BMM Developer Notes

Table of contents

Overview

Last update: 13.08.2024

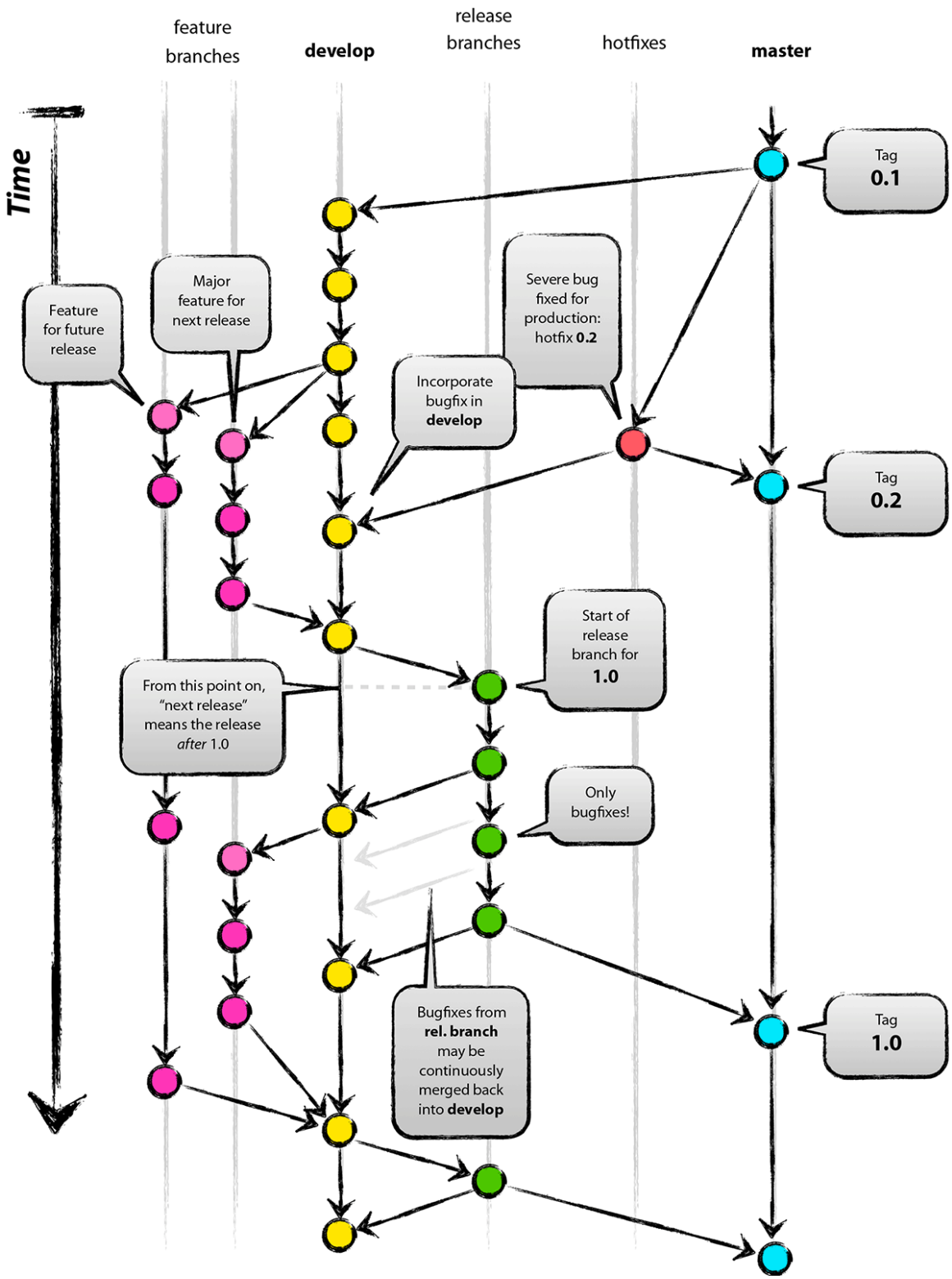
This guide aims to help developers contribute new models to bmm. It is a work in progress and will be updated as the package evolves. It explains how to set-up your system for package development, the structure of the package, and the workflow for contributing new models to the package.

The current guide is up to date with **bmm v1.0.1** and it might not yet reflect changes implemented afterwards. If you run into problems, don' 't hesitate to open an issue on github.

We follow a github flow workflow. The repository contains two main branches:

- Master (contains the latest released stable version of the `bmm` package)
- Develop (contains the latest stable development branch)

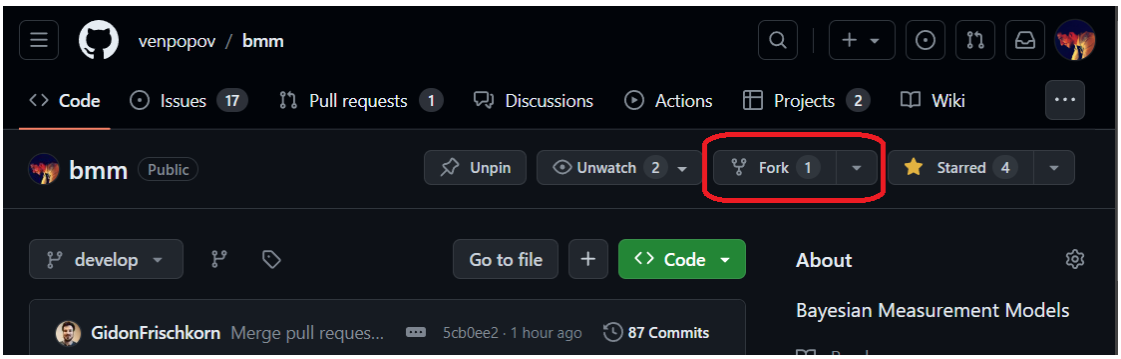
All new feature development should occur on an independent branch from Develop. If you want to contribute a new model to the `bmm` package, you need to fork the repository, create a new branch for your model, extensively test the model, and eventually submit a pull request for your changes to be merged into the Develop branch of the main repository. Your changes will be reviewed by someone from the core team. Once your changes are merged into the Develop branch, they will be included in the next release of the package.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Feature for future release
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

1 System setup

1.1 Setting up a local fork of the bmm repository

1. Fork the bmm github repository. This will create a copy of the current development branch into your own github account



2. Clone your fork to your local machine
3. Create a new branch for your model, typically named `feature/name-of-my-model`

1.2 Package development via RStudio and devtools

The bmm package is setup as an RStudio project. Opening the `bmm.Rproj` file will open a new RStudio instance, which facilitates package development with a few commands from the `devtools` package. A great tutorial on package development can be found here. Below is a summary of the most important steps

1. Make sure you have the devtools package and a few others installed and loaded

```
install.packages(c("devtools", "roxygen2", "testthat", "knitr"))
library(devtools)
install_dev_deps()
```

   To avoid having to load the **devtools** package every time, you can add the following code to your `.Rprofile` file

```
if (interactive()) {
  suppressMessages(require(devtools))
}
```

As noted here, you can create and open an `.Rprofile` file, if you don't already have one with

```
use_devtools()
```

2. Load the current version of the `bmm` package based on your local files

   ```
   load_all()  # or ctrl+shift+L
   ```

   you can use this command whenever you make changes to the package code to see the changes in action. You should not call `library(bmm)` or source the files manually, as this will load the installed version of the package, not the one you are developing.

