a graphical overview of the simulations, the `mult_plot()` function is convenient to display trace plots and/or density plots for all model coefficients.

The `free_scale` argument is often useful to obtain a meaningful graphical output, if the parameters are on different numerical scales. Setting `latex = TRUE` transforms the coefficient names to their corresponding greek symbols, which, although being a purely aesthetic feature, required a surprisingly nontrivial implementation.

```
mult_plot(samples = toy_fit, type = "trace", free_scale = TRUE, latex = TRUE)
```

Figure 1 shows trace plots for all coefficients of the `toy_fit` model. Due to the high number of simulations, the plot facets appear a bit overloaded. Considering the y-axis scales, the samples for the $\boldsymbol{\beta}$ vector show a small variance and fast convergence, whereas the samples for γ_0 and γ_1 indicate a significant autocorrelation and no sign of convergence.

The `log` argument can be useful for displaying variance parameters such as τ^2 and ξ^2, which are strictly positive. Figure 2 illustrates the effect of this option on the y-axis of the trace plots as well as the x-axis of the density plots.

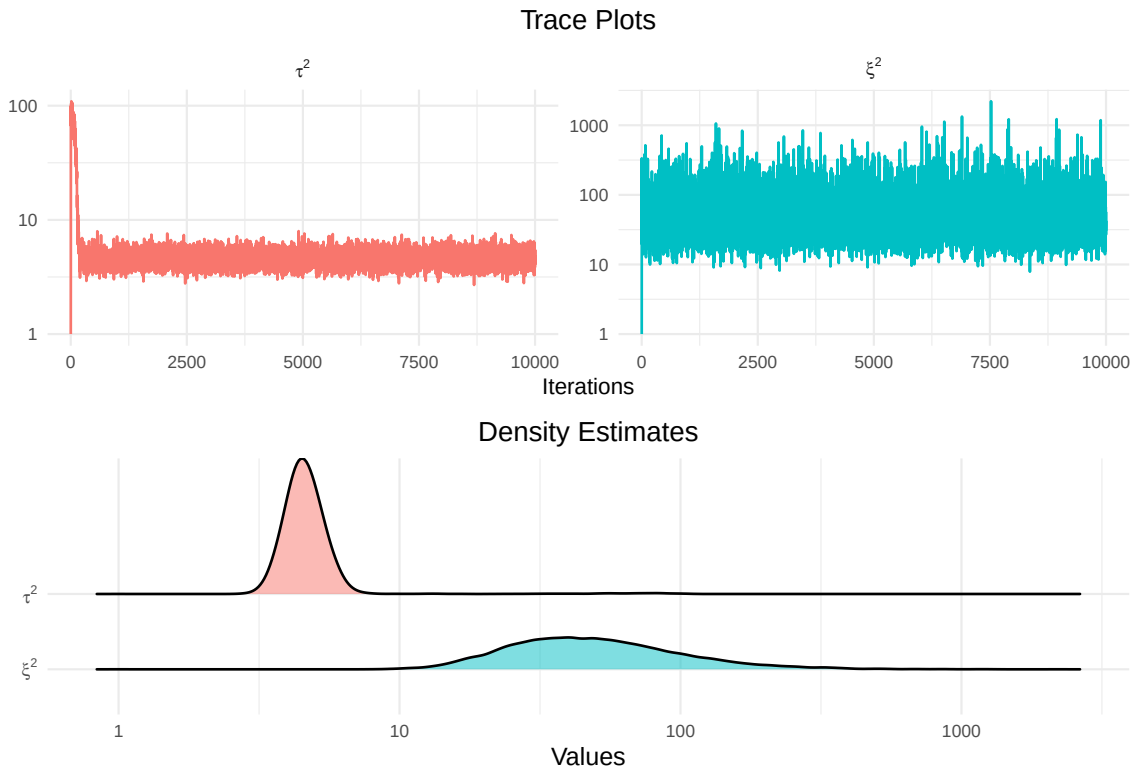

Figure 2: Trace and Density Plots for the Prior Variances, data: abdom

```
variance_samples <- cbind(
  abdom_fit$sampling_matrices$tau_samples,
  abdom_fit$sampling_matrices$xi_samples
)

mult_plot(
  samples = variance_samples, type = "both", free_scale = TRUE,
  log = TRUE, latex = TRUE
)
```

Note that the `mult_plot()` function accepts various kinds of input data, such as a complete `lmls` model in the first case or simply one or multiple sampling matrices in the latter case, and correctly extracts the required samples.

