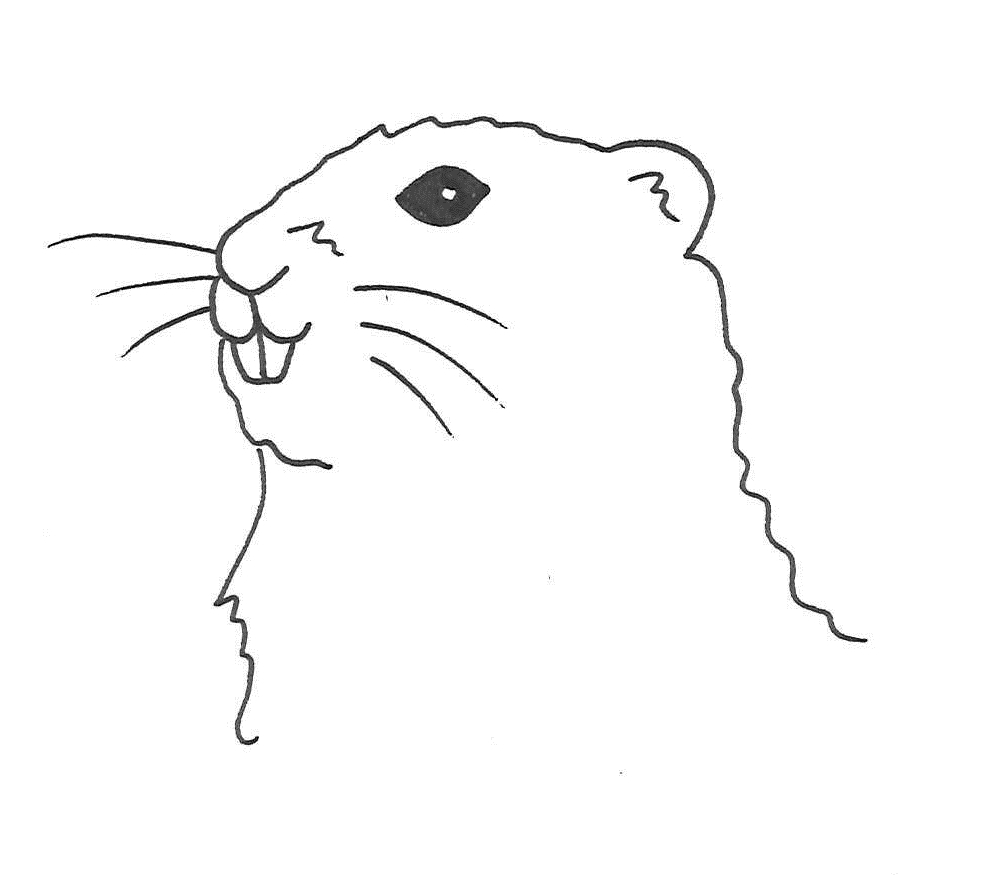
**Modular Assessment of Rainfall-Runoff Models Toolbox (MARRMoT)**

**User Manual**

**Version 2.-**



**Luca Trotter  
Wouter Knoben  
Keirnan Fowler  
Margarita Saft  
Murray Peel**

**December 2021**

**Authors**

Luca Trotter1

Wouter J. M. Knoben2

Keirnan J. A. Fowler1

Margarita Saft1

Murray C. Peel1

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

2Centre 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

[Disclaimer 3](#_Toc90711091)

[1 Introduction 5](#_Toc90711092)

[1.1 Place within MARRMoT documentation 5](#_Toc90711093)

[1.2 Content overview 5](#_Toc90711094)

[1.3 General toolbox outline 5](#_Toc90711095)

[1.4 Folder structure 7](#_Toc90711096)

[1.5 Definitions 8](#_Toc90711097)

[2 Using MARRMoT v2.- 9](#_Toc90711098)

[2.1 Model objects 10](#_Toc90711099)

