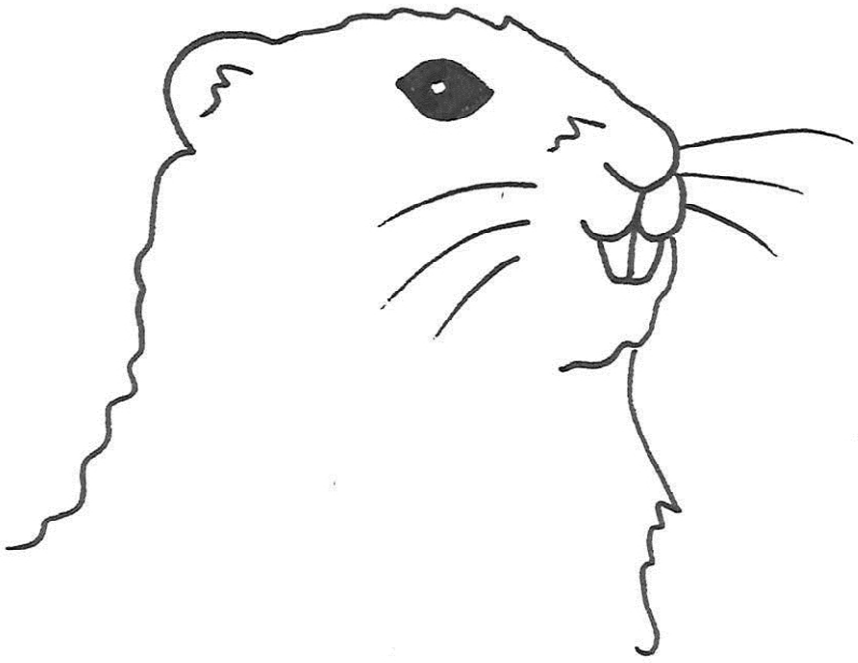
Modular Assessment of Rainfall-Runoff Models Toolbox (MARRMoT)

User Manual

Version 2.-



Luca Trotter
Wouter Knoben
Keirnan Fowler
Margarita Saft
Murray Peel

December 2021

Authors

Luca Trotter[1]

Wouter J. M. Knoben[2]

Keirnan J. A. Fowler[1]

Margarita Saft[1]

Murray C. Peel[1]

[1]Department of Infrastructure Engineering, University of Melbourne, Melbourne, Australia

[2]Centre for Hydrology, University of Saskatchewan, Canmore, Canada

Email contact:

l.trotter@unimelb.edu.au

MARRMoT download

https://github.com/wknoben/MARRMoT

Disclaimer

MARRMoT ("the program") is licensed under the GNU GPL v3.0 license. You should have received a copy of the GNU General Public License along with this program. If not, see https://www.gnu.org/licenses/. Please take note of the following:

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

In practical terms, this means that:

1. The developers do not and cannot warrant that the program meets your requirements or that the program is error free or bug free, nor that these errors or bugs can be corrected;
2. You install and use the program at your own risk;
3. The developers do not accept responsibility for the accuracy of the results obtained from using the program. In using the program, you are expected to make the final evaluation of any results in the context of your own problem.

Contents

1 Introduction

1.1 Place within MARRMoT documentation

This document provides practical guidance for users who want to use or adapt the base Modular Assessment of Rainfall-Runoff Models Toolbox (MARRMoT) code. The following documents give details about various aspects of MARRMoT:

1. **Journal papers**
    a. Trotter et al. (in preparation) present the object-oriented implementation of MARRMoT, its benefits and the technical changes from the previous version;
    b. Knoben et al. (2019) describe the rationale behind MARRMoT development and its original implementation.
2. **Appendix A – Model Descriptions**: this contains descriptions of 47 models currently included in MARRMoT, giving the Ordinary Differential Equations (ODEs) that describe changes in model storage per time, and the constitutive functions that describe the model's fluxes;
3. **Appendix B – Equations table:** describes how the constitutive equations given in the model descriptions are implemented as Matlab code;
4. **Appendix C – Unit Hydrographs table:** describes the currently implemented Unit Hydrograph routing functions.

1.2 Content overview

This manual provides practical guidance for MARRMoT users. It is divided into three parts, with increasing level of detail for different target audiences: Section 2 is intended for the general user, it contains descriptions and examples of how to run a simulation or calibration using the MARRMoT framework; Sections 3 and 4 are targeted at the more advanced user who is interested in understanding how the underlying code works. Whereas section 3 focuses on understanding parts of the framework that should not need to be modified, section 4 focuses on ways to expand and modify the existing framework.

1.3 General toolbox outline

MARRMoT currently provides model code for 47 different hydrological models of the conceptual (bucket) type. Input requirements are standardized across all models, and model output is provided in a standardized way as well.

The framework is set up in a modular fashion with individual *flux files* as the basic building blocks. *Model files* define a class of objects for each model by specifying its inner workings, whereas a *superclass* file defines all the procedures that are common to all models. Figure 1 shows a schematic overview of the toolbox structure.

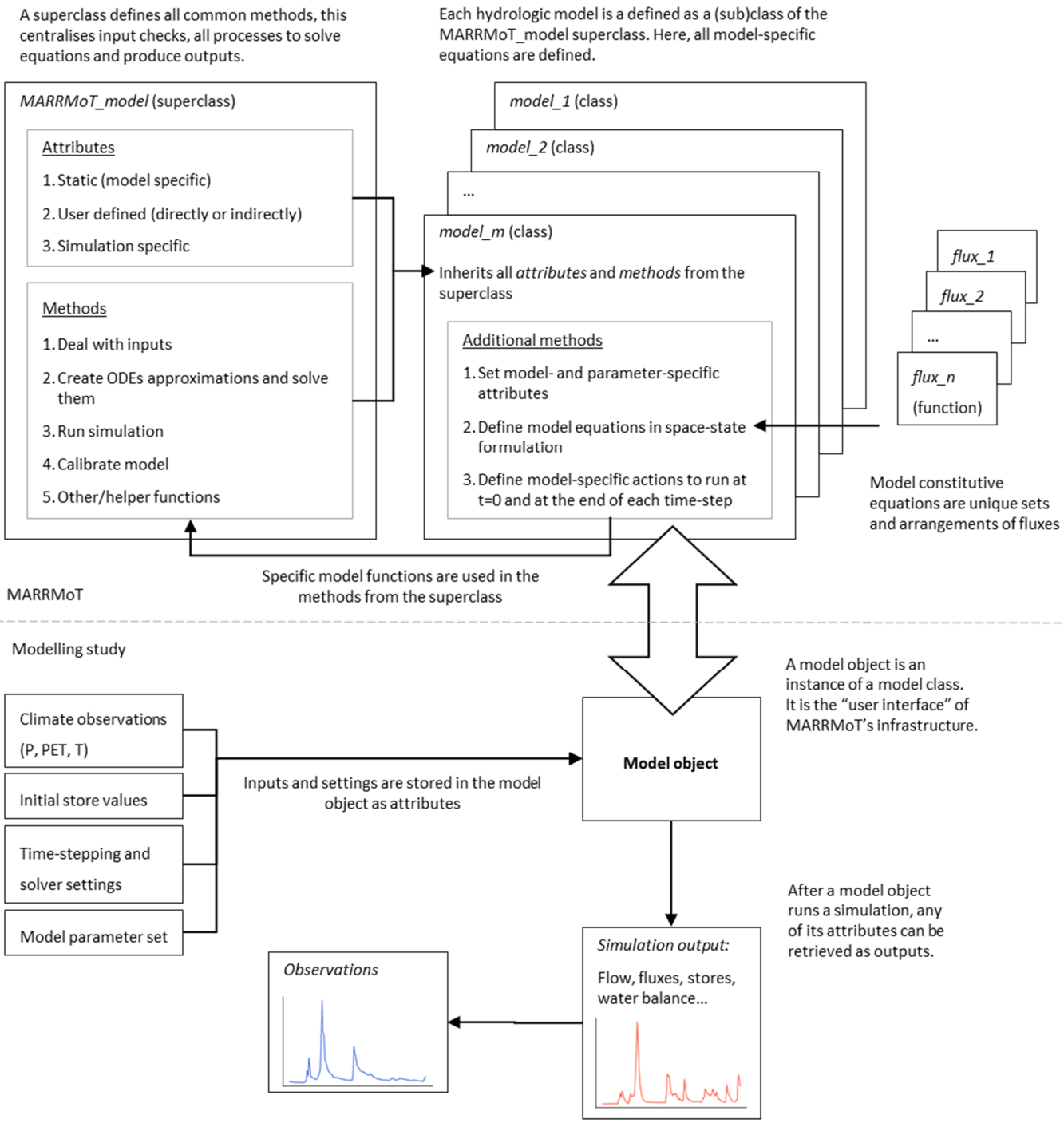


Figure 1: Schematic overview of the MARMMoT framework.