To focus on a single Markov chain, the `diagnostic_plots()` function is useful:

```
diagnostic_plots(
  samples = abdom_fit$sampling_matrices$beta_samples[, "beta_1", drop = FALSE],
  lag_max = 100, latex = TRUE
)
```

Figure 3 clearly shows the need for a `burnin()` and `thinning()` step for β_1, since the chain varies heavily among the first iterations and the samples are strongly autocorrelated. The `lag_max` argument controls the number of lags that are included in the autocorrelation plot.

In order to confront these issues, we remove the first 500 samples and additionally only keep every 10th iteration:

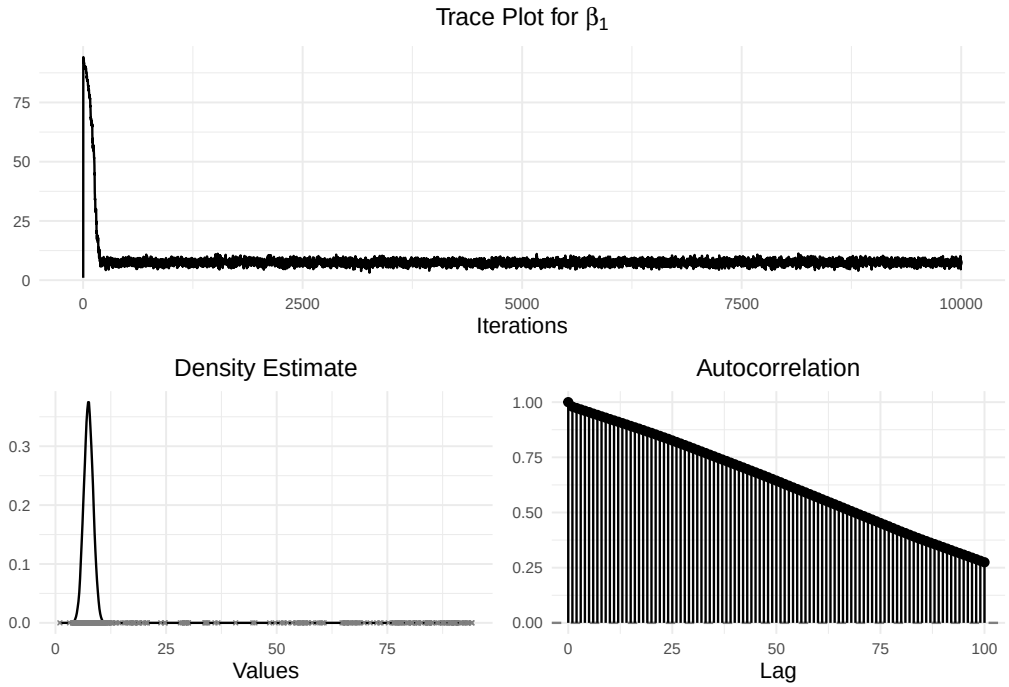

Figure 3: Diagnostic Plots for β_1 before thinning, data: abdom

```
abdom_fit$sampling_matrices$beta_samples[, "beta_1", drop = FALSE] %>%
  burnin(num_burn = 500) %>%
  thinning(freq = 10) %>%
  diagnostic_plots(lag_max = 100, latex = TRUE)
```

The diagnostic plots for β_1 in Figure 4 look much better. The chain has converged and there is little residual autocorrelation left after thinning out the samples. Additionally, the posterior density approximately follows a normal distribution.

Although removing posterior estimates in this way (and arguably losing valuable information) is not uncontroversial in the Bayesian community, statistical estimates from the remaining, well behaved chain are usually more stable and reliable. In any case, the above procedure clearly emphasizes the usefulness of a graphical diagnosis of the sampling results as a preliminary step before drawing any far reaching conclusions.

1.1.3 Statistical Analysis

At the end of a Bayesian statistical analysis, summary statistics of the posterior distribution are of major interest. These include estimates for *centrality*, such as the Posterior Mean or the Posterior Median, estimates for the *spread*, e.g. the Posterior Variance, and quantile estimates in the tails of the posterior distribution as bounds for credible intervals.

The `asp21bridge` package offers two options to quickly access this information: If only a quick look at the numerical results without further investigation is desired, the `summary()` function can be used. The `lmls` package adapts this generic S3 method to the `lmls` class with an additional `type` argument. Specifying `mcmc_ridge` displays the estimates of the `mcmc_ridge()` function, in this case the Posterior Mean, the Posterior Median and the 0.025 and 0.975 quantiles of the Posterior distribution.