[2.2 Required input 10](#_Toc90711100)

[2.3 Simulation output 12](#_Toc90711101)

[2.4 Model calibration 13](#_Toc90711102)

[3 Understanding MARRMoT v2.- 16](#_Toc90711103)

[3.1 Class definition files 16](#_Toc90711104)

[3.1.1 The MARRMoT\_model *superclass* 16](#_Toc90711105)

[3.1.2 Model files 18](#_Toc90711106)

[3.2 Detailed code descriptions 19](#_Toc90711107)

[3.2.1 Numerical ODEs solving 19](#_Toc90711108)

[3.2.2 Model simulation 21](#_Toc90711109)

[3.2.3 Model calibration 22](#_Toc90711110)

[4 Editing MARRMoT v2.- 24](#_Toc90711111)

[4.1 Creating a new model 24](#_Toc90711112)

[4.1.1 Create the model description 24](#_Toc90711113)

[4.1.2 Create the *model file* 26](#_Toc90711114)

[4.2 Creating a new flux function 31](#_Toc90711115)

[4.2.1 Basic example 32](#_Toc90711116)

[4.2.2 Adding constraints 32](#_Toc90711117)

[4.2.3 Using logistic smoothing of equations 33](#_Toc90711118)

[4.3 Creating a new unit hydrograph 34](#_Toc90711119)

[4.4 Creating a new objective function 35](#_Toc90711120)

# Introduction

## 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**
   1. Trotter et al. (in preparation) present the object-oriented implementation of MARRMoT, its benefits and the technical changes from the previous version;
   2. 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.