1.4 Folder structure

The main directory (./MARRMoT/) contains the following folders:

- Functions
    - Flux smoothing: contains logistic smoothing functions for storage and temperature thresholds
    - Objective functions: contains a few example objective functions that can be used to compare simulated and observed streamflow.
    - Optimisation functions: contains a wrapper around CMA-ES (Hansen et al., 2003) to use to calibrate models.
    - Solver functions: contains a function to re-run a solver if accuracy of a solution is below a user-specified threshold and a Newton-Raphson solver.
- Models
    - Auxiliary files: contains files that are required within (a) model(s) but are not fluxes or unit hydrographs. Usually used only to keep *model files* more readable.
    - Flux files: contains *flux files*.
    - Model files: contains *model files* and the class definition file of the `MARRMoT_model` *superclass*.
    - Unit hydrograph files: contains unit hydrograph files and helper functions.
- User Manual: contains this manual and files belonging to the examples in this manual.

1.5 Definitions

This section provides definitions for several words/phrases. These are *italicized* in the main text.

Word/phrase	**Definition**
Attribute	In object-oriented programming, an *attribute* (also called property or field) is a piece of data associated with a *class*. This can be static or dynamic, it can be user-defined or automatically calculated. E.g. for each model class, model properties such as number of parameters and number of stores are pre-defined *attributes*; while initial store values and parameter set are user-input.
Class	A class, in object-oriented programming, is the set of definitions of the data format and procedures for a given type of object. A class definition will, in general, contain definition of *attributes* and *methods*. Classes can inherit *attributes* and *methods* from other parent classes (or *superclasses*).
Flux equation	Equation that represents a certain understanding of a hydrological process in mathematical terms. In MARRMoT, *flux equations* are implemented as functions using *flux files*.
Flux file	File that contains code to create a single *flux equation* function.
Method	In object-oriented programming, a *method* (or procedure) is a function that performs operations on an *object* and its *attributes*.
Model descriptions	Document that gives model equations. See **Appendix A.**
Model file	A file unique to a given model, containing its class definition. Each model is defined as a (sub)class of the `MARRMoT_model` *superclass*. Each *model file* specifies which *flux files* are used within the model and the ODE's that describe the change in model storage through time.
Object	An *object* is an instance or a realisation of a *class*. In MARRMoT v2.-, model objects are the user interface of the MARRMoT architecture.
Superclass	In object-oriented programming, a superclass is a class defining *attributes* and/or *methods* that are shared (inherited) by other (sub)classes. In MARRMoT v2.-, each model is defined as a subclass of a single *superclass* called `MARRMoT_model`.

2 Using MARRMoT v2.-

This section gives the details of how to use the framework, including how to run a simulation and how to calibrate a model. It contains an overview of a typical object-oriented workflow as well as details of required inputs, available outputs and useful methods for common applications.

All the explanations given here are exemplified in four workflow examples whose Matlab code can be found in "./MARRMoT/User manual/". For each example, we use 5 years' worth of climate and streamflow data from Buffalo River near Flat Woods, Tennessee, USA, to illustrate. The catchment was randomly selected from those provided within the CAMELS dataset (Addor et al., 2017). The USGS gauge ID for this catchment is 3604000. This data is also included in the MARRMoT repository in the same folder, with the name "MARRMoT_example_data.mat".

While reading this section and familiarising themselves with the framework, the user is advised to read through and follow the example code given. The workflow examples include:

1. Workflow: 1 model, 1 parameter set, 1 catchment
   In this example a version of the HyMOD model (Wagener et al., 2001) is applied to the Buffalo River catchment using a single parameter set. Three different objective functions are calculated to determine the similarity between observed and simulated flows. This example is shown in the file "workflow_example_1".
2. Workflow: 1 model, N parameter sets, 1 catchment

   In this example the HyMOD model is applied to the Buffalo River catchment with N different parameter sets, randomly sampled within the provided HyMOD parameter ranges. This example is shown in the file "workflow_example_2".
3. Workflow: 3 models, 1 random parameter set, 1 catchment

   In this example, the HyMOD model, TANK model (Sugawara, 1995) and Collie1 model (Jothityangkoon et al., 2001) are applied to the Buffalo River catchment. Parameters for each model are randomly taken from the provided parameter ranges. This example is shown in the file "workflow_example_3".
4. Workflow: calibration of 1 parameter set for 1 model and 1 catchment

   In this example, the HyMOD model is calibrated for streamflow simulation in the Buffalo River catchment using a custom Matlab function from the File Exchange. A single parameter set is calibrated using 2 years of data and evaluated using 2 different years of data. MARRMoT's provided parameter ranges are used to constraint the parameter space. This example is shown in the file "workflow_example_4".

As seen in the workflow example files, running a model simulation in MARRMoT will generally involve 3 steps:

1. Creating a model object (section 2.1)
2. Populating user-defined attributes (i.e. model inputs, section 2.2)
3. Running the simulation and retrieving the outputs (section 2.3)

Additionally, in this version of MARRMoT, models also have a dedicated *method* to calibrate their parameters to some observed data. Details of how to calibrate a MARRMoT model are given in section 2.4.

2.1 Model objects

A model object is an instance of a specific model. Once a user has decided which model to use for their simulation, they create a model object by calling the function with the equivalent model's name. The naming convention for all models is the same as in the previously published version of MARRMoT: "m_%n_%name_%pp_%ss".

Where:
%n = model number within the framework
%name = name of the model
%p = number of parameters
%s = number of stores.

For example, GR4J (Perrin et al., 2003) is the 7th model in the framework and is called "m_07_gr4j_4p_2s"; to create a GR4J object and assign it to a variable `m`, the syntax is:

```
m = m_07_gr4j_4p_2s();
```

2.2 Required input

Each model object has a number of attributes that are needed to properly run a simulation (see section **Error! Reference source not found.**), of these only a handful need to be specified by the user. These are equivalent to the inputs needed in the previous version of MARRMoT. They are:

`input_climate` Climate data input. This is expected as a Matlab structure with the following fields:

- example.delta_t
- example.precip
- example.pet
- example.temp

.delta_t is a field within the structure "example" which contains the time step size of the climate data, expressed in units [days]. E.g. daily climate data has Δt = 1 [d], whereas hourly data would have Δt = 1/24 [d].

.precip, .pet, .temp are fields within the *structure* that contain a time series of precipitation, potential evapotranspiration and temperature respectively. Not every model requires temperature data for its calculations. In these cases, a placeholder input can be used instead (e.g. example.temp = NaN;).

Note: the names of these fields are hard-coded in each current *model file*. User input for these models must be defined using these field names.

Alternatively, it is possible to input the climate data as an three-column array, where each column corresponds to .precip, .pet, .temp respectively. This is the format that climate data is stored in as *attribute* of the *model object*. If this input format is used, it is necessary to specify delta_t separately.

`delta_t` If input_climate is specified as a three-column array, delta_t needs to be specified separately. See .delta_t above for data specification.

`S0` Initial values for each model store. This is expected as a vector with a length equal to the number of stores.

`theta` Parameter values for each model parameter. This is expected as a vector with a length equal to the number of stores.

`solver_opts` Settings for the solver and time stepping scheme. This is expected as a Matlab structure with the following fields:

- example.resnorm_tolerance
- example.rerun_maxiter
- example.NewtonRaphson
- example.fsolve
- example.lsqnonlin

.resnorm_tolerance specifies the required accuracy for estimates of new storage values. Ideally, the solver returns an exact solution for each new storage (i.e. $\frac{S_{new}-S_{old}}{\Delta t} - \left(P(t) - Q(S_{new})\right) = 0$ using the Implicit Euler estimate the change in storage S). In practice, the solution is an approximation that is not quite 0. .resnorm_tolerance is the allowed summed, squared deviation from zero [mm]. For *n* stores, resnorm is:

$$resnorm = \sum_{S=1}^{S=n} \left(\frac{S_{n,new} - S_{n,old}}{\Delta t} - \left(P(t) - Q(S_n)\right) \right)^2$$

If the solver has not found an accurate enough solution, the storages are calculated ones more with a more thorough but slower solver. In its current implementation each model runs through three consecutive solvers, starting from `NewtonRaphson`, followed by `fsolve` and `lsqnonlin`. Each subsequent solver is used if the previous one cannot find a solution satisfying .resnorm_tolerance.

.rerun_maxiter specifies the maximum number of iterations that can be spent to recalculate storage values with `fsolve` (`lsqnonlin`), when `NewtonRaphon` (`fsolve`) fails.

.NewtonRaphson, .fsolve, .lsqnonlin are structures containing specific options for each of the three solver functions. .NewtonRaphson can be created with the matlab function `optimset;` while for .fsolve and .lsqnonlin, `optimoptions` should be used.

Default values are available for each of these sets of options, once a model object `m` has been created, the set of all default options can be retrieved with
```
m.default_solver_opts();
```

Note: the names of these fields are hard-coded. User input must be defined using these field names.

Note: it is possible to set only some of the settings, in this case, all the others will be set to their default values. E.g.
```
m.solver_opts.resnorm_tolerance = 1E-3;
```
will set the tolerance to 1E-3 and set all other settings to default.

For a model object `m`, each of these inputs can be specified by setting the *attribute* to the desired value with an assignment statement. Model objects also have a few static *attributes* that should not be modified by the user, but are useful to create parameter sets or initial store values, such as

`numStores` and `numParams`, which contain the number of stores and parameters respectively; and `parRanges`, wich contains a [2 x `numParams`] array with suggested minima and maxima for each parameter. All model *attributes* are accessible (and assignable) using the same syntax as structure fields. For example, to assign empty initial stores and midpoint parameters to a model object `m`, the syntax is:

```
m.S0    = zeros(m.numStores,1);
m.theta = mean(m.parRanges,2);
```

2.3 Simulation output

After all the necessary inputs have been set, there are four different *methods* that can be used to run a simulation, the only difference between them is the number and type of input that they produce. They are:

`run`

Doesn't produce any output, it runs the model and collects all fluxes and store values and all simulation info within the *attributes* of the model object. The list of model *attributes*, all of which can be retrieved after running a simulation, is in section **Error! Reference source not found.**.