Note, that this option is only applicable for the `toy_fit` model, which inherits the `lmls` class:

```
summary(toy_fit, type = "mcmc_ridge")
##
```

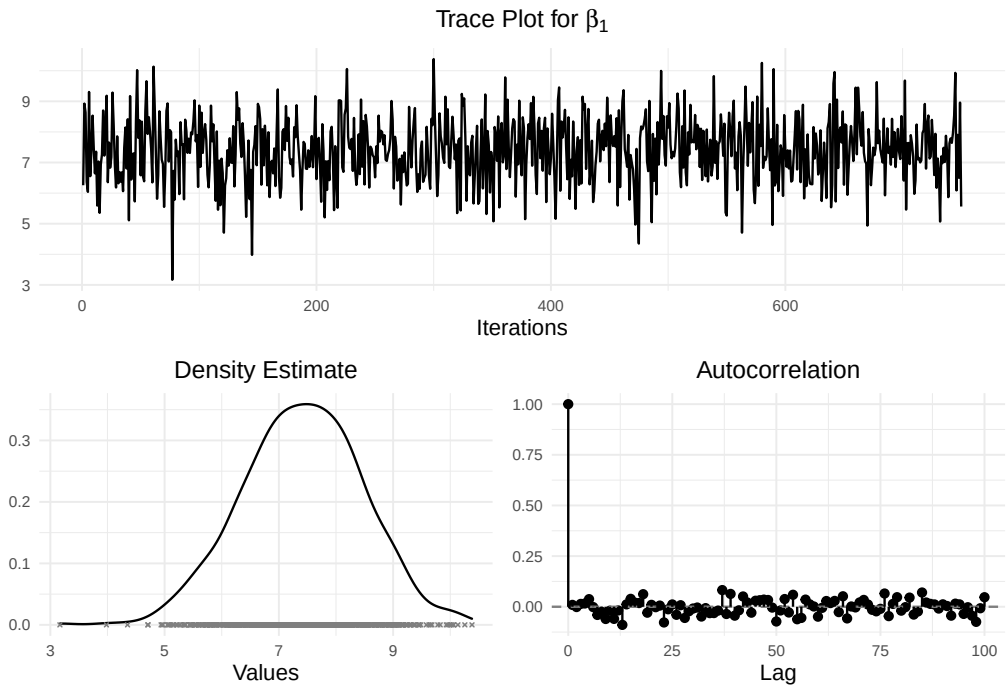

Figure 4: Diagnostic Plots for β_1 after thinning, data: abdom

```
## Call:
## lmls(location = y ~ x1 + x2 + z1 + z2, scale = ~z1 + z2, data = toy_data,
##     light = FALSE)
##
## Pearson residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -3.29800 -0.38660  0.11770 -0.01354  0.57410  2.54600
##
## Location coefficients (identity link function):
##              Mean       2.5%        50%  97.5%
## beta_0  0.0003161 -0.1488290 -0.0013337  0.156
## beta_1 -2.0015679 -2.0082464 -2.0015799 -1.995
## beta_2 -1.0044943 -1.0179012 -1.0044382 -0.991
## beta_3  1.0014607  0.9802516  1.0017075  1.022
## beta_4  2.0080638  2.0015876  2.0081264  2.014
##
## Scale coefficients (log link function):
##            Mean    2.5%     50%  97.5%
## gamma_0  0.9373 -0.7053  0.8277  3.074
## gamma_1 -1.1471 -1.5038 -1.1357 -0.862
## gamma_2  0.9325  0.7238  0.9305  1.176
##
## Residual degrees of freedom: 42
## Log-likelihood: 32.28
## AIC: -48.57
## BIC: -33.27
```

One downside of this approach is that the displayed values are not saved anywhere by default and, thus, cannot be immediately accessed. This issue is solved by the more informative `summary_complete()` function,

which conveniently saves all relevant quantities in a data frame, the central object for data analysis in R.

