RRMPG Documentation

Release 0.0.1

Frederik Kratzert

Jun 30, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

This repository is a work in progress repository.

Official documentation can be found here: http://rrmpg.readthedocs.io

Read the *Idea section* for further information about the background and aim of this project.

1.1 Idea

One of the fundamental parts of hydrology is rainfall-runoff-modelling. The task here is to model the response of a catchment to meteorological input data and to forecast the river discharge. There are different approaches to tackle the problem, namely: conceptual models, physical-based models and data-driven models.

Although this is taught at university, often hands-on experience is missing or is done on using very simple modelling approaches. One of the main reasons I see is, that most (at least the complex ones) hydrological models are implemented in Fortran but very few students of the field of hydrology know Fortran, when they first get in touch with RR-Models. So all they can probably do is simply apply a model to their data and play manually with parameter tuning, but not explore the model and see the effect of code changes.

This might be different if there would exist well performing implementations of hydrological models in a more simplistic and readable language, such as Python. What was hindering this step was always the speed of Python and the nature of RR-Models - they mostly have to be implemented using loops over all timesteps. And well, big surprise: Pure Python and for-loops is not the best combination in terms of performance.

This could be changed e.g. by using Cython for the hydrological model, but this again might hinder the code understanding, since Cython adds non-pythonic complexity to the code, which might be hard for beginners to understand and therefore play/experiment with the code.

Another option could be PyPy. The problem I see with PyPy is, that the user would be forced to install a different Python interpreter, while most I know of are quite comfortable using e.g. Anaconda.

Numba is another way to speed up array-oriented and math-heavy Python code but without changing the language/interpreter and just by few code adaptions. Using numba, the code stays easily readable and therefore better understandable for novices. I won't spend much time now on explaining how numba works, but I'll definitely add further information in the future. First performance comparisons between Fortran implementations and numba optimized Python code have shown, that the speed is roughly the same (Fortran is about ~1-2 times faster, using the GNU Fortran compiler).

Summary: The idea of this code repository is to provide fast (roughly the speed of Fortan) implementations of hydrological models in Python to make it easier to play and experiment with rainfall-runoff models.

1.2 You want to contribute?

At the moment I'm looking for a selection of hydrological models I'll implement in Python. If you want to see any (your?) model in this project, feel free to contact me. There is also a How to contribute section at the official documentation, were you can read more on the various ways you can contribute to this repository.

1.3 Contributors

I'll add later a better looking section to the official documentation. For now I list everybody, who contributed to this repository here:

- Ondřej Čertík with pull request #3: Optimized Fortran code and compilation procedure for fair speed comparision.
- Daniel Klotz with pull request #4 , #5 and #9: All spell checking.
- Andrew MacDonald for providing HBV-Edu simulation data from the original MATLAB implementation (see ##10)
- Martijn Visser with pull request #13 to update the unittest for pandas 1.0
- Martin Gauch with pull request #14 to fix a bug in the HBV model, when running multiple parameter sets at once.

1.4 Contact

Raise an issue here in this repository or contact me by mail f.kratzert(at)gmail.com

CHAPTER TWO

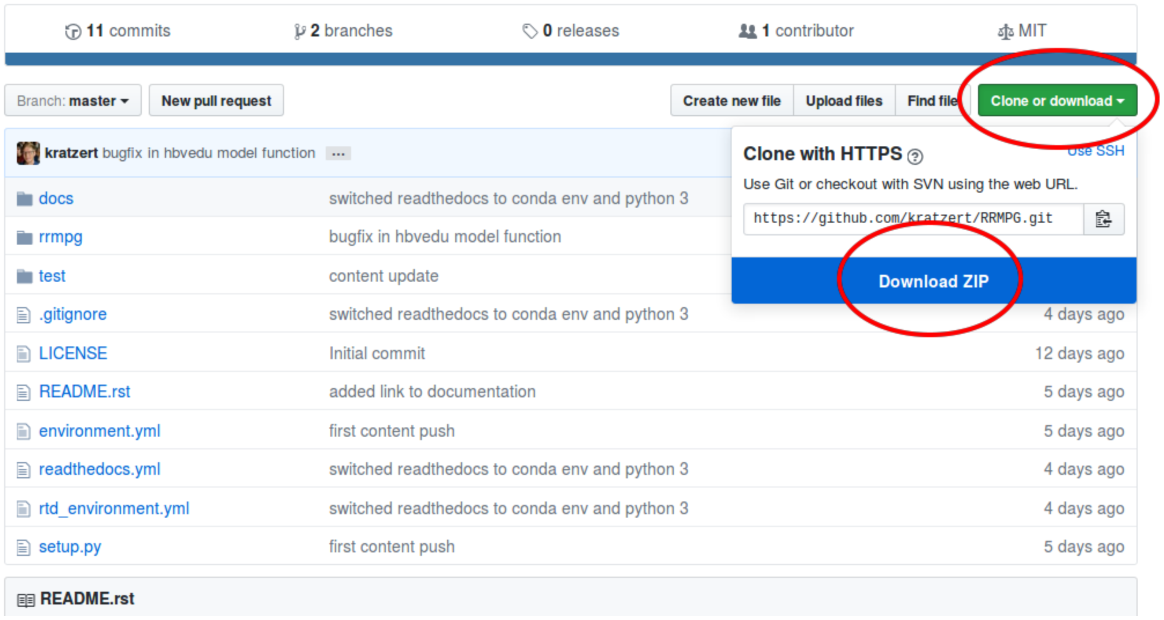
GETTING STARTED

2.1 Setting up Python

First of all you need to have Python. I highly recommend you to use Anaconda. Go to their homepage and download and install the latest Python 3 installer.

2.2 Downloading the source code

Since this package is in the early stages of development I haven't added it to PyPI, Pythons Package Index. Therefore you can't install it at the moment using `pip`. For the moment you have to download the source code from GitHub. To do so you can either download the entire repository as `.zip` and extract it to any destination on your machine:



kratzert bugfix in hbvedu model function

Clone with HTTPS

Use Git or checkout with SVN using the web URL.

`https://github.com/kratzert/RRMPG.git`

docs	switched readthedocs to conda env and python 3	
rrmpg	bugfix in hbvedu model function	
test	content update	
.gitignore	switched readthedocs to conda env and python 3	4 days ago
LICENSE	Initial commit	12 days ago
README.rst	added link to documentation	5 days ago
environment.yml	first content push	5 days ago
readthedocs.yml	switched readthedocs to conda env and python 3	4 days ago
rtd_environment.yml	switched readthedocs to conda env and python 3	4 days ago
setup.py	first content push	5 days ago

Or you use your terminal and the following commands:

```
git clone https://github.com/kratzert/RRMPG
```

2.3 Setting up RRMPGs Python dependencies

To use/develope this package, you have two different options:

1. (recommended) You use a conda environment. In the main directory a file included (`environment.yml`) that will setup everything for you by the following command:

From the terminal go to the `RRMPG` main directory and enter:

```
conda env create -f environment.yml
```

You can then activate the environment by entering:

```
# on linux and macOS
source activate rrmpg

# on windows
activate rrmpg
```

To leave an environment enter

```
# on linux and macOS
source deactivate

# on windows
deactivate
```

2. (not recommend but possible) You make sure that you have all dependencies installed in your normal/root environment.

The list of dependencies are:

- Python 3.x
- Numpy
- Matplotlib
- Numba
- Jupyter
- IPython
- Pandas
- Scipy

2.4 Installing the RRMPG package

To be able to import this package the usual way in your python environment, enter the following command in the terminal, assuming you are in the RRMPG root directory:

```
python setup.py install
```

This should install everything correctly.

To confirm, that everything worked as expected you can test the following lines of code:

```
# start Python console
python

# now in Python try the following
>>>from rrmpg.models import ABCModel
```

If no error is raised: Congratulations, you now can use the package on your local machine.

CHAPTER

THREE

DOCUMENTATION

3.1 Models

Documentation of all hydrological models implemented in the `rrmpg.models` module.

3.1.1 ABC-Model

Implementation of the model described in:

```
Myron B. Fiering "Streamflow synthesis" Cambridge, Harvard University
Press, 1967. 139 P. (1967).
```

Explanation of the model parameters:

- a: describes the fraction of precipitation that percolates through the soil to the groundwater.
- b: describes the fraction of the precipitaion that is directly lost to the atmosphere through evapotranspiration.
- c: describes the amount of groundwater that leaves the storage and drains into the stream.

Model inputs for simulation:

- prec: Array of (summed) precipitation for each timestep. [mm/day]

Class documentation

class `rrmpg.models.`**`ABCModel`**(*params=None*)

Interface to the the ABC-Model.

This model implements the classical ABC-Model. It was developed for educational purpose and represents a simple linear model.

Original Publication: Myron B. Fiering “Streamflow synthesis” Cambridge, Harvard University Press, 1967. 139 P. (1967).