`run` is useful to run a quiet simulation and save all results for later use, this can be achieved by saving the model object as is in a .mat file.

`get_output`

Produces output compatible with MARRMoT v1, specifically, it returns 5 optional outputs:

1. `fluxOutput` contains the fluxes 'leaving' the model. It is a Matlab structure with at least the fields .Q and .Ea, these contain timeseries of total simulated streamflow and evapotranspiration respectively. These timeseries have the same timestep of the cimate input and in most cases are the sum of various internal fluxes.

   In several cases, other model-specific fields are included in this output structure representing external fluxes such as groundwater exchange or abstraction. By convention all fluxes leaving the model (such as Q and Ea) are positive, and all fluxes entering the model are given a negative sign.

2. `fluxInternal` contains all model fluxes, given as a structure with model-specific fields. Each field contains a time series of flux values during the simulation period. These are essentially all the fluxes used in the model including the ones used to calculate the fields of `fluxOutput`. See the model descriptions in **Supporting Material S2** for schematics that show the flux names.

3. `storeInternal` contains storage values throughout the simulation. It is also a structure with a number of fields equal to the number of stores in the model. Currently, all models include at least one store and hence at least one field .S1, which contains a time series of storage values of the first model store in the same time resolution as the climate input. The field name is always 'S' followed by a number. If the models contains more than one store, subsequent stores are named .S2, .S3, etc.

4. `waterBalance` is the sum of all incoming and outgoing fluxes and changes in storage. This is approximately zero in a well-performing model. When this output is requested, a summary showing the main fluxes and storage changes is also printed to the screen.
5. `solverSteps` is a structure containing all information about the equation solver for each timestep. It contains three fields: .resnorm, .solver, .iter. These each contain a timeseries indicating, for each timestep, the quality of the numerical solution to the ODEs, which solver was used and how many iterations were run, respectively.

`get_streamflow`	Only returns the timeseries of total simulated streamflow (see point 2 above).
`check_waterbalance`	Only returns the water balance, printing a summary of fluxes and storage changes to screen (see point 4 above).

All of these methods don't require any input, but they can each take four optional inputs: `input_climate`, `S0`, `theta` and `solver_opts`. The expected format of these inputs is described in section 2.2 above. Using these optional inputs entails that it is possible to run a simulation using a syntax nearly identical to the syntax used in MARRMoT v1. After creating the model object, a simulation can be run directly with:

```
[fluxOutput, fluxInput, storeInternal, waterBalance] = ...
                  m.get_output(input_climate, S0, theta, solver_opts);
```

`get_output`, `get_streamflow` and `check_waterbalance` call `run` under the hood, but only if the simulation hasn't run before and no new inputs are given. This means that it is possible to use each of these methods on an already-run model object to just obtain the outputs in their neat format, without needing the simulation to run again. *Model objects* contain an *attribute* `status` indicating whether they already ran (`status = 1`) or not (`status = 0`) .

Note that after a *model object* has run a simulation using any of these *methods*, any of the object's *attributes* can be retrieved for computation, this includes a raw array of all fluxes and stores (attributes `fluxes` and `stores` respectively) as well as a structure equivalent to solverSteps in point 5 above under the attribute `solver_data`. All attributes retrievable from a model object are listed in **Error! Reference source not found.**.

2.4 Model calibration

While the `get_streamflow` *method* described above can be useful to calibrate a model using any optimiser and any objective function, MARRMoT *model objects* have a specific *method* to perform a calibration, called `calibrate`. The user is required to input the *attributes* `input_climate`, `delta_t`, `S0` and `solver_opts` as described in section 2.2 ahead of calling the `calibrate` *method*. Additionally, the following inputs are necessary when calling the method:

`Q_obs`	Vector of observed streamflow values.
`cal_idx`	Indices indicating timesteps to include in the calibration period. Can either be a numerical vector of indices or a boolean vector. If an empty vector is entered the entire series is used for calibration.

`optim_fun` Function to use to optimise the objective function. It can be given as string or a function handle. The function it directs to must require the following inputs:

1. `fun` – function to optimise (i.e. objective function);
2. `x0` – initial estimate;
3. `options` – structure of options to the optimiser.

And it must provide the following outputs:

1. `x` – solution of the optimisation;
2. `fval` – value of the objective function at `x`;
3. `exitflag` – integer indicating reason the optimiser stopped (see specification of specific optimiser function);
4. `output` – additional information about the optimisation (see specification of specific optimiser function).

Note that these are the same inputs and outputs of most of Matlab's proprietary optimisers such as `fminsearch`.

`par_ini` Initial parameter set. It will be used as `x0` input to the `optim_fun` (see above). If empty the midpoint of the parameter ranges is used as starting point.

`optim_opts` Options to the optimiser. It will be used as the `options` input to the `optim_fun` (see above). It will usually be a structure, but its format will depend on the specific optimiser chosen.

`of_name` Objective function to optimise for. This can be a string or a function handle. It can be one of the objective functions included in the MARRMoT repository (in the folde "./MARRMoT/Functions/Objective functions") or it can be any user-specified function, provided the function requires the following inputs:

1. `obs` – vector of observed values;
2. `sim` – vector of simulated values;
3. `idx` – vector of indices of steps in `obs` and `sim` to use in calculating the objective function (see `cal_idx` above);
4. `varargin` – any additional input.

And it must provide the following outputs:

1. `val` – fitness value, i.e. value of the objective function;
2. `idx` – vector of indices used to calculate the objective function.

`inverse_flag` Boolean flag (`1 = true`, `0 = false`) indicating whether the objective function in of_name should be inverted before optimisation. Most optimisers are minimisers only, therefore objective functions such as KGE and NSE need to be inverted before optimisation.

`varargin` Additional arguments to the objective function. Will be passed to `of_name` as `varargin`.

Once the calibrate method is called, it will call the optimiser chosen and provide its outputs. Therefore, the outputs of calibrate are equivalent to those of the optimiser:

`par_opt` Optimal parameter set. I.e. the result of the optimisation (`x`).

`fval`	Value of the objective function with parameter set `par_opt`. If the objective function was inverted (i.e. `inverse_flag = 1`), it is inverted back before calculating `fval`.
`stopflag`	Integer indicating the exit status of the optimiser (i.e. `exitflag`). The meaning of specific values will depend on the optimiser chosen, in general a positive value indicates a successful optimisation.
`output`	Additional information about the optimisation process (such as number of iterations, algorithm used, etc.). The specific format will depend on the optimiser chosen.

After creating a model object `m` (see 2.1) and adding the required *attributes* (i.e. `input_climate`, `delta_t`, `S0` and `solver_opts`, as described in section 2.2). The syntax to call the `calibration` *method* is the following.

```
[par_opt, of_cal,...
 stopflag, output] = m.calibrate(Q_obs, cal_idx,...
                                 optim_fun, par_ini, optim_opts,...
                                 of_name, inverse_flag, varargin);
```

After obtaining `par_opt` through calibration, the model object is ready to run a simulation with the optimised parameter set. Given that `input_climate`, `delta_t`, `S0` and `solver_opts` are already set, the user will only need to run

```
m.run([],[],par_opt);
```

or alternatively

```
m.theta = par_opts; m.run();
```

3 Understanding MARRMoT v2.-

This section contains a more thorough description of the structure and functioning of the MARRMoT framework, it is indended for the more advanced user who is interested in understanding how the framework works.

The core of MARRMoT v2.- is the *class* definition files, these include the definition of the `MARRMoT_model` *superclass* as well as the *model files*, defining the *classes* of each individual model. Model *classes* are defined as subclasses of the `MARRMoT_model` *superclass*, meaning that they inherit all of its *attributes* and *methods*. In this section, we start by providing a comprehensive list of the *methods* and *attributes* defined in the *superclass* as well as in the *model files* (section 3.1), and follow up with a thorough run through the code of the *superclass* definition which highlights the functioning of MARRMoT models and the interactions between the superclass *methods* and the model-specific *methods*.

3.1 Class definition files

3.1.1 The **`MARRMoT_model`** *superclass*

The `MARRMoT_model` *superclass* is defined in the file "./MARRMoT/Models/Model files/MARRMoT_model.m" as a subclass of the MATLAB handle class (see 'handle class' in MATLAB's documentation). It contains the definition of all *attributes* and *methods* that are shared amongst all MARRMoT models.

Each MARRMoT model has three sets of *attributes* defined in the *superclass*:

1. Model-static attributes, set for each model in its own class definition.
2. Simulation-static attributes, set by the user directly or indirectly for a specific simulation.
3. Dynamic attributes, created and updated automatically throughout a simulation.

Details of each attribute in each of these groups are given in the table below.