In the following section we focus on the `toy_data` model, since there the true coefficient values are known, and use the popular `dplyr` package for further data frame manipulations. We are primarily interested in Posterior Mean estimates for all coefficients, such that we only select a subset of the output columns and add a new column containing the true data generating values:

```
library(dplyr)

true_beta <- c(0, -2, -1, 1, 2)
true_gamma <- c(0, -1, 1)

summary_complete(samples = toy_fit) %>%
  filter(stringr::str_detect(Parameter, pattern = "beta|gamma")) %>%
  mutate(Truth = c(true_beta, true_gamma)) %>%
  select(Parameter, `Posterior Mean`, Truth, `Standard Deviation`)
## # A tibble: 8 x 4
##   Parameter `Posterior Mean` Truth `Standard Deviation`
##   <chr>                <dbl> <dbl>                <dbl>
## 1 beta_0            0.000316     0               0.0769
## 2 beta_1           -2.00        -2               0.00339
## 3 beta_2           -1.00        -1               0.00676
## 4 beta_3            1.00         1               0.0104
## 5 beta_4            2.01         2               0.00315
## 6 gamma_0           0.937        0               0.967
## 7 gamma_1          -1.15        -1               0.163
## 8 gamma_2           0.933        1               0.109
```

Except for γ_0, all coefficients are estimated very accurately with a slightly larger deviation from the true values for the remaining $\boldsymbol{\gamma}$ vector, which is sampled by the Metropolis-Hastings algorithm, compared to the $\boldsymbol{\beta}$ vector, which is drawn directly from a multivariate normal distribution.

Similar to the plotting functions introduced in section 1.1.2, the `summary_complete()` function is very robust with respect to its data input: Besides the whole model object as in the example above, just the `toy_fit$mcmc_ridge` list entry or even simply the sampling matrices `toy_fit$mcmc_ridge$sampling_matrices` are valid inputs, all leading to the same output.

Further, the function has the optional `include_plot` argument. If set to `TRUE`, one additional `Plot` column is added to the data frame, which leverages the `diagnostic_plots()` function with some default settings and, thus, contains diagnostic plot objects for all coefficients. This option comes in handy in an interactive workflow: Particularly interesting findings from the `summary_complete()` output can be quickly extracted from the data frame without interrupting the current thought process by using the `diagnostic_plots()` function separately.

In the example above, the `gamma_0` row shows a large posterior variance in addition to the large deviation of the Posterior Mean estimate from the true value. In order to investigate if the chain suffers from a lack of convergence or a significant autocorrelation, we could create the desired (yet here omitted) output with the following simple command:

```
summary_complete(toy_fit, include_plot = TRUE) %>%
  filter(Parameter == "gamma_0") %>%
  pull(Plot)
```

This introductory tutorial is not intended to validate the sampler's performance nor to interpret the obtained coefficient estimates, but rather to provide an overview of the various applications, where the `asp21bridge` package turns out to be useful. A more in depth analysis of the `mcmc_ridge()` results, also in comparison to alternatives like the `lmls()` and `mcmc()` functions of the underlying `lmls` package, can be found in chapter

?? after explaining some of the internal implementations in section 1.2.

1.2 Implementation of the Markov Chain Monte Carlo Sampler

Of all components that the `asp21bridge` package is built upon, the implementation of the `mcmc_ridge()` function is of special interest from a statistical perspective. Thus, section 1.2.1 explains both the overall structure of the `mcmc_ridge()` function and the Gibbs Sampler that is responsible for inducing the Ridge penalty. Section 1.2.2 is dedicated to the integrated Metropolis-Hastings step, which is used for sampling $\boldsymbol{\gamma}$ and, if desired, $\boldsymbol{\beta}$ as well.

1.2.1 Iterative Parameter Sampling

The `mcmc_ridge()` function implements a Markov chain Monte Carlo (MCMC) sampler that iteratively simulates from the full conditional distribution of each parameter of interest. Thus, the resulting values can be interpreted as samples from the joint Posterior Distribution (as described in chapter **??**), which in this case cannot be obtained analytically.

Overall Structure:

The function signature shows all mandatory as well as optional input parameters and their default values:

```
mcmc_ridge(m, X, Z, y, num_sim = 1000,
  beta_start = 1, gamma_start = 1, tau_start = 1, xi_start = 1,
  a_tau = 100, b_tau = 50, a_xi = 2, b_xi = 200,
  prop_var_scale = 3, mh_location = FALSE, prop_var_loc = 200
)
```

Many of the inputs have already been introduced in section 1.1.1. Explanations of each input can be found in the documentation, e.g. by calling `?mcmc_ridge()` after loading the `asp21bridge` package.

