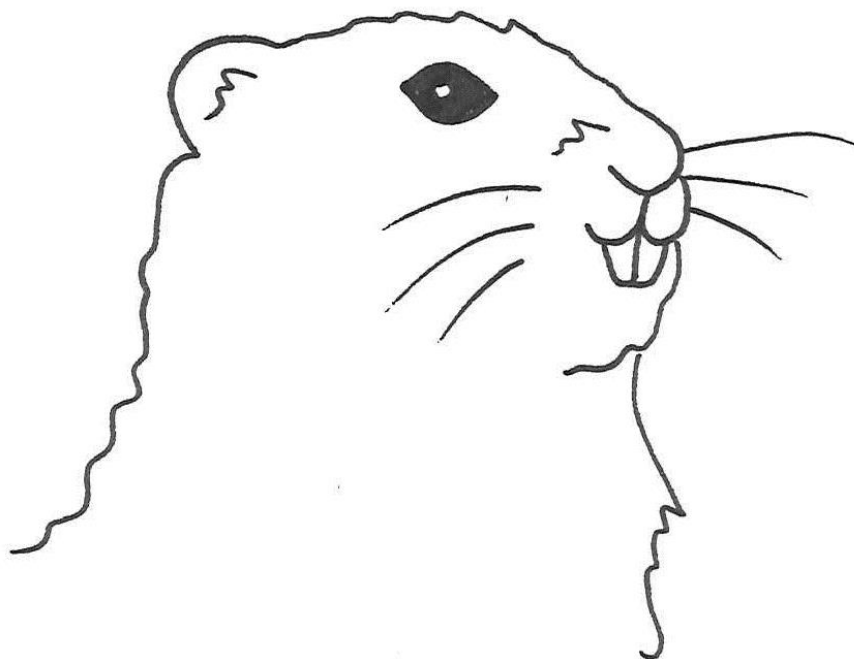


Modular Assessment of Rainfall-Runoff Models Toolbox (MARRMoT)

User Manual

Version 1.0



**Wouter Knoben, Jim Freer, Keirnan Fowler, Murray Peel,
Ross Woods**

December 2018

Authors

Wouter J. M. Knoben¹

Jim E. Freer²

Keirnan J. A. Fowler³

Murray C. Peel³

Ross A. Woods¹

¹ Department of Civil Engineering, University of Bristol, United Kingdom

² School of Geographical Sciences, University of Bristol, United Kingdom

³ Department of Infrastructure Engineering, University of Melbourne, Australia

Email contact:

w.j.m.knoben@bristol.ac.uk

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
1 Introduction	5
1.1 Place within MARRMoT documentation	5
1.2 Contents	5
1.3 General toolbox outline	6
1.4 Folder structure	7
1.5 Definitions	8
2 Navigating a model file	9
2.1 Setup of a model file	9
2.2 Step-by-step description of a model file	12
3 Using the framework	20
3.1 Setup: add MARRMoT folders to the Matlab path	20
3.2 Workflow: 1 model, 1 parameter set, 1 catchment	20
3.3 Workflow: 1 model, N parameter sets, 1 catchment	20
3.4 Workflow: 3 models, 1 random parameter set, 1 catchment	20
3.5 Workflow: calibration of 1 parameter set for 1 model and 1 catchment	20
4 How to create a new model	21
4.1 Create the model description	21
4.2 Create the model file	22
4.3 Create the parameter range file	37
4.4 Recommended quality control tests	38
5 Create a new <i>flux function</i>	39
5.1 General approach	39
5.2 The linear reservoir – using one parameter and one store	39
5.3 The non-linear reservoir - using multiple parameters	40
5.4 The capillary rise flux – using multiple parameters and stores	41
5.5 The store overflow – using logistic smoothing of equations	41
5.6 The store overflow 2.0 – using optional parameters	42
6 Matlab root-finding optimization	44
6.1 Fzero modifications	44
6.2 Fsolve modifications	44
6.3 Lsqnonlin modifications	45
7 Running MARRMoT in Octave	46
7.1 Set the path	46
7.2 Load the optimization package	46
7.3 Caution	46
8 References	47

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 paper** – “Modular Assessment of Rainfall-Runoff Models Toolbox (MARRMoT) v1.0: an open-source, extendable framework providing implementations of 46 conceptual hydrologic models as continuous space-state formulations” [doi]: describes the rationale behind MARRMoT development;
2. **Supporting Material S2 – Model Descriptions**: this contains descriptions of 46 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. **Supporting Material S3 – Equations table**: describes how the constitutive equations given in the model descriptions are implemented as Matlab code;
4. **Supporting Material S4 – Unit Hydrographs table**: describes the input requirements and general functioning of the currently implemented Unit Hydrograph routing functions;
5. **Supporting Material S5 – Parameter ranges**: describes the reasoning and provides references to support the provided MARRMoT parameter ranges.

1.2 Contents

This manual provides practical guidance for MARRMoT users. Topics covered:

1. Understanding *model files*, data requirements and time step sizes (section 2);
2. How to use a model that is part of the framework (section 3);
3. How to create a new model from scratch (section 4);
4. How to create a new flux equation for a model that is part of the framework (section 5);
5. Possible speed-ups to Matlab root-finding methods (section 6);
6. Quick guide to running MARRMoT in Octave (section 7).

Certain words/phrases in the text of this manual are *italicized*. These are words with a specific meaning, defined in section 1.5.

1.3 General toolbox outline

MARRMoT currently provides model code for 46 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 (Figure 1). *Model files* specify the inner workings of a given model.

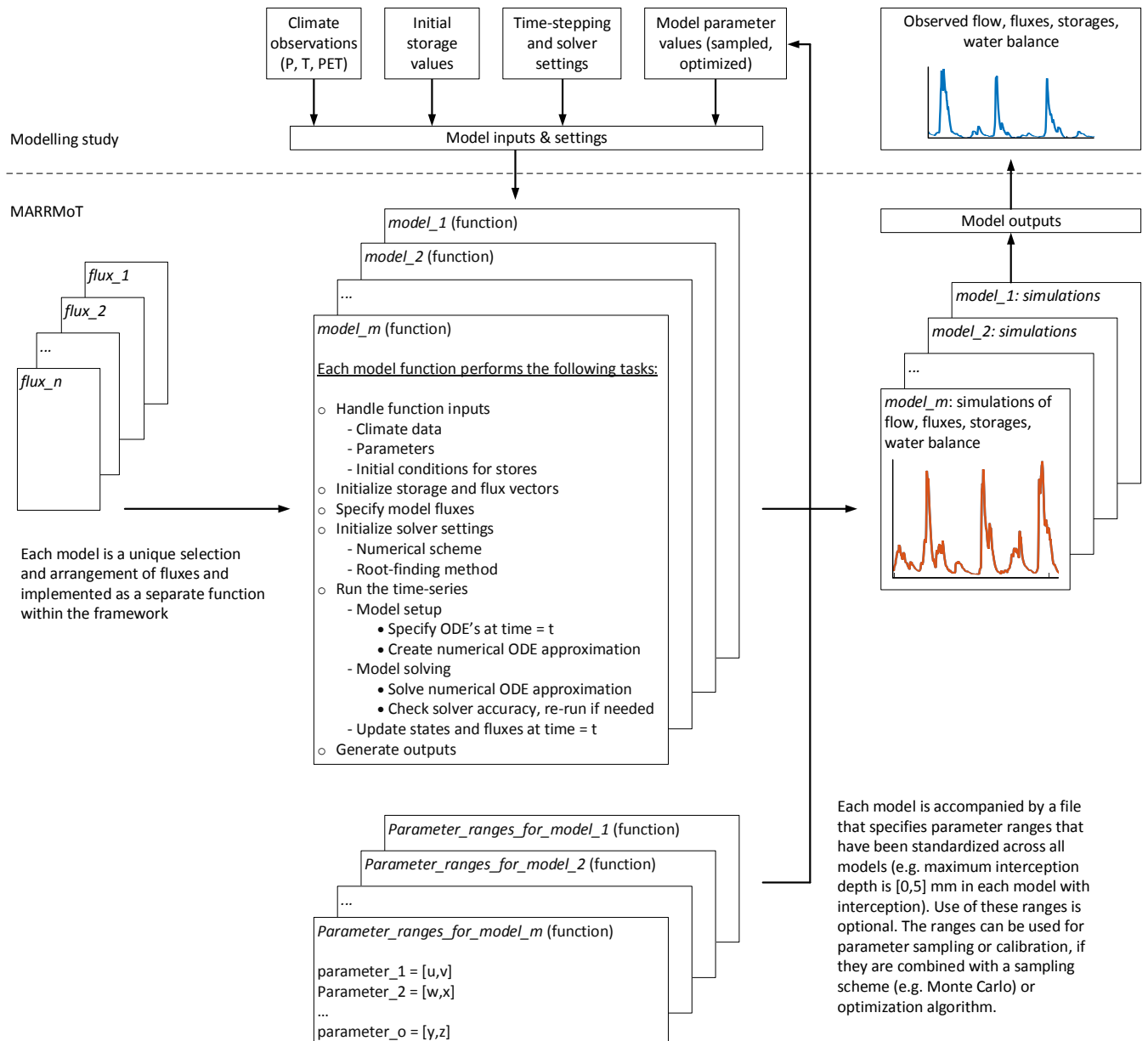


Figure 1: Schematic overview of the MARRMoT framework (Figure 1 in the MARRMoT paper). MARRMoT provides 46 conceptual models implemented in a standardized way (part below the dotted line). Each model is a unique collection and arrangement of fluxes, but the code-wise setup of each model is the same. Inputs required to run a model are time series of climate variables, values for the model parameters (which can optionally be sampled or optimized using provided, standardized ranges), and initial conditions for each model store. The model returns time series of simulated flow, fluxes and storages and a summary of the simulated water balance.

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. These are the Kling-Gupta efficiency (Gupta et al., 2009) calculated on regular flows, inverted flows (e.g. Garcia et al., 2017) and a combination of the two.
 - Solver functions: contains a function to re-run a solver if accuracy of a solution is below a user-specified threshold.
 - Time stepping: contains functions that create numerical approximations of any Ordinary Differential Equations (ODEs) that describe a model's change in storage per time. Currently contains functions for Explicit Euler and Implicit Euler.
 - Water balance: contains a function that calculates the water balance for most models.
- 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*
 - Parameter range files: contains *parameter range files*
 - Unit hydrograph files: contains Unit Hydrograph functions. These spread a single input pulse over a user-specified number of time steps. Used in various models to mimic flow routing.
- 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
Flux equation	<p>Equation that represents a certain understanding of a hydrological process in mathematical terms. In MARRMoT, <i>flux equations</i> are implemented as anonymous functions using <i>flux files</i>.</p> <p>Example: Baseflow is sometimes understood to have a linear relationship with catchment storage. A suitable equation to represent this behaviour is $q_b = k_s * S$. Where q_b is simulated baseflow [$mm\ t^{-1}$], S the current catchment storage [mm], and k_s a coefficient that connects storage to baseflow [t^{-1}]. In code:</p> <pre> Example using the notation above Q = @(ks,S) ks.*S; Same code with generalized notation func = @(p1,S) p1.*S; </pre>
Flux file	<p>File that contains code to create a single anonymous function. The anonymous function is of the shape “handle = @(inputs) f(inputs)”. Here f(inputs) is the <i>flux equation</i> that this flux file creates. Example:</p> <pre> File name baseflow_1.m File contents function [func] = baseflow_1(~) %baseflow_1 Creates function for baseflow: linear reservoir func = @(p1,S) p1.*S; end </pre>
Model descriptions	Document that gives model equations. See Supporting Material S2 .
Model file	A file unique to a given model. It specifies which <i>flux files</i> are used within the model and the ODE’s that describe the change in model storage through time. It also contains all the code necessary to run the model (see the block labelled “ <i>model_m (function)</i> ” in Figure 1).
Parameter file	A file that accompanies every model. Its output is a matrix with minimum and maximum values for each parameter in the model.
Structure	Arrays with named fields that can contain data of varying types and sizes (Matlab documentation). In MARRMoT, <i>structures</i> are used to define certain <i>model file</i> inputs. Most <i>model file</i> output comes as <i>structures</i> too. The user can specify the <i>structure’s</i> name. The names of the fields within each <i>structure</i> must follow certain naming conventions. See sections 2.1.1 and 2.1.2 for details.