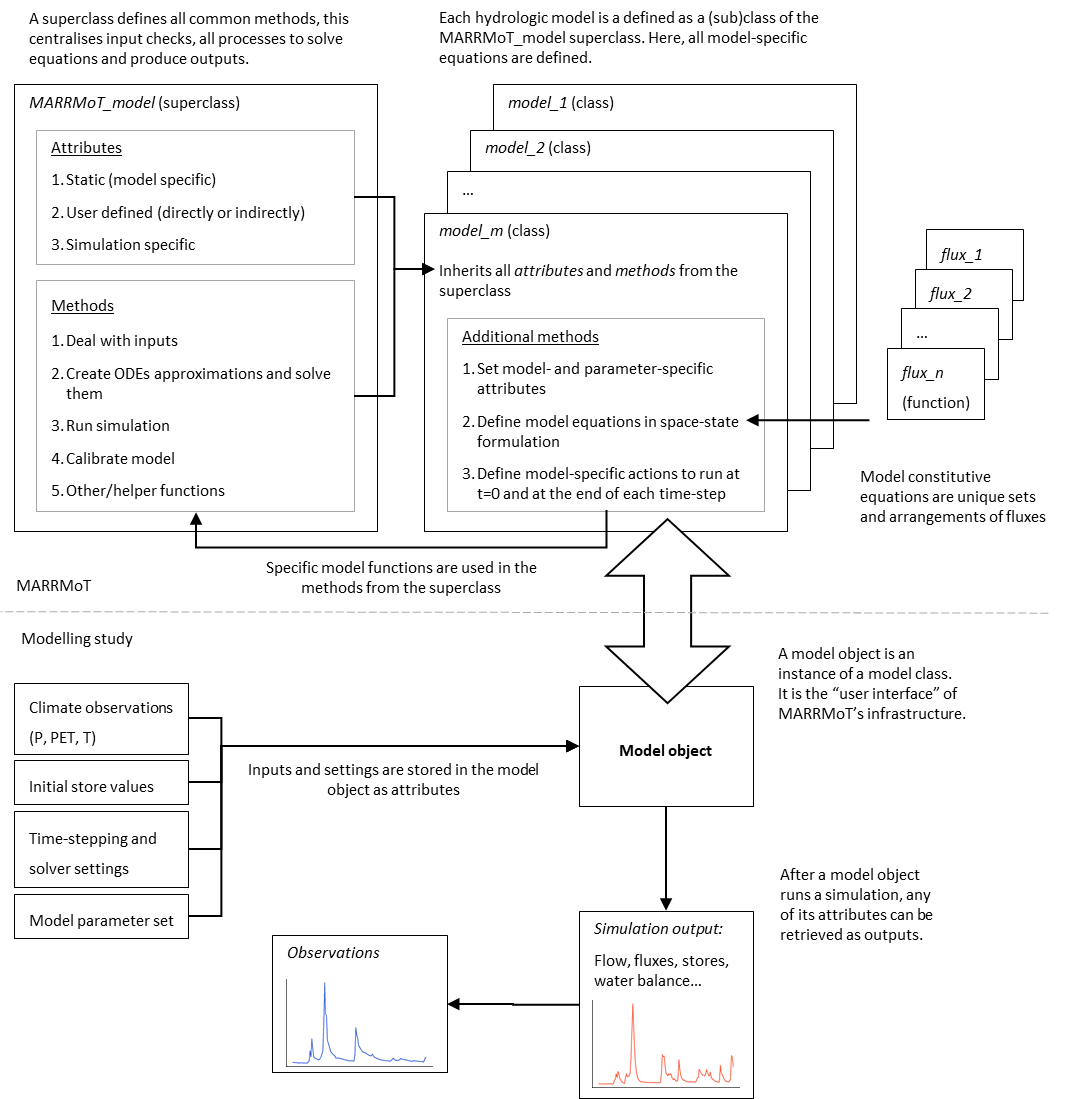
## Content overview

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.

## 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.

## 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.

## 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. |

# 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”.

1. 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”.

1. 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”.

1. 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.

## 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(); |

## 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 structurewith 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. 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:  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); |

## 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.   1. 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. 2. 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. 3. 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. 4. 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.**.

## 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(); |

# 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*.

## Class definition files

### 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 | 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). |
| set.theta |
| set.input\_climate |
| set.S0 |
| set.solver\_opts |
| 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. |

### 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. |

## 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.

### 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  140 | function err = ODE\_approx\_IE (obj, S)  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  142  143  144 | delta\_S = obj.model\_fun(S);  if obj.t == 1; Sold = obj.S0(:);  else; Sold = obj.stores(obj.t-1,:)';  end |

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

|  |  |
| --- | --- |
| 145  146 | err = (S - Sold)/obj.delta\_t - delta\_S';  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  ...  156 | solver\_opts = obj.solver\_opts;  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  162  163 | Snew\_v = zeros(3, obj.numStores);  resnorm\_v = Inf(3, 1);  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  167  168  169  170  171  172  173 | [tmp\_Snew, tmp\_fval] = ...  NewtonRaphson(@obj.ODE\_approx\_IE,...  Sold,...  solver\_opts.NewtonRaphson);  tmp\_resnorm = sum(tmp\_fval.^2);    Snew\_v(1,:) = tmp\_Snew;  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  177  178  179  180  181  182  183  184  185  186 | if tmp\_resnorm > resnorm\_tolerance  [tmp\_Snew,tmp\_fval,~,tmp\_iter] = ...  obj.rerunSolver('fsolve',...  tmp\_Snew,...  Sold);    tmp\_resnorm = sum(tmp\_fval.^2);    Snew\_v(2,:) = tmp\_Snew;  resnorm\_v(2) = tmp\_resnorm;  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  190  191  192  193  194  195  196  197  198  199  200  201  202 | if tmp\_resnorm > resnorm\_tolerance  [tmp\_Snew,tmp\_fval,~,tmp\_iter] = ...  obj.rerunSolver('lsqnonlin',...  tmp\_Snew,...  Sold);    tmp\_resnorm = sum(tmp\_fval.^2);  Snew\_v(3,:) = tmp\_Snew;  resnorm\_v(3) = tmp\_resnorm;  iter\_v(3) = tmp\_iter;    end  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  206  207  208  209  210  211 | [resnorm, solver\_id] = min(resnorm\_v);  Snew = Snew\_v(solver\_id,:);  iter = iter\_v(solver\_id);  solvers = ["NewtonRaphson", "fsolve", "lsqnonlin"];  solver = solvers(solver\_id);  end % closes function opened on line 142 |