The `mcmc_ridge()` function is structured in a modular manner; it contains various internal helper functions collected in the `mcmc_ridge_helpers` file in order to maintain a concise and readable code structure.

Valid inputs can be provided by a `lmls` model object or separate design matrices $\mathbf{X}$ and $\mathbf{Z}$ and observation vector $\mathbf{y}$. In case both are given, the function prioritizes the additional manually specified inputs, such that for instance a single covariate can be added for the Bayesian sampler while using all remaining components from the fitted `lmls` model.

The `validate_input()` function handles these aspects and extracts all input parameters that are required for the sampling process. Further, it checks for missing input data and improper dimensions of the input parameters, in which case it returns an informative error message with one example given in section 1.3.2. Next, the `includes_intercept()` function checks both design matrices for intercept columns, while the `add_intercept()` function adds those for the sampling procedure, if necessary.

Now, that the raw input data is processed into a modified and convenient form, the Gibbs Sampler starts to iteratively simulate coefficient values. Since this part lies right at the heart of the `mcmc_ridge()` function, it is explained in more detail in the following subsection.

Whereas the simulation process itself is captured by a unified framework independent of the *input* type, the *output* type of the `mcmc_ridge()` function does depend on this distinction. If a `lmls` model is given as input, the sampling results are added to the existing model object in the new list entry `mcmc_ridge`. In this case, the sampling matrices for $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are assigned the names `location` and `scale`, respectively, which plays nicely into the existing code base of the `lmls` package and in particular for adjusting the `summary()` output to the `mcmc_ridge()` estimates.

In contrast, the output is structured in a list with the original names, if the inputs are specified manually without an existing model object. This conditional output type based upon the common sampling results is generated by the `create_output()` helper function.

Gibbs Sampler:

This paragraph explains the iterative Gibbs updates of the `mcmc_ridge()` function in a stepwise fashion. The specific form of the updates are based on the conditional distributions, derived in section **??**. The following code snippets intend to merely emphasize the most important steps from a conceptual perspective and are therefore not syntactically exact. If the specific components of each step are of interest, they can of course be examined in the `mcmc_ridge()` source code.

1. Initialize matrices for collecting all simulations with the `init_sampling_matrix()` helper function. Start a counter `acc_count_scale` for calculating the acceptance rate of the Metropolis-Hastings step for $\boldsymbol{\gamma}$.
2. Repeat steps 3 - 5 for $i = 2, \ldots,$ `num_sim`:
3. Update $\boldsymbol{\beta}^{(i)}$ by sampling from a closed form multivariate normal distribution (see section **??**), using values from the previous iteration for all remaining coefficients:

   ```
   beta_samples[i, ] <- rmvnorm(1, mu = beta_mean, chol_sig_inv = chol(beta_var_inv))
   ```

   This step can also be performed via an additional Metropolis-Hastings step analogous to sampling $\boldsymbol{\gamma}$ below.
4. Sample $\boldsymbol{\gamma}^{(i)}$ via an integrated Metropolis-Hastings step discussed in section 1.2.2 with the inputs $\boldsymbol{\beta}^{(i)}$, $\boldsymbol{\gamma}^{(i-1)}$, $(\tau^2)^{(i-1)}$ and $(\xi^2)^{(i-1)}$. Update the value of `acc_count_scale` if the proposed value for $\boldsymbol{\gamma}$ was accepted:

   ```
   gamma_list <- mh_gamma(
     beta_samples[i, ], gamma_samples[i - 1, ], xi_samples[i - 1, ], ...
   )
   gamma_samples[i, ] <- gamma_list$gamma
   acc_count_scale <- acc_count_scale + gamma_list$accepted
   ```
5. Update the values for τ^2 and ξ^2 with samples from Inverse-Gamma distributions (refer to sections **??** and **??**), using values $\boldsymbol{\beta}^{(i)}$ and $\boldsymbol{\gamma}^{(i)}$ from the current iteration:

   ```
   tau_samples[i, ] <- invgamma::rinvgamma(...)
   xi_samples[i, ] <- invgamma::rinvgamma(...)
   ```

1.2.2 Metropolis Hastings Step

As the Full Conditional of $\boldsymbol{\gamma}$ cannot be assigned to a known closed form distribution (see section **??**), the MCMC update for $\boldsymbol{\gamma}$ cannot be sampled directly, but requires an Metropolis Hastings step.
This step is implemented by the `mh_gamma()` function.