Attribute	Description	Type, size
Model-static Attributes		
`numStores`	Number of model stores	Integer, [1,1]
`numFluxes`	Number of model fluxes	Integer, [1,1]
`numParams`	Number of model parameters	Integer, [1,1]
`parRanges`	Default parameter ranges	Double, [`numParams`,2]
`JacobPattern`	Pattern of the Jacobian matrix of the model's ODEs	Boolean, [`numStores`,`numStores`] (See 4.1.2-6 for details)
`StoreNames`	Names of the stores	String, [1,`numStores`]
`FluxNames`	Names of the fluxes	String, [1,`numFluxes`]
`FluxGroups`	Grouping of fluxes leaving the model	Struct (See 4.1.2-6 for details)
`StoreSigns`	Signs to assign stores in the water balance (-1 for a deficit store)	Integer, [1,`numStores`]
Simulation-static attributes		
`theta`	Parameter set	Double, [`numParams`,1]
`delta_t`	Time step in days	Double, [1,1]
`S0`	Initial store values	Double, [`numStores`,1]
`input_climate`	Rainfall, PET and temperature input for model simulation	Double, [`t_end`,3]

`solver_opts`	Options for numerical solving of ODEs	Struct (see 2.2 for details)
`store_min`	Mimimum values of stores	Double, [1,`numStores`]
`store_max`	Maximum values of stores	Double, [1,`numStores`]
Dynamic attributes		
`t`	Current timestep	Integer, [1,1]
`fluxes`	Fluxes for this simulation	Double, [`t_end`, `numFluxes`]
`stores`	Store values for this simulation	Double, [`t_end`, `numStores`]
`uhs`	Unit hydrographs and still-to-flow fluxes	Cell array (see point 3 in 4.1.2-7 for more detail)
`solver_data`	Step-by-step info of ODEs solver output	Struct (see 2.3 for details)
`status`	Flag to indicate if simulation has run (1) or not (0)	Boolean, [1,1]

The superclass also defines 16 methods which perform all the operations that are common to all models, these include:

1. Checking user-specified inputs;
2. Initialising models (i.e. setting up attributes such as unit hydrographs or store limits based on user inputs);
3. Solving model equations;
4. Running simulations and producing outputs; and
5. Calibrating a model.

The full list with their description is contained in the table below.

Method	**Description**
Input checks	

`set.delta_t`
`set.theta`
`set.input_climate`
`set.S0`
`set.solver_opts`
	These methods provide input checking to make sure that user input values are consistent with what is expected and store the input as object attributes.
	`set` methods are invoked when an attribute is assigned (e.g. `obj.theta = [1,2,3,4]` will invoke the method `set.theta`).

`default_solver_opts`	Returns default set of solver options.
`add_to_def_opts`	Adds user defined options to the default set.
Model initialisation	
`init_`	Runs before each model run to initialise store limits, auxiliary parameters and unit hydrographs based on parameter set. It calls `init`, which is the model-specific initialiser (see 3.1.2).
`reset`	Resets a model object, removing all simulation-specific attributes.
Equation solving	
`ODE_approx_IE`	Produces ODE approximations using Implicit Euler stepping scheme.
`solve_stores`	Produces values of stores and fluxes for an individual timestep by solving the ODE approximations defined above.
`rerunSolver`	Restarts a root-finding solver with different starting points.
Simulation (see 2.3)	
`run`	Runs a model simulation.

`get_output`	Produces output consistent with previous MARRMoT versions. Runs the simulation if it hasn't run already.
`check_waterbalance`	Prints the water balance to screen and returns its value.
`get_streamflow`	Only returns streamflow, useful to calibrate a model with an external optimiser. It runs the simulation if it hasn't run already.
Calibration (see 2.4)	
`calibrate`	Uses a specified optimiser and objective function to find an optimal parameter set.

3.1.2 Model files

Each model file is named according to the MARRMoT naming convention (see 2.1) and are contained in the folder "./MARRMoT/Models/Model files". For example, the file containing the definition for GR4J model class is called "m_07_gr4j_4p_2s.m".

Each individual model class is defined as a subclass of the superclass, meaning it inherits all of the *attributes* and *methods* described in section 3.1.1 above. Model files do not require the definition of any additional *attributes*, but it is possible to define additional model specific *attributes*, for example to store auxiliary parameters.

Each model, however requires the definition of four model-specific methods in its class definition file:

constructor — The constructor method is defined using the same name as the class (i.e. `m_07_gr4j_4p_2s` in the case of GR4J) and it is run automatically every time a new object for a given class is created. It is used to set up all the model-static *attributes* of the model object. Note that while these *attributes* are defined in the *superclass* (since every model has these attributes), these are populated (i.e. their values are assigned) in the *model files*, since their values depend on the model chosen.

`init` — It is called by the model-generic initialiser `init_` and it is run once at the beginning of a simulation. It performs all the operations needed to set the simulation ready to run. Typical operations included in the `init` method are: set store maxima and minima based on parameters, initiate unit hydrographs based on parameters and set auxiliary parameter sets.

`model_fun` — This is the *method* defining the model equations in space-state formulation. It takes one input S, a vector of storage values of size [1,`numStores`], and outputs two vectors:

- `dS` – vector of changes in store levels [1,`numStores`]; and
- `fluxes` – vector of internal model fluxes [1,`numFluxes`].

On top of the explicit input (S), the method accesses all model *attributes*, including the parameter set `theta`, the current timestep `t`, the climate forcing in `input_climate`, etc.

More details on the specific format of `model_fun` is given in section 4.1.2-8 below.

`step` — Finally, the `step` method is run at the end of every timestep. It is currently only used to update still-to-flow fluxes from unit hydrographs and step fluxes.

3.2 Detailed code descriptions (v2.1)

In this section, we provide a detailed description of relevant parts of the code in the `MARRMoT_model` class definition file, this file should in general not be modified by the user. Detailed code explanation of the *methods* in the *model files*, as well as flux function files, unit hydrograph files and objective function files are given in section 4. Note that the pieces of code described here are specific to MARRMoT v2.1, numerical ODEs solving works differently in v2.0 and line numbers might differ for other pieces of code in that version.

3.2.1 Numerical ODEs solving

The procedures used to numerically solve stores' differential equations at every step are contained in the `solve_stores` methods, ODEs are solved using an Implicit Euler time-stepping scheme, which is defined in the `ODE_approx_IE` method, which is defined in lines 139-146 of the `MARRMoT_model` superclass file.

`ODE_approx_IE` takes a vector of storage values S as input, and outputs an approximation error `err`, the objective of solving model equations is to find values of S so that `err = 0`. If S is given as a row vector it is here transformed to a column vector (line 140).

```
139  function err = ODE_approx_IE (obj, S)
140      S = S(:);
```

Changes in store values are calculated using model equations from the storage values at the current timestep (line 141) and old storage values are retrieved from the model attributes based on the current timestep (lines 142-144).

```
141      delta_S = obj.model_fun(S);
142      if obj.t == 1; Sold = obj.S0(:);
143      else; Sold = obj.stores(obj.t-1,:)';
144      end
```

The approximation error `err` is computed using the Implicit Euler numerical scheme formula.

```
145      err = (S - Sold)/obj.delta_t - delta_S';
146  end % closes function opened on line 139
```

The `solve_stores` method contains the iterative algorithms used to solve the ODE approximation defined above, it is defined in lines 149-211 of the `MARRMoT_model` superclass file. This method takes old store values as input (`Sold`, store values at `t-1`) and outputs new store values `Snew` as well as information about the quality of the approximation `resnorm`, the solver used `solver` and the number of iterations required `iter`.

```
149  function [Snew, resnorm, solver, iter] = solve_stores(obj, Sold)
```

Solver options are extracted from the model attributes (line 151) and the tolerance is adjusted in case store values are very small to improve the quality of the solution (line 156).

```
151      solver_opts = obj.solver_opts;
...
156      resnorm_tolerance = solver_opts.resnorm_tolerance *...
                                          min(min(abs(Sold)) + 1E-5, 1);
```

In order to solve the ODE approximations, this method uses three different solvers, in order of complexity `NewtonRaphson` (included in the MARRMoT repository in the file called "./MARRMoT/Functions/Solver functions/NewtonRaphson.m"), `fsolve` and `lsqnonlin`, which are

part of Matlab's optimisation toolbox. First three empty vectors are created to store the solutions, residual values and number of iterations for each solver (lines 161-163).

```
161      Snew_v    = zeros(3, obj.numStores);
162      resnorm_v = Inf(3, 1);
163      iter_v    = ones(3,1);
```

Then, we run the first solver (`NewtonRaphson`) on the `ODE_approx_IE` function to obtain a temporary solution `tmp_Snew` (lines 166-169), we evaluate the norm of the residuals associated with this solution `tmp_resnorm` (line 170) and store both of those in the relevant vector we just created (lines 172-173).

```
166      [tmp_Snew, tmp_fval] = ...
167                              NewtonRaphson(@obj.ODE_approx_IE,...
168                                            Sold,...
169                                            solver_opts.NewtonRaphson);
170      tmp_resnorm = sum(tmp_fval.^2);
171
172      Snew_v(1,:)  = tmp_Snew;
173      resnorm_v(1) = tmp_resnorm;
```