### 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  308  309  310  311 | function [] = run(obj,...  input\_climate,...  S0,...  theta,...  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  314  315  316  317  318  319  320  321  322  323  324 | if nargin > 4 && ~isempty(solver\_opts)  obj.solver\_opts = solver\_opts;  end  if nargin > 3 && ~isempty(theta)  obj.theta = theta;  end  if nargin > 2 && ~isempty(S0)  obj.S0 = S0;  end  if nargin > 1 && ~isempty(input\_climate)  obj.input\_climate = input\_climate;  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  332  333  334  335  336  337 | t\_end = size(obj.input\_climate, 1);    for t = 1:t\_end  obj.t = t;  if t == 1; Sold = obj.S0(:);  else; Sold = obj.stores(t-1,:)';  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  342  343  344  345  346  347  348 | [dS, f] = obj.model\_fun(Snew);    obj.fluxes(t,:) = f \* obj.delta\_t;  obj.stores(t,:) = Sold + dS' \* obj.delta\_t;  obj.solver\_data.resnorm(t) = resnorm;  obj.solver\_data.solver(t) = solver;  obj.solver\_data.iter(t) = iter; |

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

|  |  |
| --- | --- |
| 350  351 | obj.step();  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  354 | obj.status = 1;  end % closes function opened on line 223 |

### 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  467  468  469  470  471  472  473  474  475  476  477  478 | function [par\_opt  of\_cal,...  stopflag,...  output] = ...  calibrate(obj,...  Q\_obs,...  cal\_idx,...  optim\_fun,...  par\_ini,...  optim\_opts,...  of\_name,...  inverse\_flag,...  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  481  482  483  484  485 | if isempty(obj.input\_climate) || isempty(obj.delta\_t) ||...  isempty(obj.S0) || isempty(obj.solver\_opts)  error(['input\_climate, delta\_t, S0 and solver\_opts '...  'attributes must be specified before calling '...  'calibrate.']);  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  490  491  ...  495  496  497  498 | if isempty(cal\_idx)  cal\_idx = 1:length(Q\_obs);  end  if islogical(cal\_idx); cal\_idx = find(cal\_idx); end  input\_climate\_all = obj.input\_climate;  obj.input\_climate = input\_climate\_all(1:max(cal\_idx),:);  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  503  504 | if isempty(par\_ini)  par\_ini = mean(obj.parRanges,2);  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  509  510  511 | function fitness = fitness\_fun(par)  Q\_sim = obj.get\_streamflow([],[],par);  fitness = (-1)^inverse\_flag\*feval(of\_name, Q\_obs,...  Q\_sim, cal\_idx, varargin{:});  end |

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

|  |  |
| --- | --- |
| 513  514  515  516  517  518  519  520 | [par\_opt,...  of\_cal,...  stopflag,...  output] = ...  feval(optim\_fun,...  @fitness\_fun,...  par\_ini,...  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  ...  527  528 | of\_cal = (-1)^inverse\_flag \* of\_cal;  obj.input\_climate = input\_climate\_all;  end % closes function opened on line 382 |

# Editing MARRMoT v2.-

## 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.

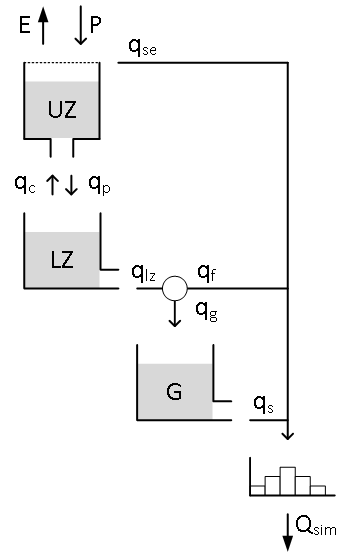
For clarity, we illustrate the process of creating a new model using an example. We assume that the new model created in this section is built according to certain assumptions of how a particular catchment functions (i.e. on some perceptual model of the catchment). Justifying these assumptions is outside 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.