3. Make any changes to the package code that you need to make (elaborated in the next section)

4. Use `check()` to check the package for errors and warnings

   ```
   check()
   ```

   you should always ensure that check() produces no errors before submitting a pull request

5. Use `document()` to update the documentation

   ```
   document()
   ```

2 Git and Github workflow

(This is not meant to be a complete guide to git, but rather a short summary of key commands)

After your fork the `bmm` repository and create a branch for your new model/feature, you can follow a typical git workflow. As you make changes and add new files, you will want to:

2.1 Git commands

Add your changes to git, commit them and then push your changes to your forked repository. You can run these commands from a terminal or from the RStudio terminal (with the project working directory)

Add all changed files to the staging area:

```
git add *
```

Commit the changes to the local repository

```
git commit -m "A short message describing the changes you made"
```

Push the changes to your forked repository

```
git push
```

You can (and should) repeat this process as many times as you need to before submitting a pull request. This will allow you to make many small changes and test them before submitting a pull request. Ideally each commit should be a small, self-contained change that can be easily reviewed.

2.2 Pull requests

When you are ready you can open a pull request from your forked repository to the main bmm repository. You can do this from the github website. Make sure to select the Develop branch as the base branch and your feature branch as the compare branch. You should add a detailed description of your changes, including the motivation for the changes and any relevant context. You should also mention any issues that your pull request resolves.

3 BMM code structure

Adding a new model is straightforward using the `use_model_template()` function, which will be described in the next section. You do not have to edit any of the files below, but it will be helpful to understand the structure of the package.

3.1 The main workhorse - `bmm()`

The main function for fitting models is `bmm()`. This function is the main entry point for users to fit models. It is set-up to be independent of the specific models that are implemented in the package.

```
bmm <- function(formula, data, model,
                prior = NULL,
                sort_data = getOption('bmm.sort_data', "check"),
                silent = getOption('bmm.silent', 1),
                backend = getOption('brms.backend', NULL),
                file = NULL, file_compress = TRUE,
                file_refit = getOption('bmm.file_refit', FALSE), ...) {
  deprecated_args(...)
  dots <- list(...)

  x <- read_bmmfit(file, file_refit)
  if (!is.null(x)) return(x)

  # set temporary global options and return modified arguments for brms
  configure_opts <- nlist(sort_data, silent, backend, parallel = dots$parallel,
                          cores = dots$cores)
  opts <- configure_options(configure_opts)
  dots$parallel <- NULL

  # check model, formula and data, and transform data if necessary
  user_formula <- formula
  model <- check_model(model, data, formula)
  data <- check_data(model, data, formula)
```

```
  formula <- check_formula(model, data, formula)

  # generate the model specification to pass to brms later
  config_args <- configure_model(model, data, formula)

  # configure the default prior and combine with user-specified prior
  prior <- configure_prior(model, data, config_args$formula, prior)

  # estimate the model
  fit_args <- combine_args(nlist(config_args, opts, dots, prior))
  fit <- call_brm(fit_args)

  # model post-processing
  fit <- postprocess_brm(model, fit, fit_args = fit_args, user_formula = user_formula,
                         configure_opts = configure_opts)

  # save the fitted model object if !is.null
  save_bmmfit(fit, file, compress = file_compress)
}
```

It calls several subroutines, implemented as generic S3 methods, to:

- `configure_options()` - to configure local options for fitting, such as parallel sampling,
- `check_model()` - check if the model exists
- `check_formula()` - check if the formula is specified correctly and transform it to a brmsformula
- `check_data()` - check whether the data contains all necessary information
- `configure_model()` - configures the model called for fitting
- `configure_prior()` - sets the default priors for the model and combines them with the user prior
- `call_brm()` - fit the model using the `brm()` function from the `brms` package
- `postprocess_brm()` - to post-process the fitted model

In addition, it also tests if the specified `bmmodel` has already been estimated and saved to a `file`. This is done via the `read_bmmfit` and `save_bmmfit` functions.

3.2 Models

All models in the package are defined as S3 classes and follow a strict template. This allows us to implement general methods for handling model fitting, data checking, and post-processing.

Each model has an internal function that defines the model and its parameters, and a user-facing alias. For a complete example model file and an explanation, see Section 4. The general model template looks like this:

```
.model_my_new_model <- function(resp_var1 = NULL, required_args1 = NULL,
                                required_arg2 = NULL, links = NULL, version = NULL,
                                call = NULL, ...) {
  out <- structure(
    list(
      resp_vars = nlist(resp_error),
      other_vars = nlist(),
      domain = "",
      task = "",
      name = "",
      version = "",
      citation = "",
      requirements = "",
      parameters = list(),
      links = list(),
      fixed_parameters = list(),
      default_priors = list(),
      version = version,
      void_mu = FALSE
    ),
    class = c("bmmodel", "my_new_model"),
    call = call
  )
  out$links[names(links)] <- links
  out
}
```

Each model is accompanied by a user-facing alias, the documentation of which is generated automatically based on the info list in the model definition.

```
# user facing alias
# information in the title and details sections will be filled in
# automatically based on the information in the .model_modelname()$info
#' @title `r .model_my_new_model()name`
#' @name Model Name#' @details `r model_info(.model_my_new_model())`
#' @param resp_var1 A description of the response variable
#' @param required_arg1 A description of the required argument
#' @param required_arg2 A description of the required argument
```

```
#' @param ... used internally for testing, ignore it
#' @return An object of class `bmmmodel`
#' @export
#' @examples
#' \dontrun{
#' # put a full example here (see 'R/bmm_model_mixture3p.R' for an example)
#' }
my_new_model <- function(resp_var1, required_arg1, required_arg2,
                         links = NULL, version = NULL, ...) {
  call <- match.call()
  stop_missing_args()
  .model_my_new_model(resp_var1 = resp_var1, required_arg1 = required_arg1,
                      required_arg2 = required_arg2, links = links, version = version,
                      call = call, ...)
}
```

Then users can fit the model using the `bmm()` function, and the model will be automatically recognized and handled by the package:

```
fit <- bmm(formula = my_bmmformula,
           data = my_data,
           model = my_new_model(resp_var1, required_arg1, required_arg2))
```

3.3 S3 methods

The package uses S3 methods to handle different models. This means that the same function can behave differently depending on the class of the object it is called with. For example, the `configure_model(model)` function called by `fit_model()`, is generally defined as:

```
configure_model <- function(model) {
   UseMethod('configure_model')
}
```

and it will call a function `configure_model.modelname()` that is specified for each model. The same is true for other functions, such as `check_data()`, `postprocess_brm()`, and `check_formula()`. This allows us to add new models without having to edit the main fitting function, `bmm()`.

