de Documentation

Release 0.1

Robin Schwemmle

Apr 02, 2020

CONTENTS:

CHAPTER

ONE

WELCOME TO DIAG-EFF'S DOCUMENTATION!

diag-eff is a package for diagnostic evaluation of hydrologic simulations. It includes one main module *de*. The module provides functions for calculating model performance and for diagnostics.

CHAPTER

TWO

HOW TO CITE

In case you use de in other software or scientific publications, please reference this module. It is published and has a DOI. It can be cited as:

> Robin Schwemmle, Dominic Demand & Markus Weiler (2020). diag-eff 0.1: Diagnostic efficiency – specific evaluation of model performance. (Version v0.1). Zenodo. http://doi.org/10.5281/zenodo...

2.1 Installation

The package can be installed directly from the Python Package Index or GitHub. The version on GitHub might be more recent, as only stable versions are uploaded to the Python Package Index.

2.1.1 PyPI

The version from PyPI can directly be installed using pip

```
pip install diag-eff
```

2.1.2 GitHub

The most recent version from GitHub can be installed like:

```
git clone https://github.com/schwemro/diag-eff.git
cd diag-eff
pip install -r requirements.txt
pip install -e .
```

2.1.3 Note

Depending on you OS, you might run into problems installing all requirements in a clean Python environment. These problems are usually caused by the scipy and numpy package, which might require to be compiled. Instead, We recommend to use an environment manager like anaconda. Then, the requirements can be installed like:

```
conda install numpy, scipy
```

2.2 Getting started

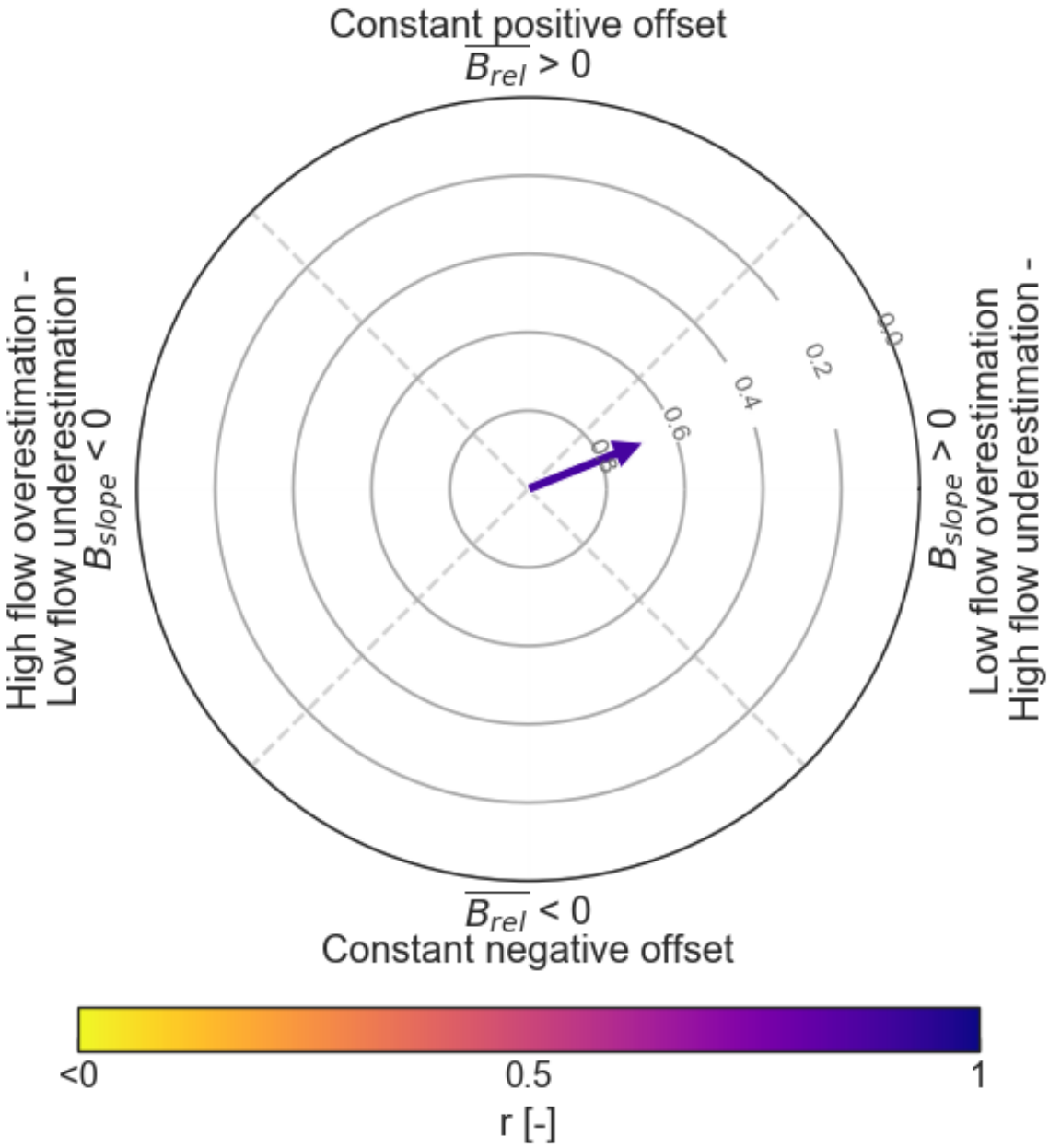
2.2.1 Diagnostic Efficiency

Load the package *de*. The calculation of the diagnostic efficiency can be easily demonstrated on the provided example dataset.

```
In [1]: from pathlib import Path  # OS-independent path handling

In [2]: from de import de

In [3]: from de import util

# path to example data
In [4]: path = Path('/Users/robinschwemmle/Desktop/PhD/diagnostic_efficiency/diag-eff/
↪examples/13331500_94_model_output.txt')

# import observed time series
In [5]: df_ts = util.import_camels_obs_sim(path)

# make numpy arrays
In [6]: obs_arr = df_ts['Qobs'].values

In [7]: sim_arr = df_ts['Qsim'].values

# calculate the diagnostic efficiency
In [8]: de.calc_de(obs_arr, sim_arr)
Out[8]: 0.6804705642526816
```

2.2.2 Diagnostic polar plot

```
In [9]: from pathlib import Path  # OS-independent path handling

In [10]: from de import de

In [11]: from de import util

# path to example data
In [12]: path = Path('/Users/robinschwemmle/Desktop/PhD/diagnostic_efficiency/diag-
↪eff/examples/13331500_94_model_output.txt')

# import observed time series
In [13]: df_ts = util.import_camels_obs_sim(path)

# make numpy arrays
In [14]: obs_arr = df_ts['Qobs'].values

In [15]: sim_arr = df_ts['Qsim'].values

# display diagnostic polar plots
In [16]: de.diag_polar_plot(obs_arr, sim_arr)
Out[16]: <Figure size 600x600 with 2 Axes>
```



2.3 Tutorials

The tutorials adopt section 3.1 and 3.2 from the paper.

2.3.1 Proof of concept

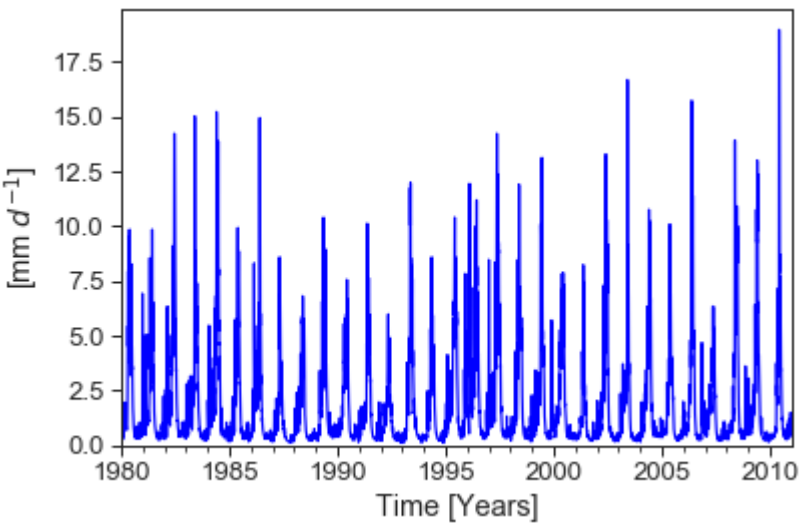
This notebook provides a proof of concept for Diagnostic Efficiency.