If the norm of the residual of this temporary solution is above the tolerance (line176), we use `fsolve` to find a better solution. `fsolve` is called using the helper method `rerunSolver` (which is defined in lines 216-300 of the *superclass* itself). `rerunSolver` will attempt to find new solutions for the current time step up to `solver.rerun_maxiter` times, and restarts the solving procedure from different initial guesses each time. This provides better chances of finding a solution with the requested accuracy. Once a new temporary solution is found with `fsolve` and `rerunSolver` (lines 177-180), the norm of the residual is calculated (line 182) and the solution, the norm and the number of iterations are stored in the relevant vectors (line 184-186).

```
176      if tmp_resnorm > resnorm_tolerance
177          [tmp_Snew,tmp_fval,~,tmp_iter] = ...
178                          obj.rerunSolver('fsolve',...
179                                          tmp_Snew,...
180                                          Sold);
181
182          tmp_resnorm = sum(tmp_fval.^2);
183
184          Snew_v(2,:)  = tmp_Snew;
185          resnorm_v(2) = tmp_resnorm;
186          iter_v(2)    = tmp_iter;
```

If the norm of the residuals of the `fsolve` solution is still above the tolerance (line 189), a new solution is searched for using `lsqnonlin`. Again, this is called within the `rerunSolver` wrapper (lines 190-193). As before, the norm of the residual is calculated and the solutions are saved (lines 195-199).

```
189          if tmp_resnorm > resnorm_tolerance
190              [tmp_Snew,tmp_fval,~,tmp_iter] = ...
191                              obj.rerunSolver('lsqnonlin',...
192                                              tmp_Snew,...
193                                              Sold);
194
195              tmp_resnorm = sum(tmp_fval.^2);
196
197              Snew_v(3,:)  = tmp_Snew;
198              resnorm_v(3) = tmp_resnorm;
```

```
199             iter_v(3)    = tmp_iter;
200
201         end
202     end % closes if on line 169
```

Finally, out of the solutions in Snew_v, the best is chosen (i.e. the one with the lowest residual norm in resnorm_v) and the outputs are produced.

```
205     [resnorm, solver_id] = min(resnorm_v);
206     Snew = Snew_v(solver_id,:);
207     iter = iter_v(solver_id);
208     solvers = ["NewtonRaphson", "fsolve", "lsqnonlin"];
209     solver = solvers(solver_id);
210
211 end % closes function opened on line 142
```

3.2.2 Model simulation

A model simulation is run using the run method. This is defined in the MARRMoT_model class definition file in lines 307-354. The run method takes four optional inputs and doesn't return any output (lines 307-311).

```
307 function [] = run(obj,...
308                   input_climate,...
309                   S0,...
310                   theta,...
311                   solver_opts)
```

If any of the inputs is provided, these are stored as model attributes (lines 313-324). Note that if the model already has a given attribute, it will be overwritten.

```
313     if nargin > 4 && ~isempty(solver_opts)
314         obj.solver_opts = solver_opts;
315     end
316     if nargin > 3 && ~isempty(theta)
317         obj.theta = theta;
318     end
319     if nargin > 2 && ~isempty(S0)
320         obj.S0 = S0;
321     end
322     if nargin > 1 && ~isempty(input_climate)
323         obj.input_climate = input_climate;
324     end
```

The init_ method is called to set up parameter-dependent model attributes (line 329)

```
329     obj.init_();
```

The number of timesteps of the simulation is extracted from the input_climate attribute (line 331) and a loop is set up (line 333). At every timestep, the attribute t is updated with the value of the current timestep (line 334) and old store values are extracted from the model attributes (lines 335-337).

```
331     t_end = size(obj.input_climate, 1);
332
333     for t = 1:t_end
334         obj.t = t;
335         if t == 1; Sold = obj.S0(:);
336         else; Sold = obj.stores(t-1,:)';
337         end
```

Next, the ODEs of the stores are solved at this timestep using the `solve_store` method described above in section 3.2.1 (line 339).

```
339          [Snew,resnorm,solver,iter] = obj.solve_stores(Sold);
```

Using the store values resulting from the solution (`Snew`), model equations are used to calculate changes in storage and flux values at this timestep (line 341), these are used to update the attributes `fluxes` and `stores` at this timestep (lines 343-344). The `solver_data` attribute is also updated with the outputs of the `solve_store` method (lines 346-348).

```
341          [dS, f] = obj.model_fun(Snew);
342
343          obj.fluxes(t,:) = f * obj.delta_t;
344          obj.stores(t,:) = Sold + dS' * obj.delta_t;
345
346          obj.solver_data.resnorm(t) = resnorm;
347          obj.solver_data.solver(t) = solver;
348          obj.solver_data.iter(t) = iter;
```

Before moving on to the next step, the model-specific `step` method is called (line 350).

```
350          obj.step();
351      end % closes for loop on line 149
```

Finally, after the final timestep (i.e. at the end of the simulation) the method status is set to 1 to indicate that the model has run with the current attributes (line 353).

```
353          obj.status = 1;
354      end % closes function opened on line 223
```

3.2.3 Model calibration

The procedure used to calibrate a model is defined in the `calibrate` method of the `MARRMoT_model` superclass (lines 466-531). This method takes in a variable number of inputs (at least 5) and returns up to 4 outputs (lines 466-478), specifications of these inputs and outputs have been described in section 2.4.

```
466  function [par_opt
467            of_cal,...
468            stopflag,...
469            output] = ...
470                      calibrate(obj,...
471                                Q_obs,...
472                                cal_idx,...
473                                optim_fun,...
474                                par_ini,...
475                                optim_opts,...
476                                of_name,...
477                                inverse_flag,...
478                                varargin)
```

As already stated, before running `calibrate`, the user should input `input_climate`, `S0`, `delta_t` and `solver_opts` manually. Whether these attributes have been set is checked and an error is returned if they aren't (lines 480-485)

```
480      if isempty(obj.input_climate) || isempty(obj.delta_t) ||...
481              isempty(obj.S0) || isempty(obj.solver_opts)
482          error(['input_climate, delta_t, S0 and solver_opts '...
483                 'attributes must be specified before calling '...
```

```
484                 'calibrate.']);
485     end
```

Input `cal_idx` is optional. If omitted, the entire length of `Q_obs` is used for the calibration (lines 489-491). Additionally, `cal_idx` can be in the form of an array of indices or an array of Booleans, if it is in the latter format, it is converted in the former one (line 495). Finally the sequence to run the simulations on is trimmed up to the last timesteps in the calibration sequence (lines 496-498)

```
489     if isempty(cal_idx)
490         cal_idx = 1:length(Q_obs);
491     end
...
495     if islogical(cal_idx); cal_idx = find(cal_idx); end
496     input_climate_all = obj.input_climate;
497     obj.input_climate = input_climate_all(1:max(cal_idx),:);
498     Q_obs = Q_obs(1:max(cal_idx));
```

Input `par_ini` is also optional, if omitted, the calibrations starts at the midpoint of the parameter ranges in the `parRanges` attribute (lines 502-504).

```
502     if isempty(par_ini)
503         par_ini = mean(obj.parRanges,2);
504     end
```

In lines 508-5011, a helper function is defined to calculate the fitness as a function of a parameter set. This function first runs the model with a given parameter set and extracts simulated streamflow (line 509), then calls the objective function in `of_name` to evaluate the parameter fitness (line 510).

```
508     function fitness = fitness_fun(par)
509         Q_sim = obj.get_streamflow([],[],par);
510         fitness = (-1)^inverse_flag*feval(of_name, Q_obs,...
                                        Q_sim, cal_idx, varargin{:});
511     end
```

The chosen optimiser (`optim_fun`), is then called to optimise this helper fitness function (lines 513-520).

```
513     [par_opt,...
514      of_cal,...
515      stopflag,...
516      output] = ...
517                feval(optim_fun,...
518                      @fitness_fun,...
519                      par_ini,...
520                      optim_opts);
```

To conclude, the value of the objective function returned by the optimiser is inversed back, if needed (line 523) and the `input_climate` attribute is restore to the original entire sequence (line 527)

```
523     of_cal = (-1)^inverse_flag * of_cal;
...
527     obj.input_climate = input_climate_all;
528 end % closes function opened on line 382
```

4 Editing MARRMoT v2.-

4.1 Creating a new model

This section shows how a new model can be created to fit within MARRMoT. The current 47 models are all created based on the following generalized principles:

- The only climate inputs are precipitation, temperature and potential evapotranspiration;
- Within the *model files*, no spatial discretization is applied (i.e. the *model file* is spatially lumped, although spatial discretization could be created by the user outside the *model file*); and
- The time step size can be specified by the user, but the internal *model file* computations use [mm/d] as the base unit.

For clarity, we illustrate the process of creating a new model using an example. We assume that the new model created in this section is built according to certain assumptions of how a particular catchment functions (i.e. on some perceptual model of the catchment). Justifying these assumptions is outside the scope of this guide. This section is intentionally divided into many small sub-sections, to make it easier to follow all steps. The headers of each sub section can be used as a check list.

4.1.1 Create the model description

Creating a new model starts with a model description: a model schematic and the model equations.

1. **Create a model schematic based on assumptions about the catchment**

The model schematic shows the behaviour the model is intended to simulate. Creating a clear model schematic helps identify assumptions and model equations.