3.4 File organization

The `bmm` package is organized into several files. The main files are:

`R/bmm.R`

It contains the main function for fitting models, `bmm()`. This function is the main entry point for users to fit models. It is set-up to be independent of the specific models that are implemented in the package.

To add new models, you do not have to edit this file. The functions above are generic S3 methods, and they will automatically recognize new models if you add appropriate methods for them (see section Adding new models).

`R/helpers-*.R`

`R/helpers-data.R`, `R/helpers-parameters.R`, `R/helpers-postprocess.R`, `R/helpers-model.R`, and `R/helpers-prior.R`

These files define the main generic S3 methods for checking data, post-processing the fitted model, configuring the model, checking the model formula, and combining priors. They contain the default methods for these functions, which are called by `bmm()` if no specific method is defined for a model. If you want to add a new model, you will need to add specific methods for these functions for your model. *You do not need to edit these files to add a new model.*

`R/bmmformula.R`

This file contains the definition of the `bmmformula` class, which is used to represent the formula for the model. It contains the `bmmformula()` function and its alias `bmf()`, which is used to create a new formula object.

In addition, it contains the definition of the `bmf2bf` S3 method that is used to convert a `bmmformula` object into a `brms_formula`‘object. This is necessary, as `brmsformula` objects are required to include the response variable in the first formula line. In contrast `bmmformula` objects only contain formulas predicting the paramaeters of a `bmmmodel`. The `bmf2bf` S3 method is used to perform this conversion and add the first formula line and including the response variable in the `brmsformula` created during model configuration.

`R/model_*.R`

Each model and it's methods is defined in a separate file. For example, the 3-parameter mixture model is defined in `model_mixture3p.R`. This file contains the internal function that defines the model and its parameters, and the specific methods for the generic S3 functions. Your new model will exist in a file like this. The name of the file should be `model_name_of_your_model.R`. You don't have to add this file manually - see section Adding new models.

`R/distributions.R`

This file contains the definition of the custom distributions that are used in the package. It specifies the density, random number generation, and probability functions for the custom distributions. If your model requires a custom distribution, you will need to add it to this file. These are not used during model fitting, but can be used to generate data from the model, and to plot the model fit.

`R/utils.R`, `R/brms-misc.R`, `R/restructure.R`, `R/summary.R`, `R/update.R`

Various utility functions.

`inst/stan_chunks/`

This directory contains the Stan chunks that are passed to the `brms::stanvar()` function. These are used to define the custom distributions that are used in the package. If you add a new custom distribution, you will need to add a new Stan chunk to this directory. Each model has several files, one for each corresponding stanvar block.

4 Example model file

All models in the package are defined as S3 classes and follow a strict template. This allows us to implement general methods for handling model fitting, data checking, and post-processing. Each model has an internal function that defines the model and its parameters, and a user-facing alias. Let's look at how two models are implemented - the IMM model, which uses both general class and specific model methods, but no custom stan code, and the SDM model, which depends heavily on custom stan code. If you use the `use_model_template()` function, templates for all sections below will be automatically generated for your model.

4.1 The Interference Measurement Model (IMM)

The model is defined in the file `R/model_imm.R`. Let's go through the different parts.

4.1.1 Model definition

The full IMM model is defined in the following internal model class:

```
.model_imm <-
  function(resp_error = NULL, nt_features = NULL, nt_distances = NULL,
           set_size = NULL, regex = FALSE, version = "full", links = NULL,
           call = NULL, ...) {
    out <- structure(
      list(
        resp_vars = nlist(resp_error),
        other_vars = nlist(nt_features, nt_distances, set_size),
        domain = "Visual working memory",
        task = "Continuous reproduction",
        name = "Interference measurement model by Oberauer and Lin (2017).",
        version = version,
        citation = glue(
          "Oberauer, K., & Lin, H.Y. (2017). An interference model \\
          of visual working memory. Psychological Review, 124(1), 21-59"
        ),
        requirements = glue(
```

```
        '- The response vairable should be in radians and \\
        represent the angular error relative to the target
        - The non-target features should be in radians and be \\
        centered relative to the target'
      ),
      parameters = list(
        mu1 = glue(
          "Location parameter of the von Mises distribution for memory \\
          responses (in radians). Fixed internally to 0 by default."
        ),
        kappa = "Concentration parameter of the von Mises distribution",
        a = "General activation of memory items",
        c = "Context activation",
        s = "Spatial similarity gradient"
      ),
      links = list(
        mu1 = "tan_half",
        kappa = "log",
        a = "log",
        c = "log",
        s = "log"
      ),
      fixed_parameters = list(mu1 = 0, mu2 = 0, kappa2 = -100),
      default_priors = list(
        mu1 = list(main = "student_t(1, 0, 1)"),
        kappa = list(main = "normal(2, 1)", effects = "normal(0, 1)"),
        a = list(main = "normal(0, 1)", effects = "normal(0, 1)"),
        c = list(main = "normal(0, 1)", effects = "normal(0, 1)"),
        s = list(main = "normal(0, 1)", effects = "normal(0, 1)")
      ),
      void_mu = FALSE
    ),
    # attributes
    regex = regex,
    regex_vars = c('nt_features', 'nt_distances'),
    class = c("bmmodel", "circular", "non_targets", "imm", paste0('imm_',version)),
    call = call
  )

  # add version specific information
  if (version == "abc") {
    out$parameters$s <- NULL
```

```
    out$links$s <- NULL
    out$default_priors$s <- NULL
    attributes(out)$regex_vars <- c('nt_features')
  } else if (version == "bsc") {
    out$parameters$a <- NULL
    out$links$a <- NULL
    out$default_priors$a <- NULL
  }

  out$links[names(links)] <- links
  out
}
```

Here is a brief explanation of the different components of the model definition:

`resp_vars`: a list of response variables that the model will be fitted to. These variables will be used to construct the `brmsformula` passed to `brms` together with the `bmmformula` and the `parameters` of the model. The user has to provide these variables in the data frame that is passed to the `bmm()` function

`other_vars:` a list of additional variables that are required for the model. This is used to check if the data contains all necessary information for fitting the model. In the example above, the IMM model requires the names of the variables specifying the non-target features relative to the target, the variables specifying the distance of the non-targets to the target, and the set_size. The user has to provide these variables in the data frame that is passed to the `bmm()` function

`domain, task, name, citation, requirements`: contains information about the model, such as the domain, task, name, citation, requirements. This information is used for generating help pages

`version:` if the model has multiple versions, this argument is specified by the user. Then it is used to dynamically adjust some information in the model object. In the case of the `imm` model, we have three versions - `full`, `bsc` and `abc`. As you can see at the end of the script, some parameters are deleted depending on the model version.