2 Navigating a model file

This section shows how each *model file* is set up (section 2.1), which inputs are needed to use a *model file* (section 2.1.1), which outputs are provided (section 2.1.3) and an in-depth look into a simple *model file* (section 2.2).

2.1 Setup of a model file

All *model files* follow the same general layout (Figure 1, box named “*model_m* (function)”). Inputs and outputs are standardized. This ensures that it is straightforward to test different models within a single study. The general layout is as follows:

```
[fluxOutput, fluxInternal, storeInternal, waterBalance] =
    modelFunction(fluxInput, storeInitial, theta, solver) (1)
```

2.1.1 Data requirements and time step size

Using any of the current 46 MARRMoT models requires time series of precipitation, temperature and potential evapotranspiration data. These must be provided as a *structure* with pre-defined names (see section 2.1.2 for an example). Currently, each model requires an input *structure* with fields “.precip”, “.temp” and “.pet”. However, not every model requires temperature data for its calculations. In these cases, a placeholder value can be used in the field “.temp”, instead of a time series (e.g. [structure_name].temp = NaN).

Climate input can use an arbitrary time resolution (e.g. hourly, daily, monthly), but it must be the same for precipitation, potential evapotranspiration and temperature time series. The time resolution of the climate data must be specified in the same *structure* as the data are in, in a field called “.delta_t”. .delta_t must be specified as a fraction or multiple of the units [days]. E.g. daily climate data has $\Delta t = 1$, hourly data has $\Delta t = 1/24$, and weekly data has $\Delta t = 7$. Model outputs (simulated runoff, internal fluxes) are given at the same temporal resolution as the climate inputs.

The internal model calculations use [mm/d] as its base unit and units for model states and model parameters are derived from this (e.g. [mm] for storages, [d⁻¹] for runoff coefficients, [d] for unit hydrograph delays). Precipitation and PET inputs are internally converted from [mm/ Δt] to [mm/d], using the user-specified time step size in .delta_t. Internal model fluxes (e.g. saturation excess, evaporation, total streamflow) are internally calculated in [mm/d] and converted back to [mm/ Δt] during model output generation.

To summarize, model input can be of any temporal resolution provided it is the same for all climate input time series. The time step size must be specified by the user. Model outputs are returned at the same time resolution as the climate input data. Model parameter values are independent of the user-provided time step size.

2.1.2 Input to a model file

Inputs to a *model file* are expected in a fixed order (eq. 1). They are:

fluxInput	Climate data input. This is expected as a Matlab <i>structure</i> 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 $\Delta t = 1$ [d], whereas hourly data would have $\Delta t = 1/24$ [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;`). It would be straightforward to allow different inputs (e.g. minimum and maximum temperature time series) in a new user-created model, by using new *structure* field names (e.g. `example.temp_min` and `example.temp_max`). Existing models would not need to be changed if the climate input *structure* has more fields than the four fields already expected.

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.

storeInitial	Initial values for each model store. This is expected as a vector with a length equal to the number of stores. The ordering of stores can be found in the <i>model file</i> , by looking at the comments in the “%% Setup section”. The header “%%INITIALISE MODEL STORES” shows the part of the code that handles initial storages.
theta	Parameter values for each model parameter. This is expected as a vector with a length equal to the number of stores. The ordering of parameters can be found in the <i>model file</i> , by looking at the comments in the “%% Setup section”. The header “% Parameters” shows the part of the code that handles parameter values.
solver	Settings for the solver and time stepping scheme. This is expected as a Matlab <i>structure</i> with the following fields: <ul style="list-style-type: none"> - <code>example.name</code> - <code>example.resnorm_tolerance</code> - <code>example.resnorm_maxiter</code>

.name contains the name of the time stepping functions that should be used. This dictates how the ODE equations that describe the model’s change in storages per time are solved on every time step. Currently Explicit Euler and Implicit Euler are provided, with names `'createOdeApprox_EE'` and `'createOdeApprox_IE'` respectively. These functions can be found in `./MARRMoT/Functions/Time stepping/`.

.resnorm_tolerance specifies the required accuracy for estimates of new storage values. Ideally, the solver returns an exact solution for each new storage value that satisfies the chosen numerical scheme (e.g. $\frac{S_{new} - S_{old}}{\Delta t} - (P(t) - Q(S_{new})) = 0$ in the case of an 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:

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

If the solver has not found an accurate enough solution, the storages are calculated ones more with a more thorough but slower solver. The current default solvers are 'fzero' for models with 1 store and 'fsolve' for models with >1 stores. 'lsqnonlin' is a more robust but slower solver. 'lsqnonlin' is the solver used when 'fzero' or 'fsolve' can't find a solution that satisfies the .resnorm_tolerance.

.resnorm_maxiter (default = 6) specifies the maximum number of iterations that can be spent to recalculate storage values with 'lsqnonlin', when 'fzero' or 'fsolve' fail to find a sufficiently accurate solution.

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

2.1.3 Output of a model file

Outputs generated by a *model file* (eq. 1) are as follows:

fluxOutput Fluxes 'leaving' the model. Given as a structure with at least the fields:

- example.Q
- example.Ea

.Q contains a time series of the total simulated streamflow in the same time resolution as the climate input. In most cases, this is the sum of various internal fluxes that represents different types of runoff (e.g. surface runoff, interflow, baseflow)

.Ea contains a time series of the total simulated evaporation in the same time resolution as the climate input. In several cases, this is the sum of various internal fluxes that represent different types of evaporation (e.g. bare soil evaporation and transpiration)

In several cases, other model-specific fields are also included in this output structure, that might represent fluxes such as a groundwater sink.

fluxInternal Fluxes internal to the model. 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 that are not given in the fluxOutput structure. See the model descriptions in **Supporting Material S2**. for schematics that show the flux names.

storeInternal Storages in the model. Given as a structure with a number of fields equal to the number of stores in the model. Currently, all models include at least 1 store:

- example.S1

.S1 contains a time series of store 1 storage values during the simulation period, 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 1 store, subsequent stores are named .S2, .S3, etc

waterBalance Returns 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.

2.2 Step-by-step description of a model file

This section gives a step-by-step overview of a *model file*. Figure 1 lists the steps taken in each *model file*:

1. Handle function inputs
2. Initialize storage and flux vectors
3. Specify model fluxes
4. Initialize solver settings
5. Run the time series
6. Generate outputs

Each step is discussed here, using a classic bucket model as an example. This model is included in MARRMoT in the file “m_01_colle1_1p_1s.m”. Each step is discussed in the next sections of this document.

2.2.1 Handle function inputs

```
%% Setup
%%INPUTS
% Time step size
delta_t = fluxInput.delta_t;
```

[Lines 31-34] The time step size is taken from the “fluxInput” *structure* and assigned to a temporary variable. This information is essential to convert climate inputs from the user’s [mm/ Δt] resolution to the internal units of [mm/d]. It is also used to calculate the numerical approximation of the model’s ODEs and to convert the internal fluxes and output fluxes back from [mm/d] to the user’s [mm/ Δt].

```
% Data: adjust the units so that all fluxes (rates) inside this model
% function are calculated in [mm/d]
P      = fluxInput.precip./delta_t;           % [mm/delta_t] > [mm/d]
Ep     = fluxInput.pet./delta_t;              % [mm/delta_t] > [mm/d]
T      = fluxInput.temp;
t_end = length(P);
```

[Lines 36-41] Climate input is taken from the “fluxInput” *structure*, converted into [mm/d] where needed and assigned to temporary variables. The number of time steps in the time series is calculated. This information is later used as the length of the modelling loop.

```
% Parameters
% [name in documentation] = theta(order in which specified in parameter file)
S1max  = theta(1);                          % Maximum soil moisture storage [mm]
```

[Lines 43-45] Parameter values are taken from the third input variable “theta” and assigned to a temporary variable. Almost always, these parameters share names with their counterparts in the *model descriptions*. Occasionally, auxiliary or derived parameters are used. This is clearly marked inside the *model file* if applicable.

```
%%INITIALISE MODEL STORES
S10     = storeInitial(1);                  % Initial soil moisture storage
```

[Lines 47-48] Initial storage values are taken out of the second input variable “storeInitial” and assigned to a temporary variable.

```
%%DEFINE STORE BOUNDARIES
store_min = [0];           % lower bounds of stores
store_upp = [];           % optional higher bounds
```

[Lines 50-52] Lower and upper storage bounds are defined. These are used within the ‘lsqnonlin’ solver, in case the default solvers (‘fzero’ and ‘fsolve’) do not provide a sufficiently accurate solution. Generally, providing store minimum bounds is useful and possible. Upper bounds are generally harder to define and do not seem to provide any reasonable benefit to the solver.

2.2.2 Initialize storage and flux vectors

```
%%INITIALISE STORAGE VECTORS (all upper case)
store_S1 = zeros(1,t_end);

flux_ea   = zeros(1,t_end);
flux_qse  = zeros(1,t_end);
```

[Lines 54-58] Zero vectors are created for all model stores and fluxes, to allocate memory efficiently. These are filled with values during the model run.

2.2.3 Specify model fluxes

```
%% 3. Specify and smooth model functions
% Store numbering:
% S1. Soil moisture
```

[Lines 66-68] The store numbering that is used in this *model file* is shown. This numbering is important during model file creation. It shows the order of expected initial storage value in the “storeInitial” input variable and serves as a memory aid when the user specifies the model equations and inputs to the *flux files*.

```
% EA(S1,Smax,Ep,delta_t): evaporation from soil moisture
EA = evap_7;

% QSE(P,S1,Smax): Saturation excess flow
QSE = saturation_1;
```

[Lines 70-74] This model uses only two *flux files*: one to simulate evaporation and one to simulate saturation excess overflow. EA and QSE use function calls to functions “evap_7” and “saturation_1” respectively. These functions return the function handle to an anonymous function each. The commented lines above each function handle assignment show which inputs each function will use (e.g. EA(S1,Smax,Ep,delta_t)) and a summary of which flux this represents (e.g. *evaporation from soil moisture*).

The anonymous function assigned to EA represents the following constitutive relationship:

$$E_a = \frac{S}{S_{max}} E_p(t), \quad E_a \leq \frac{S}{\Delta t} \quad (2)$$

And looks like:

```
func = @(S,Smax,Ep,dt) min(S./Smax.*Ep,S/dt);
```

[Line 20 in “evap_7.m”] This function gives a mathematical implementation of evaporation that occurs at the potential rate E_p when soil moisture S is at maximum capacity S_{\max} and decreases linearly as storage drops below its maximum level. Evaporation is not allowed to be larger than the total available storage.