If no model parameters are passed upon initialization, generates random parameter set.

Parameters **`params`** – (optional) Dictonary containing all model parameters as a seperate key/value pairs.

`fit`(*qobs*, *prec*, *initial_state=0*)

Fit the model to a timeseries of discharge using.

This functions uses scipy's global optimizer (differential evolution) to find a good set of parameters for the model, so that the observed discharge is simulated as good as possible.

Parameters

- **qobs** – Array of observed streaflow discharge.
- **prec** – Array of precipitation data.
- **initial_state** – (optional) Initial value for the storage.

Returns A scipy OptimizeResult class object.

Return type res

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.

`get_default_bounds`()

Return the dictionary containing the default parameter bounds.

`get_dtype`()

Return the custom model datatype.

`get_parameter_names`()

Return the list of parameter names.

`get_params`()

Return a dict with all model parameters and their current value.

`get_random_params`(*num=1*)

Generate random sets of model parameters for the ABC-model.

The ABC-model has specific parameter constraints, therefore we will overwrite the function of the Base-Model, to generated random model parameters, that satisfy the ABC-Model constraints.

Parameters num – (optional) Integer, specifying the number of parameter sets, that will be generated. Default is 1.

Returns A dict containing one key/value pair for each model parameter.

`set_params`(*params*)

Set model parameters to values passed in params.

Parameters params – Either a dictonary containing model parameters as key/value pairs or a numpy array of the models own custom dtype. For the case of dictionaries, they can contain only one model parameter or many/all. The naming of the parameters must be identical to the names specified in the _param_list. All parameter values must be numerical.

Raises

- `ValueError` – If any parameter is not a numerical value.
- `AttributeError` – If the parameter dictonary contains a key, that doesn't match any of the parameter names.
- `TypeError` – If the numpy array of the parameter does not fit the custom data type of the model or it's neither a dict nor a numpy ndarray.

simulate(*prec*, *initial_state=0*, *return_storage=False*, *params=None*)

Simulate the streamflow for the passed precipitation.

This function makes sanity checks on the input and then calls the externally defined ABC-Model function.

Parameters

- **prec** – Precipitation data for each timestep. Can be a List, numpy array or pandas.Series
- **initial_state** – (optional) Initial value for the storage.
- **return_storage** – (optional) Boolean, wether or not to return the simulated storage for each timestep.
- **params** – (optional) Numpy array of parameter sets, that will be evaluated a once in parallel. Must be of the models own custom data type. If nothing is passed, the parameters, stored in the model object, will be used.

Returns An array with the simulated stream flow for each timestep and optional an array with the simulated storage.

Raises

- ValueError – If one of the inputs contains invalid values.
- TypeError – If one of the inputs has an incorrect datatype.

3.1.2 HBV education

Implementation of the model described in:

```
Aghakouchak, Amir, and Emad Habib. "Application of a conceptual hydrologic
model in teaching hydrologic processes." International Journal of
Engineering Education 26.4 (S1) (2010).
```

Explanation of the model parameters:

- T_t: Threshold temperature. Decides if snow is melting or accumulating.
- DD: Degree-day factor. Indicates the decrease of the water content in the snow cover.
- FC: Field capacity. Describes the maximum soil moisture storage in the subsurface zone.
- Beta: Shape coefficient. Controls the amount of liquid water (Precipitation + melting Snow), which contributes to runoff.
- C: Improves model performance, when mean daily temperature deviates considerably from long-term mean.
- PWP: Permanent Wilting Point. Is a soil-moisture limit for evapotranspiration.
- K_0: Near surface flow storage coefficient.
- K_1: Interflow storage coefficient. K_1 should be smaller than K_0.
- K_2: Baseflow storage coefficient. K_2 should be smaller than K_1.
- K_p: Percolation storage coefficient.
- L: Threshold of the water level in the upper storage.

Model inputs for simulation:

- temp: Array of (mean) temperature for each timestep.
- prec: Array of (summed) precipitation for each timestep. [mm/day]
- month: Array of integers indicating for each timestep to which month it belongs [1,2, …, 12]. Used for adjusted potential evapotranspiration.
- PE_m: long-term mean monthly potential evapotranspiration.
- T_m: long-term mean monthly temperature.

Class documentation

`class rrmpg.models.HBVEdu(params=None)`

Interface to the educational version of the HBV model.

This class builds an interface to the HBV educational model as presented in [1]. This model should only be used with daily data.

If no model parameters are passed upon initialization, generates random parameter set.

Parameters `params` – (optional) Dictonary containing all model parameters as a seperate key/value pairs.

Raises `ValueError` – If a dictionary of model parameters is passed but one of the parameters is missing.

[1] Aghakouchak, Amir, and Emad Habib. "Application of a conceptual hydrologic model in teaching hydrologic processes." International Journal of Engineering Education 26.4 (S1) (2010).

`fit(qobs, temp, prec, month, PE_m, T_m, snow_init=0.0, soil_init=0.0, s1_init=0.0, s2_init=0.0)`

Fit the HBVEdu model to a timeseries of discharge.

This functions uses scipy's global optimizer (differential evolution) to find a good set of parameters for the model, so that the observed discharge is simulated as good as possible.

Parameters

- `qobs` – Array of observed streamflow discharge.
- `temp` – Array of (mean) temperature for each timestep.
- `prec` – Array of (summed) precipitation for each timestep.
- `month` – Array of integers indicating for each timestep to which month it belongs [1,2, …, 12]. Used for adjusted potential evapotranspiration.
- `PE_m` – long-term mean monthly potential evapotranspiration.
- `T_m` – long-term mean monthly temperature.
- `snow_init` – (optional) Initial state of the snow reservoir.
- `soil_init` – (optional) Initial state of the soil reservoir.
- `s1_init` – (optional) Initial state of the near surface flow reservoir.
- `s2_init` – (optional) Initial state of the base flow reservoir.

Returns A scipy OptimizeResult class object.

Return type res

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.
- **`RuntimeErrror`** – If the monthly arrays are not of size 12 or there is a size mismatch between precipitation, temperature and the month array.

`get_default_bounds`()
Return the dictionary containing the default parameter bounds.

`get_dtype`()
Return the custom model datatype.

`get_parameter_names`()
Return the list of parameter names.

`get_params`()
Return a dict with all model parameters and their current value.

`get_random_params`(*num=1*)
Generate random sets of model parameters in the default bounds.

Samples num values for each model parameter from a uniform distribution between the default bounds.

Parameters **`num`** – (optional) Integer, specifying the number of parameter sets, that will be generated. Default is 1.

Returns A numpy array of the models custom data type, containing the at random generated parameters.

`set_params`(*params*)
Set model parameters to values passed in params.

Parameters **`params`** – Either a dictonary containing model parameters as key/value pairs or a numpy array of the models own custom dtype. For the case of dictionaries, they can contain only one model parameter or many/all. The naming of the parameters must be identical to the names specified in the _param_list. All parameter values must be numerical.

Raises

- `ValueError` – If any parameter is not a numerical value.
- `AttributeError` – If the parameter dictonary contains a key, that doesn't match any of the parameter names.
- `TypeError` – If the numpy array of the parameter does not fit the custom data type of the model or it's neither a dict nor a numpy ndarray.

`simulate`(*temp*, *prec*, *month*, *PE_m*, *T_m*, *snow_init=0*, *soil_init=0*, *s1_init=0*, *s2_init=0*, *return_storage=False*, *params=None*)
Simulate rainfall-runoff process for given input.

This function bundles the model parameters and validates the meteorological inputs, then calls the optimized model routine. The meteorological inputs can be either list, numpy array or pandas Series.

Parameters

- **`temp`** – Array of (mean) temperature for each timestep.
- **`prec`** – Array of (summed) precipitation for each timestep.
- **`month`** – Array of integers indicating for each timestep to which month it belongs [1,2, . . . , 12]. Used for adjusted potential evapotranspiration.
- **`PE_m`** – long-term mean monthly potential evapotranspiration.

- **T_m** – long-term mean monthly temperature.
- **snow_init** – (optional) Initial state of the snow reservoir.
- **soil_init** – (optional) Initial state of the soil reservoir.
- **s1_init** – (optional) Initial state of the near surface flow reservoir.
- **s2_init** – (optional) Initial state of the base flow reservoir.
- **return_storage** – (optional) Boolean, indicating if the model storages should also be returned.
- **params** – (optional) Numpy array of parameter sets, that will be evaluated a once in parallel. Must be of the models own custom data type. If nothing is passed, the parameters, stored in the model object, will be used.

Returns An array with the simulated streamflow and optional one array for each of the four reservoirs.

Raises

- ValueError – If one of the inputs contains invalid values.
- TypeError – If one of the inputs has an incorrect datatype.
- **RuntimeErrror** – If the monthly arrays are not of size 12 or there is a size mismatch between precipitation, temperature and the month array.