`parameters:` contains a named list of all parameters in the model that can be estimated by the user and their description. This information is used internally to check if the `bmmformula` contains linear model formulas for all model parameters, and to decide what information to include in the summary of `bmmfit` objects.

`links:` a named list providing the link function for each parameter. For example, `kappa` in the `imm` models has to be positive, so it is sampled on the log scale. This information is used in defining the model family and for the summary methods. If you want the user to be able to

specify custom link functions, the next to last line of the script replaces the links with those provided by the user

`fixed_parameters` in the `imm` several parameters are fixed to constant values internally to identify the model. Only one of them, `mu1` is also part of the `parameters` block - this is the only fixed parameters that users can choose to estimate instead of leaving it fixed. `mu2` and `kappa2` cannot be freely estimated.

`default_priors` a list of lists for each parameter in the model. Each prior has two components: `main`, the prior that will be put on the Intercept or on each level of a factor if the intercept is suppressed; `effects`, the prior to put on the regression coefficients relative to the intercept. The priors are described as in the `set_prior` function from `brms`. This information is used by the `configure_prior()` S3 method to automatically set the default priors for the model. The priors that you put here will be used by `bmm()` unless the users chooses to overwrite them.

`void_mu:` For models using a custom family that do not contain a `location` or `mu` parameter, for example the diffusion model, we recommend setting up a `void_mu` parameter. This avoids arbitrarily using one of the model parameters as the `mu` parameter.

`regex:` For the `imm` models, the `nt_features` and `nt_distances` variables can be specified with regular expressions, if the user sets `regex = TRUE`

`call`: this automatically records how the model was called so that the call can be printed in the summary after fitting. Leave it as is.

`class`: is the most important part. It contains the class of the model. This is used by generic S3 methods to perform data checks and model configuration. The classes should be ordered from most general to most specific. A general class exists when the same operations can be performed on multiple models. For example, the '3p', 'imm_abc', 'imm_bsc' and 'imm_full' models all have non-targets and set_size arguments, so the same data checks can be performed on all of them, represented by the class `non_targets`. The first class should always be `bmmodel`, which is the main class for all models. The last class should be the specific model name, in this case `imm_full`, `imm_abc` or `imm_bsc`, which is automatically constructed if a `version` argument is provided. Otherwise the last class will be just the name of the model.

4.1.2 Model alias

The model alias is a user-facing function that calls the internal model function. It is defined as follows:

```
#' @title `r .model_imm()$name`
#' @description Three versions of the `r .model_imm()$name` - the full, bsc, and abc.
#' `IMMfull()`, `IMMbsc()`, and `IMMabc()` are deprecated and will be removed in the future.
#' Please use `imm(version = 'full')`, `imm(version = 'bsc')`, or `imm(version = 'abc')` inst
#'
```

```
#' @name imm
#' @details `r model_info(.model_imm(), components =c('domain', 'task', 'name', 'citation'))
#' #### Version: `full`
#' `r model_info(.model_imm(version = "full"), components = c('requirements', 'parameters',
#' #### Version: `bsc`
#' `r model_info(.model_imm(version = "bsc"), components = c('requirements', 'parameters', '
#' #### Version: `abc`
#' `r model_info(.model_imm(version = "abc"), components =c('requirements', 'parameters', 'f
#'
#' Additionally, all imm models have an internal parameter that is fixed to 0 to
#' allow the model to be identifiable. This parameter is not estimated and is not
#' included in the model formula. The parameter is:
#'
#'   - b = "Background activation (internally fixed to 0)"
#'
#' @param resp_error The name of the variable in the provided dataset containing
#'   the response error. The response Error should code the response relative to
#'   the to-be-recalled target in radians. You can transform the response error
#'   in degrees to radian using the `deg2rad` function.
#' @param nt_features A character vector with the names of the non-target
#'   variables. The non_target variables should be in radians and be centered
#'   relative to the target. Alternatively, if regex=TRUE, a regular
#'   expression can be used to match the non-target feature columns in the
#'   dataset.
#' @param nt_distances A vector of names of the columns containing the distances
#'   of non-target items to the target item. Alternatively, if regex=TRUE, a regular
#'   expression can be used to match the non-target distances columns in the
#'   dataset. Only necessary for the `bsc` and `full` versions.
#' @param set_size Name of the column containing the set size variable (if
#'   set_size varies) or a numeric value for the set_size, if the set_size is
#'   fixed.
#' @param regex Logical. If TRUE, the `nt_features` and `nt_distances` arguments
#'   are interpreted as a regular expression to match the non-target feature
#'   columns in the dataset.
#' @param version Character. The version of the IMM model to use. Can be one of
#'  `full`, `bsc`, or `abc`. The default is `full`.
#' @param ... used internally for testing, ignore it
#' @return An object of class `bmmodel`
#' @keywords bmmodel
#' @examplesIf isTRUE(Sys.getenv("BMM_EXAMPLES"))
#' # load data
#' data <- oberauer_lin_2017
```

```
#'
#' # define formula
#' ff <- bmmformula(
#'   kappa ~ 0 + set_size,
#'   c ~ 0 + set_size,
#'   a ~ 0 + set_size,
#'   s ~ 0 + set_size
#' )
#'
#' # specify the full IMM model with explicit column names for non-target features and distan
#' # by default this fits the full version of the model
#' model1 <- imm(resp_error = "dev_rad",
#'               nt_features = paste0('col_nt', 1:7),
#'               nt_distances = paste0('dist_nt', 1:7),
#'               set_size = 'set_size')
#'
#' # fit the model
#' fit <- bmm(formula = ff,
#'            data = data,
#'            model = model1,
#'            cores = 4,
#'            backend = 'cmdstanr')
#'
#' # alternatively specify the IMM model with a regular expression to match non-target featur
#' # this is equivalent to the previous call, but more concise
#' model2 <- imm(resp_error = "dev_rad",
#'               nt_features = 'col_nt',
#'               nt_distances = 'dist_nt',
#'               set_size = 'set_size',
#'               regex = TRUE)
#'
#' # fit the model
#' fit <- bmm(formula = ff,
#'            data = data,
#'            model = model2,
#'            cores = 4,
#'            backend = 'cmdstanr')
#'
#' # you can also specify the `bsc` or `abc` versions of the model to fit a reduced version
#' model3 <- imm(resp_error = "dev_rad",
#'               nt_features = 'col_nt',
#'               set_size = 'set_size',
```

```
#'                regex = TRUE,
#'                version = 'abc')
#' fit <- bmm(formula = ff,
#'            data = data,
#'            model = model3,
#'            cores = 4,
#'            backend = 'cmdstanr')
#' @export
imm <- function(resp_error, nt_features, nt_distances, set_size, regex = FALSE, version = "fu
  call <- match.call()
  dots <- list(...)
  if ("setsize" %in% names(dots)) {
    set_size <- dots$setsize
    warning("The argument 'setsize' is deprecated. Please use 'set_size' instead.")
  }
  if (version == "abc") {
    nt_distances <- NULL
  }
  stop_missing_args()
  .model_imm(resp_error = resp_error, nt_features = nt_features,
             nt_distances = nt_distances, set_size = set_size, regex = regex,
             version = version, call = call, ...)
}
```