Recall from chapter **??** that the Full Conditional of $\boldsymbol{\gamma}$ is given by

$$f(\boldsymbol{\gamma} \mid \cdot) \propto \exp\left(-\frac{1}{2} \cdot \left[\frac{1}{\xi^2}\tilde{\boldsymbol{\gamma}}^T\tilde{\boldsymbol{\gamma}} + 2 \cdot \mathbf{1}_n^T \mathbf{Z}\boldsymbol{\gamma} + \sum_{i=1}^{n}\left(\frac{y_i - \mathbf{x}_i^T\boldsymbol{\beta}}{\exp\left(\mathbf{z}_i^T\boldsymbol{\gamma}\right)}\right)^2\right]\right).$$

For numerical reasons, we used the Log - Full Conditionals for the Metropolis Hastings implementation. Since some parts of the expression above are used for sampling $\boldsymbol{\beta}$ as well, we computed the relevant terms only once in the `mcmc_ridge()` function and handed them as input to `mh_gamma()`. Similarly, the current state of the Markov Chains for $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, required for evaluating the Full Conditionals, are provided by the `mcmc_ridge()` function in each iteration.

The implementation of the Metropolis-Hastings algorithm can be summarized by the following steps:

1. Compute the covariance matrix for the proposal distribution

   ```
   Sigma <- diag(1 / colMeans(Z^2)) * prop_var_scale / (n * length(gamma))
   ```

and sample a proposed value for $\boldsymbol{\gamma}$ as

$$\boldsymbol{\gamma}_{prop} \sim \mathcal{N}(\boldsymbol{\gamma}_{curr},\ \boldsymbol{\Sigma})$$

with $\boldsymbol{\gamma}_{curr}$ being the current state of $\boldsymbol{\gamma}$ in the Markov chain.

Note, that the term `Z^2` squares all elements of `Z` elementwise and `n` denotes the length of the observation vector $\boldsymbol{y}$. Furthermore, the `colMeans()` function returns a numeric vector with the column means of a matrix.

2. Compute the Log - Full Conditional distribution of $\boldsymbol{\gamma}$ at the states $\boldsymbol{\gamma}_{curr}$ and $\boldsymbol{\gamma}_{prop}$.

3. Since we used a multivariate normal random walk proposal, the acceptance probability α for the proposed value can be computed by evaluation of the Full Conditionals only. Thus, we obtain

```
log_acceptance_prob <- min(0, log_full_cond_proposal - log_full_cond_curr)
```

The log - acceptance probability is restricted to be smaller or equal to zero to ensure that the acceptance probability does not exceed the value of 1. In order to compare the acceptance probability α with a standard uniform random variable $U \sim \mathcal{U}(0, 1)$, the exponential function is used for transforming α to the original scale:

```
acceptance_prob <- exp(log_acceptance_prob)
```

4. Finally, the magnitude of the acceptance probability α relative to the realized value of U controls the new state of the chain. In addition, we kept track of the overall acceptance *rate* across all proposals by introducing boolean variables `accepted` in each iteration.

```
if (stats::runif(1) < acceptance_prob) {
  new_gamma <- gamma_proposal
  accepted <- TRUE
} else {
  new_gamma <- gamma
  accepted <- FALSE
}
```

The covariance matrix $\boldsymbol{\Sigma}$ of the proposal distribution in step 1 was chosen by simulating various combinations of the remaining model parameters to keep the acceptance rates in recommended ranges for a random walk proposal. The parameter `prop_var_scale` as a scaling factor of $\boldsymbol{\Sigma}$ has major impact on the acceptance rates and is included as an input parameter in the `mcmc_ridge()` function to provide manual control for the user.

The default value `prop_var_scale` $= 3$ was obtained from the same simulations and serves as a solid benchmark that works well in many scenarios. Larger values lead to larger jumps from the current to the proposed state, such that the acceptance probability decreases and the Markov Chain for $\boldsymbol{\gamma}$ might stay at the same point for a long time. By similar logic, smaller values of `prop_var_scale` lead to larger acceptance probabilities. Since the resulting steps will be small, the chain might converge to the Posterior Distribution very slowly and not explore the Posterior space sufficiently. Therefore both extremes will result in strongly correlated samples.

The Metropolis Hastings algorithm for $\boldsymbol{\beta}$, implemented by the `mh_beta()` function, works in a similar way but is of minor importance, since the $\boldsymbol{\beta}$ updates can also be sampled directly from a multivariate normal distribution.