[4]:
```
import os
import sys
from pathlib import Path  # OS-independent path handling
os.chdir(Path('../..'))
PATH = os.getcwd()
sys.path.insert(0, PATH)

import pandas as pd
from de import de
from de import generate_errors
from de import kge
from de import nse
from de import util
from de import fdc

import warnings
warnings.filterwarnings('ignore')
```

Observed streamflow time series from CAMELS dataset

The observed and simulated streamflow time series are part of the open-source CAMELS dataset (Addor et al., 2017). The data can be downloaded from https://ncar.github.io/hydrology/datasets/CAMELS_timeseries.

[5]:
```
area = 619.11  # catchment area in km2 to convert runoff to mm/day
path = os.path.join(PATH, Path('examples/13331500_streamflow_qc.txt'))
df_ts = util.import_camels_ts(path, sep=r"\s+", catch_area=area) # import observed␣
↪time series
fig_ts = util.plot_ts(df_ts)
```

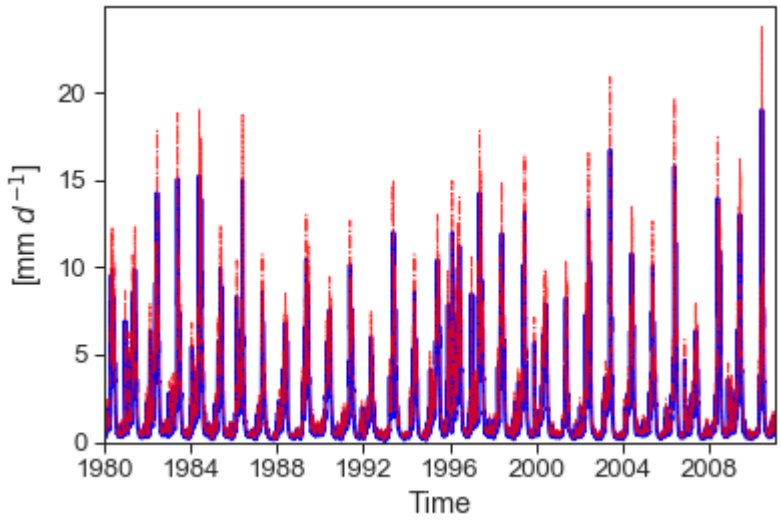


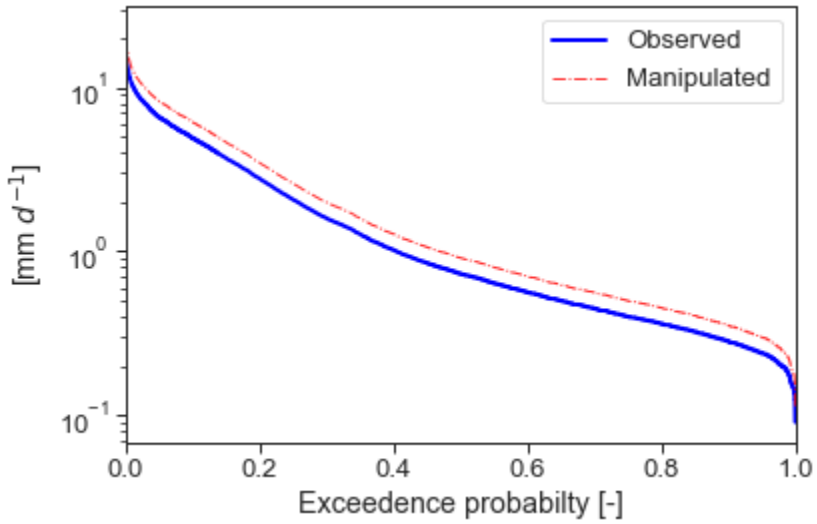
Mimicking errors

- constant error (e.g. caused by consistently overestimated precipitation)
- dynamic error (e.g. caused by storage routine)
- timing error (e.g. caused by model parameters)

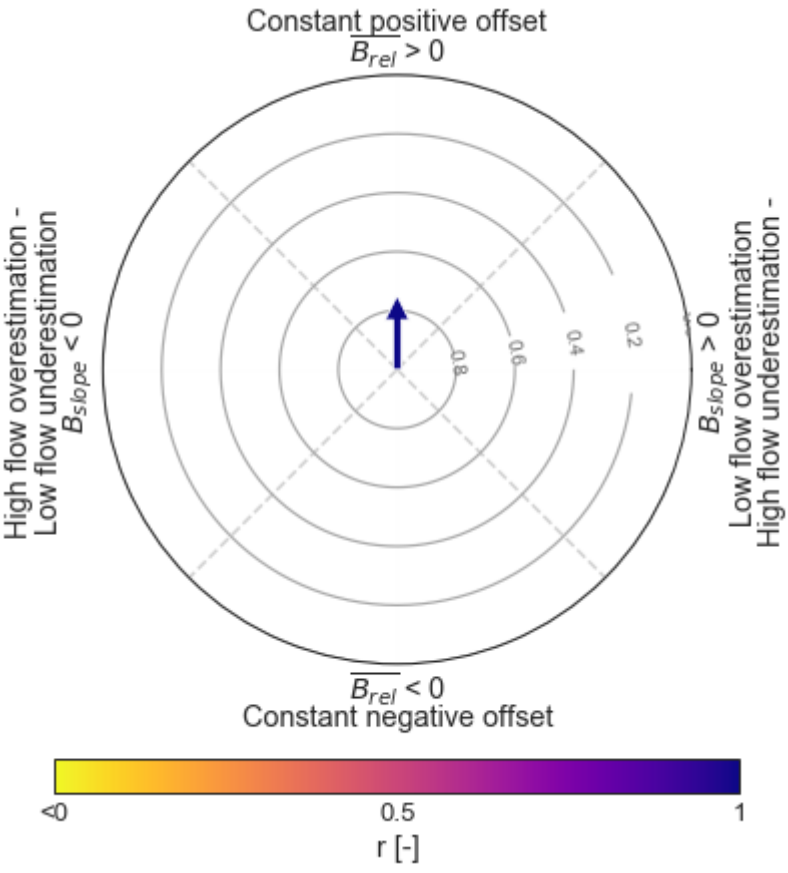
Positive constant error

[3]:
```
obs_sim = pd.DataFrame(index=df_ts.index, columns=['Qobs', 'Qsim'])
obs_sim.loc[:, 'Qobs'] = df_ts.loc[:, 'Qobs']
# generate positive constant error
obs_sim.loc[:, 'Qsim'] = generate_errors.constant(df_ts['Qobs'].values,
                                                  offset=1.25)

# plot time series
fig_ts = util.plot_obs_sim(obs_sim['Qobs'], obs_sim['Qsim'])
# plot flow duration curve
fig_fdc = fdc.fdc_obs_sim(obs_sim['Qobs'], obs_sim['Qsim'])
```