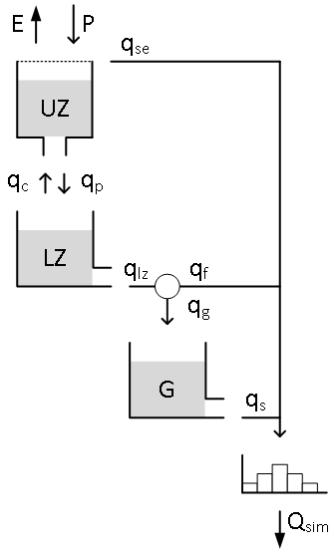


Figure 2 shows the schematic of the model used for this example, the assumptions in this model are as follows:

- There is no snowfall
- Precipitation enters the upper zone
- Evaporation is taken from the upper zone
- Saturation excess surface flow occurs when the upper zone is full
- Percolation drains the upper zone and refills the lower zone
- Capillary rise drains the lower zone and refills the upper zone
- Lower zone drainage occurs while water is available
- Part of the lower zone drainage is fast flow
- The remainder of lower zone drainage goes to groundwater
- Groundwater generates slow flow
- Surface runoff, fast flow and slow flow combine and are sent through a triangular routing scheme to form Q_{sim}

Figure 2: Model schematic

2. **Specify the model Ordinary Differential Equations (ODEs)**

Model schematics are a useful aid in the next step: defining the ODEs that specify the changes in model storages. In practical terms, this means identifying which fluxes enter and exit each store. This model has three stores, so three ODEs are needed, these are shown in equations 4.1 to 4.3 below.

$$\frac{dUZ}{dt} = P + q_c - E - q_{se} - q_p \tag{4.1}$$

$$\frac{dLZ}{dt} = q_p - q_{lz} - q_c \tag{4.2}$$

$$\frac{dG}{dt} = q_g - q_s \tag{4.3}$$

3. Specify the constitutive functions that define the model fluxes

Next, the constitutive equations that describe the individual fluxes need to be defined. These equations are based on a conceptual understanding of how the catchment functions. For example, if there is reason to believe that actual evaporation rates decline when the available soil moisture reduces, the flux equation *E* in our model should reflect this. A model must define constitutive equations for each of their fluxes, these equations will generally be functions of the input climate (i.e. P, PET and/or T), store values, parameters of the models and other fluxes. Some fluxes may also be defined using routing delays schemes (i.e. unit hydrographs). Equations 4.4 to 4.12 below are the constitutive equations for each flux of this model. In equation 4.12, *triangular* is a routing delay function of base *d*.

$$q_{se} = \begin{cases} P, & if\ UZ = UZ_{max} \\ 0, & otherwise \end{cases} \tag{4.4}$$

$$E = E_p \frac{UZ}{UZ_{max}} \tag{4.5}$$

$$q_c = c_{rate}\left(1 - \frac{UZ}{UZ_{max}}\right) \tag{4.6}$$

$$q_p = p_{rate} \tag{4.7}$$

$$q_{lz} = k_{lz} * LZ \tag{4.8}$$

$$q_g = \alpha * q_{lz} \tag{4.9}$$

$$q_s = k_g * G \tag{4.10}$$

$$q_f = (1 - \alpha) * q_{lz} \tag{4.11}$$

$$Q = (q_f + q_s + q_{se}) * triangular(d) \tag{4.12}$$

Based on these equations, this model has 7 parameters: maximum capillary rise rate c_{rate} [mm/d], maximum upper zone storage UZ_{max} [mm], constant percolation rate p_{rate} [mm/d], lower zone runoff coefficient k_{lz} [d-1], fraction of lower zone runoff to groundwater α [-] , groundwater runoff coefficient k_g [d-1], and routing delay *d* [d].

4.1.2 Create the *model file*

The easiest way to create a new model file is to start by copy-pasting the template model file ("./MARRMoT/Models/Model files/m_00_template_5p_2s.m") and making the necessary adjustments from there. The following steps outline the necessary changes to be made.

4. Define model name

First, the name of the model needs to be edited in three places:

1. The name of the file;
2. The name of the class (on line 1); and
3. The creator method (on line 16).

Model names should follow MARRMoT's naming convention: "m_%n_%name_%pp_%ss".

Where:

%n = model number within the framework
%name = name of the model
%p = number of parameters
%s = number of stores.

Our model has 3 stores and 7 parameters, we will name it "m_nn_example_7p_3s". We name the new model file "./MARRMoT/Models/Model files/ m_nn_example_7p_3s.m" and edit the file itself as follows (changes are in red).

```
1    classdef m_nn_example_7p_3s < MARRMoT_model
...
16          function obj = m_nn_example_7p_3s()
```

5. Define model-specific *attributes* (if needed)

If the model needs any attribute beyond the ones in section 3.1.1, these should be defined in lines 4—5 between the keywords `properties` and `end`. Currently, this is only used to define eventual auxiliary parameters where needed. For this model, this is not necessary, so no modifications are done. The interested user can see the *model file* "m_33_sacramento_11p_5s.m" for an example of this.

6. Populate model-static *attributes* using the creator *method*

The class creator method is used to populate model-static *attributes* (i.e. assign their values). This method is called automatically when a new object of a class is created and it bears the same name as the class. In the example file we are modifying, it starts on line 16. Here the values of the following attributes need to be assigned:

- `numStores`, `numFluxes`, `numParams`: scalar integers of numbers of stores, fluxes and parameters respectively. In our example 3 (same as the number of ODEs), 9 (as the number of constitutive equations) and 7.
- `JacobPattern`: pattern of the Jacobian matrix, this is a square array of zeros (FALSE) and ones (TRUE), with as many rows and columns are there are stores and ODEs in the model. Each element in the matrix responds to the question of whether the row-numbered ODE depends on the column-numbered store. For example, our first ODE (eq. 4.1) depends on the first store (*UZ*) via *E* and q_c (see eqq. 4.5 and 4.6) and on the second store (*LZ*) via q_c again (this is because even in *LZ* does not appear in eq. 4.6, current storage in *LZ* is the upper bound for

q_c). The first ODE does not depend on the third store, hence, the first row of `JacobPattern` is [1 1 0]. With the same logic, we generate the other two rows as:

	UZ	LZ	G
ΔUZ	1	1	0
ΔLZ	1	1	0
ΔG	0	1	1

Note that it is possible to not assign any value to `JacobPattern`, this is equivalent to filling out a matrix of ones. While this doesn't affect the output of a simulation, defining the pattern of the Jacobian matrix speeds up computation.

- `parRanges`: array of suggested parameter ranges. This should be a 2-column array with the first column containing parameter minima and the second column containing parameter maxima. This attribute is never used internally by MARRMoT and is only a reference for other users.
- `StoreNames` and `FluxNames`: two 1-row arrays of strings containing the names of stores and fluxes, these are used when producing outputs (e.g. using the method `get_output`). While any name can be given to both fluxes and stores, here we maintain the convention of MARRMoT v1.x of using the letter S and a progressive integer for naming stores. Outside of output production, MARRMoT uses indices to identify and store fluxes (and stores). It is therefore paramount that the order of fluxes and stores defined here is maintained also in the definition of the model equations (see…). Here, we use the same order as the fluxes and stores are defined in the constitutive equations (4.4 to 4.12) and ODEs (4.1 to 4.3) respectively.
- `FluxGroups`: a structure to define how the fluxes should be grouped for the output. It should contain at least two fields: .Q, containing the indices of the fluxes that contribute to streamflow (in our case 9, i.e. flux *Q*); and .Ea, containing the indices of the fluxes that contribute to actual evapotranspiration (in our case, that is flux *E*, which has index 2). Additional fields must be added to cover all fluxes leaving (or entering) the model in order to make sure that water balance calculations are correct; for these, the sign of the indices indicate whether the flux (when positive) is leaving (+) or entering (-) the model. See file "m_07_gr4j_4p_2s.m" for an example of this: here flux 13 is a groundwater exchange, a positive exchange enters the model (opposite to a positive streamflow or ET, which exit the model), therefore on line 30 we assign `obj.FluxGroups.Exchange = -13;`.
- `StoreSigns`: an array with as many elements as there is stores to indicate their signs, this is used to calculate the water balance. A 1 indicates a "regular" store, whereas a -1 indicates a deficit store (i.e. its values will be inverted in the water balance calculations). If all the elements are ones, this can be omitted, such as in our example. See model file "m_05_ihacres_7p_2s" for an example of a deficit store.

The creator method doesn't take any inputs and returns a model object of the class being defined (in this case of class `m_nn_example_7p_3s`). The following code defines the creator class for our example model.

```
16          function obj = m_nn_example_7p_3s()
17              obj.numStores = 3;
18              obj.numFluxes = 9;
19              obj.numParams = 7;
20
21              obj.JacobPattern  = [1,1,0;
```

```
22                                      1,1,0;
23                                      0,1,1];
24
25              obj.parRanges = [ 0,    4;  % crate [mm/d]
26                                1, 2000;  % uzmax [mm]
27                                0,   20;  % prate [mm/d]
28                                0,    1;  % klz   [mm/d]
29                                0,    1;  % alpha [d-1]
30                                0,    1;  % kg    [-]
31                                1,  120]; % d     [d]
32
33              obj.StoreNames = ["S1", "S2", "S3"];
34              obj.FluxNames  = ["qse", "e",  "qp", "qc", "qlz",...
35                                "qf",  "qg", "qs", "q"];
36
37              obj.FluxGroups.Ea = 2;
38              obj.FluxGroups.Q  = 9;
39
40          end
```

We now need to write the code for three additional *methods* to define how the model actually works. These are the `init` *method*, the `model_fun` *method* and the `step` *method*. These are the subjects of points 7, 8 and 9 below, respectively.