The details will be filled out automatically from the model definition. This does some fancy formatting to include documentation about all versions of the model in the same help file.

4.1.3 check_data() methods

Each model should have a `check_data.modelname()` method that checks if the data contains all necessary information for fitting the model. For the IMM, the `bsc` and `full` versions require a special check for the `nt_distances` variables:

```
#' @export
check_data.imm_bsc <- function(model, data, formula) {
  data <- .check_data_imm_dist(model, data, formula)
  NextMethod("check_data")
}

#' @export
check_data.imm_full <- function(model, data, formula) {
```

```
  data <- .check_data_imm_dist(model, data, formula)
  NextMethod("check_data")
}

.check_data_imm_dist <- function(model, data, formula) {
  nt_distances <- model$other_vars$nt_distances
  max_set_size <- attr(data, 'max_set_size')

  stopif(!isTRUE(all.equal(length(nt_distances), max_set_size - 1)),
         "The number of columns for non-target distances in the argument \\
         'nt_distances' should equal max(set_size)-1})")

  # replace nt_distances
  data[,nt_distances][is.na(data[,nt_distances])] <- 999

  stopif(any(data[,nt_distances] < 0),
         "All non-target distances to the target need to be postive.")
  data
}
```

The IMM models share methods with the `mixture3p` model, all of which are of class `non_targets` so the `check_data.non_targets` method is defined in the general file `R/helpers-data.R`. If you are adding a new model, you should check if the data requirements are similar to any existing model and define the `check_data` method only for the methods that are unique to your model.

The `check_data.mymodel()` function should always take the arguments `model`, `data`, and `formula` and return the data with the necessary transformations. It should also call `data = NextMethod("check_data")` to call the check_data method of the more general class.

4.1.4 configure_model() methods

The configure_model.mymodel() method is where you specify the model formula, the family, any custom code. The method is defined as follows for the IMM model:

(we show only the `IMMfull` version)

```
configure_model.imm_full <- function(model, data, formula) {
  # retrieve arguments from the data check
  max_set_size <- attr(data, 'max_set_size')
  lure_idx <- attr(data, "lure_idx_vars")
  nt_features <- model$other_vars$nt_features
```

```
  set_size_var <- model$other_vars$set_size
  nt_distances <- model$other_vars$nt_distances

  # construct main brms formula from the bmm formula
  formula <- bmf2bf(model, formula) +
    brms::lf(kappa2 ~ 1) +
    brms::lf(mu2 ~ 1) +
    brms::nlf(theta1 ~ log(exp(c) + exp(a))) +
    brms::nlf(kappa1 ~ kappa) +
    brms::nlf(expS ~ exp(s))

  # additional internal terms for the mixture model formula
  kappa_nts <- paste0("kappa", 3:(max_set_size + 1))
  theta_nts <- paste0("theta", 3:(max_set_size + 1))
  mu_nts <- paste0("mu", 3:(max_set_size + 1))

  for (i in 1:(max_set_size - 1)) {
    formula <- formula +
      glue_nlf("{kappa_nts[i]} ~ kappa") +
      glue_nlf("{theta_nts[i]} ~ {lure_idx[i]} * log(exp(c-expS*{nt_distances[i]}) + exp(a))"
               "+ (1 - {lure_idx[i]}) * (-100)") +
      glue_nlf("{mu_nts[i]} ~ {nt_features[i]}")
  }

  # define mixture family
  formula$family <- brms::mixture(brms::von_mises("tan_half"),
                                  brms::von_mises("identity"),
                                  nmix = c(1, max_set_size),
                                  order = "none")

  nlist(formula, data)
}
```

The configure_model method should always take the arguments `model`, `data`, and `formula` (as a `bmmformula`) and return a named list with the formula (as a `brmsformula`) and the data. The `brmsfamily` should be stored within the formula.

Inside the configure_model method the `brmsformula` is generated using the `bmf2bf` function. This function converts the `bmmformula` passed to `bmm()` function into a `brmsformula` based on the information for the response variables provided in the `bmmmodel` object. There is a general method in `R/bmmformula.R` to construct the formula for all models with a single response variable.

```r
# default method to paste the full brms formula for all bmmodels
#' @export
bmf2bf.bmmodel <- function(model, formula) {
  # check if the model has only one response variable and extract if TRUE
  brms_formula <- NextMethod("bmf2bf")

  # for each dependent parameter, check if it is used as a non-linear predictor of
  # another parameter and add the corresponding brms function
  for (pform in formula) {
    if (is_nl(pform)) {
      brms_formula <- brms_formula + brms::nlf(pform)
    } else {
      brms_formula <- brms_formula + brms::lf(pform)
    }
  }
  brms_formula
}

# paste first line of the brms formula for all bmmodels with 1 response variable
#' @export
bmf2bf.default <- function(model, formula){
  # set base brms formula based on response
  brms::bf(paste0(model$resp_vars[[1]], "~ 1"))
}
```

The `bmf2bf.bmmodel` method initializes the conversion of the `bmmformula` to a `brms_formula`. The first step for this is to paste the first line of a `brmsformula` that includes the response as the dependent variable on the left-hand side. For models with a single response variable this is done in the `bmf2bf.default` method. For models with more than one response variable, you will have to provide a model specific method of `bmf2bf.myModel` to convert the `bmmformula` into the `brmsformula` . This conversion from a `bmmformula` object into a `brmsformula` object is done to avoid users having to specify complicated and long formulas or specifying all additional response information in the `brmsformula` themselves. For more detailed information on the use of additional response information in a `brmsformula` please see the `brmsformula` documentation.