3.1.3 GR4J

Implementation of the model described in:

```
Perrin, Charles, Claude Michel, and Vazken Andréassian. "Improvement of a
parsimonious model for streamflow simulation." Journal of hydrology 279.1
(2003): 275-289.
```

Explanation of the model parameters:

- x1: maximum capacity of the production store [mm]
- x2: groundwater exchange coefficient [mm]
- x3: one day ahead maximum capacity of the routing store [mm]
- x4: time base of the unit hydrograph UH1 [days]

Model inputs for simulation:

- prec: Array of precipitation [mm/day]
- etp: Array mean potential evapotranspiration [mm]

Class documentation

`class rrmpg.models.GR4J(params=None)`

Interface to the GR4J hydrological model.

This class builds an interface to the GR4J model, as presented in [1]. This model should only be used with daily data.

If no model parameters are passed upon initialization, generates random parameter set.

[1] Perrin, Charles, Claude Michel, and Vazken Andréassian. "Improvement of a parsimonious model for streamflow simulation." Journal of hydrology 279.1 (2003): 275-289.

Parameters **params** – (optional) Dictionary containing all model parameters as a separate key/value pairs.

Raises `ValueError` – If a dictionary of model parameters is passed but one of the parameters is missing.

`fit(qobs, prec, etp, s_init=0.0, r_init=0.0)`

Fit the GR4J model to a timeseries of discharge.

This functions uses scipy's global optimizer (differential evolution) to find a good set of parameters for the model, so that the observed discharge is simulated as good as possible.

Parameters

- **qobs** – Array of observed streamflow discharge [mm/day]
- **prec** – Array of daily precipitation sum [mm]
- **etp** – Array of mean potential evapotranspiration [mm]
- **s_init** – (optional) Initial value of the production storage as fraction of x1.
- **r_init** – (optional) Initial value of the routing storage as fraction of x3.

Returns A scipy OptimizeResult class object.

Return type res

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.
- `RuntimeErrror` – If there is a size mismatch between the precipitation and the pot. evapotranspiration input.

`get_default_bounds()`

Return the dictionary containing the default parameter bounds.

`get_dtype()`

Return the custom model datatype.

`get_parameter_names()`

Return the list of parameter names.

`get_params()`

Return a dict with all model parameters and their current value.

`get_random_params(num=1)`

Generate random sets of model parameters in the default bounds.

Samples num values for each model parameter from a uniform distribution between the default bounds.

Parameters `num` – (optional) Integer, specifying the number of parameter sets, that will be generated. Default is 1.

Returns A numpy array of the models custom data type, containing the at random generated parameters.

`set_params`(*params*)
Set model parameters to values passed in params.

Parameters `params` – Either a dictonary containing model parameters as key/value pairs or a numpy array of the models own custom dtype. For the case of dictionaries, they can contain only one model parameter or many/all. The naming of the parameters must be identical to the names specified in the _param_list. All parameter values must be numerical.

Raises

- `ValueError` – If any parameter is not a numerical value.
- `AttributeError` – If the parameter dictonary contains a key, that doesn't match any of the parameter names.
- `TypeError` – If the numpy array of the parameter does not fit the custom data type of the model or it's neither a dict nor a numpy ndarray.

`simulate`(*prec*, *etp*, *s_init=0.0*, *r_init=0.0*, *return_storage=False*, *params=None*)
Simulate rainfall-runoff process for given input.

This function bundles the model parameters and validates the meteorological inputs, then calls the optimized model routine. The meteorological inputs can be either list, numpy arrays or pandas Series.

Parameters

- `prec` – Array of daily precipitation sum [mm]
- `etp` – Array of mean potential evapotranspiration [mm]
- `s_init` – (optional) Initial value of the production storage as fraction of x1.
- `r_init` – (optional) Initial value of the routing storage as fraction of x3.
- `return_stprage` – (optional) Boolean, indicating if the model storages should also be returned.
- `params` – (optional) Numpy array of parameter sets, that will be evaluated a once in parallel. Must be of the models own custom data type. If nothing is passed, the parameters, stored in the model object, will be used.

Returns An array with the simulated streamflow and optional one array for each of the two storages.

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.
- `RuntimeError` – If there is a size mismatch between the precipitation and the pot. evapotranspiration input.

3.1.4 Cemaneige

Implementation of the model described in:

```
Valéry, A. "Modélisation précipitations - débit sous influence nivale.
Élaboration d'un module neige et évaluation sur 380 bassins versants".
PhD thesis, Cemagref (Antony), AgroParisTech (Paris), 405 pp. (2010)
```

Explanation of the model parameters:

- CTG: snow-pack inertia factor
- Kf: day-degree factor

Model inputs for simulation:

- prec: Array of daily precipitation sum [mm]
- mean_temp: Array of the mean temperature [C]
- min_temp: Array of the minimum temperature [C]
- max_temp: Array of the maximum temperature [C]
- met_station_height: Height of the meteorological station [m]. Needed to calculate the fraction of solid precipitation and optionally for the extrapolation of the meteorological inputs.
- altitudes: (optionally) List of the median elevation of each elevation layer.

Class documentation

class rrmpg.models.**Cemaneige**(*params=None*)

Interface to the Cemaneige snow routine.

This class builds an interface to the implementation of the Cemaneige snow acounting model, originally developed by A. Valery [1] (french) and also presented in [2] (english). This model should only be used with daily data.

If no model parameters are passed upon initialization, generates random parameter set.

[1] Valéry, A. "Modélisation précipitations – débit sous influence nivale. Élaboration d'un module neige et évaluation sur 380 bassins versants". PhD thesis, Cemagref (Antony), AgroParisTech (Paris), 405 pp. (2010)

[2] Audrey Valery, Vazken Andreassian, Charles Perrin. "'As simple as possible but not simpler': What is useful in a temperature-based snow- accounting routine? Part 2 - Sensitivity analysis of the Cemaneige snow accounting routine in 380 Catchments." Journal of Hydrology 517 (2014) 1176-1187.

Parameters **params** – (optional) Dictonary containing all model parameters as a seperate key/value pairs.

Raises ValueError – If a dictionary of model parameters is passed but one of the parameters is missing.

fit(*obs*, *prec*, *mean_temp*, *min_temp*, *max_temp*, *met_station_height*, *snow_pack_init=0*, *thermal_state_init=0*, *altitudes=[]*)

Fit the Cemaneige model to a observed timeseries

This functions uses scipy's global optimizer (differential evolution) to find a good set of parameters for the model, so that the observed timeseries is simulated as good as possible.

Parameters

- `obs` – Array of the observed timeseries [mm]
- `prec` – Array of daily precipitation sum [mm]
- `mean_temp` – Array of the mean temperature [C]
- `min_temp` – Array of the minimum temperature [C]
- `max_temp` – Array of the maximum temperature [C]
- `met_station_height` – Height of the meteorological station [m]. Needed to calculate the fraction of solid precipitation and optionally for the extrapolation of the meteorological inputs.
- `snow_pack_init` – (optional) Initial value of the snow pack storage
- `thermal_state_init` – (optional) Initial value of the thermal state of the snow pack
- `altitudes` – (optional) List of median altitudes of each elevation layer [m]

Returns A scipy OptimizeResult class object.

Return type res

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.
- `RuntimeErrror` – If there is a size mismatch between the precipitation and the pot. evapotranspiration input.

`get_default_bounds()`
Return the dictionary containing the default parameter bounds.

`get_dtype()`
Return the custom model datatype.

`get_parameter_names()`
Return the list of parameter names.

`get_params()`
Return a dict with all model parameters and their current value.

`get_random_params(`*num=1*`)`
Generate random sets of model parameters in the default bounds.

Samples num values for each model parameter from a uniform distribution between the default bounds.

Parameters `num` – (optional) Integer, specifying the number of parameter sets, that will be generated. Default is 1.

Returns A numpy array of the models custom data type, containing the at random generated parameters.

`set_params(`*params*`)`
Set model parameters to values passed in params.

Parameters `params` – Either a dictonary containing model parameters as key/value pairs or a numpy array of the models own custom dtype. For the case of dictionaries, they can contain only one model parameter or many/all. The naming of the parameters must be identical to the names specified in the _param_list. All parameter values must be numerical.

Raises

- `ValueError` – If any parameter is not a numerical value.
- `AttributeError` – If the parameter dictonary contains a key, that doesn't match any of the parameter names.
- `TypeError` – If the numpy array of the parameter does not fit the custom data type of the model or it's neither a dict nor a numpy ndarray.

simulate(*prec*, *mean_temp*, *min_temp*, *max_temp*, *met_station_height*, *snow_pack_init=0*, *thermal_state_init=0*, *altitudes=[]*, *return_storages=False*, *params=None*)

Simulate the snow-routine of the Cemaneige model.

This function checks the input data and prepares the data for the actual simulation function, which is kept outside of the model class (due to restrictions of Numba). Meteorological input arrays can be either lists, numpy arrays or pandas Series.