The anonymous function assigned to QSE represents the following constitutive relationship:

$$Q_{se} \begin{cases} P(t), & \text{if } S = S_{\max} \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

And looks like:

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

[Lines 21-27 in “saturation_1.m”] This function gives a mathematical implementation of flow that only occurs when storage S is at its maximum capacity S_{\max} . This is achieved through a logistic smoothing function (Clark et al., 2008; Kavetski and Kuczera, 2007) that is 0 when $S < S_{\max}$ and 1 when $S \geq S_{\max}$. The smoothing function has two parameters, r and e . These can be specified as optional arguments when “saturation_1” is first called (e.g. QSE = saturation_1(r,e)). If no values are specified the defaults $r = 0.01$, $e = 5.00$ are used (Clark et al., 2008). See the files “smoothThreshold_storage_logistic” and “smoothThreshold_temperature_logistic” in the folder ./MARRMoT/Functions/Flux smoothing/ for more details.

2.2.4 Initialize solver settings

```
%% 4. Determine numerical scheme and solver settings
% Function name of the numerical scheme
scheme = solver.name;
```

[Lines 76-78] Find the name of the time stepping scheme function from the fourth input *structure* “solver”.

```
% Define which storage values should be used to update fluxes
[~,store_fun] = feval(scheme,storeInitial,delta_t); % storeInitial = number
stores
```

[Lines 80-81] This section evaluates the time stepping function and requests the second output only. This returns a string that tells Matlab which variables it should use to update the model fluxes. These variables differ in different numerical ODE approximations; e.g. with an Explicit Euler scheme the variable $S(t-1)$ is used to update flux(t), while with an Implicit Euler scheme the variable $S(t)$ is estimated iteratively and used to update flux(t). In the Explicit Euler case, the returned string reads:

```
`tmp_sFlux = [S1old]'
```

While in the Implicit Euler case the string reads (for this 1-store model):

```
`tmp_sFlux = [tmp_sNew(1)]'
```

In both cases, `tmp_sFlux` is the variable used in the remainder of the model function (lines 138-139), but the variables assigned to it are different (and appropriate for the chosen numerical scheme). The appropriate variables are assigned to `tmp_sFlux` in line 135.

```
% settings of the root finding method
fzero_options = optimset('Display','off');
lsqnonlin_options = optimoptions('lsqnonlin',...
                                'Display','none',...
                                'MaxFunEvals',1000);
```

[Lines 83-87] Options for the root-finding method are defined. This *model file* uses only a single store and thus has only a single ODE that needs to be solved on each time step (i.e. the change in storage of the single model store). For this problem the ‘fzero’ solver can be used. For a multi-store model, several ODEs need to be solved simultaneously and this requires use of the ‘fsolve’ solver. The ‘lsqnonlin’ solver is only called when the first solver (‘fzero’ in this case) is not sufficiently accurate as specified by the `resnorm tolerance` user input (see section 2.1.1). Through a large number of trials, it seems that ‘lsqnonlin’ usually needs in the order of 10 to 100 iterations to converge. Therefore, a default maximum value of 1000 evaluations is used. In general, the only cases where both solvers struggle to find an accurate solution is when unrealistically small store sizes are used (<1 mm store depth).

By default, all solver display settings are turned off to avoid unnecessary printing to the display. It can be helpful to turn the display settings on for debugging purposes.

Note: for models with multiple stores, an additional option of ‘fsolve’ and ‘lsqnonlin’ is used, that allows a user to specify a Jacobian matrix for the multi-store problem. Specifying the Jacobian is not required but can significantly reduce computational times. An example is provided in section 0.

2.2.5 Run the time series

On every time step, three different actions are performed:

1. First, the ODEs for this time step are defined with current climate inputs and rewritten in terms of the chosen time stepping scheme;
2. Next, the (collection of) storage equations are solved for the given time step, and the accuracy of these solutions is compared to a user-specified threshold;
3. Finally, model fluxes and storages are updated.

```
%% 5. Solve the system for the full time series
for t = 1:t_end
```

[Lines 89-90] Start of the time loop.

```
% Model setup -----
% Determine the old storages (at t-1)
if t == 1; S1old = S10; else; S1old = store_S1(t-1); end
```

[Lines 92-94] Storages at t-1 are stored into temporary variables. These are later used as starting points for the numerical solver(s).

```
% Create temporary store ODE's that need to be solved
tmpf_S1 = @(S1)      (P(t) - ...
                     EA(S1,S1max,Ep(t),delta_t) - ...
                     QSE(P(t),S1,S1max));
```

[Lines 96-99] The right-hand side of each ODE is created. ODEs are specified in the *model descriptions*. For this model, it would read:

$$\frac{dS}{dt} = P(t) - E_a(S, S_{max}, E_p(t), \Delta t) - Q_{se}(S, S_{max}, P(t)) \quad (4)$$

With constitutive functions

$$E_a = \frac{S}{S_{max}} E_p(t) \quad (5)$$

$$Q_{se} \begin{cases} P(t), & \text{if } S = S_{max} \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

The constitutive functions have been defined before as function handles assigned by *flux files*. Here, the proper variables and parameters are inserted into these and a new anonymous function `tmpf_S1` is created. The unknown in `tmpf_S1` is the storage value `S1`. Parameter S_{max} , climate $P(t)$ and $E_p(t)$ and time step size Δt [d] are known.

```
% Call the numerical scheme function to create the ODE approximations.
% This returns a new anonymous function that we solve in the next step.
solve_fun = feval(scheme,...
                 [S1old],...
                 delta_t,...
                 tmpf_S1);
```

[Lines 101-106] Here the entire ODE is re-written to a form that allows the use of a root-finding method:

$$\frac{dS}{dt} - (P(t) - E_a(S, S_{max}, E_p(t)) - Q_{se}(S, S_{max}, P(t))) = 0 \quad (7)$$

The choice of time stepping method specified in `scheme` determines how the differential equation is treated and which storage value the constitutive functions depend on. In the case of an Implicit Euler scheme, `solve_fun` is:

$$\frac{S_{new} - S1old}{\delta_{\Delta t}} - (P(t) - E_a(S_{new}, S_{max}, E_p(t)) - Q_{se}(S_{new}, S_{max}, P(t))) = 0 \quad (8)$$

Where the part between the brackets is the anonymous function `tmpf_S1` with input S_{new} . However, due to its general nature, the function that generates `solve_fun` can not show this specific equation. Instead `solve_fun` shows a generalized version:

$$\frac{S_{new} - S1old}{\delta_{\Delta t}} - varargin(S_{new}) = 0 \quad (9)$$

Where $\text{varargin}(S_{\text{new}})$ fulfils the same function as the specific equation in eq. 8. In case of a multi-store model, `solve_fun` would be a matrix containing a similarly re-written ODE for each store in the model.

```
% Model solving -----
% --- Use the specified numerical scheme to solve storages ---
[tmp_sNew,tmp_fval] = fzero(solve_fun,...
                           S1old,...
                           fzero_options);
```

[Lines 108-112] Here the equation stored in `solve_fun` (eq. 8) is solved using Matlab's 'fzero' algorithm. The starting point for the solver is the storage value at t-1 `S1old`. `fzero_options` contains solver settings specified before. Section 6.1 details a small modification that can be made to Matlab's 'fzero' to suppress output message generation. The gains in computational efficiency are significant however.

Function output `tmp_sNew` contains the solver's estimate of the storage value at time = t. Output `tmp_fval` contains the resulting function value if the new store estimate from `tmp_sNew` were to be used as S_{new} in equation 8. If the solver has found a proper solution, this value is approximately zero.

```
% --- Check if the solver has found an acceptable solution and re-run
% if not. The re-run uses the 'lsqnonlin' solver which is slower but
% more robust. It runs solver.resnorm_iterations times, with different
% starting points for the solver on each iteration ---
tmp_resnorm = sum(tmp_fval.^2);
```

[Lines114-118] Normalized residuals are calculated using the new estimated storage value(s).

```
if tmp_resnorm > solver.resnorm_tolerance
    [tmp_sNew,~,~] = rerunSolver('lsqnonlin', ...
                                lsqnonlin_options, ...
                                @(eq_sys) solve_fun(...
                                    eq_sys(1)), ...
                                solver.resnorm_maxiter, ...
                                solver.resnorm_tolerance, ...
                                tmp_sNew, ...
                                [S1old], ...
                                store_min, ...
                                store_upp);
end
```

[Lines 120-131] If the residuals are above a user-specified threshold `solver.resnorm_tolerance`, new storages are calculated again with the more robust solver 'lsqnonlin'. Solver options `lsqnonlin_options` have been specified before. "`@(eq_sys) solve_fun(eq_sys(1))`" is a construction that lets 'lsqnonlin' interact properly with the function it needs to solve. Multi-store solver 'fsolve' uses the same construction. `tmp_sNew` and `S1old` are the (inaccurately estimated) new storages and the old storages from t-1 respectively. Both are used as starting points for 'lsqnonlin' in its attempts to find a better solution. `store_min` and `store_upp` are lower and upper store bounds respectively. Both are optional, but the lower bounds are occasionally useful to constrain the solver to realistic store estimates. Upper bounds are generally less useful because they are harder/impossible to define.

```
% Model states and fluxes -----
% Find the storages needed to update fluxes: update 'tmp_sFlux'
eval(store_fun);
```

[Lines 133-135] This evaluates the string created in “%% 4. Determine numerical scheme and solver settings”, which tells Matlab which variables it should use to update the model fluxes.

```
% Calculate the fluxes
flux_ea(t) = EA(tmp_sFlux(1), S1max, Ep(t), delta_t);
flux_qse(t) = QSE(P(t), tmp_sFlux(1), S1max);

% Update the stores
store_S1(t) = S1old + (P(t) - flux_ea(t) - flux_qse(t)) * delta_t;
```

[Lines 137-142] The time series of flux and storage values are updated using the appropriate storage values; e.g. $S(t-1)$ with an Explicit Euler scheme and final estimates of $S(t)$ with an Implicit Euler scheme. Storages are updated based on the calculated fluxes, converted into the proper time step size.

```
end
```

[Line 144] End of the time loop.

2.2.6 Generate outputs

```
%% 6. Generate outputs
% --- Fluxes leaving the model ---
% 'Ea' and 'Q' are used outside the function and should NOT be renamed
fluxOutput.Ea = flux_ea * delta_t;
fluxOutput.Q = flux_qse * delta_t;

% --- Fluxes internal to the model ---
fluxInternal.noInternalFluxes = NaN;

% --- Stores ---
storeInternal.S1 = store_S1;
```

[Lines 146-156] Temporary vectors with flux and storage values are assigned to the appropriate output *structures*. Because this is such a simple model, both fluxes describe processes the ‘leave’ the model (evaporation and streamflow), so the fluxInternal *structure* is filled with a placeholder value. In the time loop fluxes have been calculated in [mm/d] for consistency with parameter and storage units. During output generation these fluxes are converted back into the user-specified [mm/Δt]. Storage values do not need to be changed, because these are already based on flux values given in [mm/Δt] (see lines 137-142).

```

% Check water balance
if nargout == 4
    waterBalance = ...
        checkWaterBalance(...
            P,...           % Incoming precipitation
            fluxOutput,...   % Fluxes Q and Ea leaving the model
            storeInternal,... % Time series of storages ...
            storeInitial,... % And initial store values to calculate delta S
            0);              % Whether the model uses a routing scheme that
                             % still contains water. Use '0' for no routing
end

```

[Lines 158-168] If requested, a water balance check is performed. This returns 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.