1.3 Package Development

This section is dedicated to more niche aspects that come along with the package development process. Part 1.3.1 focuses on more formal components of the package, whereas some of the ideas behind the `asp21bridge` functions are discussed in section 1.3.2.

1.3.1 Code Coverage

Arguably the most important component of a package for new users is the **documentation**. All exported functions of the `asp21bridge` package as well as the `toy_data` data set are fully documented according to common standards for `R` packages.

In particular, we put effort into informative, yet not excessively long descriptions of the overall function and their parameters with highlighted default settings. The help pages are designed in a consistent manner across all functions. The `examples` section usually starts with the most basic applications signaling the function's *intent* and proceeds with more complex use cases that cover many of the function's optional arguments.

A more elaborate, publicly accessible tutorial similar to section 1.1 of this report can be found in the `README` file or at the front page of the `asp21bridge` GitLab website.

A second major aspect, that contributes to the quality of a package, are **unit tests**. Up to this point, a total of 251 unit tests have been written for the `lmls` and the `asp21bridge` packages combined. The number of implemented tests alone is, of course, not a meaningful metric, since the tests might be highly redundant and still capture only a small fraction of the entire package functionality. Thus, we prioritized both *width* and *depth* of the test coverage.

More specifically, unit tests are written for every single exported function with a larger focus on the more complex and more central ones such as the main `mcmc_ridge()` function.

Further, not only simple input checks (which are important nonetheless!), but also more esoteric edge cases and especially very common use cases of combining inputs and outputs of multiple `asp21bridge` functions are covered. Whenever discovering and fixing an unexpected error during the development phase, we implemented corresponding unit tests, such that this specific error will not reoccur in the future. Tests are continuously written and updated to ensure a stable and enjoyable user experience.

At a larger scope, the `R CMD CHECK`, or equivalently the `devtools::check()` command, produces 0 errors, 0 warnings and 0 messages. Besides working unit tests, this includes correctly specified `DESCRIPTION` and `NAMESPACE` files for all imported and exported functions as well as functioning examples in the documentation. The master branch of the `asp21bridge` project will maintain this standard in the foreseeable future; possible new features that could involve breaking changes will first be implemented on separate Git branches and at a later stage merged into the master branch after a thorough testing procedure.

1.3.2 Design Choices

Throughout the development process, we tried to adhere to some 'best practices' for general software development. These include several different aspects:

Default Settings:

For many `asp21bridge` functions, the user has to specify very few inputs manually since many input options are set to sensible default arguments. These inputs are chosen in a *neutral* way, such that the output aligns as best as possible with the user's expectation.

As an example, the default starting values for $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$ are the Maximum Likelihood estimates from the `lmls()` function, if a model object is provided as input to the `mcmc_ridge()` function. This serves as guidance for users with little prior knowledge about their model. It is worth noting, however, that there are multiple input parameters which *can* be set manually if desired, enabling fine control over the sampling procedure.

Moreover, the default values for the hyperparameters a_τ, b_τ, a_ξ and b_ξ are set according to the results of our conducted simulation studies in sections **??** and **??**, which correspond to a conservative penalty effect and, thus, a moderate induced bias.

Input Flexibility / Defensive Programming:

As already mentioned in section 1.1, all exported functions are designed to be as flexible as possible with respect to their input arguments. Thus, many functions accept multiple data structures like (nested) lists,

data frames, matrices or even simple vectors as data input and internally transforms the input into the desired format.

These transformations, however, are only performed as long as the user's *intention* is clear, e.g. providing the samples as input to a plotting functions regardless of the exact data structure, where the samples are stored. Whenever there is ambiguity or the input is simply not valid / incomplete, we have put effort into writing informative error messages.

To illustrate this last point, assume that we had forgotten to provide the second design matrix `Z` as input to our second application example from section 1.1 using the `abdom` data set. This might happen quite frequently, especially if the user is not particularly familiar with the structure of location-scale regression models:

```
y <- abdom$y
X <- as.matrix(abdom$x)

abdom_fit <- mcmc_ridge(
  y = y, X = X, beta_start = 1, gamma_start = 1, num_sim = 10000
)
## Error in validate_input(m = m, X = X, Z = Z, y = y, beta_start = beta_start, :
## At least either all model matrices (X, Z, y) and coefficients
## (beta_start, gamma_start) or a model object (m) must be given.
```

From the error message, it is immediately obvious which part of the input is missing.