### Create the model description

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

#### 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:

**Figure 2:** Model schematic

* 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 Qsim

#### 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.

|  |  |
| --- | --- |
|  | (4.1) |
|  | (4.2) |
|  | (4.3) |
|  |  |

#### 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*.

|  |  |
| --- | --- |
|  | (4.4) |
|  | (4.5) |
|  | (4.6) |
|  | (4.7) |
|  | (4.8) |
|  | (4.9) |
|  | (4.10) |
|  | (4.11) |
|  | (4.12) |

Based on these equations, this model has 7 parameters: maximum capillary rise rate crate [mm/d], maximum upper zone storage *UZmax* [mm], constant percolation rate *prate* [mm/d], lower zone runoff coefficient *klz* [d-1], fraction of lower zone runoff to groundwater *α* [-] , groundwater runoff coefficient *kg* [d-1], and routing delay *d* [d].

### 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.

#### 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  ...  16 | classdef m\_nn\_example\_7p\_3s < MARRMoT\_model    function obj = m\_nn\_example\_7p\_3s() |
|  |  |

#### 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.

#### 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 *qc* (see eqq. 4.5 and 4.6) and on the second store (*LZ*) via *qc* again (this is because even in *LZ* does not appear in eq. 4.6, current storage in *LZ* is the upper bound for *qc*). 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  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | function obj = m\_nn\_example\_7p\_3s()  obj.numStores = 3;  obj.numFluxes = 9;  obj.numParams = 7;    obj.JacobPattern = [1,1,0;  1,1,0;  0,1,1];    obj.parRanges = [ 0, 4; % crate [mm/d]  1, 2000; % uzmax [mm]  0, 20; % prate [mm/d]  0, 1; % klz [mm/d]  0, 1; % alpha [d-1]  0, 1; % kg [-]  1, 120]; % d [d]    obj.StoreNames = ["S1", "S2", "S3"];  obj.FluxNames = ["qse", "e", "qp", "qc", "qlz",...  "qf", "qg", "qs", "q"];    obj.FluxGroups.Ea = 2;  obj.FluxGroups.Q = 9;    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.

#### 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.

1. 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 *UZmax*, which is the second parameter.

1. 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  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | function obj = init(obj)  % extract theta and delta\_t from attributes  theta = obj.theta;  delta\_t = obj.delta\_t;    % needed parameters  uzmax = theta(2); % Maximum upper zone storage [mm]  d = theta(7); % Routing delay [d]    % min and max of stores  obj.store\_max(1) = uzmax;    % unit hydrographs  uh = uh\_4\_full(d,delta\_t);  obj.uhs = {uh};  end |
|  |  |