In the original Cemaneige model, the catchment is divided into 5 subareas of different elevations with the each of them having the same area. For each elevation layer, the snow routine is calculated separately. Therefore, the meteorological input is extrapolated from the height of the measurement station to the median height of each sub-area. This feature is optional (also the number of elevation layer) in this implementation an can be activated if the corresponding heights of each elevation layer is passed as input. In this case, also the height of the measurement station must be passed.

Parameters

- **prec** – Array of daily precipitation sum [mm]
- **mean_temp** – Array of the mean temperature [C]
- **min_temp** – Array of the minimum temperature [C]
- **max_temp** – Array of the maximum temperature [C]
- **met_station_height** – Height of the meteorological station [m]. Needed to calculate the fraction of solid precipitation and optionally for the extrapolation of the meteorological inputs.
- **snow_pack_init** – (optional) Initial value of the snow pack storage
- **thermal_state_init** – (optional) Initial value of the thermal state of the snow pack
- **altitudes** – (optional) List of median altitudes of each elevation layer [m]
- **return_storages** – (optional) Boolean, indicating if the model storages should also be returned.
- **params** – (optional) Numpy array of parameter sets, that will be evaluated a once in parallel. Must be of the models own custom data type. If nothing is passed, the parameters, stored in the model object, will be used.

Returns An array with the simulated stream flow and optional one array for each of the two storages.

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.
- `RuntimeError` – If there is a size mismatch between meteorological input arrays.

3.1.5 CemaneigeGR4J

This model couples the Cemaneige snow routine with the GR4J model into one model.

```
Valéry, A. "Modélisation précipitations - débit sous influence nivale.
Élaboration d'un module neige et évaluation sur 380 bassins versants".
PhD thesis, Cemagref (Antony), AgroParisTech (Paris), 405 pp. (2010)
```

Explanation of the model parameters:

- CTG: snow-pack inertia factor
- Kf: day-degree factor
- x1: maximum capacity of the production store [mm]
- x2: groundwater exchange coefficient [mm]
- x3: one day ahead maximum capacity of the routing store [mm]
- x4: time base of the unit hydrograph UH1 [days]

Model inputs for simulation:

- prec: Array of daily precipitation sum [mm]
- mean_temp: Array of the mean temperature [C]
- min_temp: Array of the minimum temperature [C]
- max_temp: Array of the maximum temperature [C]
- etp: Array mean potential evapotranspiration [mm]
- met_station_height: Height of the meteorological station [m]. Needed to calculate the fraction of solid precipitation and optionally for the extrapolation of the meteorological inputs.
- altitudes: (optionally) List of the median elevation of each elevation layer.

Class documentation

`class rrmpg.models.`**`CemaneigeGR4J`**`(`*params=None*`)`

Interface to the Cemaneige + GR4J coupled hydrological model.

This class builds an interface to the coupled model, consisting of the Cemaneige snow routine [1] and the GR4J model [2]. This model should only be used with daily data.

If no model parameters are passed upon initialization, generates random parameter set.

[1] Valéry, A. "Modélisation précipitations – débit sous influence nivale. Élaboration d'un module neige et évaluation sur 380 bassins versants". PhD thesis, Cemagref (Antony), AgroParisTech (Paris), 405 pp. (2010)

[2] Perrin, Charles, Claude Michel, and Vazken Andréassian. "Improvement of a parsimonious model for streamflow simulation." Journal of hydrology 279.1 (2003): 275-289.

Parameters **`params`** – (optional) Dictionary containing all model parameters as a separate key/value pairs.

Raises `ValueError` – If a dictionary of model parameters is passed but one of the parameters is missing.

fit(*obs*, *prec*, *mean_temp*, *min_temp*, *max_temp*, *etp*, *met_station_height*, *snow_pack_init=0*, *thermal_state_init=0*, *s_init=0*, *r_init=0*, *altitudes=[]*)

Fit the Cemaneige + GR4J coupled model to a observed timeseries

This functions uses scipy's global optimizer (differential evolution) to find a good set of parameters for the model, so that the observed timeseries is simulated as good as possible.

Parameters

- **obs** – Array of the observed timeseries [mm]
- **prec** – Array of daily precipitation sum [mm]
- **mean_temp** – Array of the mean temperature [C]
- **min_temp** – Array of the minimum temperature [C]
- **max_temp** – Array of the maximum temperature [C]
- **etp** – Array of mean potential evapotranspiration [mm]
- **met_station_height** – Height of the meteorological station [m]. Needed to calculate the fraction of solid precipitation and optionally for the extrapolation of the meteorological inputs.
- **snow_pack_init** – (optional) Initial value of the snow pack storage
- **thermal_state_init** – (optional) Initial value of the thermal state of the snow pack
- **s_init** – (optional) Initial value of the production storage as fraction of x1.
- **r_init** – (optional) Initial value of the routing storage as fraction of x3.
- **altitudes** – (optional) List of median altitudes of each elevation layer [m]

Returns A scipy OptimizeResult class object.

Return type res

Raises

- **ValueError** – If one of the inputs contains invalid values.
- **TypeError** – If one of the inputs has an incorrect datatype.
- **RuntimeErrror** – If there is a size mismatch between the precipitation and the pot. evapotranspiration input.

get_default_bounds()

Return the dictionary containing the default parameter bounds.

get_dtype()

Return the custom model datatype.

get_parameter_names()

Return the list of parameter names.

get_params()

Return a dict with all model parameters and their current value.

get_random_params(*num=1*)

Generate random sets of model parameters in the default bounds.

Samples num values for each model parameter from a uniform distribution between the default bounds.

Parameters **num** – (optional) Integer, specifying the number of parameter sets, that will be generated. Default is 1.

Returns A numpy array of the models custom data type, containing the at random generated parameters.

`set_params`(*params*)
Set model parameters to values passed in params.

Parameters `params` – Either a dictonary containing model parameters as key/value pairs or a numpy array of the models own custom dtype. For the case of dictionaries, they can contain only one model parameter or many/all. The naming of the parameters must be identical to the names specified in the _param_list. All parameter values must be numerical.

Raises

- `ValueError` – If any parameter is not a numerical value.
- `AttributeError` – If the parameter dictonary contains a key, that doesn't match any of the parameter names.
- `TypeError` – If the numpy array of the parameter does not fit the custom data type of the model or it's neither a dict nor a numpy ndarray.

`simulate`(*prec*, *mean_temp*, *min_temp*, *max_temp*, *etp*, *met_station_height*, *snow_pack_init=0*, *thermal_state_init=0*, *s_init=0*, *r_init=0*, *altitudes=[]*, *return_storages=False*, *params=None*)
Simulate the Cemaneige + GR4J coupled hydrological model.

This function checks the input data and prepares the data for the actual simulation function, which is kept outside of the model class (due to restrictions of Numba). Meteorological input arrays can be either lists, numpy arrays or pandas Series.

In the original Cemaneige model, the catchment is divided into 5 subareas of different elevations with the each of them having the same area. For each elevation layer, the snow routine is calculated separately. Therefore, the meteorological input is extrapolated from the height of the measurement station to the median height of each sub-area. This feature is optional (also the number of elevation layer) in this implementation an can be activated if the corresponding heights of each elevation layer is passed as input. In this case, also the height of the measurement station must be passed.

Parameters

- `prec` – Array of daily precipitation sum [mm]
- `mean_temp` – Array of the mean temperature [C]
- `min_temp` – Array of the minimum temperature [C]
- `max_temp` – Array of the maximum temperature [C]
- `etp` – Array of mean potential evapotranspiration [mm]
- `met_station_height` – Height of the meteorological station [m]. Needed to calculate the fraction of solid precipitation and optionally for the extrapolation of the meteorological inputs.
- `snow_pack_init` – (optional) Initial value of the snow pack storage
- `thermal_state_init` – (optional) Initial value of the thermal state of the snow pack
- `s_init` – (optional) Initial value of the production storage as fraction of x1.
- `r_init` – (optional) Initial value of the routing storage as fraction of x3.
- `altitudes` – (optional) List of median altitudes of each elevation layer [m]
- `return_storages` – (optional) Boolean, indicating if the model storages should also be returned.

- **params** – (optional) Numpy array of parameter sets, that will be evaluated a once in parallel. Must be of the models own custom data type. If nothing is passed, the parameters, stored in the model object, will be used.

Returns An array with the simulated stream flow and optional one array for each of the two storages.

Raises

- `ValueError` – If one of the inputs contains invalid values.
- `TypeError` – If one of the inputs has an incorrect datatype.
- `RuntimeError` – If there is a size mismatch between meteorological input arrays.

3.2 Tools

Documentation of all functions defined within the `rrmpg.tools` module.