7. Write the initialisation *method* (`init`)

`init` runs once at the beginning of a simulation and is called by the `init_` *method* defined in the *superclass* (`init_` is the same for every model, `init` is model-specific). In general, there are three types of operations that one might need to include in the init method:

1. Produce and store auxiliary parameters

   If any auxiliary parameter is defined in the properties of the model class (see point 5 above), these should be assigned their values here. As already mentioned, we don't have any auxiliary parameters defined for our example model, so this isn't necessary.

2. Update store minima and maxima

   Minimum and maximum values for each store are saved as attributes `store_min` and `store_max` respectively. In the `init_` method, these are assigned arrays of zeros and infinities respectively (of size [1,`numStores`]). In the model specific *method*, we can update any of these limits. Here we will specify that the maximum value of store 1 (*UZ*) is the parameter UZ_{max}, which is the second parameter.

3. Initialise and save routing schemes

   To initialise routing schemes (unit hydrographs), we use the functions in the folder "./MARRMoT/Models/Unit Hydrograph files". Unit hydrograph functions are described in more detail in section 4.3, each of them returns a 2-row array whose number of columns indicate the number of steps forward that the fluxes will be routed into. The first row contains the coefficients of the unit hydrograph (which will be static throughout the simulation), the second row is zeros for now, these are still-to-flow fluxes and will be updated at every timestep. Unit hydrograph arrays are stored in the `uhs` attribute as elements of a cell array. For our example we use the function `uh_4_full` to create the triangular unit hydrograph we need to route the streamflow *Q*.

Note that `init` is not limited to these three uses: in theory, any operation that needs to happen once at the beginning of a simulation should be coded here. For example, in model m_47_IHM19_16p_4s, initial store values are determined based on parameters, these are therefore assigned within the `init` method. Also note that the `init` method needs to exist even if it is left blank (i.e. no auxiliary parameters, no parameter-defined store limits and no routing).

For our example model, the following code produces the operations described.

```
47          function obj = init(obj)
48              % extract theta and delta_t from attributes
49              theta   = obj.theta;
50              delta_t = obj.delta_t;
51
52              % needed parameters
53              uzmax = theta(2); % Maximum upper zone storage [mm]
54              d     = theta(7); % Routing delay [d]
55
56              % min and max of stores
57              obj.store_max(1) = uzmax;
58
59              % unit hydrographs
60              uh = uh_4_full(d,delta_t);
61              obj.uhs = {uh};
62          end
```

8. Code the model equations (`model_fun`)

The next step is to translate the model equations and ODEs (equations 4.1 to 4.12) into code. This is done inside the method `model_fun`. `model_fun` takes one input, an array of store values S, and returns two outputs: arrays of store differentials and fluxes. Writing the `model_fun` method is a two-step approach: first we define formulas to calculate fluxes (i.e. the constitutive equations) and then use fluxes entering and leaving every store to calculate store differentials (i.e. the ODEs).

Flux files contain functions to be used to write constitutive equations, see section 0 to create a new one if needed. Flux functions can take all sorts of inputs, all inputs that aren't store values (which are the input to the `model_fun` method), should be retrieved from the model attributes. In general, the attributes needed are: `t`, the current timestep; `theta`, the parameter set; `climate_input`, the vector of P, PET and T for the simulation; and `uhs`, the cell array of the unit hydrographs and still-to-flow vectors. Note that any attribute can be used here, including fluxes and stores at any previous timestep through the attributes `fluxes` and `stores`.

If a flux is the result of the application of a routing scheme to another flux (like *Q* in equation 4.12 in our example model), the function `route(flux_in, uh)` should be used to calculate its value. This function (see "./MARRMoT/Models/Unit Hydrograph files/route.m" for details) calculates the routing of `flux_in` trough the unit hydrograph specified in `uh` for this step and adds to it the still-to-flow value for this step (remember that `uh` contain both the routing coefficients and still-to-flow vector). Note that `route` does not update the still-to-flow vector, this is done only at the end of each timestep, once fluxes are calculated, using the step method described below.

Once all fluxes are calculated and, from them, the stores differentials, both are returned as arrays. The order of fluxes and stores in each of these arrays should match the names in the `StoreNames` and `FluxNames` attributes.

All of the equations in our model are already coded as flux functions in MARRMoT, hence we use those to code our model functions. The full code implementation is the following.

```
 65          function [dS, fluxes] = model_fun(obj, S)
 66              % parameters
 67              theta = obj.theta;
 68              crate = theta(1);     % Max capillary rise rate [mm/d]
 69              uzmax = theta(2);     % Max upper zone storage [mm]
 70              prate = theta(3);     % Max percolation rate [mm/d]
 71              klz   = theta(4);     % Lower zone runoff coeff [d-1]
 72              alpha = theta(5);     % Frac. of lz runoff to gw [-]
 73              kg    = theta(6);     % Gw runoff coefficient [d-1]
 74              d     = theta(7);     % Routing delay [d]
 75
 76              % delta_t
 77              delta_t = obj.delta_t;
 78
 89              % unit hydrographs
 80              uhs = obj.uhs;
 81              uh = uhs{1};
 82
 83              % stores
 84              S1 = S(1);
 85              S2 = S(2);
 86              S3 = S(3);
 87
 88              % climate input
 89              t = obj.t;                      % this time step
 90              c = obj.input_climate(t,:);   % climate at this step
 91              P  = c(1);
 92              Ep = c(2);
 93              T  = c(3);
 94
 95              % fluxes functions
 96              flux_qse = saturation_1(P,S1,uzmax);
 97              flux_e   = evap_7(S1,uzmax,Ep,delta_t);
 98              flux_qp  = percolation_1(prate,S1,delta_t);
 99              flux_qc  = capillary_1(crate,S1,uzmax,S2,delta_t);
100              flux_qlz = baseflow_1(klz,S2);
101              flux_qf  = split_1(1-alpha,flux_qlz);
102              flux_qg  = split_1(alpha,flux_qlz);
103              flux_qs  = baseflow_1(kg,S3);
104              flux_qt  = route(flux_qse + flux_qf + flux_qs, uh);
105
106              % stores ODEs
107              dS1 = P       + flux_qc - flux_e - flux_qse - flux_qp;
108              dS2 = flux_qp - flux_qc - flux_qlz;
109              dS3 = flux_qg - flux_qs;
110
111              % outputs
112              dS = [dS1 dS2 dS3];
113              fluxes = [flux_qse flux_e flux_qp flux_qc flux_qlz ...
114                        flux_qf flux_qg flux_qs flux_qt];
115          end
```

9. Write the stepping *method* (`step`)

Finally, the last *method* that needs defining is `step`. This is run once at the end of every timestep after the ODEs are solved and the fluxes and store values for the step are calculated. Currently this *method* is only used to update still-to-flow fluxes in the `uhs` attribute (i.e. the second row of the arrays). The route function used in `model_fun` above calculates the output at the current timestep, but does not update the still-to-flow vector: `model_fun` is called multiple times to numerically solve the ODEs and the update only needs to happen once the ODEs are solved and the fluxes evaluated.

For this we use the function `update_uh(uh, flux_in)`, whose full code can be seen at "./MARRMoT/Models/Unit Hydrograph files/update_uh.m". This function returns a new uh with unchanged first row (the coefficients) and updated second row (the still-to-flow values). After calculation, these are stored again as cell arrays in the `uhs` *attribute*.

Just like the `init` method, also `step` needs to exist even if there are no unit hydrographs to update.

The following code update the still-to-flow vector in our example model's uhs attribute.

```
119         function obj = step(obj)
120             % unit hydrographs and still-to-flow vectors
121             uhs = obj.uhs;
122             uh = uhs{1};
123 
124             % input fluxes to the unit hydrographs at this timestep
125             fluxes = obj.fluxes(obj.t,:);
126             flux_qse = fluxes(1);
127             flux_qf  = fluxes(6);
128             flux_qs  = fluxes(8);
129 
130             % update still-to-flow vectors using fluxes at current
131             % step and unit hydrographs
132             uh = update_uh(uh, flux_qse + flux_qf + flux_qs);
133 
134             obj.uhs = {uh};
135         end
```

The full code for the model example file can be found in the file named "./MARRMoT/User manual/m_nn_example_7p_3s.m".

4.2 Creating a new flux function

This section gives a few examples that show how to create flux functions. This involves three steps:

1. Define the function that should be used
2. Specify any constraints that should be used
3. Apply a smoothing scheme if the function is discontinuous

Note: smoothing schemes exist for both threshold discontinuities and angle discontinuities. However, smoothing an equation means a fundamental change to the flux equation. Threshold discontinuities are smoothed in MARRMoT because this improves the accuracy of store estimates. Matlab solvers are able to function with angle discontinuities however, and these are not smoothed in MARRMoT to keep the original flux equations intact wherever possible.

In MARRMoT, flux equations are created in separate files from the *model files*,The flux is defined as a function which outputs flux values based on a variety of parameters, storage values and climate inputs.

We present five flux functions in the following sections to exemplify this general approach.