3 Using the framework

This section provides several examples accompanied by computer code in the folder “./MARRMoT/User manual/”. The examples show how to use a model from the framework in a few basic applications. The first example shows how a pre-defined model can be used to simulate runoff in a catchment using a single parameter set. The second example shows how the provided parameter ranges for each model can be used to generate random parameter sets for a model. The third example shows how several pre-defined models can be used in a single loop. The fourth example shows how a model can be calibrated using a few of MARRMoT’s provided functions.

This guide uses 5 years’ worth of climate and streamflow data from Buffalo River near Flat Woods, Tennessee, USA, to illustrate examples. 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.

3.1 Setup: add MARRMoT folders to the Matlab path

MARRMoT files are spread out in different folders within the main ./MARRMoT/ folder. These must be added to the Matlab path:

1. Open Matlab
2. Navigate Matlab’s “current folder” to the folder that contains MARRMoT
3. Right-click the MARRMoT folder
4. Select [Add to Path] > [Selected Folders and Subfolders]
5. **Note:** ensure that the folder “Octave” is not part of this folder structure. Remove the folder if present

3.2 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”.

3.3 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”.

3.4 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”.

3.5 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”. **Note:** this workflow example does not work with Octave.

4 How to create a new model

This section shows how a new model can be created to fit within MARRMoT. The current 46 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*)
- The time step size can be specified by the user, but the internal *model file* computations use [mm/d] as the base unit

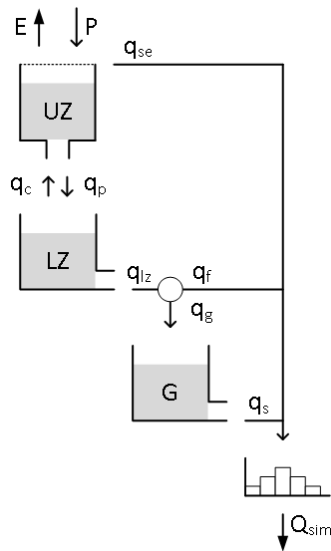
For simplicity, we assume that the new model created in this section is built according to certain assumptions of how a particular catchment functions (i.e. on some perceptual model of the catchment). Justifying these assumptions is outside the scope of this guide. This section is intentionally divided 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 Create the model description

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

4.1.1 Create a model schematic based on assumptions about the catchment

Create a model schematic that shows the behaviour the model is intended to simulate (Figure 2).



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

4.1.2 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. This model has three stores, so three ODEs are needed:

$$\frac{dUZ}{dt} = P + q_c - E - q_{se} - q_p \quad (10)$$

$$\frac{dLZ}{dt} = q_p - q_{lz} - q_c \quad (11)$$

$$\frac{dG}{dt} = q_g - q_s \quad (12)$$

4.1.3 Specify the constitutive functions that define the model fluxes

Next, define the constitutive equations that describe the individual fluxes. 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. The following equations reflect several of such assumptions, but it is beyond the scope of this guide to justify these.

$$E = E_p \frac{UZ}{UZ_{max}} \quad (13)$$

$$q_c = c_{rate} \left(1 - \frac{UZ}{UZ_{max}}\right) \quad (14)$$

$$q_{se} = \begin{cases} P, & \text{if } UZ = UZ_{max} \\ 0, & \text{otherwise} \end{cases} \quad (15)$$

$$q_p = p_{rate} \quad (16)$$

$$q_{lz} = k_{lz} * LZ \quad (17)$$

$$q_g = \alpha * q_{lz} \quad (18)$$

$$q_s = k_g * G \quad (19)$$

In some cases (such as this one) not all fluxes are directly part of an ODE. The fraction of lower zone outflow that becomes fast flow is not yet specified:

$$q_f = (1 - \alpha) * q_{lz} \quad (20)$$

Last, the triangular routing scheme distributes the incoming runoff in a triangular way over a certain time period. By definition, the area under the triangle sums to 1 (see section 4.2.8 for details). Now all required equations are known. Our 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].

4.2 Create the model file

The next step is creating the model file.

4.2.1 Copy and rename a *model file*

Navigate to the folder “./MARRMoT/Models/Main” and copy the file “m_00_template_5p_2s.m”. Paste this file in the same directory and rename it (Figure 3). The new name should follow the same structure as the current *model files*:

“m_[number]_[name in lower case]_[number of parameters]p_[number of stores]s.m”

The example model created in this manual can be found in the folder “./MARRMoT/User manual”.

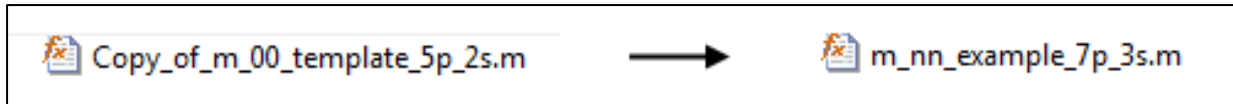


Figure 3: Copy the template model file and rename it. Left: template model. Right: example model

4.2.2 Open the new *model file* and change the function name

Open the renamed *model file* and change the function's name to match the file name. Optional: change the comments to reflect the model's name and provide a reference (Figure 4).

```
function [ fluxOutput, fluxInternal, storeInternal, waterBalance ] = ...
    m_00_template_5p_2s( fluxInput, storeInitial, theta, solver )
% Hydrologic conceptual model: [xxx]
%
% Model reference
% [reference]
```

```
function [ fluxOutput, fluxInternal, storeInternal, waterBalance ] = ...
    m_nn_example_7p_3s( fluxInput, storeInitial, theta, solver )
% Hydrologic conceptual model: [MARRMoT User Manual example model]
%
% Model reference
% MARRMoT User Manual, 2018.
```

Figure 4: Change the function's name to match the file name. Top: template model. Bottom: example model

4.2.3 Do not change the “INPUTS – Time step size and Data” section

Climate input is handled in a standardized way across all models. This does not need to be changed (Figure 5).

```
%%INPUTS
% Time step size
delta_t = fluxInput.delta_t;

% Data
P      = fluxInput.precip./delta_t;          % [mm/delta_t] > [mm/d]
Ep     = fluxInput.pet./delta_t;             % [mm/delta_t] > [mm/d]
T      = fluxInput.temp;
t_end = length(P);
```

Figure 5: Do not change the “Inputs – time step size” and “Inputs – data” sections

4.2.4 Update the “INPUTS – Parameters” section

This part of the code assigns parameter values from the *model file* input variable “theta” to temporary variables. The easiest approach is using the parameter names from the *model description* as names for these variables (Figure 6). **Note:** make sure to increment the index of “theta” for models with more than 5 parameters.

<pre> % Parameters % [name in documentation] = theta(order in which specified in parameter file) S1max = theta(1); % Maximum soil moisture storage [mm] kc = theta(2); % Maximum capillary rise [mm/d] kp = theta(3); % Maximum percolation [mm/d] ks = theta(4); % Runoff coefficient [d-1] delay = theta(5); % Routing delay [d] % ... </pre>
<pre> % Parameters % [name in documentation] = theta(order in which specified in parameter file) crate = theta(1); % Maximum capillary rise rate [mm/d] uzmax = theta(2); % Maximum upper zone storage [mm] prate = theta(3); % Maximum percolation rate [mm/d] klz = theta(4); % Lower zone runoff coefficient [d-1] alpha = theta(5); % Fraction of lower zone runoff to groundwater [-] kg = theta(6); % Groundwater runoff coefficient [d-1] d = theta(7); % Routing delay [d] </pre>

Figure 6: Update the parameter section. Top: template model. Bottom: example model

4.2.5 Update the “INITIALISE MODEL STORES” section if the model has 1 or >2 stores

This part of the code assigns initial storage from the *model file* input variable “storeInitial” to temporary variables. If the model has 1 or more than 2 stores, this section needs to be updated. **Note:** make sure to increment the index of “storeInitial” for models with more than 2 stores.

<pre> %%INITIALISE MODEL STORES S10 = storeInitial(1); % Initial soil moisture storage S20 = storeInitial(2); % Initial groundwater storage % ... </pre>
<pre> %%INITIALISE MODEL STORES S10 = storeInitial(1); % Initial upper zone storage S20 = storeInitial(2); % Initial lower zone storage S30 = storeInitial(3); % Initial groundwater storage </pre>

Figure 7: Update initial storages section. Top: template model. Bottom: example model

4.2.6 Define store boundaries

This section defines upper and lower store boundaries that can be used by the ‘lsqnonlin’ solver. The example model has three stores, all of which have a defined lower boundary of $S = 0$. The upper zone has a defined maximum storage given by the parameter UZ_{max} , but the lower zone and groundwater stores have a theoretical infinite storage. In practice however, defining the upper store boundaries has little to no benefit for the solver. The option to define them is included in MARRMoT but not used by any of the included models. For this example model, the upper store boundaries vector is kept empty (Figure 8).

<pre> %%DEFINE STORE BOUNDARIES store_min = [0,0]; % lower bounds of stores store_upp = []; % optional higher bounds </pre>
<pre> %%DEFINE STORE BOUNDARIES store_min = [0,0,0]; % lower bounds of stores store_upp = []; % optional higher bounds </pre>

Figure 8: Update the store boundary vector(s). Top: template model. Bottom: example model

4.2.7 Define empty flux and storage vectors

Allocating vectors of the right size before using them (in contrast to increasing their size by one per iteration) results in increased computational efficiency. Create an empty vector for each store and each flux (Figure 9).

<pre>%%INITIALISE STORAGE VECTORS store_S1 = zeros(1,t_end); store_S2 = zeros(1,t_end); % ... flux_cap = zeros(1,t_end); flux_ea = zeros(1,t_end); flux_qo = zeros(1,t_end); flux_perc = zeros(1,t_end); flux_qs = zeros(1,t_end); flux_qt = zeros(1,t_end); % ...</pre>		<pre>%%INITIALISE STORAGE VECTORS store_S1 = zeros(1,t_end); store_S2 = zeros(1,t_end); store_S3 = zeros(1,t_end); flux_qse = zeros(1,t_end); flux_e = zeros(1,t_end); flux_qp = zeros(1,t_end); flux_qc = zeros(1,t_end); flux_qlz = zeros(1,t_end); flux_qf = zeros(1,t_end); flux_qg = zeros(1,t_end); flux_qs = zeros(1,t_end); flux_qt = zeros(1,t_end);</pre>
---	--	--

Figure 9: Create empty storage and flux vectors. Left: template model. Right: example model

4.2.8 Select the routing weighting scheme if applicable

MARRMoT includes several routing schemes (Table 1: Overview of Unit Hydrograph based routing schemes in MARRMoT (see also **Supporting Material S4**)). All are based on a Unit Hydrograph (UH) principle and quantify how a single unit of input is distributed over n time steps.

The example model uses a triangular routing scheme with time base “d” (Figure 10). In the current setup of *model files*, the Unit Hydrograph (i.e. the percentage-based distribution of 1 flow unit in time) is defined before the time loop starts. Hence, only the second output of the UH *flux file* is required, and the inputs to the UH *flux file* are 1 (flow unit) and the time delay parameter “d”. “uh_full” contains the percentage-wise distribution of incoming flow over subsequent time steps.