4.2 The Signal Discrimination Model (SDM)

The SDM model is defined in the file `R/model_sdm.R`. The SDM model differs in the configuration compared to the IMM model, as it requires custom STAN code. Let's go through the different parts. As before, we start with the model definition.

4.2.1 Model definition

```
.model_sdm <- function(resp_error = NULL, links = NULL, version = "simple", call = NULL, ...)
  out <- structure(
    list(
      resp_vars = nlist(resp_error),
      other_vars = nlist(),
      domain = 'Visual working memory',
      task = 'Continuous reproduction',
      name = 'Signal Discrimination Model (SDM) by Oberauer (2023)',
      citation = glue(
        'Oberauer, K. (2023). Measurement models for visual working memory - \\
        A factorial model comparison. Psychological Review, 130(3), 841-852'
      ),
      version = version,
      requirements = glue(
        '- The response variable should be in radians and represent the angular \\
        error relative to the target'
      ),
      parameters = list(
        mu = glue('Location parameter of the SDM distribution (in radians; \\
                  by default fixed internally to 0)'),
        c = 'Memory strength parameter of the SDM distribution',
        kappa = 'Precision parameter of the SDM distribution'
      ),
      links = list(
        mu = 'tan_half',
        c = 'log',
        kappa = 'log'
      ),
      fixed_parameters = list(mu = 0),
      default_priors = list(
        mu = list(main = "student_t(1, 0, 1)"),
        kappa = list(main = "student_t(5, 1.75, 0.75)", effects = "normal(0, 1)"),
        c = list(main = "student_t(5, 2, 0.75)", effects = "normal(0, 1)")
      ),
      void_mu = FALSE
    ),
    class = c('bmmodel', 'circular', 'sdm', paste0("sdm_", version)),
    call = call
  )
  out$links[names(links)] <- links
```

```
  out
}
```

The model definition is similar to the IMM model, but the SDM model only requires the user to specify the response error, but not additional variables such as non-target variables. The `class` is also different, as the SDM model is not a subclass of the IMM model. We'll skip the alias for the SDM model, as it is similar for every model.

4.2.2 check_data() methods

The SDM shares a class with other `circular` models, so most of the data checks are performed by `check_data.circular` method, defined in the general file `R/helpers-data.R`. The `sdm` however, samples much more quickly in Stan, if the data is sorted by the predictor variables, so we have the following custom data check method for the sdm:

```
#' @export
check_data.sdm <- function(model, data, formula) {
  # data sorted by predictors is necessary for speedy computation of normalizing constant
  data <- order_data_query(model, data, formula)
  NextMethod("check_data")
}
```

4.2.3 configure_model() methods

The configure_model method for the SDM model is different compared to the IMM model, as it requires custom STAN code. The method is defined as follows:

```
#' @export
configure_model.sdm <- function(model, data, formula) {
  # construct the family
  # note - c has a log link, but I've coded it manually for computational efficiency
  sdm_simple <- brms::custom_family(
    "sdm_simple",
    dpars = c("mu", "c", "kappa"),
    links = c("tan_half", "identity", "log"),
    lb = c(NA, NA, NA),
    ub = c(NA, NA, NA),
    type = "real", loop = FALSE,
    log_lik = log_lik_sdm_simple,
    posterior_predict = posterior_predict_sdm_simple
```

```
  )

  # prepare initial stanvars to pass to brms, model formula and priors
  sc_path <- system.file("stan_chunks", package = "bmm")
  stan_funs <- read_lines2(paste0(sc_path, "/sdm_simple_funs.stan"))
  stan_tdata <- read_lines2(paste0(sc_path, "/sdm_simple_tdata.stan"))
  stan_likelihood <- read_lines2(paste0(sc_path, "/sdm_simple_likelihood.stan"))
  stanvars <- brms::stanvar(scode = stan_funs, block = "functions") +
    brms::stanvar(scode = stan_tdata, block = "tdata") +
    brms::stanvar(scode = stan_likelihood, block = "likelihood", position = "end")

  # construct main brms formula from the bmm formula
  formula <- bmf2bf(model, formula)
  formula$family <- sdm_simple

  # set initial values to be sampled between [-1,1] to avoid extreme SDs that
  # can cause the sampler to fail
  init <- 1

  # return the list
  nlist(formula, data, stanvars, init)
}
```

Lines 5-14 use the `brms::custom_family` function to define a custom family for the SDM model. The `dpars` argument specifies the parameters of the model, and the `links` argument specifies the link functions for the parameters. For more information, see here

Lines 17-23 read the custom STAN code from the `inst/stan_chunks` directory. This has to be specified with the system.file() command to ensure that the code is found when the package is installed. The `stanvars` object is used to pass custom STAN code to the `brms` package. The `stanvars` object is a list of `brms::stanvar` objects, each of which contains the STAN code for a specific part of the model. There is a separate `.stan` file for each part of the STAN code, and each file is read into a separate `brms::stanvar` object.

Converting the `bmmformula` to a `brmsformula` and collecting all arguments is done entirely using the `bmf2bf` method.

4.2.4 Post-processing methods

Unlike the `imm` model, the `sdm` model requires some special post-processing because of the way the link functions are coded. These methods are applied *after* the brmsfit object is returned, at the very end of the *bmm()* pipeline:

```
#' @export
postprocess_brm.sdm <- function(model, fit, ...) {
  # manually set link_c to "log" since I coded it manually
  fit$family$link_c <- "log"
  fit$formula$family$link_c <- "log"
  fit
}

#' @export
revert_postprocess_brm.sdm <- function(model, fit, ...) {
  fit$family$link_c <- "identity"
  fit$formula$family$link_c <- "identity"
  fit
}
```

we also have a couple of special functions for custom families in brms (see the `log_lik`and `posterior_predict` argument in the call to `brms::custom_familiy()`), which allow other typical tools from `brms` such posterior_predict of bridgesampling to work:

```
log_lik_sdm_simple <- function(i, prep) {
  mu <- brms::get_dpar(prep, "mu", i = i)
  c <- brms::get_dpar(prep, "c", i = i)
  kappa <- brms::get_dpar(prep, "kappa", i = i)
  y <- prep$data$Y[i]
  dsdm(y, mu, c, kappa, log = T)
}

posterior_predict_sdm_simple <- function(i, prep, ...) {
  mu <- brms::get_dpar(prep, "mu", i = i)
  c <- brms::get_dpar(prep, "c", i = i)
  kappa <- brms::get_dpar(prep, "kappa", i = i)
  rsdm(length(mu), mu, c, kappa)
}
```

We will now look at how to construct all these parts for a new model. **Hint**: you don't have to do it manually, you can use the `use_model_template()` function to generate templates for your model.