#### 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  66  67  68  69  70  71  72  73  74  75  76  77  78  89  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115 | function [dS, fluxes] = model\_fun(obj, S)  % parameters  theta = obj.theta;  crate = theta(1); % Max capillary rise rate [mm/d]  uzmax = theta(2); % Max upper zone storage [mm]  prate = theta(3); % Max percolation rate [mm/d]  klz = theta(4); % Lower zone runoff coeff [d-1]  alpha = theta(5); % Frac. of lz runoff to gw [-]  kg = theta(6); % Gw runoff coefficient [d-1]  d = theta(7); % Routing delay [d]    % delta\_t  delta\_t = obj.delta\_t;    % unit hydrographs  uhs = obj.uhs;  uh = uhs{1};    % stores  S1 = S(1);  S2 = S(2);  S3 = S(3);    % climate input  t = obj.t; % this time step  c = obj.input\_climate(t,:); % climate at this step  P = c(1);  Ep = c(2);  T = c(3);    % fluxes functions  flux\_qse = saturation\_1(P,S1,uzmax);  flux\_e = evap\_7(S1,uzmax,Ep,delta\_t);  flux\_qp = percolation\_1(prate,S1,delta\_t);  flux\_qc = capillary\_1(crate,S1,uzmax,S2,delta\_t);  flux\_qlz = baseflow\_1(klz,S2);  flux\_qf = split\_1(1-alpha,flux\_qlz);  flux\_qg = split\_1(alpha,flux\_qlz);  flux\_qs = baseflow\_1(kg,S3);  flux\_qt = route(flux\_qse + flux\_qf + flux\_qs, uh);  % stores ODEs  dS1 = P + flux\_qc - flux\_e - flux\_qse - flux\_qp;  dS2 = flux\_qp - flux\_qc - flux\_qlz;  dS3 = flux\_qg - flux\_qs;    % outputs  dS = [dS1 dS2 dS3];  fluxes = [flux\_qse flux\_e flux\_qp flux\_qc flux\_qlz ...  flux\_qf flux\_qg flux\_qs flux\_qt];  end |
|  |  |

#### 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  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135 | function obj = step(obj)  % unit hydrographs and still-to-flow vectors  uhs = obj.uhs;  uh = uhs{1};    % input fluxes to the unit hydrographs at this timestep  fluxes = obj.fluxes(obj.t,:);  flux\_qse = fluxes(1);  flux\_qf = fluxes(6);  flux\_qs = fluxes(8);    % update still-to-flow vectors using fluxes at current  % step and unit hydrographs  uh = update\_uh(uh, flux\_qse + flux\_qf + flux\_qs);    obj.uhs = {uh};  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”.

## 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.

### Basic example

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

|  |  |
| --- | --- |
|  | (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* < 1)*.* If *S = 0*, *q = 0*, regardless of *k.* The flux file (“./MARRMoT/Models/Flux files/baseflow\_1.m”) looks as follows:

|  |  |
| --- | --- |
| 1  ...  10  11  12  13  14  15  16  17  18  19 | function [out] = baseflow\_1(p1,S)  % Flux function  % ------------------  % Description: Outflow from a linear reservoir  % Constraints: -  % @(Inputs): p1 - time scale parameter [d-1]  % S - current storage [mm]  out = p1.\*S;  end |

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

### Adding constraints

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

|  |  |
| --- | --- |
|  | (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 < S/Δt* 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 > 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  ...  10  11  12  13  14  15  16  17  18  19  20  21  22 | function [out] = baseflow\_2(S,p1,p2,dt)  % Flux function  % ------------------  % Description: Non-linear outflow from a reservoir  % Constraints: f <= S/dt  % S >= 0 prevents issues with complex numbers  % @(Inputs): S - current storage [mm]  % p1 - time coefficient [d]  % p2 - exponential scaling parameter [-]  % dt - time step size [d]  out = min((1./p1\*max(S,0)).^(1./p2),max(S,0)/dt);  end |

### 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:

|  |  |
| --- | --- |
|  | (4.15) |

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

|  |  |
| --- | --- |
|  | (4.16) |

where *ϕ(S,Smax)* is the smoothing function (Kavetski and Kuczera, 2007).

|  |  |
| --- | --- |
|  | (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  ...  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | function [out] = interception\_1(In,S,Smax,varargin)  % Flux function  % ------------------  % Description: Interception excess when max capacity is reached  % Constraints: -  % @(Inputs): In - incoming flux [mm/d]  % 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:

|  |  |
| --- | --- |
|  | (4.18) |

Where *T* is the current temperature and *Tthr* 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.

## 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.

## 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≥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≥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≥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).