<pre>%%PREPARE UNIT HYDROGRAPHS % [Optional] [~,uh_full] = uh_4_full(1,delay,delta_t); % ...</pre>		<pre>%%PREPARE UNIT HYDROGRAPHS [~,uh_full] = uh_4_full(1,d,delta_t);</pre>
--	--	---

Figure 10: Choose and parameterize the routing scheme if applicable. Left: template model. Right: example model

4.2.9 Update or remove the “INITIALISE ROUTING VECTOR” section


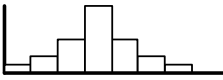
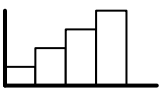
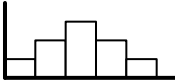
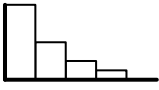
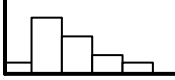
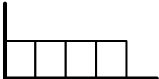
The example model has a routing component, so this section needs to be kept. The empty vector “tmp_Qt_old” will be used later when flow routing is computed.

If the model has no routing component, this section can be removed. Model 07 (GR4J) and model 34 (FLEX-IS) are good examples of models with different types of routing schemes.

<pre>%%INITIALISE ROUTING VECTORS tmp_Qt_old = zeros(1,length(uh_full));</pre>		<pre>%%INITIALISE ROUTING VECTORS tmp_Qt_old = zeros(1,length(uh_full));</pre>
--	--	--

Figure 11: Update routing storage vector with the correct parameter. Left: template model. Right: example model

Table 1: Overview of Unit Hydrograph based routing schemes in MARRMoT (see also **Supporting Material S4**)

<i>Flux file</i>	Inputs	Diagram	Description	Used in model
uh_1_half	1: amount to be routed 2: time base 3: Δt		Exponentially increasing scheme	7
uh_2_full	1: amount to be routed 2: time base (time is doubled inside the function) 3: Δt		Exponential triangular scheme	7
uh_3_half	1: amount to be routed 2: time base 3: Δt		Triangular scheme: linearly increasing	13, 15, 21, 26, 34
uh_4_full	1: amount to be routed 2: time base 3: Δt		Triangular scheme: linearly increasing and decreasing	0 (template), 16, 37, nn (example)
uh_5_half	1: amount to be routed 2: time base 3: Δt		Exponentially decreasing scheme	5
uh_6_gamma	1: amount to be routed 2: gamma parameter [-] 3: time for flow to reduce by factor e [d] 4: length of time series		Gamma function-based	40
uh_7_uniform	1: amount to be routed 2: time base 3: Δt		Uniform distribution	39

4.2.10 Specify how the model stores are numbered in the *model file*

It is generally easiest to number model stores from the top-left of the model diagram to the bottom-right. The numbering determines which variable name (e.g. “S1”, “S2”) is used to refer to each store’s current value. These comments serve the purpose of clarifying upfront how stores are ordered and prevent possible confusion when inputs for each *flux function* are determined.

```
%% 3. Specify and smooth model functions
% Store numbering:
% S1. Soil moisture
% S2. Groundwater
```

```
%% 3. Specify and smooth model functions
% Store numbering:
% S1. Upper zone
% S2. Lower zone
% S3. Groundwater
```

Figure 12: Determine store order and numbering. Top: template model. Bottom: example model

4.2.11 Update the *flux* file selection

Select the appropriate *flux* files for the model using **Table S1 (Supporting Material S3)**. Per flux, specify a function handle name (e.g. “E” for evaporation) and assign the proper *flux function* to the handle (e.g. “E = evap_7;”). Use the comments to clarify which inputs (i.e. climate, parameters, storage values, time step size) the function needs (e.g. E(S1,uzmax,Ep(t),delta_t); Figure 13).

Detailed explanation: we have specified that evaporation in the model decreases linearly as a fraction of potential evapotranspiration E_p as the upper zone dries. The *flux file* “evap_7.m” contains an anonymous function that describes the desired evaporation behaviour. In the *model file* we use the *flux file* “evap_7” to assign this anonymous function to the function handle “E”, with the line “E = evap_7;”. The function generated by “evap_7” requires 3 inputs (this can be checked by opening the *flux file* “evap_7”: S, Smax and E_p . Here S is the current storage in the store where evaporation is taken from; Smax is the maximum storage value of this store; and E_p is the current evaporation demand. We have determined that evaporation is taken from the upper zone and numbered this store as “S1”. Thus, the first input for evaporation function “E” is “S1”. The maximum value of “S1” is given by parameter “uzmax”. Thus, the second input in “E” is “uzmax”. The current evaporation demand is found in the climate vector “Ep”. Thus, the third input to “E” is “Ep(t)”.

```
% E(S1,uzmax,Ep(t),delta_t): evaporation from upper zone (S1).
E = evap_7;
```

Figure 13: Example showing how to use flux files to create flux equations in a model file

The example model only uses *flux* files that have already been included in MARRMoT. See section 5 for help with creating a new flux file. The example model uses the following *flux* files (Figure 14):

- evap_7 for evaporation E (eq. 13)
- saturation_1 for saturation excess q_{se} (eq. 15)
- percolation_1 for percolation to the lower zone q_p (eq. 16)
- capillary_1 for capillary rise from the lower to upper zone q_c (eq. 14)
- baseflow_1 for outflow from the lower zone q_{lz} (17)
- split_1 for the division between fast flow q_f and groundwater recharge q_g (eq. 20, 18)
- baseflow_1 for slow flow from groundwater q_s (eq. 19)

```

% Ea: evaporation from soil moisture. Angle discontinuity
EA = @(S1,Ep,delta_t) min(S1/delta_t,Ep);

% Qo: overflow from soil moisture. This formula uses a threshold - this
% gives a threshold discontinuity which we deal with using a logistic smoother
QO = @(P,S1,S1max) P.*(1-smoothThreshold_storage_logistic(S1,S1max,0.001));

% cap: capillary rise frm groundwater to soil moisture. This can use min
% function, this leads to an angle discontinuity
CAP = @(kc,S1,S1max,S2,delta_t) min(max(kc*(S1max-S1)/S1max,0),S2/delta_t);

% perc: percolation from soil moisture to groundwater. Angle discontinuity
% at S1 = 0
PERC = @(kp,S1,S1max,delta_t) min(kp.*S1/S1max,S1/delta_t);

% Qs: flow from groundwater. An angle discontinuity at S2 = 0
QS = @(ks,S2) ks*S2;

```

```

% E(S1,uzmax,Ep(t),delta_t): evaporation from upper zone (S1).
E = evap_7;

% QSE(P(t),S1,uzmax): saturation excess from upper zone (S1).
% Has a threshold discontinuity and needs logistic smoothing
QSE = saturation_1;

% QP(prate,S1,delta_t): percolation from upper zone (S1) to lower zone (S2)
QP = percolation_1;

% QC(crate,S1,uzmax,S2,delta_t): capillary rise from lower (S2) to upper
% zone (S1)
QC = capillary_1;

% QLZ(klz,S2): outflow from lower zone (S2)
QLZ = baseflow_1;

% QF(1-alpha,QLZ(klz,S2)): fraction (1-alpha) of lower zone outflow (QLZ)
% that is fast flow
QF = split_1;

% QG(alpha,QLZ(klz,S2)): fraction (alpha) of lower zone outflow (QLZ) that
% goes to groundwater (S3)
QG = split_1;

% QS(kg,S3): outflow from groundwater (S3)
QS = baseflow_1;

```

Figure 14: Selection of flux equations. Top: template model. Bottom: example model

4.2.12 Do not change the “Settings for the numerical scheme” section

This part of the code handles user-defined inputs that specify the choice of time-stepping scheme. These are handled in a standardized way and this section does not need to be changed (Figure 15).

```

%% 4. Determine numerical scheme and solver settings
% Function name of the numerical scheme
scheme = solver.name;

% Define which storage values should be used to update fluxes
[~,store_fun] = feval(scheme,storeInitial,delta_t);

```

Figure 15: Do not change the numerical time stepping scheme initialization section

4.2.13 Remove unnecessary solver options

‘fzero’ is the proper option for a model with 1 store, ‘fsolve’ is used for multiple stores. ‘lsqnonlin’ is used as a back-up in case the first solver is unable to find a sufficiently accurate solution. If ‘fzero’ is not used, the line that generates an option structure for ‘fzero’ can be removed. Similarly, if ‘fsolve’ is not used, the corresponding lines can be deleted.

The example model has three stores, so ‘fsolve’ must be used (Figure 16).

4.2.14 Specify the Jacobian

Specifying the Jacobian matrix increases computational efficiency in multi-store models. It specifies how the ODEs that quantify storage changes in a given store depend on storage levels in stores besides themselves (see option ‘JacobPattern’, Figure 16).

Extended reasoning: E.g. in the example model, the first ODE (eq. 10) quantifies the change in store 1 level ($\Delta S1$) over time. This depends on several fluxes: precipitation P , evaporation E , surface runoff q_{se} , percolation q_p and capillary rise q_c . P and E are store-independent; i.e. their value does not depend on the current level of any of the model stores. q_{se} and q_p are store-dependent: their value depends on the current storage in the upper zone ($S1$). q_c is multi-store-dependent: its value depends on the current (lack of) storage in the upper zone ($S1$) and the available water in the lower zone ($S2$). Thus, ODE 1 (eq. 9) describes the change in storage levels of store $S1$, and this change depends on the current values of both $S1$ and $S2$ – because these storages control the magnitude of the aforementioned fluxes and these in turn dictate the change in storage. $\Delta S1$ is not influenced by the level in the groundwater store $S3$. In the Jacobian pattern this can be indicated as shown in Table 2:

Table 2: Partly filled Jacobian matrix for the example model

	Depends on current value of		
	S1	S2	S3
$\Delta S1$	1	1	0
$\Delta S2$			
$\Delta S3$			

A Jacobian matrix must be square and show all dependencies between ODEs. Similar to store $S1$, the change in store $S2$ level depends on the current storage in both $S1$ and $S2$ but is independent from store $S3$. The change in store $S3$ is dependent on the current level in $S2$ (because this controls the magnitude of the groundwater inflow flux q_g) and its own current level (because this controls the magnitude of the groundwater outflow flux q_s). The full Jacobian is (Table 3: Filled Jacobian matrix for the example model):

Table 3: Filled Jacobian matrix for the example model

	Depends on current value of		
	S1	S2	S3
$\Delta S1$	1	1	0
$\Delta S2$	1	1	0
$\Delta S3$	0	1	1

```

%% 4. Determine numerical scheme and solver settings
% Function name of the numerical scheme
scheme = solver.name;

% Define which storage values should be used to update fluxes
[~,store_fun] = feval(scheme,storeInitial,delta_t);

% Root-finding options
fsolve_options = optimoptions('fsolve','Display','none',...
                              'JacobPattern', [1,1;
                                                1,1]);

Specify the Jacobian pattern
% fzero_options = optimset('Display','off');
lsqnonlin_options = optimoptions('lsqnonlin',...
                                 'Display','none',...
                                 'JacobPattern', [1,1;
                                                1,1],...
                                 'MaxFunEvals',1000);

```

```

%% 4. Determine numerical scheme and solver settings
% Function name of the numerical scheme
scheme = solver.name;

% Define which storage values should be used to update fluxes
[~,store_fun] = feval(scheme,storeInitial,delta_t);

% Root-finding options
fsolve_options = optimoptions('fsolve','Display','none',...
                              'JacobPattern', [1,1,0;
                                                1,1,0;
                                                0,1,1]);

Specify the Jacobian pattern
lsqnonlin_options = optimoptions('lsqnonlin',...
                                 'Display','none',...
                                 'JacobPattern', [1,1,0;
                                                1,1,0;
                                                0,1,1],...
                                 'MaxFunEvals',1000);

```

Figure 16: Update the numerical scheme and solver settings/options. Top: template model. Bottom: example model