4.2.1 Basic example

To understand the basic structure of all flux files, consider the example of the linear reservoir. Its equation is:

$$q = kS \tag{4.13}$$

where q is the store's outflow, k a runoff coefficient and S the current storage. No constraints are needed, because q relates directly to S (provided $k \leq 1$). If $S = 0$, $q = 0$, regardless of k. The flux file ("./MARRMoT/Models/Flux files/baseflow_1.m") looks as follows:

```
  1 function [out] = baseflow_1(p1,S)
...
 10 % Flux function
 11 % ------------------
 12 % Description:  Outflow from a linear reservoir
 13 % Constraints:  -
 14 % @(Inputs):    p1   - time scale parameter [d-1]
 15 %               S    - current storage [mm]
 16
 17 out = p1.*S;
 18
 19 end
```

`p1` represents parameter k and S is the current storage. `out` is the calculate flux and the output of this flux function.

4.2.2 Adding constraints

To show how to add constraints, we use the non-linear reservoir as example. The equation for a non-linear reservoir is:

$$q = kS^a \tag{4.14}$$

where q is the store's outflow, k a runoff coefficient, a the non-linearity coefficient and S the current storage. No lower constraint is needed, because $q = 0$, if $S = 0$, regardless of k and a. However, for large values of k and a, it is possible to generate values $q > S$. This is logically impossible so a constraint of the form $q \leq S/\Delta t$ is needed. Thus the *flux equation* has two parameters, one store input and one constraint.

An additional complication arises from very small numerical inaccuracies, that can result in stores having very slightly negative values for some time steps. These errors are generally in the order of -1E-5 or smaller. However, in a non-linear equation this can result in mathematically correct, but physically meaningless complex estimates of fluxes. An additional constraint is introduced to avoid this which ensures $S \geq 0$.

To introduce constraints, we use Matlab's functions `min` and `max`. With two parameters, one store input and two constraints, the flux file for the non-linear reservoir ("./MARRMoT/Models/Flux files/baseflow_1.m") looks as follows:

```
  1 function [out] = baseflow_2(S,p1,p2,dt)
...
 10 % Flux function
 11 % ------------------
```

```
12 % Description:  Non-linear outflow from a reservoir
13 % Constraints:  f <= S/dt
14 %               S >= 0      prevents issues with complex numbers
15 % @(Inputs):    S    - current storage [mm]
16 %               p1   - time coefficient [d]
17 %               p2   - exponential scaling parameter [-]
18 %               dt   - time step size [d]
19
20 out = min((1./p1*max(S,0)).^(1./p2),max(S,0)/dt);
21
22 end
```

4.2.3 Using logistic smoothing of equations

A logistic smoothing function (Kavetski and Kuczera, 2007) can be used to modify equations with threshold discontinuities to be continuous over their domain. An example of a threshold equation is effective rainfall after an interception store is filled:

$$P_{eff} = \begin{cases} P(t), if\ S = S_{max} \\ 0, \quad otherwise \end{cases} \tag{4.15}$$

Where the effective flow P_{eff} is zero until the store reaches maximum capacity, after which all inflow to the store $P(t)$ becomes P_{eff}. A smoothing function makes this transition more gradual. The equation becomes:

$$P_{eff} = P(t)[1 - \phi(S, S_{max})] \tag{4.16}$$

where $\phi(S,Smax)$ is the smoothing function (Kavetski and Kuczera, 2007).

$$\phi(S, S_{max}) = \frac{1}{1 + exp\left[\frac{S - S_{max} + r * e * S_{max}}{r * S_{max}}\right]} \tag{4.17}$$

This smoothing function in equation 4.17 uses two parameters, *r* and *e*.

In MARRMoT, the smoothing function in equation 4.17 is implemented in the function file "./MARRMoT/Functions/Flux smoothing/smoothThreshold_storage_logistic.m". It defines the function `smoothThreshold_storage_logistic`, which takes four arguments: the first two (`S` and `Smax`,) are mandatory; the second two (`r` and `e`) are optional, when not specified their default values are 0.01 and 5.00 respectively (Clark et al., 2008). When using the smoothing function `smoothThreshold_storage_logistic` within a flux file, we use the argument `varargin` to allow the user to specify different values to these two parameters.

Overall, the store overflow equation has one mandatory parameter and needs one store input, plus it can take two additional parameters to define the smoothing. The flux file ("./MARRMoT/Models/Flux files/interception_1.m") to define it looks as follows:

```
  1 function [out] = interception_1(In,S,Smax,varargin)
...
 10 % Flux function
 11 % ------------------
 12 % Description:  Interception excess when max capacity is reached
 13 % Constraints:  -
 14 % @(Inputs):    In   - incoming flux [mm/d]
 15 %               S    - current storage [mm]
```

```
% Smax - maximum storage [mm]
% varargin(1) - smoothing variable r (default 0.01)
% varargin(2) - smoothing variable e (default 5.00)

if size(varargin,2) == 0
    out = In.*(1-smoothThreshold_storage_logistic(S,Smax));
elseif size(varargin,2) == 1
    out = In.*(1-smoothThreshold_storage_logistic(S,Smax,...
                                                  varargin(1)));
elseif size(varargin,2) == 2
    out = In.*(1-smoothThreshold_storage_logistic(S,Smax,...
                                        varargin(1),varargin(2)));
end

end
```

There is a different function to smooth temperatures threshold (e.g. for snowmelt or snowfall). This has the following form:

$$\phi_t(T, T_{thr}) = \frac{1}{1 + exp\left[\frac{T - T_{thr}}{r}\right]} \tag{4.18}$$

Where T is the current temperature and T_{thr} is the threshold. Equation 4.18 has an additional smoothing parameter r. In MARRMoT, function `smoothThreshold_temperature_logistic` (in file "./MARRMoT/Functions/Flux smoothing/smoothThreshold_temperature_logistic.m") codes the output of equation 4.18. Similarly to the storage smoothing function, this takes an optional argument `r`, and uses 0.01 as default value if this is not given. It's use within a *flux file* is identical to the example shown.

4.3 Creating a new unit hydrograph

Unit hydrograph functions files are found in the folder "./MARRMoT/Models/Unit Hydrograph files". These are functions that take at least two inputs (i.e. a unit hydrograph base time and the step size, `delta_t`) and return a 2-row array called UH. The first row contains the coefficients of the routing scheme: when a flux is routed using the unit hydrograph, these are the multiplier for the flux at every future timestep. The values of the coefficients must sum to one (i.e. all the flux needs to be routed at some point and nothing more than the whole flux). The second row of zeros will keep track of fluxes whose routing in future timesteps has been calculated already, and will be routed, these are called still-to-flow fluxes and are updated using the `update_uh` function (see section 4.1.2-9 above).

In the most general case, creating a new unit hydrograph function for any shape will start by defining a pdf as a function of time and integrating it at every timestep `t` (i.e. between `t-1` and `t`) to discretise it. In practice, this is rarely necessary as for the most common routing schemes, unit hydrograph coefficients can be derived analytically without the need of integration. The user is advised to browse the unit hydrograph files provided to grasp the different methods used to define their coefficients, including: using cumulative distribution functions (`uh_1_half` and `uh_2_full`), calculating individual step sizes for every step analytically (`uh_3_half`, `uh_7_uniform`, `uh_8_delay`), and integrating the pdf (`uh_4_full`, `uh_5_half` and `uh_6_gamma`). In every case, the following should be considered:

1. Ensure flux is routed with at least one future step – if the base time of the unit hydrograph is below zero, this might happen and precautions should be used to prevent it.

2. Ensure all flux is routed and nothing more than the flux (i.e. sum of coefficients = 1) – this happens when an infinite pdf is integrated for a finite number of timesteps; the residual flow should be redistributed proportionally to every timestep.
3. Ensure to add the second row of zeros to store still-to-flow flux values is added to the coefficients before returning.

4.4 Creating a new objective function

Objective functions are defined in a series of files in the folder "./MARRMoT/Functions/Objective functions". These functions are needed for the `calibrate` *method* (see section 2.4) and can be used to evaluate the ability of a model to reproduce an observed timeseries. All objective functions in MARRMoT take at least two inputs (`sim` and `obs`, the simulated and observed timeseries to compare) and one additional optional input (`idx`, either a boolean array of the same size of `sim` and `obs`, or an array of indices to use to calculate the value of the objective function – if not specified all the timesteps where $obs \geq 0$ are used). Additionally, they can take additional inputs as parameters of the functions themselves (e.g. weights of the three components, in the case of KGE).

To create a new objective function, the easiest way to proceed is to start from an existing objective function file. There, the code under the header "`Check inputs and select timesteps`" should not be modified: this piece of code calls a helper function `check_and_select` (see its definition in "./MARRMoT/Functions/Objective functions/check_and_select.m") which: (1) ensures `sim` and `obs` are of comparable lengths; (2) ensures `idx` has the correct format (i.e. either a boolean array of the same size of `sim` and `obs` or a numeric array); and (3) extracts from `sim` and `obs` the timesteps specified by `idx` intersected with the timesteps where $obs \geq 0$. It returns the selected indices of `sim` and `obs` as well as the array of indices from the original series used to subset them (that is the intecetion of `idx` and $obs \geq 0$).

After this checks and subsetting, the new arrays `sim` and `obs` can be used freely to define a new objective function, with the only precaution that if an other objective function from MARRMoT is called (e.g. to calculate the average of two objective functions), this should be called without `idx` (i.e. `idx = []`) as the input vectors have already been filtered.

All objective functions return at least their value (`val`) and the list of indices used to calculate it (i.e. the `idx` which was the output of `check_and_select`).