BMM Developer Notes

Table of contents

Overview

This article 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.

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

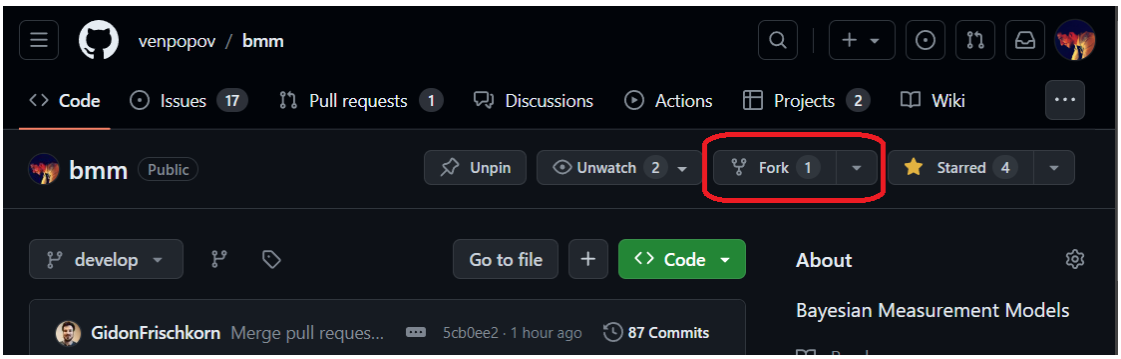
- Master (contains the latest released stable version of the `bmm` package
- Develop (contains the latest stable development branch

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.

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)
```

   To avoid having to load the 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 - `fit_model()`

The main function for fitting models is `fit_model()`. 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:

```
fit_model <- function(formula, data, model, parallel = FALSE,
                      chains = 4, prior = NULL, ...) {
  # enable parallel sampling if parallel equals TRUE
  opts <- configure_options(nlist(parallel, chains))

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

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

  # combine the default prior plus user given prior
  config_args <- combine_prior(config_args, prior)

  # estimate the model
  dots <- list(...)
  fit_args <- c(config_args, opts, dots)
  fit <- call_brm(fit_args)

  # model postprocessing
  fit <- postprocess_brm(model, fit)
```

```
  return(fit)
}
```

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
- `check_data()` - whether the data contains all necessary information
- `configure_model()` - to configure the model for fitting
- `combine_priors()` - to combine a user specified prior with the default priors we provide for each model
- `call_brm()` - to fit the model using the `brm()` function from the `brms` package
- `postprocess_brm()` - to postprocess the fitted model

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 postprocessing. 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_modelname <- function(required_arg1, required_arg2, ...) {
   out <- list(
      vars = nlist(required_arg1, required_arg2),
      info = list(
         domain = '',
         task = '',
         name = '',
         citation = '',
         version = '',
         requirements = '',
         parameters = list()
      ))
   class(out) <- c('bmmmodel', 'modelname')
   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()$info
#' @title `r .model_modelname()$info$name`
#' @details `r model_info(modelname(NA,NA))`
#' @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)
#' }
modelname <- .model_modelname
```

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

```
fit <- fit_model(formula,
                 data = mydata,
                 model = modelname(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, `fit_model()`.

3.4 File organization

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

`R/fit_model.R`

It contains the main function for fitting models, `fit_model()`. 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-postprocess.R`, `R/helpers-model.R`, `R/helpers-formula.R` and `R/helpers-prior.R`

These files define the main generic S3 methods for checking data, postprocessing 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 `fit_model()` 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/bmm_model_*.R`

Each model and it's methods is defined in a separate file. For example, the 3-parameter mixture model is defined in `bmm_model_3p.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 `bmm_model_name_of_your_model.R`. You don't have to add this file manually - see section Adding new models.

`R/bmm_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/random_data_generation.R`

Functions for generating random data from the models. This is useful for testing the models.

`R/utils.R`, `R/brms-misc.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 postprocessing. 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, all sections bellows will be automatically generated for your model.

4.1 The Interference Measurement Model (IMM)

The model is defined in the file `R/bmm_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_IMMfull <- function(non_targets, setsize, spaPos, ...) {
  out <- list(
    vars = nlist(non_targets, setsize, spaPos),
    info = list(
      domain = "Visual working memory",
      task = "Continuous reproduction",
      name = "Interference measurement model by Oberauer and Lin (2017).",
      version = "full",
      citation = paste0("Oberauer, K., & Lin, H.Y. (2017). An interference model ",
                        "of visual working memory. Psychological Review, 124(1), 21-59"),
      requirements = paste0('- The response vairable should be in radians and ',
                            'represent the angular error relative to the target\n  ',
                            '- The non-target variables should be in radians and be ',
                            'centered relative to the target'),
      parameters = list(
        kappa = "Concentration parameter of the von Mises distribution (log scale)",
        a = "General activation of memory items",
```

```
        b = "Background activation (internally fixed to 0)",
        c = "Context activation",
        s = "Spatial similarity gradient"
      )
    ))
  class(out) <- c("bmmmodel","vwm","nontargets","IMMspatial","IMMfull")
  out
}
```