4.2.15 Inside the time loop, update the “old storages” section if applicable

This section only needs to be changed if the new model does not have 2 stores (Figure 17).

```

% Model setup -----
% Determine the old storages
if t == 1; S1old = S10; else; S1old = store_S1(t-1); end
if t == 1; S2old = S20; else; S2old = store_S2(t-1); end

% Model setup -----
% Determine the old storages
if t == 1; S1old = S10; else; S1old = store_S1(t-1); end
if t == 1; S2old = S20; else; S2old = store_S2(t-1); end
if t == 1; S3old = S30; else; S3old = store_S3(t-1); end

```

Figure 17: Update these lines to reflect the right number of model stores. Top: template model. Bottom: example model

4.2.16 Inside the time loop, update the temporary ODEs

Define an anonymous function for each of the model's ODEs (eq. 9, 10, 11). Use the earlier defined flux equations for this (section 4.2.11). The only inputs to each ODE must be store values (S1, S2, S3; Figure 18).

```

% Create temporary store ODE's that need to be solved
tmpf_S1 = ...
    @(S1,S2) ...
    (P(t) + ...
    CAP(kc,S1,S1max,S2,delta_t) - ...
    EA(S1,Ep(t),delta_t) - ...
    QO(P(t),S1,S1max) - ...
    PERC(kp,S1,S1max,delta_t));

tmpf_S2 = ...
    @(S1,S2) ...
    (PERC(kp,S1,S1max,delta_t) - ...
    QS(ks,S2) - ...
    CAP(kc,S1,S1max,S2,delta_t));

% Create temporary store ODE's that need to be solved
tmpf_S1 = ...
    @(S1,S2,S3) ...
    (P(t) + ...
    QC(crate,S1,uzmax,S2,delta_t) - ...
    E(S1,uzmax,Ep(t),delta_t) - ...
    QSE(P(t),S1,uzmax) - ...
    QP(prate,S1,delta_t));

tmpf_S2 = ...
    @(S1,S2,S3) ...
    (QP(prate,S1,delta_t) - ...
    QC(crate,S1,uzmax,S2,delta_t) - ...
    QLZ(klz,S2));

tmpf_S3 = ...
    @(S1,S2,S3) ...
    (QG(alpha,QLZ(klz,S2)) - ...
    QS(kg,S3));

```

Figure 18: Updated ODEs. Top: template model. Bottom: example model

4.2.17 Inside the time loop, update the “ODE approximation” section if applicable

This section only needs to be changed if the new model has fewer than 2, or more than 2 stores. Two sections need to be changed: the line that specifies the values of each store at t-1, and the line that specifies which ODEs need to be re-written (Figure 19).

This part of the code calls the time-stepping function specified by the user (e.g. “createOdeApprox_IE”).

<pre>% Call the numerical scheme function to create the ODE approximations. % This returns a new anonymous function that we solve in the next step. solve_fun = feval(scheme,... [S1old,S2old],... delta_t,... tmpf_S1,tmpf_S2);</pre>
<pre>% Call the numerical scheme function to create the ODE approximations. % This returns a new anonymous function that we solve in the next step. solve_fun = feval(scheme,... [S1old,S2old,S3old],... % time-stepping function delta_t,... % Store values at t-1 tmpf_S1,tmpf_S2,tmpf_S3); % time step size % anonymous functions of ODEs</pre>

Figure 19: Update the time stepping scheme section. Top: template model. Bottom: example model

4.2.18 Inside the time loop, update the “Model solving” section

This section only needs to be changed if the new model has fewer than 2, or more than 2 stores. In case of a 1-store model, remove the ‘fsolve’ lines and activate the ‘fzero’ lines. In case of 2+-store models, change the ‘fsolve’ lines to reflect the correct number of stores. Note that both the ‘eq_sys(x)’ and ‘[S1old,...]’ lines need to be changed (Figure 20). Section 0 details two improvements that can be made to ‘fsolve’ to achieve increased computational efficiency.

<pre>% --- Use the specified numerical scheme to solve storages --- [tmp_sNew,tmp_fval] = fsolve(@(eq_sys) solve_fun(... eq_sys(1),eq_sys(2)),... [S1old,S2old],... fsolve_options);</pre>
<pre>% [tmp_sNew, tmp_fval] = fzero(solve_fun,... % S1old,... % fzero_options);</pre>
<pre>% --- Use the specified numerical scheme to solve storages --- [tmp_sNew,tmp_fval] = fsolve(@(eq_sys) solve_fun(... eq_sys(1),eq_sys(2),eq_sys(3)),... [S1old,S2old,S3old],... fsolve_options);</pre>

Figure 20: Update the solver settings, so that the right number of stores are represented. Top: template model. Bottom: example model

4.2.19 Inside the time loop, update the solver accuracy section if applicable

This section only needs to be changed if the new model has fewer than 2, or more than 2 stores. In case of a 1-store model, remove ‘eq_sys(2)’ and ‘S2old’. In case of a 2+-store model, add subsequent elements for the total number of stores in the model (Figure 21). Section 0 shows a small modification that can be made to ‘lsqnonlin’ to gain some computational efficiency.


```

% --- Check if the solver has found an acceptable solution and re-run
% if not. The re-run uses the 'lsqnonlin' solver which is slower but
% more robust. It runs solver.resnorm_iterations times, with different
% starting points for the solver on each iteration ---
tmp_resnorm = sum(tmp_fval.^2);

if tmp_resnorm > solver.resnorm_tolerance
    [tmp_sNew,~,~] = rerunSolver('lsqnonlin', ...
                                lsqnonlin_options, ...
                                @(eq_sys) solve_fun(...
                                    eq_sys(1),eq_sys(2)), ...
                                solver.resnorm_maxiter, ...
                                solver.resnorm_tolerance, ...
                                tmp_sNew, ...
                                [S1old,S2old], ...
                                store_min, ...
                                store_upp);
end

% --- Check if the solver has found an acceptable solution and re-run
% if not. The re-run uses the 'lsqnonlin' solver which is slower but
% more robust. It runs solver.resnorm_iterations times, with different
% starting points for the solver on each iteration ---
tmp_resnorm = sum(tmp_fval.^2);

if tmp_resnorm > solver.resnorm_tolerance
    [tmp_sNew,~,~] = rerunSolver('lsqnonlin', ...
                                lsqnonlin_options, ...
                                @(eq_sys) solve_fun(...
                                    eq_sys(1),eq_sys(2),...
                                    eq_sys(3)), ...
                                solver.resnorm_maxiter, ...
                                solver.resnorm_tolerance, ...
                                tmp_sNew, ...
                                [S1old,S2old,S3old], ...
                                store_min, ...
                                store_upp);
end

```

Figure 21: Update the solver accuracy control section. Top: template model. Bottom: example model

4.2.20 Inside the time loop, do not change the 'Find storages to update fluxes' section

This part of the code evaluates an earlier defined function (section 4.2.12) that specifies which storage variables should be used to update the model fluxes ().

```

% Model states and fluxes -----
% This line creates/updates a variable called 'tmp_sFlux' which is used
% to update the model fluxes for the current time step. Which variables
% get assigned to 'tmp_sFlux' is a feature of the chosen numerical time
% stepping scheme (see line 123-124).
eval(store_fun);

% Model states and fluxes -----
% This line creates/updates a variable called 'tmp_sFlux' which is used
% to update the model fluxes for the current time step. Which variables
% get assigned to 'tmp_sFlux' is a feature of the chosen numerical time
% stepping scheme (see line 133-134).
eval(store_fun);

```

Figure 22: Do not change the "Find storage needed to update fluxes" section. Top: template model. Bottom: example model

4.2.21 Inside the time loop, update the “Model fluxes” section

In this part of the code, the time series of flux values are updated (Figure 23). This uses the variable “tmp_sFlux” which gets its values assigned based on the choice of time stepping scheme. “tmp_sFlux” is a vector that contains a value for each model store, and is thus of size [1,number of stores]. The flux equations can be copied directly from the ODEs, but the temporary variables “S1”, “S2”, etc must be replaced with “tmp_sFlux(1)”, “tmp_sFlux(2)”, etc.

Note: fluxes in the time loop are calculated in units [mm/d]. Conversion back to the user’s specified [mm/ Δt] occurs when outputs are generated, after the time loop has completed.

<pre>% Calculate the fluxes flux_cap(t) = CAP(kc,tmp_sFlux(1),S1max,tmp_sFlux(2),delta_t); flux_ea(t) = EA(tmp_sFlux(1),Ep(t),delta_t); flux_qo(t) = QO(P(t),tmp_sFlux(1),S1max); flux_perc(t) = PERC(kp,tmp_sFlux(1),S1max,delta_t); flux_qs(t) = QS(ks,tmp_sFlux(2));</pre>
<pre>% Calculate the fluxes flux_qse(t) = QSE(P(t),tmp_sFlux(1),uzmax); flux_e(t) = E(tmp_sFlux(1),uzmax,Ep(t),delta_t); flux_qp(t) = QP(prate,tmp_sFlux(1),delta_t); flux_qc(t) = QC(crate,tmp_sFlux(1),uzmax,tmp_sFlux(2),delta_t); flux_qlz(t) = QLZ(klz,tmp_sFlux(2)); flux_qf(t) = QF(1-alpha,flux_qlz(t)); flux_qg(t) = QG(alpha,flux_qlz(t)); flux_qs(t) = QS(kg,tmp_sFlux(3));</pre>

Figure 23: Update the time series of flux values. First copy the flux equations, then change the temporary variables ‘S1’, ‘S2’ ‘S..’ to use the new storage values in variable ‘tmp_sNew’. Change the references to other fluxes (e.g. QLZ(klz,S2) in QF(1-alpha,QLZ(..))) to use the updated flux values. Top: template model. Bottom: example model

4.2.22 Inside the time loop, update the “Model stores” section

In this part of the code, the time series of model storages are updated (Figure 24). These equations are a numerical approximation of the ODEs (eq. 10, 11, 12) at time = t. Therefore, flux values must be multiplied by the user-specified time step size Δt .

<pre>% Update the stores store_S1(t) = S1old + (P(t) + flux_cap(t) - flux_ea(t) - ... flux_qo(t) - flux_perc(t)) * delta_t; store_S2(t) = S2old + (flux_perc(t) - flux_qs(t) - ... flux_cap(t)) * delta_t;</pre>
<pre>% Update the stores store_S1(t) = S1old + (P(t) + flux_qc(t) - flux_e(t) - ... flux_qse(t) - flux_qp(t)) * delta_t; store_S2(t) = S2old + (flux_qp(t) - flux_qc(t) - ... flux_qlz(t)) * delta_t; store_S3(t) = S3old + (flux_qg(t) - flux_qs(t)) * delta_t;</pre>

Figure 24: Update the time series of model storage values. Top: template model. Bottom: example model

4.2.23 Inside the time loop, update the “Routing” section if applicable

If a routing scheme is used, this section needs to be updated. Use the model schematic to find out which flows combine and enter the routing scheme. In the example model, surface runoff Q_{se} , fast flow Q_f and slow flow Q_s are combined and routed together (Figure 25). The flux that represents lagged flow is called Q_t in the example model, so no further changes to the routing code are necessary.

Note: the template and example *model files* contain more detailed comments that explain how the routing code works.

Note: time step size is accounted for during generation of the Unit Hydrograph (ensure that the UH has the correct length; section 4.2.8) and during output generation (ensure that the fluxes are properly converted to [mm/Δt]; section 4.2.24).