5 Adding a new model

If you have read Section 3 and Section 4, you should have a pretty good idea of how the `bmm` package functions. Now it's time to add your new model.

The good news are, you don't have to add any of the files manually. The `bmm` package includes a function `use_model_template()` that generates all the files with templates for all necessary functions. Thus, you can focus on the important work of filling these templates with the relevant information without worrying about missing something critical. If you type `?use_model_template` you will see the following description:

Function `use_model_template()`

Description

Create a file with a template for adding a new model (for developers)

Usage

```
use_model_template(
  model_name,
  custom_family = FALSE,
  stanvar_blocks = c("data", "tdata", "parameters",
                     "tparameters", "model", "likelihood",
                     "genquant", "functions"),
  open_files = TRUE,
  testing = FALSE
)
```

Arguments

Argument	Description
`model_name`	A string with the name of the model. The file will be named bmm_model_model_name.R and all necessary functions will be created with the appropriate names and structure. The file will be saved in the R/ directory
`custom_family`	Logical; Do you plan to define a brms::custom_family()? If TRUE the function will add a section for the custom family, placeholders for the stan_vars and corresponding empty .stan files in inst/stan_chunks/, that you can fill For an example, see the sdmSimple model in /R/bmm_model_sdmSimple.R. If FALSE (default) the function will not add the custom family section nor stan files.
`stanvar_blocks`	A character vector with the names of the blocks that will be added to the custom family section. See brms::stanvar() for more details. The default lists all the possible blocks, but it is unlikely that you will need all of them. You can specify a vector of only those that you need. The function will add a section for each block in the list
`open_files`	Logical; If TRUE (default), the function will open the template files that were created in RStudio
`testing`	Logical; If TRUE, the function will return the file content but will not save the file. If FALSE (default), the function will save the file

5.1 Example

Let's add a new model called `gcm`. Let's assume that you have tested the model in Stan and you have the Stan code ready. We want to define a custom family for the `gcm` model, and we want to define the following blocks: `likelihood`, `functions` (see `?brms::stanvar` for an explanation of the blocks).

First you set up your system and git environment as described in Section 1. Then you can run the following code from RStudio in the root directory of the `bmm` package:

```
use_model_template("gcm", custom_family = TRUE, stanvar_blocks = c("likelihood", "functions")
```

This will create the file `bmm_model_gcm.R` in the `R/` directory and the files `gcm_likelihood.stan` and `gcm_functions.stan` in the `inst/stan_chunks/` directory. The function will also open the files in RStudio. You will see the following output in the console:

- Modify 'inst/stan_chunks/gcm_likelihood.stan'
- Modify 'inst/stan_chunks/gcm_functions.stan'
- Modify 'R/model_gcm.R'

Now you can fill the files with the appropriate code. The Stan files will be empty, but the R file will have the following structure:

```
##############################################################################!
# MODELS                                                                     ####
##############################################################################!
# see file 'R/model_mixture3p.R' for an example

.model_gcm <- function(resp_var1 = NULL, required_arg1 = NULL, required_arg2 = NULL, links =
   out <- structure(
     list(
       resp_vars = nlist(resp_var1),
       other_vars = nlist(required_arg1, required_arg2),
       domain = '',
       task = '',
       name = '',
       citation = '',
       version = version,
       requirements = '',
       parameters = list(),
       links = list(),
       fixed_parameters = list(),
       default_priors = list(par1 = list(), par2 = list()),
       void_mu = FALSE
     ),
     class = c('bmmodel', 'gcm'),
     call = call
   )
   if(!is.null(version)) class(out) <- c(class(out), paste0("gcm_",version))
   out$links[names(links)] <- links
   out
}
# user facing alias
# information in the title and details sections will be filled in
# automatically based on the information in the .model_gcm()$info

#' @title `r .model_gcm()$name`
#' @name Model Name#' @details `r model_info(.model_gcm())`
```

```r
#' @param resp_var1 A description of the response variable
#' @param required_arg1 A description of the required argument
#' @param required_arg2 A description of the required argument
#' @param links A list of links for the parameters.
#' @param version A character label for the version of the model. Can be empty or NULL if the
#' @param ... used internally for testing, ignore it
#' @return An object of class `bmmodel`
#' @export
#' @examples
#' \dontrun{
#' # put a full example here (see 'R/model_mixture3p.R' for an example)
#' }
gcm <- function(resp_var1, required_arg1, required_arg2, links = NULL, version = NULL, ...) {
   call <- match.call()
   stop_missing_args()
   .model_gcm(resp_var1 = resp_var1, required_arg1 = required_arg1, required_arg2 = required_
               links = links, version = version,call = call, ...)
}


#############################################################################!
# CHECK_DATA S3 methods                                                     ####
#############################################################################!
# A check_data.* function should be defined for each class of the model.
# If a model shares methods with other models, the shared methods should be
# defined in helpers-data.R. Put here only the methods that are specific to
# the model. See ?check_data for details.
# (YOU CAN DELETE THIS SECTION IF YOU DO NOT REQUIRE ADDITIONAL DATA CHECKS)

#' @export
check_data.gcm <- function(model, data, formula) {
   # retrieve required arguments
   required_arg1 <- model$other_vars$required_arg1
   required_arg2 <- model$other_vars$required_arg2

   # check the data (required)

   # compute any necessary transformations (optional)

   # save some variables as attributes of the data for later use (optional)

   NextMethod('check_data')
```

```
}


################################################################################!
# Convert bmmformula to brmsformla methods                                    ####
################################################################################!
# A bmf2bf.* function should be defined if the default method for consructing
# the brmsformula from the bmmformula does not apply (e.g if aterms are required).
# The shared method for all `bmmodels` is defined in bmmformula.R.
# See ?bmf2bf for details.
# (YOU CAN DELETE THIS SECTION IF YOUR MODEL USES A STANDARD FORMULA WITH 1 RESPONSE VARIABL

#' @export
bmf2bf.gcm <- function(model, formula) {
   # retrieve required response arguments
   resp_var1 <- model$resp_vars$resp_var1
   resp_var2 <- model$resp_vars$resp_arg2

   # set the base brmsformula based
   brms_formula <- brms::bf(paste0(resp_var1," | ", vreal(resp_var2), " ~ 1" ),)

   # return the brms_formula to add the remaining bmmformulas to it.
   brms_formula
}


################################################################################!
# CONFIGURE_MODEL S3 METHODS                                                  ####
################################################################################!
# Each model should have a corresponding configure_model.* function. See
# ?configure_model for more information.

#' @export
configure_model.gcm <- function(model, data, formula) {
   # retrieve required arguments
   required_arg1 <- model$other_vars$required_arg1
   required_arg2 <- model$other_vars$required_arg2

   # retrieve arguments from the data check
   my_precomputed_var <- attr(data, 'my_precomputed_var')

   # construct brms formula from the bmm formula
```

```
   formula <- bmf2bf(model, formula)

   # construct the family & add to formula object
   gcm_family <- brms::custom_family(
     'gcm',
     dpars = c(),
     links = c(),
     lb = c(), # upper bounds for parameters
     ub = c(), # lower bounds for parameters
     type = '', # real for continous dv, int for discrete dv
     loop = TRUE, # is the likelihood vectorized
   )
   formula$family <- gcm_family

   # prepare initial stanvars to pass to brms, model formula and priors
   sc_path <- system.file('stan_chunks', package='bmm')
   stan_likelihood <- read_lines2(paste0(sc_path, '/gcm_likelihood.stan'))
   stan_functions <- read_lines2(paste0(sc_path, '/gcm_functions.stan'))

   stanvars <- stanvar(scode = stan_likelihood, block = 'likelihood') +
      stanvar(scode = stan_functions, block = 'functions')

   # return the list
   nlist(formula, data, stanvars)
}


################################################################################!
# POSTPROCESS METHODS                                                          ####
################################################################################!
# A postprocess_brm.* function should be defined for the model class. See
# ?postprocess_brm for details

#' @export
postprocess_brm.gcm <- function(model, fit) {
   # any required postprocessing (if none, delete this section)
   fit
}
```