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

`vars`: a list of 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 non-target variables, the setsize and the variable specifying the spatial distance between the memory items. The user has to provide these variables in the data frame that is passed to the `fit_model` function

`info`: contains information about the model, such as the domain, task, name, citation, version, requirements, and parameters. This information is used to generate the documentation for the model.

`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', 'IMMabc', 'IMMbsc' and 'IMMfull' models all have non-targets and setsize arguments, so the same data checks can be performed on all of them, represented by the class `nontargets`. The first class should always be `bmmmodel`, which is the main class for all models. The last class should be the specific model name, in this case `IMMfull`.

4.1.2 Model alias

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

```
# user facing alias
#' @title `r .model_IMMfull(NA, NA, NA)$info$name`
#' @name IMM
#' @details `r model_info(IMMfull(NA, NA, NA), components =c('domain', 'task', 'name', 'cita
#' #### Version: `IMMfull`
#' `r model_info(IMMfull(NA, NA, NA), components =c('requirements', 'parameters'))`
#' #### Version: `IMMbsc`
#' `r model_info(IMMbsc(NA, NA, NA), components =c('requirements', 'parameters'))`
```

```
#' #### Version: `IMMabc`
#' `r model_info(IMMabc(NA, NA, NA), components =c('requirements', 'parameters'))`
#' @param non_targets 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.
#' @param setsize Name of the column containing the set size variable (if
#'   setsize varies) or a numeric value for the setsize, if the setsize is
#'   fixed.
#' @param spaPos A vector of names of the columns containing the spatial distances of
#'   non-target items to the target item. Only necessary for the `IMMbsc` and `IMMfull` model
#' @param ... used internally for testing, ignore it
#' @return An object of class `bmmmodel`
#' @keywords bmmmodel
#' @export
IMMfull <- .model_IMMfull

#' @rdname IMM
#' @keywords bmmmodel
#' @export
IMMbsc <- .model_IMMbsc

#' @rdname IMM
#' @keywords bmmmodel
#' @export
IMMabc <- .model_IMMabc
```

The details will be filled out automatically from the model definition. The example for the IMM model also includes the aliases for the other versions of the IMM model, which are `IMMbsc` and `IMMabc`, and 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, all three versions of the model share the same data requirements, so check_data is defined for the more general class, `IMMspatial`. The method is defined as follows:

```
#' @export
check_data.IMMspatial <- function(model, data, formula) {
  spaPos <- model$vars$spaPos
```

```
  max_setsize <- attr(data, 'max_setsize')

  if (length(spaPos) < max_setsize - 1) {
    stop(paste0("The number of columns for spatial positions in the argument",
                "'spaPos' is less than max(setsize)-1"))
  } else if (length(spaPos) > max_setsize-1) {
    stop(paste0("The number of columns for spatial positions in the argument",
                "spaPos'' is more than max(setsize)-1"))
  }

  if (max(abs(data[,spaPos]), na.rm=T) > 10) {
    data[,spaPos] <- data[,spaPos]*pi/180
    warning('It appears your spatial position variables are in degrees. We will transform it
  }
  # wrap spatial position variables around the circle (range = -pi to pi)
  data[,spaPos] <- bmm::wrap(data[,spaPos])
  data = NextMethod("check_data")

  return(data)
}
```

The IMM models share methods with the `mixture3p` model, all of which are of class `nontargets` so the `check_data.nontargets` 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, and the priors. The method is defined as follows for the IMM model:

(we show only the `IMMfull` version)

```
#' @export
configure_model.IMMfull <- function(model, data, formula) {
  # retrieve arguments from the data check
  max_setsize <- attr(data, 'max_setsize')
```

```r
lure_idx_vars <- attr(data, "lure_idx_vars")
non_targets <- model$vars$non_targets
setsize_var <- model$vars$setsize
spaPos <- model$vars$spaPos

# names for parameters
kappa_nts <- paste0('kappa', 2:max_setsize)
kappa_unif <- paste0('kappa',max_setsize + 1)
theta_nts <- paste0('theta',2:max_setsize)
mu_nts <- paste0('mu', 2:max_setsize)
mu_unif <- paste0('mu', max_setsize + 1)

# construct formula
formula <- formula +
  brms::lf(mu1 ~ 1) +
  glue_lf(kappa_unif,' ~ 1') +
  glue_lf(mu_unif, ' ~ 1') +
  brms::nlf(theta1 ~ c + a) +
  brms::nlf(kappa1 ~ kappa) +
  brms::nlf(expS ~ exp(s))
for (i in 1:(max_setsize-1)) {
  formula <- formula +
    glue_nlf(kappa_nts[i], ' ~ kappa') +
    glue_nlf(theta_nts[i], ' ~ ', lure_idx_vars[i], '*(exp(-expS*',spaPos[i],')*c + a) + '
             '(1-', lure_idx_vars[i], ')*(-100)') +
    glue_nlf(mu_nts[i], ' ~ ', non_targets[i])
}

# define mixture family
vm_list = lapply(1:(max_setsize+1), function(x) brms::von_mises(link="identity"))
vm_list$order = "none"
family <- brms::do_call(brms::mixture, vm_list)

# define prior
prior <- # fix mean of the first von Mises to zero
  brms::prior_("constant(0)", class = "Intercept", dpar = "mu1") +
  # fix mean of the guessing distribution to zero
  brms::prior_("constant(0)", class = "Intercept", dpar = mu_unif) +
  # fix kappa of the second von Mises to (alomst) zero
  brms::prior_("constant(-100)", class = "Intercept", dpar = kappa_unif) +
  # set reasonable priors fpr the to be estimated parameters
  brms::prior_("normal(2, 1)", class = "b", nlpar = "kappa") +
```

```
    brms::prior_("normal(0, 1)", class = "b", nlpar = "c") +
    brms::prior_("normal(0, 1)", class = "b", nlpar = "a") +
    brms::prior_("normal(0, 1)", class = "b", nlpar = "s")

  # if there is setsize 1 in the data, set constant prior over thetant for setsize1
  if ((1 %in% data$ss_numeric) && !is.numeric(data[[setsize_var]])) {
    prior <- prior +
      brms::prior_("constant(0)", class="b", coef = paste0(setsize_var, 1), nlpar="a")+
      brms::prior_("constant(0)", class="b", coef = paste0(setsize_var, 1), nlpar="s")
  }

  out <- nlist(formula, data, family, prior)
  return(out)
}
```

The configure_model method should always take the arguments `model`, `data`, and `formula` and return a named list with the formula, the data, the family, and the prior.

4.2 The Signal Discrimination Model (SDM)

The SDM model is defined in the file `R/bmm_model_SDM.R`. The SDM model is a bit more complex than 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_sdmSimple <- function(...) {
   out <- list(
      vars = nlist(),
      info = list(
         domain = 'Visual working memory',
         task = 'Continuous reproduction',
         name = 'Signal Discrimination Model (SDM) by Oberauer (2023)',
         citation = paste0('Oberauer, K. (2023). Measurement models for visual working memory
                           'A factorial model comparison. Psychological Review, 130(3), 841-8
         version = 'Simple (no non-targets)',
         requirements = '- The response variable should be in radians and represent the angul
         parameters = list(
            # mu = 'Location parameter of the SDM distribution (in radians; fixed internally
```

```
            c = 'Memory strength parameter of the SDM distribution',
            kappa = 'Precision parameter of the SDM distribution (log scale)'
        )
    ))
  class(out) <- c('bmmmodel', 'vwm', 'sdmSimple')
  out
}
```

The model definition is similar to the IMM model, but the SDM model does not require non-target variables, so the `vars` list is empty. 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 does not require special data checks beyond it's shared class with other `vwm` models, so we don't need to define a `check_data.sdmSimple` method. The `check_data.vwm` method is defined in the general file `R/helpers-data.R`.

4.2.3 configure_model() methods

The configure_model method for the SDM model is a bit more complex than for the IMM model, as it requires custom stan code. The method is defined as follows:

```
#' @export
configure_model.sdmSimple <- 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("identity","identity", "log"), lb = c(NA, NA, NA),
      type = "real", loop=FALSE,
    )
    family <- sdm_simple

    # prepare initial stanvars to pass to brms, model formula and priors
    stan_funs <- readChar('inst/stan_chunks/sdmSimple_funs.stan',
                          file.info('inst/stan_chunks/sdmSimple_funs.stan')$size)
    stan_tdata <- readChar('inst/stan_chunks/sdmSimple_tdata.stan',
                           file.info('inst/stan_chunks/sdmSimple_tdata.stan')$size)
```