<pre> % Routing ----- % Total runoff Qt = Qo + Qs. Apply a triangular routing scheme with % time base 'delay' (parameter 5) tmp_Qt_cur = (flux_qo(t) + flux_qs(t)).*uh_full; tmp_Qt_old = tmp_Qt_old + tmp_Qt_cur; flux_qt(t) = tmp_Qt_old(1); tmp_Qt_old = circshift(tmp_Qt_old,-1); tmp_Qt_old(end) = 0; </pre>
<pre> % Routing ----- % Total runoff Q = Qse + Qf + Qs. Apply a pre-determined (line 82) % traingular Unit Hydrograph routing scheme to find lagged flow Qt. tmp_Qt_cur = (flux_qse(t) + flux_qf(t) + flux_qs(t)).*uh_full; tmp_Qt_old = tmp_Qt_old + tmp_Qt_cur; flux_qt(t) = tmp_Qt_old(1); tmp_Qt_old = circshift(tmp_Qt_old,-1); tmp_Qt_old(end) = 0; </pre>

Figure 25: Update the routing code, if applicable. The red fluxes are the total incoming amount of water on this time step that needs to be routed using Unit Hydrograph "uh_full". The dotted vector represents the lagged flow after routing has been applied and might need to be renamed if this flux is named differently in the model description. Top: template model. Bottom: example model

4.2.24 Update the “Generate outputs” section

Update this section so that all time series (fluxes and stores) are included in one of the output structures. Combine different elements together if necessary (e.g. fluxOutput.Ea = flux_bareSoilEvap + flux_transpiration;” if the model has two different evaporation components). In this case it is good practice to include the individual fluxes in the “fluxInternal”-structure as well.

```

% --- Fluxes leaving the model ---
% 'Ea' and 'Q' are used outside the function and should NOT be renamed
fluxOutput.Ea      = flux_ea * delta_t;
fluxOutput.Q       = flux_qt * delta_t;

% --- Fluxes internal to the model ---
fluxInternal.cap   = flux_cap * delta_t;
fluxInternal.perc  = flux_perc * delta_t;
fluxInternal.Qo    = flux_qo  * delta_t;
fluxInternal.Qs    = flux_qs  * delta_t;

% --- Stores ---
storeInternal.S1   = store_S1;
storeInternal.S2   = store_S2;

% --- Fluxes leaving the model ---
% 'Ea' and 'Q' are used outside the
% function and should NOT be renamed
fluxOutput.Ea      = flux_e  * delta_t;
fluxOutput.Q       = flux_qt * delta_t;

% --- Fluxes internal to the model ---
fluxInternal.qse   = flux_qse * delta_t;
fluxInternal.qp    = flux_qp  * delta_t;
fluxInternal.qc    = flux_qc  * delta_t;
fluxInternal.qlz   = flux_qlz * delta_t;
fluxInternal.qf    = flux_qf  * delta_t;
fluxInternal.qg    = flux_qg  * delta_t;
fluxInternal.qs    = flux_qs  * delta_t;

% --- Stores ---
storeInternal.S1   = store_S1;
storeInternal.S2   = store_S2;
storeInternal.S3   = store_S3;

```

Figure 26: update the output generation section. Left: template model. Right: example model

4.2.25 Optional: update the “Check water balance” section

Change the variable “tmp_Qt_old” to 0 if no routing scheme is used (Figure 27).

The function “checkWaterBalance(..)” can only handle a fairly basic model layout. The expected elements are incoming precipitation, outgoing evaporation and streamflow and storages that track how much water is currently held inside the model. An optional argument tracks water that is still held in the routing vector if applicable. The “checkWaterBalance(..)” function currently has no functionality to deal with model stores that track a moisture deficit instead of the presence of moisture, and it has no way to deal with additional fluxes that leave the model (e.g. subsurface leakage or between-catchment water exchange). Disable this function and use the commented code to the water balance manually if either a deficit store or sink flows are present in the model.

MARRMoT model 07 (GR4J) shows an example of a manual water balance that accounts for a groundwater exchange flow (lines 293-312). MARRMoT model 05 (IHACRES) gives an example of a deficit store (store 1, lines 139-141) and a manual water balance that accounts for this deficit store (lines 219-236).

```

% Check water balance
if nargin == 4
    waterBalance = ...
    checkWaterBalance(...
        P,... % Incoming precipitation
        fluxOutput,... % Fluxes Q and Ea leaving the model
        storeInternal,... % Time series of storages ...
        storeInitial,... % And initial store values to calculate delta S
        tmp_Qt_old); % Whether the model uses a routing scheme that
                    % still contains water. Use '0' for no routing
end

```

Figure 27: This function checks the water balance. If a routing scheme is used and part of the flow is not yet fully routed at the end of the time series, the remaining flow is stored in variable "tmp_Qt_old". If no routing scheme is used, change "tmp_Qt_old" to

4.3 Create the parameter range file

The final step is creating a *parameter range file* which contains ranges for the model parameters.

4.3.1 Copy and rename the template *parameter range file*

Navigate to the folder “./MARRMoT/Models/Parameter ranges” and copy the file “m_00_template_5p_2s_parameter_ranges.m”. Paste this file in the same directory and rename it (Figure 28). The new name should follow the same structure as the current *parameter range files*:

“[model file name]_parameter_ranges.m”

The example *parameter range file* created in this manual can be found in the folder “./MARRMoT/User manual”.

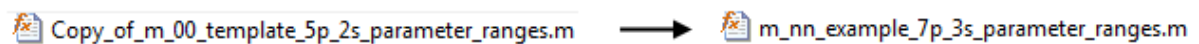


Figure 28: Create a dedicated parameter range file file. Left: template mode parameter file. Right: example model parameter file

4.3.2 Open the file and change the function name

Open the renamed *parameter range file* and change the function’s name to match the file name. Optional: change the comments to reflect the model’s name and provide a reference (Figure 29).

```

function [ theta ] = m_00_template_5p_2s_parameter_ranges( )
% m_00_template_5p_2s_parameter_ranges Provides parameter ranges for
calibration of the 2-store test model, created by W. Knoben in 02-2018.

function [ theta ] = m_nn_example_7p_3s_parameter_ranges( )
% m_nn_example_7p_3s_parameter_ranges Provides parameter ranges for
calibration of the 3-store example model, created by W. Knoben in 09-2018.

```

Figure 29: Change the function name to match the file name. Update the description in the comments for clarity. Top: template mode parameter file. Bottom: example model parameter file

4.3.3 Change the parameter ranges and follow the ordering in the *model file*

Define parameter ranges for each parameter used by the *model file*. Ensure that the order of parameters in this file is the same as the order in the *model file*. MARRMoT attempts to provide consistent parameter ranges across all models to facilitate model comparison studies. Use **Table S3 (Supporting Materials S5)** to determine appropriate parameter ranges for the new model if the new model is intended to be consistent with the other MARRMoT models.

theta = [1 , 40; % Smax [mm]		
0 , 2 ; % kc, capillary rise [mm/d]		
0 , 3 ; % kp, percolation rate [mm/d]		
0.5 , 1; % ks, base flow time parameter [d-1]		
1 , 5]; % time delay of routing scheme [d]		
theta = [0, 4; % crate, Maximum capillary rise rate [mm/d]		
1, 2000; % uzamx, Maximum upper zone storage [mm]		
0, 20; % prate, Maximum percolation rate [mm/d]		
0, 1; % klz, Lower zone runoff coefficient [d-1]		
0, 1; % alpha, Fraction lower zone runoff to groundwater [-]		
0, 1; % kg, Groundwater runoff coefficient [d-1]		
1, 120]; % d, Routing delay [d]		

Figure 30: choose parameter ranges that are as consistent as possible between different models and follow the parameter order specified in the model file. Top: template mode parameter file. Bottom: example model parameter file

4.4 Recommended quality control tests

Users are strongly encouraged to perform several quality control tests after creating a new model. This section describes two tests that have been valuable during MARRMoT development. Both tests are implemented in the file “workflow_crashTest.m”.

4.4.1 Parameter extremes crash test

Ensure that the model functions properly at all combinations of minimum and maximum parameter values. Provided that flux functions are continuous between the parameter extremes, if the model can simulate runoff with extreme parameter values it should work with intermediate parameter values too. This check ensures that the model can at least simulate a full time series without crashing.

4.4.2 Random parameter value water balance check

Ensure that no mistakes occurred when creating the *model file*. An easy way to check this is by running the model with several random parameter sets and investigating the model’s water balance. If e.g. any fluxes have been forgotten, counted double, added to or subtracted from the wrong stores, the water balance will show a discrepancy. During development, water balance errors were generally in the order of 1E-12 or smaller.

5 Create a new *flux function*

This section gives several examples that show how to create flux functions. See section 4 for guidance about using flux functions inside model files.

5.1 General approach

Creating a new flux function requires several 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 an anonymous function, and the handle to this anonymous function is the output of each *flux function*. These handles are used inside the *model files* to calculate flux sizes based on a variety of parameters, storage values and climate inputs.

5.2 The linear reservoir – using one parameter and one store

The equation for a linear reservoir is:

$$q = kS$$

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 looks as follows:

```
function [func] = baseflow_1(~)
%baseflow_1
%
% Anonymous function
% -----
% Description: Outflow from a linear reservoir
% Constraints: -
% @(Inputs):  p1    - time scale parameter [d-1]
%              S     - current storage [mm]
%
% WK, 05/10/2018

func = @(p1,S) p1.*S;

end
```

$p1$ represents parameter k and S is the current storage. *func* is the function handle passed as the *flux file's* output. This *flux function* might be used in a *model file* as follows:

```
% Baseflow from groundwater
QB = baseflow_1;

...

% Update baseflow flux
QB_vector(t) = QB(parameter_k, storage_value);
```

Where QB is a temporary function handle used in the *model file*. The construction QB = baseflow_1, with the flux equation specified in *flux file* “baseflow_1” is functionally identical to typing: QB = @(p1,S) p1.*S; in the *model file*.

5.3 The non-linear reservoir - using multiple parameters

The equation for a non-linear reservoir is:

$$q = kS^a$$

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, 1 store value and 1 constraint:

```
function [func] = baseflow_7(~)
%baseflow_7
%
% Anonymous function
% -----
% Description:  Non-linear outflow from a reservoir
% Constraints:  f <= S/dt
% @(Inputs):   p1  - time coefficient [d-1]
%              p2  - exponential scaling parameter [-]
%              S   - current storage [mm]
%              dt  - time step size [d]
%
% WK, 05/10/2018

func = @(p1,p2,S,dt) min(S/dt,p1.*S.^p2);

end
```

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 $-1\text{E-}5$ or smaller. However, in a non-linear equation this can result in mathematically correct, but physically meaningless complex estimates of fluxes (e.g. $(-1\text{E-}5)^{0.1} = 0.3008 + 0.0977i$). An additional constraint is introduced to avoid this which ensures $S \geq 0$:


```

function [func] = baseflow_7(~)
%baseflow_7
%
% Anonymous function
% -----
% Description:  Non-linear outflow from a reservoir
% Constraints:  f <= S/dt
%              S >= 0
% @(Inputs):   p1   - time coefficient [d-1]
%              p2   - exponential scaling parameter [-]
%              S    - current storage [mm]
%              dt   - time step size [d]
%
% WK, 05/10/2018

func = @(p1,p2,S,dt) min(S/dt,p1.*max(0,S).^p2);

end