Now you have to:

1. Fill the `.model_gcm` function with the appropriate code. This function should return a

list with all the variables specified above. The class of the list should be `c('bmmmodel', 'gcm')`. Rename the response arguments and the other required arguments, or delete the other arguments if you do not have any. You can see an example in the `model_sdm.R` file. Specify the parameters of the model, the link functions, what if any parameters are fixed (and to what value). It's crucial that you set default priors for every parameter of the model, which should be informed by knowledge in the field.

2. Adjust the user-facing alias. Here you should only rename the required arguments and fill in the `@examples` section with a full example. Everything else will be filled in automatically based on the information in the `.model_gcm` function.

3. Fill the `check_data.gcm` function with the appropriate code. This function should check the data and return the data. You may or may not need to compute any transformations or save some variables as attributes of the data.

4. If necessary define the `bmf2bf.gcm` method to convert the `bmmformula` to a `brmsformula`. The first step for this is always to specify the response variable and additional response information. Keep in mind that `brms` automatically interprets this formula as the linear model formula for the `mu` parameter of your custom family. Currently, `brms` requires all custom families to have a `mu` parameter. However, we recommend to code this parameter as a `void_mu`, and fix the intercept of this parameter to zero using constant priors. This way, the `bmmformula` can be used to only specify the linear or non-linear model for the parameters of a `bmmmodel`. *If your model has a single response variable, you can delete this section.*

5. Fill the `configure_model.gcm` function with the appropriate code. This function should construct the formula, the family, the stanvars. You can also retrieve any arguments you saved from the data check. Depending on your model, some of these parts might not be necessary. For example, for the mixture models (e.g. `mixture3p`), we construct a new formula, because we want to rename the arguments to make it easier for the user. For the `sdmSimple` model, we define the family ourselves, so we don't need to change the formula.

   You need to fill information about your custom family, and then fill the STAN files with your STAN code. Conveniently, you don't have to edit lines 134-140: loading the STAN files and setting up the stanvars is set up automatically when calling the `use_model_template` function. Should you need to add more STAN files after you created the template, you can add the files in `init/stan_chunks/` manually and edit those lines to additionally load the manually added files.

6. Fill the `postprocess_brm.gcm` function with the appropriate code. By post-processing, we mean changes to the fitted brms model - like renaming parameters, etc. If you don't need any post-processing, you can delete this section.

5.2 Testing

Unit testing is extremely important. You should test your model with the `testthat` package. You can use the `use_test()` function to create a test file for your model. See file `tests/testthat/test-bmm.R` for an example of how we test the existing models. BRMS models take a long time to fit, so we don't test the actual fitting process. The `bmm()` function provides an argument `backend="mock"`, which will return a mock object instead of fitting the model. This ensures that the entire pipeline works without errors. For example, here's a test of the `IMMfull` model:

```
test_that('Available mock models run without errors',{
  withr::local_options('bmm.silent'=2)
  skip_on_cran()
  dat <- data.frame(
    resp_err = rIMM(n = 5),
    Item2_rel = 2,
    Item3_rel = -1.5,
    spaD2 = 0.5,
    spaD3 = 2
  )

  # two-parameter model mock fit
  f <- bmmformula(kappa ~ 1, c ~ 1, a ~ 1, s ~ 1)
  mock_fit <- bmm(f, dat,
                  imm(resp_err = "resp_err",
                      setsize = 3,
                      nt_features = paste0('Item',2:3,'_rel'),
                      nt_distance=paste0('spaD',2:3)),
                  backend = "mock", mock_fit = 1, rename=FALSE)
  expect_equal(mock_fit$fit, 1)
  expect_type(mock_fit$bmm$fit_args, "list")
  expect_equal(names(mock_fit$fit_args[1:4]), c("formula", "data"))
})
```

The tests based on the `testthat` package are run every time you call the `check()` command. Before you submit your changes, make sure that all tests pass.

> **!** Important
>
> Additionally, you should perform a full test of the model by running it in a separate script and ensuring it gives meaningful results. At the very least, you should perform basic parameter recovery simulations for hyper-parameters (i.e. means and standard deviations)

as well as subject-level parameters to give users an idea of how much data they need to adequately estimate the model.
We are in the process of establishing guidelines for that.

5.3 Add an example dataset

All new models should come with an example dataset, that can be loaded by users and can be used in the examples section. This should be either:

- A new dataset that you add to the package
- A dataset that already exists in the package but that can be used with the new model
- A dataset that exists in another package that you can load with `data()` and use with the new model

For example, the vignettes for the `mixture2p` and `mixture3p` use an external dataset from the `mixtur` package that can be loaded with `data('bays2009_full', package='mixtur')`. The `IMM` models use a dataset included in the current package. For instructions on how to add a new dataset see here.

5.4 Add an article

All new models should come with an article that explains some basic information about the model and how to estimate it with `bmm`. You can use the `use_article()` function to create a new article. See here for more information. The articles will be published automatically on the package website under "Articles" when the pull request is approved. You can browse the source code for the existing articles in the `vignettes/articles/` directory. You can see the published version of the existing vignettes here.

And that's it! You have added a new model to the `bmm` package. You can now submit your changes to the `bmm` package repository.