One further quite neat feature is implemented for the generic `summary()` function. Since the available options for the `type` argument are specific to the `lmls` class and therefore a potential reason for confusion, the error message provides some guidance in case of spelling mistakes, suggesting the closest valid input option:

```
summary(toy_fit, type = "mcmc-ridge")
## Error: `type` must be one of "ml", "boot", "mcmc", or "mcmc_ridge".
## Did you mean "mcmc_ridge"?
```

Modularity:

The `lmls` package served as a fantastic example how to design and compose functions in a modular way, leading to independent applications of single components in various contexts, easier debugging and arguably better code transparency and readability.

The `asp21bridge` package similarly splits large functions into multiple pieces according to the well known premise, that functions should be doing one thing only, but one thing well. Therefore the `mcmc_ridge_helpers` file contains various helper functions for the main `mcmc_ridge()` function, which perform tasks like input validation or inclusion of the Metropolis-Hastings step for the γ vector as explained in section 1.2.1.

However, there is certainly a trade off to keep in mind: Since naming functions and variables is notoriously challenging, excessive modularity with poor naming choices can hinder code comprehension by inducing a wrong mental model of the function's task. Moreover, it can make sense to keep related functions close together in a physical sense and avoid spreading them across multiple files across the project.

Coding Style:

Before starting our work on different parts of the code base, we agreed to a consistent coding style. This includes using only lowercase letters and underscores in function and variable names and a limit of 80 characters per line. Here, we mostly conformed to the recommendations from the tidyverse style guide by Hadley Wickham. Moreover, we leveraged the styler package for aesthetically pleasing code formatting.

To extend consistent naming schemes even further, we chose the same argument names for all exported functions whenever possible. For instance, the data input for all functions that expect the MCMC simulations is called `samples` and all shared arguments of the 5 plotting functions are assigned the same name and the

same functionality. Hence, it often suffices to be familiar with one function of a certain family, since that knowledge can be easily transferred to all related functions.

Finally, there are two design choices, that are very specific to the `asp21bridge` context:

Operation Chaining:

First, we aimed to allow for frictionless function composition via the popular `%>%` operator. As illustrated in section 1.1, this comes in particularly handy when including `burnin()` and `thinning()` operations in a larger pipeline without the need to define dummy variables for temporary objects:

```
lmls(
  location = y ~ x1 + x2 + z1 + z2, scale = ~ z1 + z2,
  data = toy_data, light = FALSE
) %>%
  mcmc_ridge(num_sim = 1000) %>%
  purrr::pluck("mcmc_ridge", "sampling_matrices") %>%
  burnin(num_burn = 100) %>%
  thinning(freq = 5) %>%
  mult_plot(type = "both", free_scale = TRUE, latex = TRUE)
```

This workflow proved to be so useful, that we provided access to the pipe operator directly by loading the `asp21bridge` package, i.e. we reexported `%>%` from the `magrittr` package. Alternatively, one could have used the native `|>` pipe introduced in `R 4.1`. However, since the release of this `R` version was so recent, it might induce a stronger dependency than just relying on the `%>%` operator, which is ubiquitous at least in the `R` community that is related to the data science field.

There are two conditions that functions must fulfill to chain multiple operations:

- The first argument must be chosen uniformly for all (except the first) involved commands. In case of the `asp21bridge` package this corresponds to the `samples` argument at the first position.
- Functions that serve as intermediary steps inside a chain should return the same object type as their input. If a `lmls` model object is provided, the `mcmc_ridge()` function returns a modified (copy of the same) `lmls` model object. Even more flexible are the `burnin()` and `thinning()` functions: They can take a list of matrices, a single matrix or a numeric vector as input and always return the same provided input data structure.

Object-Oriented Programming:

Secondly, we thought about initializing our own S3 class when calling the `mcmc_ridge()` function, but finally decided against it for two reasons: We always considered the `asp21bridge` package as a strict extension to the `lmls` package. Thus, we did not want to make the underlying `lmls()` results less accessible after using `mcmc_ridge()`, i.e. generic functions like `coef()`, `plot()` or `print()` should in our mind create the same output before and after extending the original model.

One exception is the `summary()` function, as we have already mentioned. Here, we added one more option to the `type` argument, which has to be explicitly specified to access information from the `boot()` and `mcmc()` function of the `lmls` package as well. Apart from that, we could not see much additional value in implementing wrapper functions, which adapt e.g. the `print()` output to the penalized MCMC results, beyond the already existing tools that were introduced in section 1.1.