3.2.1 Monte-Carlo

`rrmpg.tools.monte_carlo.`**`monte_carlo`**`(`*model*, *num*, *qobs=None*, ***kwargs*`)`

Perform Monte-Carlo-Simulation.

This function performs a Monte-Carlo-Simulation for any given hydrological model of this repository.

Parameters

- **model** – Any instance of a hydrological model of this repository.
- **num** – Number of simulations.
- **qobs** – (optional) Array of observed streamflow.
- ****kwargs** – Keyword arguments, matching the inputs the model needs to perform a simulation (e.g. qobs, precipitation, temperature etc.). See help(model.simulate) for model input requirements.

Returns A dictonary containing the following two keys [‘params’, ‘qsim’]. The key ‘params’ contains a numpy array with the model parameter of each simulation. ‘qsim’ is a 2D numpy array with the simulated streamflow for each simulation. If an array of observed streamflow is provided, one additional key is returned in the dictonary, being ‘mse’. This key contains an array of the mean-squared-error for each simulation.

Raises

- `ValueError` – If any input contains invalid values.
- `TypeError` – If any of the inputs has a wrong datatype.

3.3 Utils

Documentation of all functions defined within the `rrmpg.utils` module.

3.3.1 Plot utils

`rrmpg.utils.plot_utils.`**`plot_qsim_range`**`(`*qsim*, *x_vals=None*, *qobs=None*`)`

Plot the range of multiple simulations and their mean.

This function plots the quantiles of multiple simulations as a filled area and the mean as a line. The (0.05, 0.95) and the (0.25, 0.75) quantile are plotted as different colored areas and the mean as a solid line. If observations are also passed, they are plotted as well as a solid line.

Parameters

- **`qsim`** – 2D array of simulations. Shape must be (num_timesteps, num_sims)
- **`x_vals`** – (optional) 1D array, that will be used as x-axes values. (e.g. date)
- **`qobs`** – (optional) 1D arary of oversations.

Returns A handle to the matplotlib figure.

Raises `ValueError` – For incorrect inputs.

3.3.2 Metrics

`rrmpg.utils.metrics.`**`calc_mse`**`(`*obs*, *sim*`)`

Calculate the mean squared error.

Parameters

- **`obs`** – Array of the observed values
- **`sim`** – Array of the simulated values

Returns The MSE value for the simulation, compared to the observation.

Raises

- `ValueError` – If the arrays are not of equal size or have non-numeric values.
- `TypeError` – If the arrays is not a supported datatype.

`rrmpg.utils.metrics.`**`calc_rmse`**`(`*obs*, *sim*`)`

Calculate the root mean squared error.

Parameters

- **`obs`** – Array of the observed values
- **`sim`** – Array of the simulated values

Returns The RMSE value for the simulation, compared to the observation.

Raises

- `ValueError` – If the arrays are not of equal size or have non-numeric values.
- `TypeError` – If the arrays is not a supported datatype.

`rrmpg.utils.metrics.calc_nse(obs, sim)`

Calculate the Nash-Sutcliffe model efficiency coefficient.

Original Publication: Nash, J. Eamonn, and Jonh V. Sutcliffe. "River flow forecasting through conceptual models part I—A discussion of principles." Journal of hydrology 10.3 (1970): 282-290.

Parameters

- **obs** – Array of the observed values
- **sim** – Array of the simulated values

Returns The NSE value for the simulation, compared to the observation.

Raises

- `ValueError` – If the arrays are not of equal size or have non-numeric values.
- `TypeError` – If the arrays is not a supported datatype.
- `RuntimeError` – If all values in qobs are equal. The NSE is not defined for this cases.

`rrmpg.utils.metrics.calc_kge(obs, sim)`

Calculate the Kling-Gupta-Efficiency.

Calculate the original KGE value following [1].

Parameters

- **obs** – Array of the observed values
- **sim** – Array of the simulated values

Returns The KGE value for the simulation, compared to the observation.

Raises

- `ValueError` – If the arrays are not of equal size or have non-numeric values.
- `TypeError` – If the arrays is not a supported datatype.
- `RuntimeError` – If the mean or the standard deviation of the observations equal 0.

[1] Gupta, H. V., Kling, H., Yilmaz, K. K., & Martinez, G. F. (2009). Decomposition of the mean squared error and NSE performance criteria: Implications for improving hydrological modelling. Journal of Hydrology, 377(1-2), 80-91.

`rrmpg.utils.metrics.calc_alpha_nse(obs, sim)`

Calculate the alpha decomposition of the NSE.

Calculate the alpha decomposition of the NSE following [1].

Parameters

- **obs** – Array of the observed values
- **sim** – Array of the simulated values

Returns The alpha decomposition of the NSE of the simulation compared to the observation.

Raises

- `ValueError` – If the arrays are not of equal size or have non-numeric values.
- `TypeError` – If the arrays is not a supported datatype.
- `RuntimeError` – If the standard deviation of the observations equal 0.

[1] Gupta, H. V., Kling, H., Yilmaz, K. K., & Martinez, G. F. (2009). Decomposition of the mean squared error and NSE performance criteria: Implications for improving hydrological modelling. Journal of Hydrology, 377(1-2), 80-91.

`rrmpg.utils.metrics.`**`calc_beta_nse`**`(`*obs*, *sim*`)`

Calculate the beta decomposition of the NSE.

Calculate the beta decomposition of the NSE following [1].

Parameters

- **`obs`** – Array of the observed values
- **`sim`** – Array of the simulated values

Returns The beta decomposition of the NSE of the simulation compared to the observation.

Raises

- `ValueError` – If the arrays are not of equal size or have non-numeric values.
- `TypeError` – If the arrays is not a supported datatype.
- `RuntimeError` – If the mean or the standard deviation of the observations equal 0.

[1] Gupta, H. V., Kling, H., Yilmaz, K. K., & Martinez, G. F. (2009). Decomposition of the mean squared error and NSE performance criteria: Implications for improving hydrological modelling. Journal of Hydrology, 377(1-2), 80-91.

`rrmpg.utils.metrics.`**`calc_r`**`(`*obs*, *sim*`)`

Calculate the pearson r coefficient.

Interface to the scipy implementation of the pearson r coeffienct.

Parameters

- **`obs`** – Array of the observed values
- **`sim`** – Array of the simulated values

Returns The pearson r coefficient of the simulation compared to the observation.

3.3.3 Array Checks

`rrmpg.utils.array_checks.`**`check_for_negatives`**`(`*arr*`)`

Check if array contains negative number.

Numba optimized function to check if a numpy array containes a negative value. Returns, whenever the first negative function is found.

Parameters **`arr`** – Numpy array

Returns True, if the array contains at least on negative number and False, if the array contains no negative number.

`rrmpg.utils.array_checks.`**`validate_array_input`**`(`*arr*, *dtype*, *arr_name*`)`

Check if array has correct type and is numerical.

This function checks if the input is either a list, numpy.ndarray or pandas.Series of numerical values, converts it to a numpy.ndarray and throws an error in case of incorrect data.

Parameters

- **`arr`** – Array of data

- **dtype** – One of numpy's dtypes
- **arr_name** – String specifing the variable name, so that the error message can be adapted correctly.

Returns A as numpy.ndarray converted array of values with a datatype specified in the input argument.

Raises

- `ValueError` – In case non-numerical data is passed
- `TypeError` – If the error is neither a list, a numpy.ndarray nor a pandas.Series

3.4 Data

Documentation of all functions and classes defined within the `rrmpg.data` module.

3.4.1 CAMELSLoader

`class rrmpg.data.CAMELSLoader`

Interface for loading basin data from the CAMELS dataset.

This class provides an easy to use interface to load different basins from the CAMELS [1] dataset provided within this Python package. CAMELS stands for Catchment Attributes for Large-Sample Studies and is a hydrological dataset provided by NCAR for 671 catchments in the USA. The data entire data can be downloaded for free at [2]. Within this package we provide the data of just a few catchments as toy data for this package.

[1] Addor, N., A.J. Newman, N. Mizukami, and M.P. Clark, 2017: The CAMELS data set: catchment attributes and meteorology for large-sample studies. version 2.0. Boulder, CO: UCAR/NCAR. doi:10.5065/D6G73C3Q

[2] https://ncar.github.io/hydrology/datasets/CAMELS_attributes

`get_basin_numbers()`

Return a list of all available basin numbers.

`get_station_height(basin_number)`

Return the elevation of the meteorological station of one basin.

Parameters **basin_number** – String of the basin number that shall be loaded.

Returns The elevation of the meteorological station.

Raises `ValueError` – If the basin number is an invalid number. Check the .get_basin_numbers() function for a list of all available basins.

`load_basin(basin_number)`

Load basin data pandas Dataframe.

Load the meteorological data, as well as observed discharge and modeled potential evapotranspiration of the specified basin from the CAMELS data set.

Parameters **basin_number** – String of the basin number that shall be loaded.

Returns A pandas DataFrame with the data of the basin.