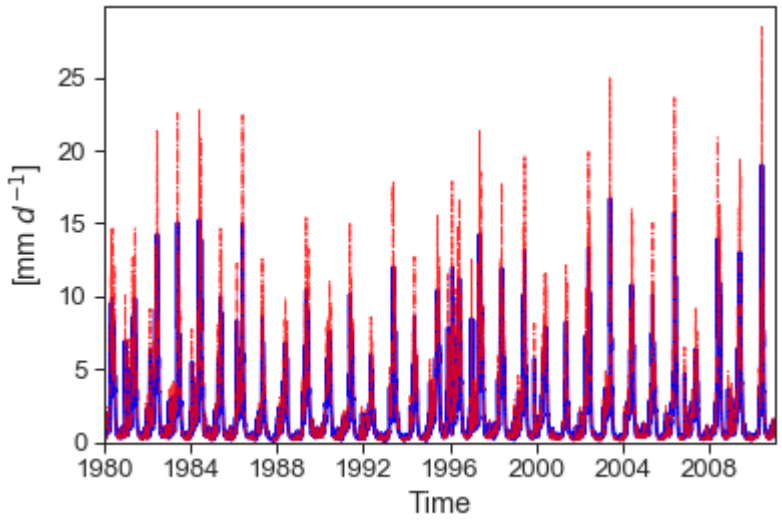


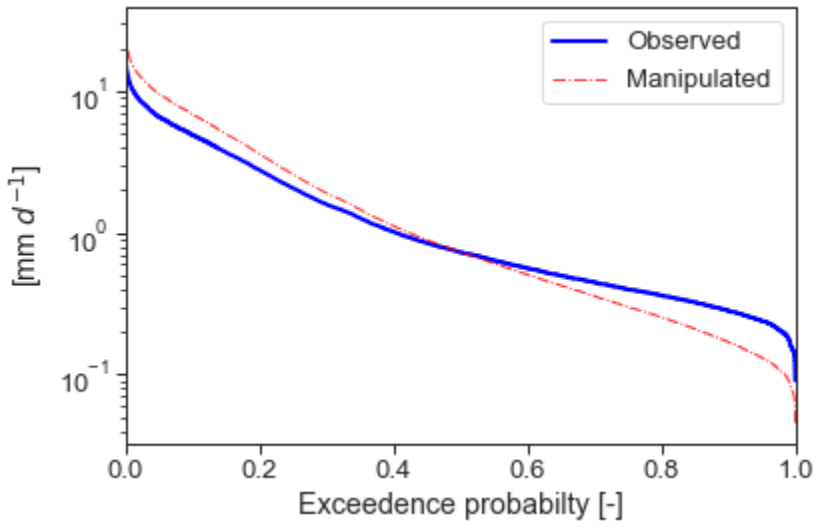
[4]:
```
# make arrays
obs_arr = obs_sim['Qobs'].values
sim_arr = obs_sim['Qsim'].values

# diagnostic polar plot
fig_dpp = de.diag_polar_plot(obs_arr, sim_arr)
```



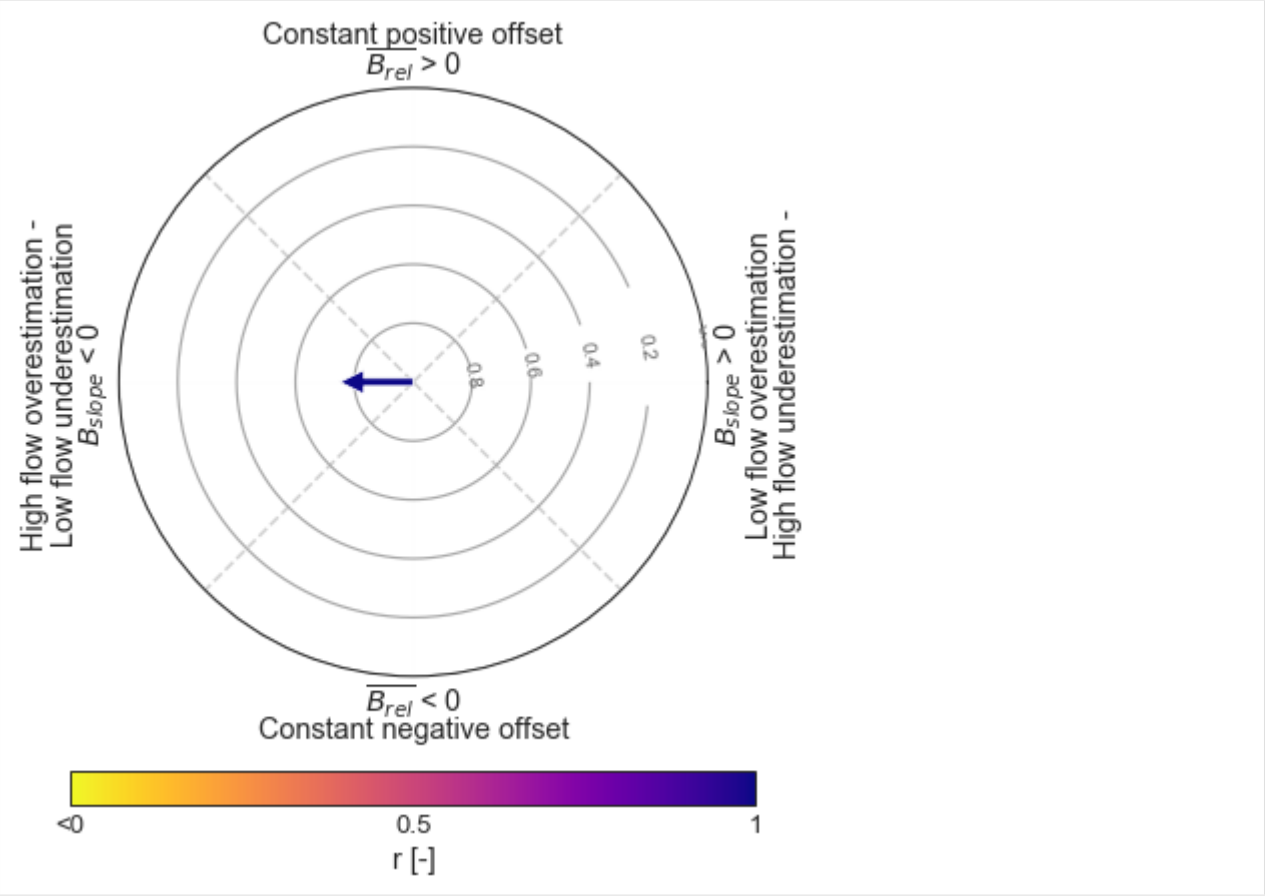
Positive dynamic error

[5]:
```
obs_sim = pd.DataFrame(index=df_ts.index, columns=['Qobs', 'Qsim'])
obs_sim.loc[:, 'Qobs'] = df_ts.loc[:, 'Qobs']
# generate positive dynamic error
tsd = generate_errors.positive_dynamic(df_ts.copy(), prop=0.5)
obs_sim.loc[:, 'Qsim'] = tsd.loc[:, 'Qsim']

# plot time series
fig_ts = util.plot_obs_sim(obs_sim['Qobs'], obs_sim['Qsim'])
# plot flow duration curve
fig_fdc = fdc.fdc_obs_sim(obs_sim['Qobs'], obs_sim['Qsim'])
```



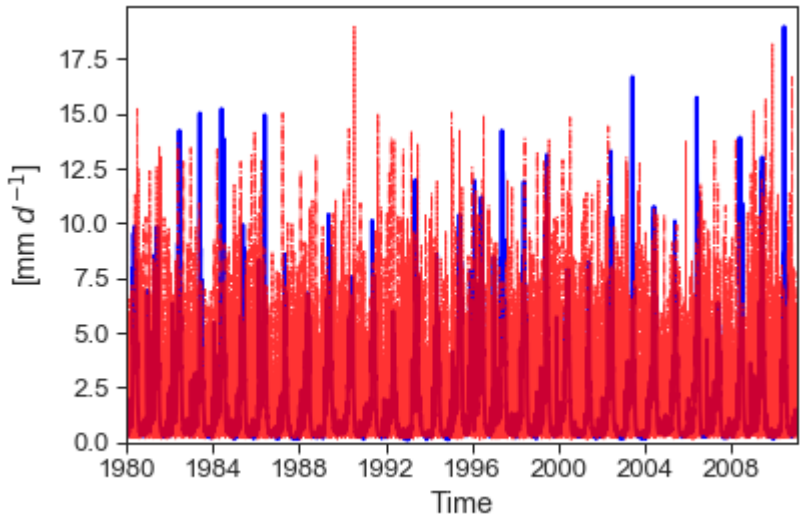


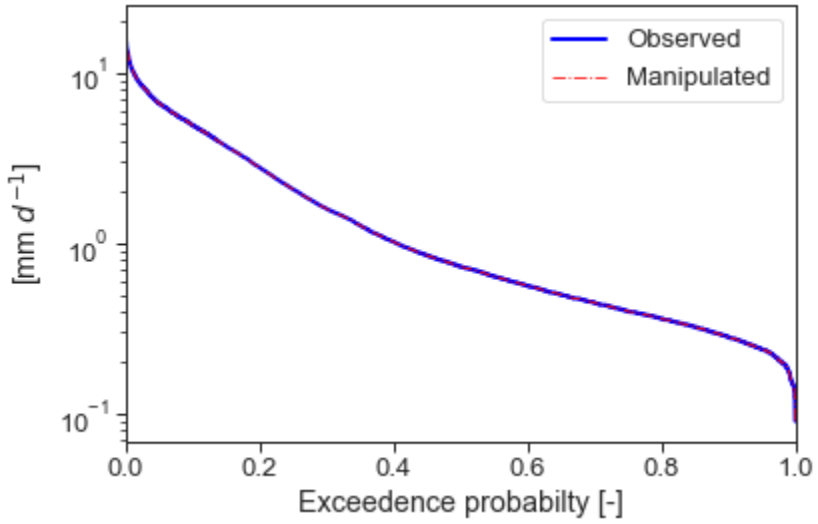
[6]:
```
# make arrays
obs_arr = obs_sim['Qobs'].values
sim_arr = obs_sim['Qsim'].values

# diagnostic polar plot
fig_dpp = de.diag_polar_plot(obs_arr, sim_arr)
```



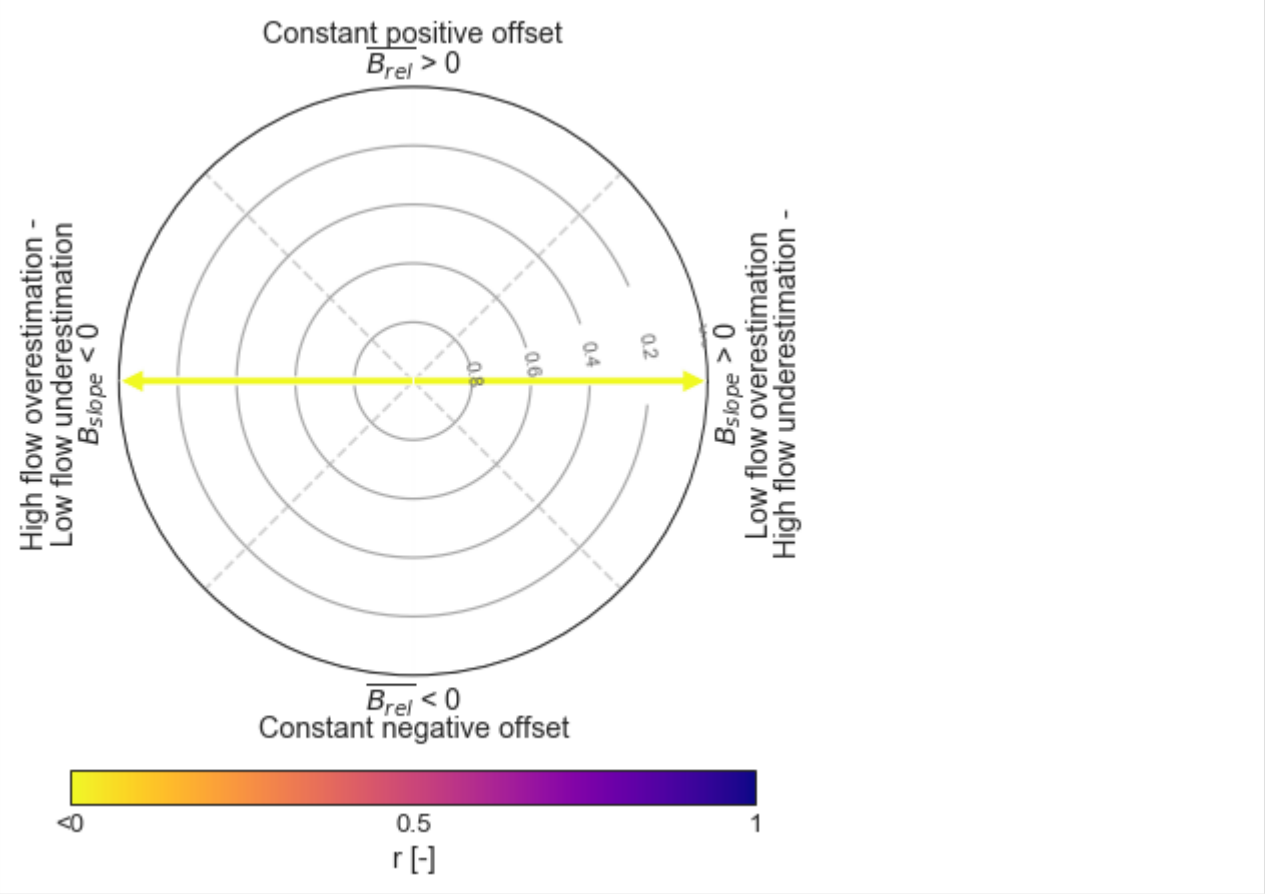
Timing error

[7]:
```
obs_sim = pd.DataFrame(index=df_ts.index, columns=['Qobs', 'Qsim'])
obs_sim.loc[:, 'Qobs'] = df_ts.loc[:, 'Qobs']
# generate timing error
tss = generate_errors.timing(df_ts.copy(), shuffle=True)  # shuffling
obs_sim.loc[:, 'Qsim'] = tss.iloc[:, 0].values

# plot time series
fig_ts = util.plot_obs_sim(obs_sim['Qobs'], obs_sim['Qsim'])
# plot flow duration curve
fig_fdc = fdc.fdc_obs_sim(obs_sim['Qobs'], obs_sim['Qsim'])
```





[8]:
```
# make arrays
obs_arr = obs_sim['Qobs'].values
sim_arr = obs_sim['Qsim'].values

# diagnostic polar plot
fig_dpp = de.diag_polar_plot(obs_arr, sim_arr)
```



In this case, dynamic error types cannot be distinguished from each other. Thus, the arrow extends in both directions.

References

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

2.3.2 Real case example

This notebook demonstrates the applicability of Diagnostic Efficiency to a real case example.

[17]:
```
import os
import sys
from pathlib import Path  # OS-independent path handling
os.chdir(Path('../..'))
PATH = os.getcwd()
sys.path.insert(0, PATH)

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from de import de
from de import kge
```

(continues on next page)

(continued from previous page)

```
from de import nse
from de import util

import warnings
warnings.filterwarnings('ignore')
```

Observed streamflow time series and simulated streamflow time series from CAMELS dataset

In order to demonstrate the applicability, we here use simulated streamflow time series which have been derived from Addor et al. (2017). Streamflow time series have been simulated by the coupled Snow-17 and SAC-SMA system for the same catchment as in Fig. 1. We briefly summarize here their modelling approach consisting of Snow-17 which "is a conceptual air- temperature-index snow accumulation and ablation model" (Newman et al., 2015) and SAC-SMA model which is "a conceptual hydrologic model that includes representation of physical processes such as evapotranspiration, percolation, surface flow, and subsurface lateral flow" (Newman et al., 2015). Snow-17 runs first to partition precipitation into rain and snow and delivers the input for SAC-SMA model. For further details about the modelling procedure we refer to section 3.1 in Newman et al. (2015). In particular, we evaluated three model runs with different parameter sets, but the same input data.

[4]:
```
path_cam1 = os.path.join(PATH, Path('examples/13331500_05_model_output.txt'))
path_cam2 = os.path.join(PATH, Path('examples/13331500_48_model_output.txt'))
path_cam3 = os.path.join(PATH, Path('examples/13331500_94_model_output.txt'))
df_cam1 = util.import_camels_obs_sim(path_cam1)
df_cam2 = util.import_camels_obs_sim(path_cam2)
df_cam3 = util.import_camels_obs_sim(path_cam3)
```

Plotting simulated streamflow time series and corresponding flow duration curves

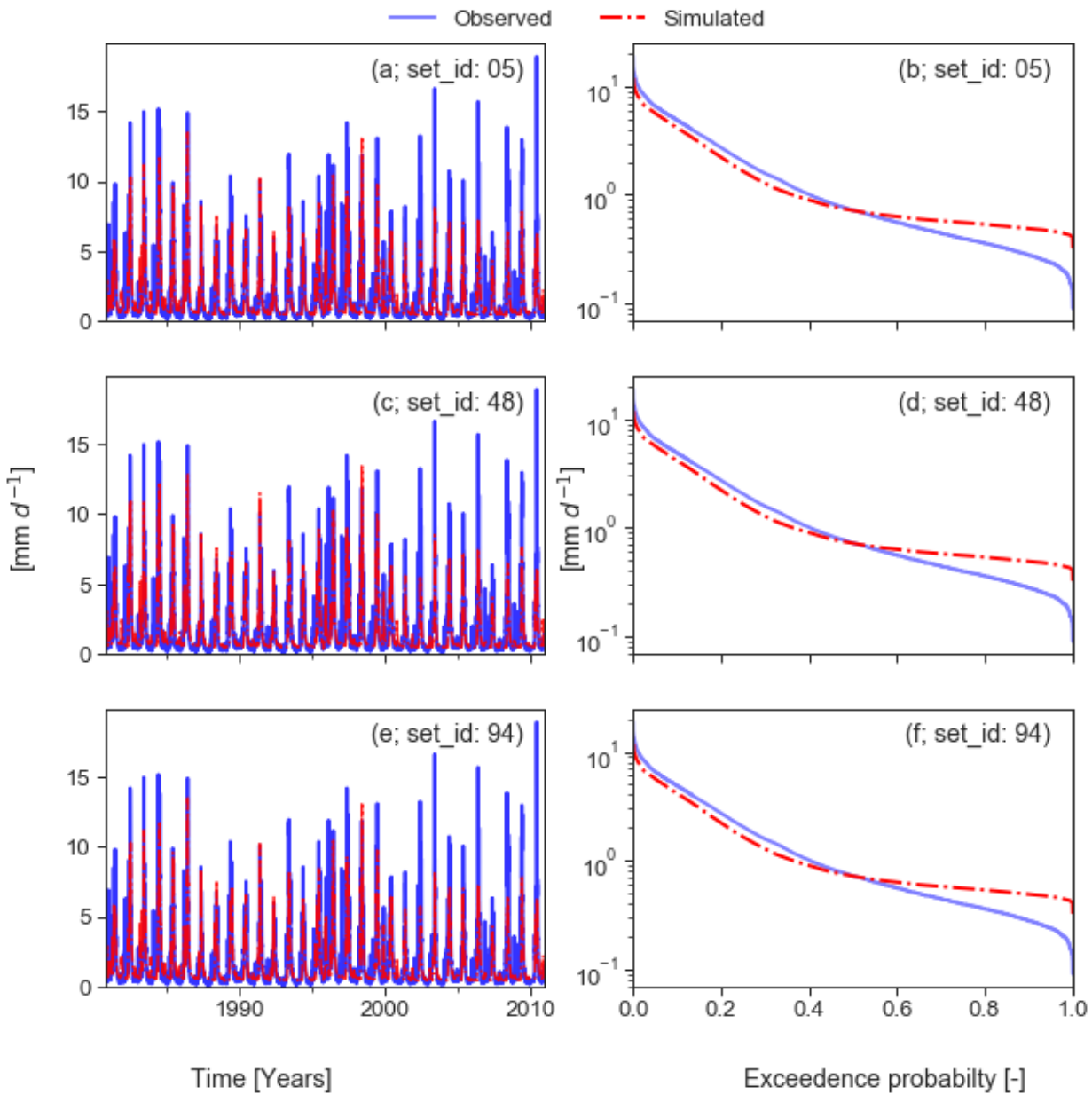
[13]:
```
idx = ['05', '48', '94']
fig, axes = plt.subplots(3, 2, figsize=(10, 10), sharex='col')
fig.text(0.06, 0.5, r'[mm $d^{-1}$]', ha='center', va='center',
         rotation='vertical')
fig.text(0.5, 0.5, r'[mm $d^{-1}$]', ha='center', va='center',
         rotation='vertical')
fig.text(0.25, 0.05, 'Time [Years]', ha='center', va='center')
fig.text(0.75, 0.05, 'Exceedence probabilty [-]', ha='center', va='center')

util.plot_obs_sim_ax(df_cam1['Qobs'], df_cam1['Qsim'], axes[0, 0], '')
axes[0, 0].text(.95, .95, '(a; set_id: {})'.format(idx[0]),
                transform=axes[0, 0].transAxes, ha='right', va='top')
# format the ticks
years_10 = mdates.YearLocator(10)
years_5 = mdates.YearLocator(5)
yearsFmt = mdates.DateFormatter('%Y')
axes[0, 0].xaxis.set_major_locator(years_10)
axes[0, 0].xaxis.set_major_formatter(yearsFmt)
axes[0, 0].xaxis.set_minor_locator(years_5)
util.fdc_obs_sim_ax(df_cam1['Qobs'], df_cam1['Qsim'], axes[0, 1], '')
axes[0, 1].text(.95, .95, '(b; set_id: {})'.format(idx[0]),
                transform=axes[0, 1].transAxes, ha='right', va='top')
# legend above plot
axes[0, 1].legend(loc=2, labels=['Observed', 'Simulated'], ncol=2,
                  frameon=False, bbox_to_anchor=(-0.6, 1.2))
```

(continues on next page)

(continued from previous page)

```
util.plot_obs_sim_ax(df_cam2['Qobs'], df_cam2['Qsim'], axes[1, 0], '')
axes[1, 0].text(.95, .95, '(c; set_id: {})'.format(idx[1]),
                transform=axes[1, 0].transAxes, ha='right', va='top')
# format the ticks
years_10 = mdates.YearLocator(10)
years_5 = mdates.YearLocator(5)
yearsFmt = mdates.DateFormatter('%Y')
axes[1, 0].xaxis.set_major_locator(years_10)
axes[1, 0].xaxis.set_major_formatter(yearsFmt)
axes[1, 0].xaxis.set_minor_locator(years_5)
util.fdc_obs_sim_ax(df_cam1['Qobs'], df_cam1['Qsim'], axes[1, 1], '')
axes[1, 1].text(.95, .95, '(d; set_id: {})'.format(idx[1]),
                transform=axes[1, 1].transAxes, ha='right', va='top')

util.plot_obs_sim_ax(df_cam1['Qobs'], df_cam1['Qsim'], axes[2, 0], '')
axes[2, 0].text(.95, .95, '(e; set_id: {})'.format(idx[2]),
                transform=axes[2, 0].transAxes, ha='right', va='top')
# format the ticks
years_10 = mdates.YearLocator(10)
years_5 = mdates.YearLocator(5)
yearsFmt = mdates.DateFormatter('%Y')
axes[2, 0].xaxis.set_major_locator(years_10)
axes[2, 0].xaxis.set_major_formatter(yearsFmt)
axes[2, 0].xaxis.set_minor_locator(years_5)
util.fdc_obs_sim_ax(df_cam1['Qobs'], df_cam1['Qsim'], axes[2, 1], '')
axes[2, 1].text(.95, .95, '(f; set_id: {})'.format(idx[2]),
                transform=axes[2, 1].transAxes, ha='right', va='top')
```

[13]:
```
Text(0.95, 0.95, '(f; set_id: 94)')
```



Evaluation of model performance

[21]:
```
# create dataframe for comparison of DE, KGE and NSE
idx = ['05', '48', '94']
cols = ['brel_mean', 'b_area', 'temp_cor', 'de', 'b_dir', 'b_slope',
        'phi', 'beta', 'alpha', 'kge', 'nse']
df_eff_cam = pd.DataFrame(index=idx, columns=cols, dtype=np.float64)
```

Calculation of DE, KGE and NSE

[22]:
```
# make arrays
obs_arr = df_cam1['Qobs'].values
sim_arr = df_cam1['Qsim'].values

# mean relative bias
brel_mean = de.calc_brel_mean(obs_arr, sim_arr)
df_eff_cam.iloc[0, 0] = brel_mean
# residual relative bias
brel_res = de.calc_brel_res(obs_arr, sim_arr)
# area of relative remaing bias
b_area = de.calc_bias_area(brel_res)
df_eff_cam.iloc[0, 1] = b_area
# temporal correlation
temp_cor = de.calc_temp_cor(obs_arr, sim_arr)
df_eff_cam.iloc[0, 2] = temp_cor
# diagnostic efficiency
df_eff_cam.iloc[0, 3] = de.calc_de(obs_arr, sim_arr)
# direction of bias
b_dir = de.calc_bias_dir(brel_res)
df_eff_cam.iloc[0, 4] = b_dir
# slope of bias
b_slope = de.calc_bias_slope(b_area, b_dir)
df_eff_cam.iloc[0, 5] = b_slope
# convert to radians
# (y, x) Trigonometric inverse tangent
df_eff_cam.iloc[0, 6] = np.arctan2(brel_mean, b_slope)

# KGE beta
df_eff_cam.iloc[0, 7] = kge.calc_kge_beta(obs_arr, sim_arr)
# KGE alpha
df_eff_cam.iloc[0, 8] = kge.calc_kge_alpha(obs_arr, sim_arr)
# KGE
df_eff_cam.iloc[0, 9] = kge.calc_kge(obs_arr, sim_arr)

# NSE
df_eff_cam.iloc[0, 10] = nse.calc_nse(obs_arr, sim_arr)

# make arrays
obs_arr = df_cam2['Qobs'].values
sim_arr = df_cam2['Qsim'].values

# mean relative bias
brel_mean = de.calc_brel_mean(obs_arr, sim_arr)
df_eff_cam.iloc[1, 0] = brel_mean
# residual relative bias
brel_res = de.calc_brel_res(obs_arr, sim_arr)
# area of relative remaing bias
b_area = de.calc_bias_area(brel_res)
df_eff_cam.iloc[1, 1] = b_area
# temporal correlation
temp_cor = de.calc_temp_cor(obs_arr, sim_arr)
df_eff_cam.iloc[1, 2] = temp_cor
# diagnostic efficiency
df_eff_cam.iloc[1, 3] = de.calc_de(obs_arr, sim_arr)
# direction of bias
```

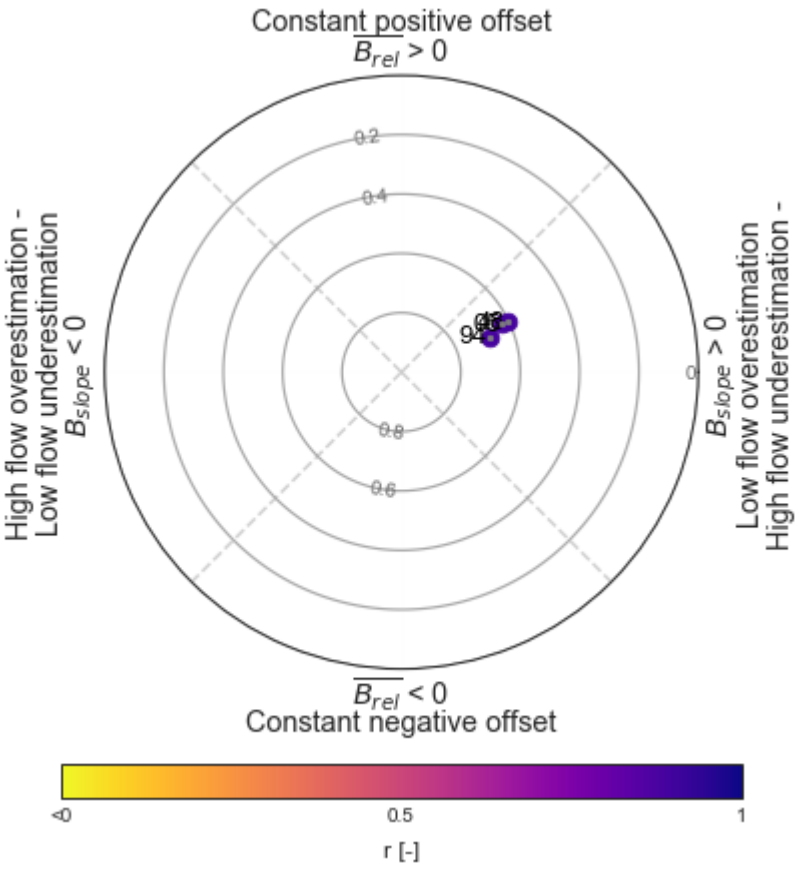
(continues on next page)

(continued from previous page)

```
b_dir = de.calc_bias_dir(brel_res)
df_eff_cam.iloc[1, 4] = b_dir
# slope of bias
b_slope = de.calc_bias_slope(b_area, b_dir)
df_eff_cam.iloc[1, 5] = b_slope
# convert to radians
# (y, x) Trigonometric inverse tangent
df_eff_cam.iloc[1, 6] = np.arctan2(brel_mean, b_slope)

# KGE beta
df_eff_cam.iloc[1, 7] = kge.calc_kge_beta(obs_arr, sim_arr)
# KGE alpha
df_eff_cam.iloc[1, 8] = kge.calc_kge_alpha(obs_arr, sim_arr)
# KGE
df_eff_cam.iloc[1, 9] = kge.calc_kge(obs_arr, sim_arr)

# NSE
df_eff_cam.iloc[1, 10] = nse.calc_nse(obs_arr, sim_arr)

# make arrays
obs_arr = df_cam3['Qobs'].values
sim_arr = df_cam3['Qsim'].values

# mean relative bias
brel_mean = de.calc_brel_mean(obs_arr, sim_arr)
df_eff_cam.iloc[2, 0] = brel_mean
# residual relative bias
brel_res = de.calc_brel_res(obs_arr, sim_arr)
# area of relative remaing bias
b_area = de.calc_bias_area(brel_res)
df_eff_cam.iloc[2, 1] = b_area
# temporal correlation
temp_cor = de.calc_temp_cor(obs_arr, sim_arr)
df_eff_cam.iloc[2, 2] = temp_cor
# diagnostic efficiency
df_eff_cam.iloc[2, 3] = de.calc_de(obs_arr, sim_arr)
# direction of bias
b_dir = de.calc_bias_dir(brel_res)
df_eff_cam.iloc[2, 4] = b_dir
# slope of bias
b_slope = de.calc_bias_slope(b_area, b_dir)
df_eff_cam.iloc[2, 5] = b_slope
# convert to radians
# (y, x) Trigonometric inverse tangent
df_eff_cam.iloc[2, 6] = np.arctan2(brel_mean, b_slope)

# KGE beta
df_eff_cam.iloc[2, 7] = kge.calc_kge_beta(obs_arr, sim_arr)
# KGE alpha
df_eff_cam.iloc[2, 8] = kge.calc_kge_alpha(obs_arr, sim_arr)
# KGE
df_eff_cam.iloc[2, 9] = kge.calc_kge(obs_arr, sim_arr)

# NSE
df_eff_cam.iloc[2, 10] = nse.calc_nse(obs_arr, sim_arr)
```

Diagnostic polar plot

[23]:
```
brel_mean_arr = df_eff_cam['brel_mean'].values
b_area_arr = df_eff_cam['b_area'].values
temp_cor_arr = df_eff_cam['temp_cor'].values
b_dir_arr = df_eff_cam['b_dir'].values
de_arr = df_eff_cam['de'].values
phi_arr = df_eff_cam['phi'].values
b_slope_arr = df_eff_cam['b_slope'].values

fig_de = util.diag_polar_plot_multi_fc(brel_mean_arr, b_area_arr,
                                       temp_cor_arr, de_arr, b_dir_arr,
                                       phi_arr, idx, ax_lim=0)
```



Simulations realised by parameter set with set_id 94 outperform the other parameter sets. All simulations have in common, that positive dynamic error type (i.e. high flows are underestimated and low flows are overestimated) dominates accompanied by a slight positive constant error. Timing contributes least to the error.

After identifying the error types and its contributions, we can infer hints on how to improve the simulations. From a process- based (perceptual) perspective, the apparent negative dynamic error described by high flow underestimation and low flow overestimation suggest that process realism (e.g. snow melt, infiltration, storage outflow) appears to be deficient. Measures for improvement could start with adjusting the model parameters (e.g. refining the calibration procedure). If necessary, a follow-up measure could be to alter the model structure (e.g. adjusting the model equations). Additionally, there is positive constant error available. Because a constant error may be linked to input data errors, this implies that adjusting the input data (e.g. precipitation correction) might amend the simulations.

References

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

Newman, A. J., Clark, M. P., Sampson, K., Wood, A., Hay, L. E., Bock, A., Viger, R. J., Blodgett, D., Brekke, L., Arnold, J. R., Hopson, T., and Duan, Q.: Development of a large-sample watershed-scale hydrometeorological data set for the contiguous USA: data set characteristics and assessment of regional variability in hydrologic model performance, Hydrol. Earth Syst. Sci., 19, 209-223, 10.5194/hess-19-209-2015, 2015.

2.4 Code Reference

2.4.1 de

Diagnostic Efficiency

de.de.**calc_de**(*obs*, *sim*, *sort=True*)

Calculate Diagnostic-Efficiency (DE).

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series
- **sort** (*boolean, optional*) – If True, time series are sorted by ascending order. If False, time series are not sorted. The default is to sort.

Returns **eff** – non-normalized Diagnostic efficiency

Return type float

Notes

$$DE = 1 - \sqrt{\overline{B_{rel}}^2 + |B_{area}|^2 + (r-1)^2}$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> de.calc_de(obs, sim)
0.8202204384691575
```

Constant error term

de.de.**calc_brel_mean**(*obs*, *sim*, *sort=True*)

Calculate arithmetic mean of relative bias.

Parameters

- **obs** (*(N,) array_like*) – observed time series as 1-D array
- **sim** (*(N,) array_like*) – simulated time series
- **sort** (*boolean, optional*) – If True, time series are sorted by ascending order. If False, time series are not sorted. The default is to sort.

Returns **brel_mean** – average relative bias

Return type float

Notes

$$\overline{B_{rel}} = \frac{1}{N} \sum_{i=1}^{N} B_{rel}(i)$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> de.calc_brel_mean(obs, sim)
0.09330065359477124
```

Dynamic error term

de.de.**calc_bias_area**(*brel_rest*)

Calculate absolute bias area for entire flow duration curve.

Parameters **brel_rest** (*(N,) array_like*) – remaining relative bias as 1-D array

Returns **b_area** – bias area

Return type float

Notes

$$|B_{area}| = \int_0^1 |B_{rest}(i)| di$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> b_rest = de.calc_brel_rest(obs, sim)
>>> de.calc_bias_area(b_rest)
0.1112908496732026
```

Timing error term

de.de.**calc_temp_cor**(*obs*, *sim*, *r='pearson'*)

Calculate temporal correlation between observed and simulated time series.

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series
- **r** (*str, optional*) – Either Spearman correlation coefficient ('spearman') or Pearson correlation coefficient ('pearson') can be used to describe the temporal correlation. The default is to calculate the Pearson correlation.

Returns **temp_cor** – correlation between observed and simulated time series

Return type float

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> de.calc_temp_cor(obs, sim)
0.8940281850583509
```

Diagnostic Polar Plot

de.de.**diag_polar_plot**(*obs*, *sim*, *sort=True*, *l=0.05*, *extended=False*)

Diagnostic polar plot of Diagnostic efficiency (DE) for a single evaluation.

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series as 1-D array
- **sort** (*boolean, optional*) – If True, time series are sorted by ascending order. If False, time series are not sorted. The default is to sort.
- **l** (*float, optional*) – Threshold for which diagnosis can be made. The default is 0.05.

- **extended** (*boolean, optional*) – If True, extended diagnostic plot is displayed. In addtion, the duration curve of B_rest is plotted besides the polar plot. The default is, that only the diagnostic polar plot is displayed.

Returns **fig** – Returns a single figure if extended=False and two figures if extended=True.

Return type Figure

Notes

$$\varphi = arctan2(\overline{B_{rel}}, B_{slope})$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> de.diag_polar_plot(obs, sim)
```

2.4.2 generate_errors

Constant error

de.generate_errors.**constant**(*ts*, *offset=1.5*)

Generate constant errors.

Constant errors are generated by multiplying with either constant positive offset or constant negative offset.

Parameters

- **ts** (*(N,)array_like*) – Observed time series
- **offset** (*float, optional*) – Offset multiplied to time series. If greater than 1 positive constant offset and if less than 1 negative constant offset. The default is 1.5.

Returns **ts_const** – Time series with constant error

Return type array_like

Dynamic error

de.generate_errors.**positive_dynamic**(*ts*, *prop=0.5*)

Generate positive dynamic errors (i.e. Overestimate high flows - Underestimate low flows)

High to medium flows are increased by linear decreasing factors. Medium to low flows are decreased by linear decreasing factors.

Parameters

- **ts** (*dataframe*) – Dataframe with time series
- **prop** (*float, optional*) – Factor by which time series is tilted.

Returns **ts_dyn** – Time series with positive dynamic error

Return type dataframe

de.generate_errors.**negative_dynamic**(*ts*, *prop=0.5*)

Generate negative dynamic error (i.e Underestimate high flows - Overestimate low flows)

High to medium flows are decreased by linear increasing factors. Medium to low flows are increased by linear increasing factors.

Parameters

- **ts** (*dataframe*) – Observed time series
- **prop** (*float, optional*) – Factor by which time series is tilted.

Returns **ts_dyn** – Time series with negative dynamic error

Return type dataframe

Timing error

de.generate_errors.**timing**(*ts*, *tshift=3*, *shuffle=True*)

Mimicking timing errors.

Timing errors are generated by either shifting or shuffling.

Parameters

- **ts** (*dataframe*) – dataframe with time series
- **tshift** (*int, optional*) – days by which time series is shifted. Both positive and negative time shift are possible. The default is 3 days.
- **shuffle** (*boolean, optional*) – If True, time series is shuffled. The default is shuffling.

Returns **ts_tim** – Time series with timing error

Return type dataframe

2.4.3 kge

Kling-Gupta Efficiency

de.kge.**calc_kge**(*obs*, *sim*, *r='pearson'*, *var='std'*)

Calculate Kling-Gupta-Efficiency (KGE).

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series as 1-D array
- **r** (*str, optional*) – Either Spearman correlation coefficient ('spearman'; Pool et al. 2018) or Pearson correlation coefficient ('pearson'; Gupta et al. 2009) can be used to describe the temporal correlation. The default is to calculate the Pearson correlation.
- **var** (*str, optional*) – Either coefficient of variation ('cv'; Kling et al. 2012) or standard deviation ('std'; Gupta et al. 2009, Pool et al. 2018) to describe the gamma term. The default is to calculate the standard deviation.

Returns **eff** – Kling-Gupta-Efficiency measure

Return type float

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> kge.calc_kge(obs, sim)
0.683901305466148
```

Notes

$$KGE = 1 - \sqrt{(\beta - 1)^2 + (\alpha - 1)^2 + (r - 1)^2}$$

$$KGE = 1 - \sqrt{(\frac{\mu_{sim}}{\mu_{obs}} - 1)^2 + (\frac{\sigma_{sim}}{\sigma_{obs}} - 1)^2 + (r - 1)^2}$$

$$KGE = 1 - \sqrt{(\beta - 1)^2 + (\gamma - 1)^2 + (r - 1)^2}$$

$$KGE = 1 - \sqrt{(\frac{\mu_{sim}}{\mu_{obs}} - 1)^2 + (\frac{CV_{sim}}{CV_{obs}} - 1)^2 + (r - 1)^2}$$

References

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, Journal of Hydrology, 377, 80-91, 10.1016/j.jhydrol.2009.08.003, 2009.

Kling, H., Fuchs, M., and Paulin, M.: Runoff conditions in the upper Danube basin under an ensemble of climate change scenarios, Journal of Hydrology, 424-425, 264-277, 10.1016/j.jhydrol.2012.01.011, 2012.

Pool, S., Vis, M., and Seibert, J.: Evaluating model performance: towards a non-parametric variant of the Kling-Gupta efficiency, Hydrological Sciences Journal, 63, 1941-1953, 10.1080/02626667.2018.1552002, 2018.

Bias term

de.kge.**calc_kge_beta**(*obs*, *sim*)

Calculate the beta term of Kling-Gupta-Efficiency (KGE).

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series as 1-D array

Returns **kge_beta** – alpha value

Return type float

Notes

$$\beta = \frac{\mu_{sim}}{\mu_{obs}}$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> de.calc_kge_beta(obs, sim)
1.0980392156862746
```

References

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, Journal of Hydrology, 377, 80-91, 10.1016/j.jhydrol.2009.08.003, 2009.

Kling, H., Fuchs, M., and Paulin, M.: Runoff conditions in the upper Danube basin under an ensemble of climate change scenarios, Journal of Hydrology, 424-425, 264-277, 10.1016/j.jhydrol.2012.01.011, 2012.

Pool, S., Vis, M., and Seibert, J.: Evaluating model performance: towards a non-parametric variant of the Kling-Gupta efficiency, Hydrological Sciences Journal, 63, 1941-1953, 10.1080/02626667.2018.1552002, 2018.

Variability term

de.kge.**calc_kge_alpha**(*obs*, *sim*)

Calculate the alpha term of the Kling-Gupta-Efficiency (KGE).

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series

Returns **kge_alpha** – alpha value

Return type float

Notes

$$\alpha = \frac{\sigma_{sim}}{\sigma_{obs}}$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> kge.calc_kge_alpha(obs, sim)
1.2812057455166919
```

References

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, Journal of Hydrology, 377, 80-91, 10.1016/j.jhydrol.2009.08.003, 2009.

Kling, H., Fuchs, M., and Paulin, M.: Runoff conditions in the upper Danube basin under an ensemble of climate change scenarios, Journal of Hydrology, 424-425, 264-277, 10.1016/j.jhydrol.2012.01.011, 2012.

Pool, S., Vis, M., and Seibert, J.: Evaluating model performance: towards a non-parametric variant of the Kling-Gupta efficiency, Hydrological Sciences Journal, 63, 1941-1953, 10.1080/02626667.2018.1552002, 2018.

de.kge.**calc_kge_gamma**(*obs*, *sim*)

Calculate the gamma term of Kling-Gupta-Efficiency (KGE).

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series as 1-D array

Returns **kge_gamma** – gamma value

Return type float

Notes

$$\gamma = \frac{CV_{sim}}{CV_{obs}}$$

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> kge.calc_kge_gamma(obs, sim)
1.166812375381273
```

References

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, Journal of Hydrology, 377, 80-91, 10.1016/j.jhydrol.2009.08.003, 2009.

Kling, H., Fuchs, M., and Paulin, M.: Runoff conditions in the upper Danube basin under an ensemble of climate change scenarios, Journal of Hydrology, 424-425, 264-277, 10.1016/j.jhydrol.2012.01.011, 2012.

Pool, S., Vis, M., and Seibert, J.: Evaluating model performance: towards a non-parametric variant of the Kling-Gupta efficiency, Hydrological Sciences Journal, 63, 1941-1953, 10.1080/02626667.2018.1552002, 2018.

Correlation term

de.kge.**calc_temp_cor**(*obs*, *sim*, *r='pearson'*)

Calculate temporal correlation between observed and simulated time series.

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series
- **r** (*str, optional*) – Either Spearman correlation coefficient ('spearman') or Pearson correlation coefficient ('pearson') can be used to describe the temporalcorrelation. The default is to calculate the Pearson correlation.

Returns **temp_cor** – correlation between observed and simulated time series

Return type float

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> de.calc_temp_cor(obs, sim)
0.8940281850583509
```

Polar Plot

de.kge.**polar_plot**(*obs*, *sim*, *r='pearson'*, *var='std'*)

Polar plot of Diagnostic efficiency (KGE) for a single evaluation.

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series as 1-D array
- **r** (*str, optional*) – Either Spearman correlation coefficient ('spearman') or Pearson correlation coefficient ('pearson') can be used to describe the temporal correlation. The default calculates the Pearson correlation.
- **var** (*str, optional*) – Either coefficient of variation ('cv') or standard deviation ('std') to describe the gamma term. The default calculates the standard deviation.

Returns **fig** – diagnostic polar plot

Return type Figure

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> kge.diag_polar_plot_kge(obs, sim)
```

2.4.4 nse

Nash-Sutcliffe Efficiency

de.nse.**calc_nse**(*obs*, *sim*)

Calculate Nash-Sutcliffe-Efficiency (NSE).

Parameters

- **obs** (*(N,) array_like*) – Observed time series as 1-D array
- **sim** (*(N,) array_like*) – Simulated time series as 1-D array

Returns **sig** – Nash-Sutcliffe-Efficiency measure

Return type float

Examples

Provide arrays with equal length

```
>>> from de import de
>>> import numpy as np
>>> obs = np.array([1.5, 1, 0.8, 0.85, 1.5, 2])
>>> sim = np.array([1.6, 1.3, 1, 0.8, 1.2, 2.5])
>>> nse.calc_nse(obs, sim)
0.5648252536640361
```

Notes

$$NSE = 1 - \frac{\sum_{t=1}^{t=T}(Q_{sim}(t) - Q_{obs}(t))^2}{\sum_{t=1}^{t=T}(Q_{obs}(t) - \overline{Q_{obs}})^2}$$

References

Nash, J. E., and Sutcliffe, J. V.: River flow forecasting through conceptual models part I - A discussion of principles, Journal of Hydrology, 10, 282-290, 10.1016/0022-1694(70)90255-6, 1970.

2.5 Changelog

2.5.1 Version 0.1

INDEX