```

5.4 The capillary rise flux – using multiple parameters and stores

It is straightforward to use multiple stores in a *flux function*. Imagine capillary rise from store S2 to store S1:

$$q_c = c_{rate} \left(1 - \frac{S1}{S1_{max}} \right)$$

where q_c is the actual capillary rise, dependent on a maximum rate c_{rate} and the storage deficit in the receiving store S1 ($S1/S1_{max}$ being the relative storage in S1). A constraint needs to be added to ensure that the capillary rise does not over drain the supplying store S2: $q_c \leq S2$. The *flux file* becomes:

```

function [func] = capillary_1(~)
%capillary_1
%
% Anonymous function
% -----
% Description:  Capillary rise: based on deficit in higher reservoir
% Constraints:  f <= S2/dt
% @(Inputs):   p1   - maximum capillary rise rate [mm/d]
%              S1   - current storage in receiving store [mm]
%              S1max- maximum storage in receiving store [mm]
%              S2   - current storage in providing store [mm]
%              dt   - time step size [d]
%
% WK, 05/10/2018

func = @(p1,S1,S1max,S2,dt) min(p1.*(1-S1/S1max),S2/dt);

end

```

5.5 The store overflow – 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:

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

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 (Figure 31). The equation becomes:

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

where $\phi(S, S_{max})$ is the smoothing function (Kavetski and Kuczera, 2007). The *flux function* is as follows:

```
function [func] = interception_1(~)
%interception_1 Creates function for store overflow: uses logistic smoother.
%
% Anonymous function
% -----
% Description: Interception excess when maximum capacity is reached
% Constraints: -
% @(Inputs):  In  - incoming flux [mm/d]
%              S   - current storage [mm]
%              Smax - maximum storage [mm]
%
% WK, 07/10/2018

func = @(In,S,Smax) In.*(1-smoothThreshold_storage_logistic(S,Smax));

end
```

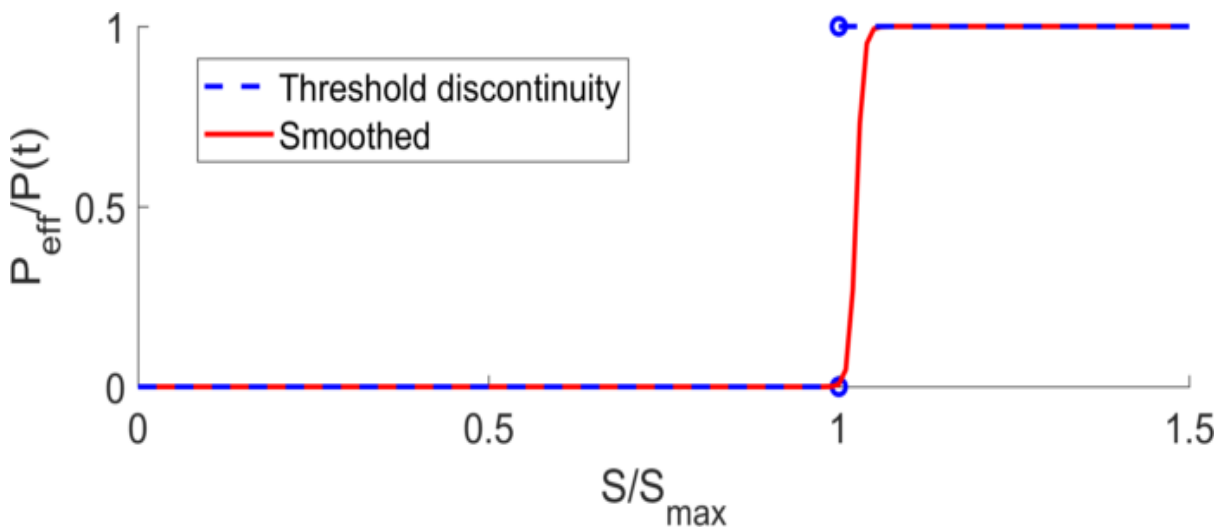


Figure 31: Example of equation smoothing

5.6 The store overflow 2.0 – using optional parameters

The smoothing functions in MARRMoT use two smoothing parameters, r and e , with default values 0.01 and 5.00 respectively (Clark et al., 2008):

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

However, users might prefer different values and the *flux function* must allow this. The “interception_1” function thus needs to allow optional parameters and revert to default smoothing parameters if the user does not specify any values:

```

function [func] = interception_1(varargin)
%interception_1 Creates function for store overflow: uses logistic smoother.
% varargin(1): value of smoothing variable r (default 0.01)
% varargin(2): value of smoothing variable e (default 5.00)
%
% Anonymous function
% -----
% Description: Interception excess when maximum capacity is reached
% Constraints: -
% @(Inputs):  In   - incoming flux [mm/d]
%              S    - current storage [mm]
%              Smax - maximum storage [mm]
%
% WK, 07/10/2018

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

end

```

6 Matlab root-finding optimization

Several small modifications can be made to Matlab's *fzero*, *fsolve* and *lsqnonlin* for several small speed gains. Due to licensing, modified files cannot be provided as part of MARRMoT.

6.1 Fzero modifications

This file is part of Matlab's optimization toolbox and can be found in the default directory:

./MATLAB/<version>/toolbox/matlab/optimfun/fzero.m

Fzero generates an output message on line 553. In certain versions (tested with R2013) generation of this message can take a long time. This line can be disabled:

```
msg
sprintf(getString(message('MATLAB:optimfun:fzero:ZeroFoundInInterval',sprin
tf('%g', savea), sprintf('%g', saveb)))));
```

6.2 Fsolve modifications

This file is part of Matlab's optimization toolbox and can be found in the default directory:

./MATLAB/<version>/toolbox/optim/optim/fsolve.m

Fsolve internally generates options for the solver in lines 151-152. Within MARRMoT, this means that these options are generated anew on every time step where *fsolve* is called. The generated options are the same however, and this line can be safely taken outside *fsolve*. *fsolve* then needs to be modified to accept the optionFeedback structure as input:

```
% Model file
...

% Root-finding options
fsolve_options = optimoptions('fsolve','Display','none',...
                             'JacobPattern',[1,0;
                                             1,1]);

% Prepare the options for the solver
[fsolve_options,optionFeedback] = prepareOptionsForSolver(fsolve_options,
'fsolve');

% Some more code
...

% --- Determine store values at the end of the time step ---
[tmp_sNew,tmp_fval] = fsolve_MODIFIED(...
    @(eq_sys) solve_fun(eq_sys(1),eq_sys(2)),...
    [S1old,S2old],...
    fsolve_options,optionFeedback);
```

fsolve's inputs must be modified to allow this (modifications in red):

```
function [x,FVAL,EXITFLAG,OUTPUT,JACOBI] = ...
    fsolve_noMSG(FUN,x,options,optionFeedback,varargin)
```

fsolve generates an output message on lines 403-413. In certain versions (tested with R2017) generation of this message can take up to 20% of *fsolve*'s total run time. This section can be safely disabled:

```

if EXITFLAG > 0 % if we think we converged:
    % Call createExitMsg with appended additional information on the
    closeness
    % to a root.
    if Resnorm > sqrtTolFunValue
        msgData = internalFlagForExitMessage(algorithmflag ==
2,msgData,EXITFLAG);
        EXITFLAG = -2;
    end
    OUTPUT.message =
createExitMsg(msgData{:},Resnorm,optionFeedback.TolFunValue,sqrtTolFunValue
);
else
    OUTPUT.message = createExitMsg(msgData{:});
end

```

6.3 Lsqnonlin modifications

This file is part of Matlab's optimization toolbox and can be found in the default directory:

./MATLAB/<version>/toolbox/shared/optimlib/lsqncommon.m

This is a shared file between various non-linear solvers. It generates an output message on line 181. This section can be disabled for speed gains:

```
OUTPUT.message = createExitMsg(msgData{:});
```

7 Running MARRMoT in Octave

The Octave distribution of MARRMoT works the same as the Matlab distribution, with the exception that a certain function must be replaced and that a Jacobian matrix cannot be specified during model computation (see section 7.3 for both points). This means that Octave does not benefit from the speed ups that can be gained by specifying the Jacobian. The impact of this is untested. In the Matlab distribution, models with more stores benefit more from specifying the Jacobian matrix, and it is not unreasonable to expect that models with more stores are thus slower to run in Octave than they are in Matlab.

This section provides a very short guide to set up MARRMoT in Octave.

7.1 Set the path

Navigate to the direct that contains “./MARRMoT”. Run the following command to add all MARRMoT files to Octave’s load path:

```
addpath(genpath('MARRMoT'))
```

7.2 Load the optimization package

MARRMoT relies on certain functions that are not loaded by default. Load the optimization package with the following command:

```
pkg load optim
```

7.3 Caution

Octave currently does not include an equivalent to Matlab’s function “optimoptions”. Additionally, Octave does not allow specification of the Jacobian matrix in the same way as Matlab allows.

MARRMoT’s Octave distribution includes a custom placeholder function “optimoptions” in the folder ./MARRMoT/Functions/Octave. This function only sets the maximum number of function evaluations and is thus *not a replacement of Matlabs optimoptions!* It merely allows MARRMoT to be used in Octave without significant changes to each model’s code.

7.4 Possible Octave errors thrown by workflow examples

Testing has shown that certain older Octave distributions do not contain the function ‘repelem’. This results in errors during use. Please update to a more recent Octave version if this error occurs (MARRMoT was tested on Octave 4.4.1).

Workflow example 4 (calibration of a model using custom Matlab function ‘fminsearchbnd’) does **not** work in Octave 4.4.1. Octave users will need to consider an alternative calibration algorithm and/or calibration approach.

8 References

- Addor, N., Newman, A. J., Mizukami, N. and Clark, M. P.: The CAMELS data set: catchment attributes and meteorology for large-sample studies, *Hydrol. Earth Syst. Sci.*, 21, 5293–5313, doi:10.5194/hess-2017-169, 2017.
- Clark, M. P., Slater, A. G., Rupp, D. E., Woods, R. a., Vrugt, J. a., Gupta, H. V., Wagener, T. and Hay, L. E.: Framework for Understanding Structural Errors (FUSE): A modular framework to diagnose differences between hydrological models, *Water Resour. Res.*, 44(12), doi:10.1029/2007WR006735, 2008.
- Garcia, F., Folton, N. and Oudin, L.: Which objective function to calibrate rainfall–runoff models for low-flow index simulations?, *Hydrol. Sci. J.*, 62(7), 1149–1166, doi:10.1080/02626667.2017.1308511, 2017.
- Gupta, H. V., Kling, H., Yilmaz, K. K. and Martinez, G. F.: Decomposition of the mean squared error and NSE performance criteria: Implications for improving hydrological modelling, *J. Hydrol.*, 377(1-2), 80–91, doi:10.1016/j.jhydrol.2009.08.003, 2009.
- Jothityangkoon, C., Sivapalan, M. and Farmer, D. .: Process controls of water balance variability in a large semi-arid catchment: downward approach to hydrological model development, *J. Hydrol.*, 254(1-4), 174–198, doi:10.1016/S0022-1694(01)00496-6, 2001.
- Kavetski, D. and Kuczera, G.: Model smoothing strategies to remove microscale discontinuities and spurious secondary optima in objective functions in hydrological calibration, *Water Resour. Res.*, 43(3), n/a–n/a, doi:10.1029/2006WR005195, 2007.
- Sugawara, M.: Tank model, in *Computer models of watershed hydrology*, edited by V. P. Singh, pp. 165–214, Water Resources Publications, USA., 1995.
- Wagener, T., Boyle, D. P., Lees, M. J., Wheater, H. S., Gupta, Hoshin, V. and Sorooshian, S.: A framework for development and application of hydrological models, *Hydrol. Earth Syst. Sci.*, 5, 13–26, 2001.