Raises `ValueError` – If the basin number is an invalid number. Check the .get_basin_numbers() function for a list of all available basins.

CHAPTER

FOUR

EXAMPLES

Here will follow some example applications of the RRMPG library.

4.1 Numba Speed-Test

In this notebook I'll test the speed of a simple hydrological model (the ABC-Model [1]) implemented in pure Python, Numba and Fortran. This should only been seen as an example of the power of numba in speeding up array-oriented python functions, that have to be processed using loops. This is for example the case for hydrological models that have to be processed timestep after timestep to update model states (depending on previous states) and calculate flows. Python is natively very slow for this kind of functions (loops). Normally hydrological (as well as meterological and environmental) models are implemented in Fortran or C/C++ which are known for their speed. The downside is, that this languages are quite harder to start with and the code often seems overly complicated for beginner. Numba is a library that performs just-in-time compilation on Python code and can therefore dramatically increase the speed of Python functions (without having to change much in the code).

Anyway, this is not meant to give an introduction to numba, but just to compare the execution speed against pure Python and Fortan. For everybody, who is interested in further explanations on Numba see: - Gil Forsyth's & Lorena Barba's tutorial from the SciPy 2017 - The numba homepage, which includes examples

If you want to reproduce the results and you have installed a conda environment using the environment.yml from the rrmpg github repository make sure to additionally install `cython`:

```
conda install -c anaconda cython
```

[1] Myron B. Fiering "Streamflow synthesis" Cambridge, Harvard University Press, 1967. 139 P. (1967).

```
# Notebook setups
import numpy as np

from numba import njit, float64
from timeit import timeit
```

We'll use an array of random numbers as input for the model. Since we only want to test the execution time, this will work for now.

```
# Let's an array of 10 mio values
rain = np.random.random(size=10000000)
```

Next we are going to define three different functions:

1. `abc_model_py`: An implementation of the ABC-Model using pure Python.

2. `abc_model_numba`: A numba version of the ABC-model. The just-in-time compilation is achieved by adding a numba decorator over the function definition. I use the `@njit` to make sure an error is raised if numba can't compile the function.
3. `abc_model_fortan`: A fortan version of the ABC-model. In previous version this was done using the f2py module which added some overhead to the function call and was no fair benchmark (see pull request #3). Now the Fortran implementation is wrapped in a Cython function.

Note how for this simple model the only difference between the pure Python version and the Numba version is the decorator. The entire code of the model is the same.

```
# pure Python implementation
def abc_model_py(a, b, c, rain):
    outflow = np.zeros((rain.size), dtype=np.float64)
    state_in = 0
    state_out = 0
    for i in range(rain.size):
        state_out = (1 - c) * state_in + a * rain[i]
        outflow[i] = (1 - a - b) * rain[i] + c * state_in
        state_in = state_out
    return outflow

# numba version of the ABC-model
@njit(['float64[:](float64,float64,float64,float64[:])'])
def abc_model_numba(a, b, c, rain):
    outflow = np.zeros((rain.size), dtype=np.float64)
    state_in = 0
    state_out = 0
    for i in range(rain.size):
        state_out = (1 - c) * state_in + a * rain[i]
        outflow[i] = (1 - a - b) * rain[i] + c * state_in
        state_in = state_out
    return outflow
```

```
%%file abc.f90

module abc
use iso_c_binding, only: c_int, c_double
implicit none
integer, parameter :: dp = kind(0d0)
private
public c_abc_model_fortran

contains


subroutine c_abc_model_fortran(n, a, b, c, inflow, outflow) bind(c)
integer(c_int), intent(in), value :: n
real(c_double), intent(in), value :: a, b, c
real(c_double), intent(in) :: inflow(n)
real(c_double), intent(out) :: outflow(n)
call abc_model(a, b, c, inflow, outflow)
end subroutine

```

(continues on next page)

(continued from previous page)

```
subroutine abc_model(a, b, c, inflow, outflow)
real(dp), intent(in) :: a, b, c, inflow(:)
real(dp), intent(out) :: outflow(:)
real(dp) :: state_in, state_out
integer :: t
state_in = 0
do t = 1, size(inflow)
    state_out = (1 - c) * state_in + a * inflow(t)
    outflow(t) = (1 - a - b) * inflow(t) + c * state_in
    state_in = state_out
end do
end subroutine


end module
```

```
Writing abc.f90
```

```
%%file abc_py.pyx

from numpy cimport ndarray
from numpy import empty, size

cdef extern:
    void c_abc_model_fortran(int n, double a, double b, double c, double *inflow, double␣
↪*outflow)

def abc_model_fortran(double a, double b, double c, ndarray[double, mode="c"] inflow):
    cdef int N = size(inflow)
    cdef ndarray[double, mode="c"] outflow = empty(N, dtype="double")
    c_abc_model_fortran(N, a, b, c, &inflow[0], &outflow[0])
    return outflow
```

```
Writing abc_py.pyx
```

Compile the Fortran and Cython module

```
%%bash
set -e
#set -x
# Debug flags
#FFLAGS="-Wall -Wextra -Wimplicit-interface -fPIC -fmax-errors=1 -g -fcheck=all -
↪fbacktrace"
#CFLAGS="-Wall -Wextra -fPIC -fmax-errors=1 -g"
# Release flags
FFLAGS="-fPIC -O3 -march=native -ffast-math -funroll-loops"
CFLAGS="-fPIC -O3 -march=native -ffast-math -funroll-loops"
gfortran -o abc.o -c abc.f90 $FFLAGS
cython abc_py.pyx
gcc -o abc_py.o -c abc_py.c -I$CONDA_PREFIX/include/python3.6m/ $CFLAGS
gcc -o abc_py.so abc_py.o abc.o -L$CONDA_PREFIX/lib -lpython3.6m -lgfortran -shared
```

```
# Now we can import it like a normal Python module
from abc_py import abc_model_fortran
```

Now we'll use the `timeit` package to measure the execution time of each of the functions

```
# Measure the execution time of the Python implementation
py_time = %timeit -o abc_model_py(0.2, 0.6, 0.1, rain)
```

```
5.83 s ± 70.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
# Measure the execution time of the Numba implementation
numba_time = %timeit -o abc_model_numba(0.2, 0.6, 0.1, rain)
```

```
33.3 ms ± 707 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
# Measure the execution time of the Fortran implementation
fortran_time = %timeit -o abc_model_fortran(0.2, 0.6, 0.1, rain)
```

```
23.7 ms ± 37.7 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

As you can see by the raw numbers, Fortran (as expected) is the fastest, but what is interesting, that the Numba version of the ABC-Model does not perform much worse. Let's compare the numbers.

First we'll compare the pure Python version, against the Numba version. Remember, everthing we did was to add a decorator to the Python function, the rest (the magic) is done by the Numba library.

```
py_time.best / numba_time.best
```

```
174.3980506055237
```

Wow, this is roughly a 205 x speed up by one single additional line of code. Note that for more complicated models, we'll have to adapt the code a bit more, but in general it will stay very close to normal Python code.

Now let's see how the Numba version performs against Fortran, which is still the standard in the modelling community of hydrology and meteorology.

```
numba_time.best / fortran_time.best
```

```
1.3963906102603512
```

So the Fortran implementation is still faster but not much. We only need less than 1,5x the time of the Fortran version if we run the Python code optimized with the Numba library.

Note that this Fortran function is compiled using the GNU Fortran compiler, which is open source and free. Using e.g. the Intel Fortran compiler will certainly increase speed of the Fortran function, but I think it's only fair to compare two open source and free-of-charge versions.

So what does this mean

We'll see, but you will now have maybe a better idea of this project. The thing is, we can implement models in Python, that have roughly the performance of Fortran, but are at the same time less complex to implement and play around with. We can also save a lot of boilerplate code we need with Fortran to compiler our code in the most optimal way. We only need to follow some rules of the Numba library and for the rest, add one decorator to the function definition. We can run 1000s of simulations and don't have to wait for ages and we can stay the entire time in one environment (for simulating and evaluating the results). The hope is, that this will help fellow students/researchers to better understand hydrological models and lose fear of what might seem intimidating at first, follwing a quote by Richard Feynman:

"What I can not create, I do not understand" - Richard Feynman

4.2 Model API Example

In this notebook, we'll explore some functionality of the models of this package. We'll work with the coupled CemaneigeGR4j model that is implemented in `rrmpg.models` module. The data we'll use, comes from the CAMELS [1] data set. For some basins, the data is provided within this Python library and can be easily imported using the `CAMELSLoader` class implemented in the `rrmpg.data` module.

In summary we'll look at: - How you can create a model instance. - How we can use the CAMELSLoader. - How you can fit the model parameters to observed discharge by: - Using one of SciPy's global optimizer - Monte-Carlo-Simulation - How you can use a fitted model to calculate the simulated discharge.

[1] Addor, N., A.J. Newman, N. Mizukami, and M.P. Clark, 2017: The CAMELS data set: catchment attributes and meteorology for large-sample studies. version 2.0. Boulder, CO: UCAR/NCAR. doi:10.5065/D6G73C3Q

```
# Imports and Notebook setup
from timeit import timeit

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from rrmpg.models import CemaneigeGR4J
from rrmpg.data import CAMELSLoader
from rrmpg.tools.monte_carlo import monte_carlo
from rrmpg.utils.metrics import calc_nse
```

4.2.1 Create a model

As a first step let us have a look how we can create one of the models implemented in `rrmpg.models`. Basically, for all models we have two different options: 1. Initialize a model **without** specific model parameters. 2. Initialize a model **with** specific model parameters.

The documentation provides a list of all model parameters. Alternatively we can look at `help()` for the model (e.g. `help(CemaneigeGR4J)`).

If no specific model parameters are provided upon intialization, random parameters will be generated that are in between the default parameter bounds. We can look at these bounds by calling `.get_param_bounds()` method on the model object and check the current parameter values by calling `.get_params()` method.

For now we don't know any specific parameter values, so we'll create one with random parameters.