```
    stan_likelihood <- readChar('inst/stan_chunks/sdmSimple_likelihood.stan',
                                file.info('inst/stan_chunks/sdmSimple_likelihood.stan')$size)
    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 the default prior
   # TODO: add a proper prior
   prior <-
     # fix mu to 0 (when I change mu to be the center, not c)
     brms::prior_("constant(0)", class = "Intercept", dpar = "mu")


   # return the list
   out <- nlist(formula, data, family, prior, stanvars)
   return(out)
}
```

Lines 5-10 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 13-22 read the custom stan code from the `inst/stan_chunks` directory. 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.

The rest is the same as for the IMM model.

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 the template 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.

You don't have to add any of the files manually. You can use the function `use_model_template()` to generate all the files with template for all necessary functions, that you can fill. 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/bmm_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/bmm_model_mixture3p.R' for an example

.model_gcm <- function(required_arg1, required_arg2, ...) {
   out <- list(
      vars = nlist(required_arg1, required_arg2),
      info = list(
         domain = '',
         task = '',
         name = '',
         citation = '',
         version = '',
         requirements = '',
         parameters = list()
      ))
   class(out) <- c('bmmmodel', 'gcm')
   out
}

# user facing alias
# information in the title and details sections will be filled in
# automatically based on the information in the .model_gcm()$info
#' @title `r .model_gcm()$info$name`
#' @details `r model_info(gcm(NA,NA))`
#' @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)
#' }
gcm <- .model_gcm



################################################################################!
```

```r
# 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 data-helpers.R. Put here only the methods that are specific to
# the model. See ?check_data for details


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


   # check the data (required)


   # compute any necessary transformations (optional)


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


   data = NextMethod('check_data')

   return(data)
}



##############################################################################!
# 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$vars$required_arg1
```

```
  required_arg2 <- model$vars$required_arg2


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


  # construct the formula
  formula <- formula + brms::lf()


  # construct the family
  gcm_family <- brms::custom_family(
    'gcm', dpars = c(),
    links = c(), lb = c(), ub = c(),
    type = '', loop=FALSE,
  )
  family <- gcm_family

  # prepare initial stanvars to pass to brms, model formula and priors
  stan_likelihood <- readChar('inst/stan_chunks/gcm_likelihood.stan',
     file.info('inst/stan_chunks/gcm_likelihood.stan')$size)
  stan_functions <- readChar('inst/stan_chunks/gcm_functions.stan',
     file.info('inst/stan_chunks/gcm_functions.stan')$size)

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


  # construct the default prior
  prior <- NULL


  # return the list
  out <- nlist(formula, data, family, prior, stanvars)
  return(out)
}



################################################################################!
# 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)


   return(fit)
}
```

Now you have to:

1. Fill the `.model_gcm` function with the appropriate code. This function should return a list with the variables that the model needs and a list with information about the model. The class of the list should be `c('bmmmodel', 'gcm')`. The information list should contain the following elements: `domain`, `task`, `name`, `citation`, `version`, `requirements`, and `parameters`. Rename the required arguments, if you have any, or delete them if you don't. You can see an example in the `bmm_model_sdmSimple.R` file.
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. Fill the `configure_model.gcm` function with the appropriate code. This function should construct the formula, the family, the stanvars, and the prior. 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. You don't have to edit lines 102-109 - loading the stan files and setting up the stanvars is done automatically. If you decide to add more stan files after you created the template, you can edit those lines.

5. Fill the `postprocess_brm.gcm` function with the appropriate code. By postprocessing, we mean changes to the fitted brms model - like renaming parameters, etc. If you don't need any postprocessing, 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-fit_model.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. `fit_model()` 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',{
  skip_on_cran()
  dat <- gen_imm_data(parms = data.frame(c=2,a=0.5,n=0,s=2,kappa=5),
                          ntrial = 2, setsize = 5)
  f <- brms::bf(respErr ~ 1, kappa ~ 1, c ~ 1, a ~ 1, s ~ 1)
  mock_fit <- fit_model(f, dat, IMMfull(setsize=5, non_targets = paste0('Item',2:5,'_rel'), s
  expect_equal(mock_fit$fit, 1)
  expect_type(mock_fit$fit_args, "list")
  expect_equal(names(mock_fit$fit_args[1:4]), c("formula", "data", "family", "prior"))
})
```

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 parameter recovery simulations. We are in the process of establishing guidelines for that.

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.