```
model = CemaneigeGR4J()
model.get_params()
```

```
{'CTG': 0.3399735717656279,
 'Kf': 0.8724652383290821,
 'x1': 427.9652389107806,
 'x2': 0.9927197563086638,
```

(continues on next page)

(continued from previous page)

```
 'x3': 288.20205223188475,
 'x4': 1.4185137324914372}
```

Here we can see the six model parameters of CemaneigeGR4J model and their current values.

4.2.2 Using the CAMELSLoader

To have data to start with, we can use the `CAMELSLoader` class to load data of provided basins from the CAMELS dataset. To get a list of all available basins that are provided within this library, we can use the `.get_basin_numbers()` method. For now we will use the provided basin number `01031500`.

```
df = CAMELSLoader().load_basin('01031500')
df.head()
```

Next we will split the data into a calibration period, which we will use to find a set of good model parameters, and a validation period, we will use the see how good our model works on unseen data. As in the CAMELS data set publication, we will use the first 15 hydrological years for calibration. The rest of the data will be used for validation.

Because the index of the dataframe is in pandas Datetime format, we can easily split the dataframe into two parts

```
# calcute the end date of the calibration period
end_cal = pd.to_datetime(f"{df.index[0].year + 15}/09/30", yearfirst=True)

# validation period starts one day later
start_val = end_cal + pd.DateOffset(days=1)

# split the data into two parts
cal = df[:end_cal].copy()
val = df[start_val:].copy()
```

4.2.3 Fit the model to observed discharge

As already said above, we'll look at two different methods implemented in this library: 1. Using one of SciPy's global optimizer 2. Monte-Carlo-Simulation

Using one of SciPy's global optimizer

Each model has a `.fit()` method. This function uses the global optimizer differential evolution from the scipy package to find the set of model parameters that produce the best simulation, regarding the provided observed discharge array. The inputs for this function can be found in the documentation or the `help()`.

```
help(model.fit)
```

```
Help on method fit in module rrmpg.models.cemaneigegr4j:

fit(obs, prec, mean_temp, min_temp, max_temp, etp, met_station_height, snow_pack_init=0,␣
↪thermal_state_init=0, s_init=0, r_init=0, altitudes=[]) method of rrmpg.models.
↪cemaneigegr4j.CemaneigeGR4J instance
    Fit the Cemaneige + GR4J coupled model to a observed timeseries
```

(continues on next page)

(continued from previous page)

```
    This functions uses scipy's global optimizer (differential evolution)
    to find a good set of parameters for the model, so that the observed
    timeseries is simulated as good as possible.

    Args:
        obs: Array of the observed timeseries [mm]
        prec: Array of daily precipitation sum [mm]
        mean_temp: Array of the mean temperature [C]
        min_temp: Array of the minimum temperature [C]
        max_temp: Array of the maximum temperature [C]
        etp: Array of mean potential evapotranspiration [mm]
        met_station_height: Height of the meteorological station [m].
            Needed to calculate the fraction of solid precipitation and
            optionally for the extrapolation of the meteorological inputs.
        snow_pack_init: (optional) Initial value of the snow pack storage
        thermal_state_init: (optional) Initial value of the thermal state
            of the snow pack
        s_init: (optional) Initial value of the production storage as
            fraction of x1.
        r_init: (optional) Initial value of the routing storage as fraction
            of x3.
        altitudes: (optional) List of median altitudes of each elevation
            layer [m]

    Returns:
        res: A scipy OptimizeResult class object.

    Raises:
        ValueError: If one of the inputs contains invalid values.
        TypeError: If one of the inputs has an incorrect datatype.
        RuntimeErrror: If there is a size mismatch between the
            precipitation and the pot. evapotranspiration input.
```

We don’t know any values for the initial states of the storages, so we will ignore them for now. For the missing mean temperature, we calculate a proxy from the minimum and maximum daily temperature. The station height can be retrieved from the `CAMELSLoader` class via the `.get_station_height()` method.

```
# calculate mean temp for calibration and validation period
cal['tmean'] = (cal['tmin(C)'] + cal['tmax(C)']) / 2
val['tmean'] = (val['tmin(C)'] + val['tmax(C)']) / 2

# load the gauge station height
height = CAMELSLoader().get_station_height('01031500')
```

Now we are ready to fit the model and retrieve a good set of model parameters from the optimizer. Again, this will be done with the calibration data. Because the model methods also except pandas Series, we can call the function as follows.

```
# We don't have an initial value for the snow storage,  so we omit this input
result = model.fit(cal['QObs(mm/d)'], cal['prcp(mm/day)'], cal['tmean'],
                   cal['tmin(C)'], cal['tmax(C)'], cal['PET'], height)
```

`result` is an object defined by the scipy library and contains the optimized model parameters, as well as some more information on the optimization process. Let us have a look at this object:

```
result
```

```
    fun: 1.6435277126036711
    jac: array([ 0.00000000e+00,  3.68594044e-06, -1.36113343e-05, -6.66133815e-06,
      -4.21884749e-07,  7.35368810e-01])
message: 'Optimization terminated successfully.'
   nfev: 2452
    nit: 25
success: True
      x: array([7.60699105e-02, 4.22084687e+00, 1.45653881e+02, 1.14318020e+00,
      5.87237837e+01, 1.10000000e+00])
```

The relevant information here is: - `fun` is the final value of our optimization criterion (the mean-squared-error in this case) - `message` describes the cause of the optimization termination - `nfev` is the number of model simulations - `sucess` is a flag wether or not the optimization was successful - `x` are the optimized model parameters

Next, let us set the model parameters to the optimized ones found by the search. Therefore we need to create a dictonary containing one key for each model parameter and as the corresponding value the optimized parameter. As mentioned before, the list of model parameter names can be retrieved by the `model.get_parameter_names()` function. We can then create the needed dictonary by the following lines of code:

```
params = {}

param_names = model.get_parameter_names()

for i, param in enumerate(param_names):
    params[param] = result.x[i]

# This line set the model parameters to the ones specified in the dict
model.set_params(params)

# To be sure, let's look at the current model parameters
model.get_params()
```

```
{'CTG': 0.07606991045128364,
 'Kf': 4.220846873695767,
 'x1': 145.6538807127758,
 'x2': 1.143180196835088,
 'x3': 58.723783711432226,
 'x4': 1.1}
```

Also it might not be clear at the first look, this are the same parameters as the ones specified in `result.x`. In `result.x` they are ordered according to the ordering of the `_param_list` specified in each model class, where ass the dictonary output here is alphabetically sorted.

Monte-Carlo-Simulation

Now let us have a look how we can use the Monte-Carlo-Simulation implemented in `rrmpg.tools.monte_carlo`.

```
help(monte_carlo)
```

```
Help on function monte_carlo in module rrmpg.tools.monte_carlo:

monte_carlo(model, num, qobs=None, **kwargs)
    Perform Monte-Carlo-Simulation.

    This function performs a Monte-Carlo-Simulation for any given hydrological
    model of this repository.

    Args:
        model: Any instance of a hydrological model of this repository.
        num: Number of simulations.
        qobs: (optional) Array of observed streamflow.
        **kwargs: Keyword arguments, matching the inputs the model needs to
            perform a simulation (e.g. qobs, precipitation, temperature etc.).
            See help(model.simulate) for model input requirements.

    Returns:
        A dictonary containing the following two keys ['params', 'qsim']. The
        key 'params' contains a numpy array with the model parameter of each
        simulation. 'qsim' is a 2D numpy array with the simulated streamflow
        for each simulation. If an array of observed streamflow is provided,
        one additional key is returned in the dictonary, being 'mse'. This key
        contains an array of the mean-squared-error for each simulation.

    Raises:
        ValueError: If any input contains invalid values.
        TypeError: If any of the inputs has a wrong datatype.
```

As specified in the help text, all model inputs needed for a simulation must be provided as keyword arguments. The keywords need to match the names specified in the `model.simulate()` function. Let us create a new model instance and see how this works for the CemaneigeGR4J model.

```
model2 = CemaneigeGR4J()

# Let use run MC for 1000 runs, which is in the same range as the above optimizer
result_mc = monte_carlo(model2, num=10000, qobs=cal['QObs(mm/d)'],
                        prec=cal['prcp(mm/day)'], mean_temp=cal['tmean'],
                        min_temp=cal['tmin(C)'], max_temp=cal['tmax(C)'],
                        etp=cal['PET'], met_station_height=height)

# Get the index of the best fit (smallest mean squared error)
idx = np.argmin(result_mc['mse'][~np.isnan(result_mc['mse'])])

# Get the optimal parameters and set them as model parameters
optim_params = result_mc['params'][idx]

params = {}
```

(continues on next page)

(continued from previous page)

```
for i, param in enumerate(param_names):
    params[param] = optim_params[i]

# This line set the model parameters to the ones specified in the dict
model2.set_params(params)
```

4.2.4 Calculate simulated discharge

We now have two models, optimized by different methods. Let's calculate the simulated streamflow of each model and compare the results! Each model has a `.simulate()` method, that returns the simulated discharge for the inputs we provide to this function.

```
# simulated discharge of the model optimized by the .fit() function
val['qsim_fit'] = model.simulate(val['prcp(mm/day)'], val['tmean'],
                                 val['tmin(C)'], val['tmax(C)'],
                                 val['PET'], height)

# simulated discharge of the model optimized by monte-carlo-sim
val['qsim_mc'] = model2.simulate(val['prcp(mm/day)'], val['tmean'],
                                 val['tmin(C)'], val['tmax(C)'],
                                 val['PET'], height)

# Calculate and print the Nash-Sutcliff-Efficiency for both simulations
nse_fit = calc_nse(val['QObs(mm/d)'], val['qsim_fit'])
nse_mc = calc_nse(val['QObs(mm/d)'], val['qsim_mc'])

print("NSE of the .fit() optimization: {:.4f}".format(nse_fit))
print("NSE of the Monte-Carlo-Simulation: {:.4f}".format(nse_mc))
```

```
NSE of the .fit() optimization: 0.8075
NSE of the Monte-Carlo-Simulation: 0.7332
```

What do this number mean? Let us have a look at some window of the simulated timeseries and compare them to the observed discharge:

```
# Plot last full hydrological year of the simulation
%matplotlib notebook
start_date = pd.to_datetime("2013/10/01", yearfirst=True)
end_date = pd.to_datetime("2014/09/30", yearfirst=True)
plt.plot(val.loc[start_date:end_date, 'QObs(mm/d)'], label='Qobs')
plt.plot(val.loc[start_date:end_date, 'qsim_fit'], label='Qsim .fit()')
plt.plot(val.loc[start_date:end_date, 'qsim_mc'], label='Qsim mc')
plt.legend()
```

```
<IPython.core.display.Javascript object>
```

```
<matplotlib.legend.Legend at 0x7f1b03d61198>
```

The result is not perfect, but it is not bad either! And since this package is also about speed, let us also check how long it takes to simulate the discharge for the entire validation period (19 years of data).

```
%%timeit
model.simulate(val['prcp(mm/day)'], val['tmean'],
                                 val['tmin(C)'], val['tmax(C)'],
                                 val['PET'], height)
```

```
2.46 ms ± 42.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

CHAPTER FIVE

WIKI

Here I'll add soon detailed explanations for each of the implemented models.

CHAPTER

SIX

HOW TO CONTRIBUTE

Since this is a community project, I would deeply encourage any developer or hydrologist to contribute to the development of this project. Different ways of how you can contribute exist and not all require programming skills. If you want to contribute, make sure you have the latest version and everything is setup correctly (see *Getting started*).

6.1 Spell checking

Since English is not my native-language, I'm sure there are a lot of mistakes in this documentation or the code documentation itself. Feel free to make a pull request on GitHub (or open an issue or write me an email, whatever is the most comfortable for you) and I'll gladly correct the mistakes.

6.2 Contribute to the wiki

The *Wiki* should give a more detailed description of the model (including e.g. historical background and application examples). If feasable, also visualizations of the model structure can be added.

The idea is, to have a summary of the model, that provides enough information, such that anyone without previous knowledge of the model understands the model capabilities/weaknesses and knows what the model is doing. Articles don't have to be written in one push and can be extended in future commits by other contributers.

6.2.1 What do you need?

The entire documentation is created using Sphinx with reStructuredText (rst: Wikipedia , Quick Ref), which is lightweight markup language. Basically you can write rst-files with any editor, I personally use the free Atom-Editor. If you want to create a new entry in the wiki for one of the models, simply create a new rst-file and start writing. In the case you want to extend/edit an existing model entry, simply edit the corresponding rst-file.

To compile the documentation and create the html output from the rst-files locally you further need Python installed with various packages. I highly recommend using Anaconda, which ships with a lot of useful packages. In general it's a good practise to have different environments of Python for different projects you are working on (Read this link for an introduction to Python environments) and Anaconda comes with its own way for organising environments. In the repository I added a rtd_environment.yml file, which creates a new environment for you with everything you need compile the documentation locally. Simply download the rtd_environment.yml file, then enter the terminal and enter:

```
conda env  create -f rtd_environment.yml
```

You can then activate the environment by entering:

```
# on linux and macOS
source activate docenv3

# on windows
activate docenv3
```

To leave an environment enter

```
# on linux and macOS
source deactivate

# on windows
deactivate
```

For further details on Anaconda environments see here.

After you have made changes to the documentation and you want to see the result as a html-page, set your terminal to `rrmpg/docs` and enter:

```
# on linux and macOS
make html

# on windows
make.bat html
```

If everything has compiled correctly you should find an `index.html` in `rrmpg/docs/build/html`.

6.2.2 Other options

Anyway, if this might seem to complicated for you, you can always send me your text by email (f.kratzert[at]gmail.com) or create an issue on GitHub and I'll do the rest.

6.2.3 Important note

This should be commonsense but I would like to remind you to cite every work of others (may it be publications, homepages, images etc.) you use in what ever you write.

6.3 Contribute to the code base

If you find any mistake/bug in the code or want to add new functionality to the code base, you should make sure that your code satisfies the following points.

1. Your code should follow the Google Python Style Guide and most importantly the docstrings (because the code documentation is autogenerated by the docstrings in the code). See the comments section for an example or look at the code of this repository.
2. The more you comment the better. Although the code should be selfexplaning at some points if you use good variable names, remember that also Python beginners might look at the code.
3. Add unittests for newly added functions/classes and make sure that all test are passed (the new, as well as the existing ones!).

6.3.1 Development of a new model

All models should inherit from the `BaseModel` defined in `models.basemodel`. By doing so, at lot of functionality is already added to your new model, without you having to write one line of code. See basemodel.py for all the functions that will be inherited. The skeleton of all implemented models should be equal (so that other functions, like `monte_carlo()`, can take any model as input) and look like this:

```python
from numba import njit
from .basemodel import BaseModel

class NewModel(BaseModel)
    """Model explanation comes here.

    Args:
        List of all input arguments comes here (mandatory and optional)

    """

    def __init__(self, params=None, **kwargs):
        """Docstring of __init__ function comes here.

        You should set the params input to None as default. By doing so,
        random parameters will be generated if no model parameters are
        passed during initialization. If the model has further mandatory
        inputs (like catchment area etc.) add them here.

        """
        super().__init__(params=params)

    def simulate(self, *args, **kwargs):
        """Docstring of simulate function comes here.

        Make sure to document all the inputs that are needed to run a
        simulation of your model.
        This function only validates and prepares all inputs and then calls
        a class extern model function, see below.

        """
        pass

    def fit(self, *args, **kwargs):
        """Docstring of fit function comes here.

        Make sure to document all the inputs that are needed to run this
        function.
        This function validates and prepares all inputs in a way, that we
        can use scipy.optimize.minimize to find an optimal parameter set.
        The loss function is defined externally (see below).

        """
        pass
```

(continues on next page)

(continued from previous page)

```
def _loss(X, *args):
    """Objective function used by the scipy optimizer.

    This function is used to calculate the model performance for a set X of
    parameters and must return a skalar. The optimizer tries to minimize
    this return value. For further explanation of how to build such a
    function read the scipy.optimizer.minimize documentation of look at the
    already implemented models.

    """

    pass


@njit
def _simulate(*args):
    """Here comes the real model simulation function.

    You have two options here:
    1. Already try to implement a numba optimized version of your model
    (add the @njit decorator).
    2. Or implement your model in pure python and I will afterwards
    optimize your function (remove @njit decorator.)

    """
    pass
```

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX