pastastore Documentation

Release 1.4.0

D.A. Brakenhoff

Mar 22, 2024

CONTENTS

pastastore is a module for managing Pastas time series and models.

The module supports storing and managing data with a database or on disk. This gives the user a simple way to manage Pastas projects, and allows the user to pick up where they left off, without having to load all data into memory.

An small example using pastastore:

```
import pastastore as pst
import pandas as pd

# initialize a connector and a pastastore
pstore = pst.PastaStore("my_store", pst.PasConnector("my_dbase", "./path_to_folder"))

# read some data
series = pd.read_csv("some.csv", index_col=[0], parse_dates=True)

# add data to store
pstore.add_oseries(series, "my_oseries", metadata={"x": 10., "y": 20.})
```

See the table of contents to get started with *pastastore*.

CHAPTER

ONE

GETTING STARTED

On this page you will find all the information to get started with *pastastore*.

1.1 Getting Python

To install *pastastore*, a working version of Python 3.7 or higher has to be installed on your computer. We recommend using the Anaconda Distribution of Python.

1.2 Installing *pastastore*

Install the module by typing:

```
pip install pastastore
```

For installing in development mode, clone the repository and install by typing *pip install -e .* from the module root directory.

1.3 Using *pastastore*

Start Python and import the module:

```
import pastastore as pst
conn = pst.DictConnector("my_connector")
store = pst.PastaStore(conn)
```

See the *Examples* section for some quick examples on how to get started.

1.4 External dependencies

This module has several external optional dependencies. These are required for storing time series and models in external databases using compression. Without these, only the in-memory option (DictConnector) and storing data on disk without compression (PasConnector) are available, which should be good enough for most users.

If you do need more performance of compression, both the *PystoreConnector* and *ArcticConnector* are dependent on external software.

- **Using Pystore requires Snappy:**
  - Snappy is a fast and efficient compression/decompression library from Google. You'll need to install Snappy on your system before installing PyStore. See links for installation instructions here: https://github.com/ranaroussi/pystore#dependencies
  - After installing Snappy, install pystore by typing *pip install pystore* in a terminal.
- **Using Arctic requires MongoDB:**
  - The recommended method of obtaining MongoDB is using Docker. The instructions for this are shown below.
  - Alternatively, get MongoDB by installing the Community edition (Windows, MacOS).
  - Additionally, install arctic by typing *pip install git+https://github.com/man-group/arctic.git* in a terminal. Note that the current version of arctic is not compatible with pandas>1.1, and a version will have to be installed that fixes that. Install arctic from a Pull Request that fixes this particular issue to bypass this problem. If you need help, feel free to get in touch.

1.4.1 Running MongoDB from docker

Follow these steps to get the MongoDB docker container up and running using *docker-compose*:

1. Install Docker (i.e. Docker Desktop)
2. Open a terminal and navigate to *./dockerfiles*.
3. Run *docker-compose up -d* to start the docker container running MongoDB. The *-d* flag runs the container in the background. This command uses the *docker-compose.yml* file by default.
4. View your running containers with *docker ps -a*.
5. If you are done and wish to stop the container, run *docker-compose stop* in a terminal.

CHAPTER

TWO

EXAMPLES

This page provides some short examples and example applications in Jupyter Notebooks. The following snippets show typical usage.

The general idea is to first define the connector object. This object manages the communication between the user and the data store. The the next step is to pass that connector to the *PastaStore* to access all of the useful methods for building time series models.

2.1 In-memory

The following snippet shows how to use PastaStore with in-memory storage of time series and models. This is the simplest implementation because everything is stored in-memory (in dictionaries):

```
import pastastore as pst

# define dict connector
conn = pst.DictConnect("my_db")

# create project for managing Pastas data and models
store = pst.PastaStore(conn)
```

2.2 Using Pastas

The following snippet shows how to use PastaStore with storage of time series and models on disk as pas-files. This is the simplest implementation that writes data to disk as no external dependencies are required:

```
import pastastore as pst

# define pas connector
path = "./data/pastas_db"
conn = pst.PasConnector("my_db", path)

# create project for managing Pastas data and models
store = pst.PastaStore(conn)
```

2.3 Using ArcticDB

The following snippet shows how to create an *ArcticDBConnector* and initialize a *PastaStore* object:

```
import pastastore as pst

# define arctic connector
uri = "lmdb://./my_path_here/"
conn = pst.ArcticDBConnector("my_db", uri)

# create project for managing Pastas data and models
store = pst.PastaStore(conn)
```

2.4 Using Arctic

The following snippet shows how to create an *ArcticConnector* and initialize a *PastaStore* object. Please note that a MongoDB instance must be running for this to work:

```
import pastastore as pst

# define arctic connector
connstr = "mongodb://localhost:27017/"
conn = pst.ArcticConnector("my_db", connstr)

# create project for managing Pastas data and models
store = pst.PastaStore(conn)
```

2.5 Using Pystore

To use the *PystoreConnector* the steps are identical, the only difference is that the user has to provide a path to a location on disk instead of a connection string to a database:

```
# define pystore connector
path = "./data/pystore"
conn = pst.PystoreConnector("my_db", path)

# create project for managing Pastas data and models
store = pst.PastasProject(conn)
```

2.6 The PastaStore object

The *PastaStore* object provides useful methods e.g. for creating models and determining which time series are closest to one another. The database read/write/delete methods can be accessed directly from the *PastaStore* object:

```
# create a new time series
series = pd.Series(index=pd.date_range("2019", "2020", freq="D"), data=1.0)

# add an observation time series
store.add_oseries(series, "my_oseries", metadata={"x": 100, "y": 200})

# retrieve the oseries
oseries = store.get_oseries("my_oseries")
```

To create a Pastas time series model use *store.create_model()*. Note that this does not automatically add the model to the database. To store the model, it has to be explicitly added to the database:

```
# create a time series model
ml = store.create_model("my_oseries", add_recharge=False)

# add to the database
store.add_model(ml)

# retrieve model from database
ml = store.get_models("my_oseries")
```

2.7 Example Notebooks

The links below link to Jupyter Notebooks with explanation and examples of the usage of the *pastastore* module:

2.7.1 Managing time series and Pastas models with a database

This notebook shows how Pastas time series and models can be managed and stored on disk.

Content

1. The Connector object
    1. PasConnector
    2. Database structure
2. Initializing a PastaStore
3. Managing time series
    1. Adding oseries and stresses
    2. Accessing time series and metadata
    3. Deleting oseries and stresses
    4. Overview of oseries and stresses

4. Managing Pastas models
    1. Creating a model
    2. Storing a model
    3. Loading a model
    4. Overview of models
    5. Deleting models
5. Bulk operations
6. Deleting databases

1. The Connector object

This sections shows how to initialize a connection to a new database (connecting to an existing database works the same way).

Import `pastastore` and some other modules:

[1]:
```
import pastastore as pst
import os
import pandas as pd
import pastas as ps

ps.show_versions()
```

```
Python version: 3.9.7
NumPy version: 1.21.2
Pandas version: 1.5.2
SciPy version: 1.10.0
Matplotlib version: 3.6.1
Numba version: 0.55.1
LMfit version: 1.0.3
Latexify version: Not Installed
Pastas version: 1.0.0b
```

1.1 PasConnector

The PasConnector requires the path to a directory and a name for the connector. Data is stored in JSON files in the given directory.

[2]:
```
path = "./pas"
name = "my_first_connector"
```

Initialize the `PasConnector` object:

[3]:
```
conn = pst.PasConnector(name, path)
```

```
PasConnector: library oseries created in /home/david/Github/pastastore/examples/
↪notebooks/pas/oseries
PasConnector: library stresses created in /home/david/Github/pastastore/examples/
↪notebooks/pas/stresses
```

(continues on next page)

(continued from previous page)

```
PasConnector: library models created in /home/david/Github/pastastore/examples/notebooks/
↪pas/models
PasConnector: library oseries_models created in /home/david/Github/pastastore/examples/
↪notebooks/pas/oseries_models
```

Let's take a look at `conn`. This shows us how many oseries, stresses and models are contained in the store:

[4]:
```
conn
```

[4]:
```
<PasConnector> 'my_first_connector': 0 oseries, 0 stresses, 0 models
```

1.2 Database structure

The database/store contains 4 libraries or collections. Each of these contains specific data related to the project. The four libraries are: - oseries - stresses - models - oseries_models

These libraries are usually not meant to be accessed directly, but they can be accessed through the internal method `conn._get_library()`:

[5]:
```
# using the PasConnector
conn._get_library("stresses")
```

[5]:
```
'/home/david/Github/pastastore/examples/notebooks/pas/stresses'
```

The library handles are not generally used directly but internally they manage the reading, writing and deleting of data from the database/store. In the case of the `PasConnector`, the library is just a path to a directory.

2. Initializing a PastaStore object

The `PastaStore` object is used process and use the data in the database. The connector objects only manage the reading/writing/deleting of data. The `PastaStore` contains all kinds of methods to actually *do* other stuff with that data.

In order to access the data the `PastaStore` object must be initialized with a Connector object.

[6]:
```
pstore = pst.PastaStore("my_first_project", conn)
```

Let's take a look at the object:

[7]:
```
pstore
```

[7]:
```
<PastaStore> my_first_project:
 - <PasConnector> 'my_first_connector': 0 oseries, 0 stresses, 0 models
```

The connector object is accessible through `store.conn`, so all of the methods defined in the connector objects can be accessed through e.g. `store.conn.<method>`. The most common methods are also registered under the `store` object for easier access. The following statements are equivalent.

[8]:
```
pstore.conn.get_oseries
```

[8]:
```
<bound method BaseConnector.get_oseries of <PasConnector> 'my_first_connector': 0␣
↪oseries, 0 stresses, 0 models>
```

[9]:
```
pstore.get_oseries
```

[9]:
```
<bound method BaseConnector.get_oseries of <PasConnector> 'my_first_connector': 0␣
↪oseries, 0 stresses, 0 models>
```

3. Managing time series

This section explains how time series can be added, retrieved or deleted from the database. We'll be using the `PastaStore` instance we created before.

3.1 Adding oseries and stresses

Let's read some data to put into the database as an oseries. The data we are using is in the `tests/data` directory.

[10]:
```
datadir = "../../tests/data/"  # relative path to data directory
oseries1 = pd.read_csv(
    os.path.join(datadir, "head_nb1.csv"), index_col=0, parse_dates=True
)
oseries1.head()
```

[10]:
```
             head
date
1985-11-14  27.61
1985-11-28  27.73
1985-12-14  27.91
1985-12-28  28.13
1986-01-13  28.32
```

Add the time series to the oseries library using `store.add_oseries`. Metadata can be optionally be provided as a dictionary. In this example a dictionary x and y coordinates is passed as metadata which is convenient later for automatically creating Pastas models.

[11]:
```
pstore.add_oseries(oseries1, "oseries1", metadata={"x": 100300, "y": 400400})
```

The series was added to the oseries library. Let's confirm by looking at the `store` object:

[12]:
```
pstore
```

[12]:
```
<PastaStore> my_first_project:
 - <PasConnector> 'my_first_connector': 1 oseries, 0 stresses, 0 models
```

Stresses can be added similarly using `pstore.add_stress`. The only thing to keep in mind when adding stresses is to pass the `kind` argument so that different types of stresses (i.e. precipitation or evaporation) can be distinguished. The code below reads the precipitation and evaporation csv-files and adds them to our project:

[13]:
```
# prec
s = pd.read_csv(os.path.join(datadir, "rain_nb1.csv"), index_col=0, parse_dates=True)
pstore.add_stress(s, "prec1", kind="prec", metadata={"x": 100300, "y": 400400})

# evap
s = pd.read_csv(os.path.join(datadir, "evap_nb1.csv"), index_col=0, parse_dates=True)
pstore.add_stress(s, "evap1", kind="evap", metadata={"x": 100300, "y": 400400})
```

[14]:
```
pstore
```

[14]:
```
<PastaStore> my_first_project:
 - <PasConnector> 'my_first_connector': 1 oseries, 2 stresses, 0 models
```

3.2 Accessing time series and metadata

Time series can be accessed through `pstore.get_oseries()` or `pstore.get_stresses()`. These methods accept just a name or a list of names. In the latter case a dictionary of dataframes is returned.

[15]:
```
ts = pstore.get_oseries("oseries1")
ts.head()
```

[15]:
```
            oseries1
1985-11-14     27.61
1985-11-28     27.73
1985-12-14     27.91
1985-12-28     28.13
1986-01-13     28.32
```

Using a list of names:

[16]:
```
stresses = pstore.get_stresses(["prec1", "evap1"])
stresses
```

[16]:
```
{'prec1':              prec1
 1980-01-01  0.0033
 1980-01-02  0.0025
 1980-01-03  0.0003
 1980-01-04  0.0075
 1980-01-05  0.0080
 ...            ...
 2016-10-27  0.0000
 2016-10-28  0.0000
 2016-10-29  0.0003
 2016-10-30  0.0000
 2016-10-31  0.0000

 [13454 rows x 1 columns],
 'evap1':              evap1
 1980-01-01  0.0002
 1980-01-02  0.0003
 1980-01-03  0.0002
 1980-01-04  0.0001
 1980-01-05  0.0001
 ...            ...
 2016-11-18  0.0004
 2016-11-19  0.0003
 2016-11-20  0.0005
 2016-11-21  0.0003
 2016-11-22  0.0005

 [13476 rows x 1 columns]}
```

The metadata of a time series can be accessed through `pstore.get_metadata()`. Provide the library and the name to load the metadata for an oseries…

[17]:
```
meta = pstore.get_metadata("oseries", "oseries1")
meta
```

[17]:
```
                x       y
name
oseries1   100300  400400
```

or for multiple stresses:

[18]:
```
meta = pstore.get_metadata("stresses", ["prec1", "evap1"])
meta
```

[18]:
```
                x         y  kind
name
prec1   100300.0  400400.0  prec
evap1   100300.0  400400.0  evap
```

4.3 Deleting oseries and stresses

Deleting time series can be done using `pstore.del_oseries` or `pstore.del_stress`. These functions accept a single name or list of names of time series to delete.

4.4 Overview of oseries and stresses

An overview of the oseries and stresses is available through `pstore.oseries` and `pstore.stresses`. These are dataframes containing the metadata of all the time series. These dataframes are cached for performance. The cache is cleared when a time series is added or modified in the database.

[19]:
```
pstore.oseries
```

[19]:
```
                x       y
name
oseries1   100300  400400
```

[20]:
```
pstore.stresses
```

[20]:
```
                x         y  kind
name
evap1   100300.0  400400.0  evap
prec1   100300.0  400400.0  prec
```

4. Managing Pastas models

This section shows how Pastas models can be created, stored, and loaded from the database.

4.1 Creating a model

Creating a new model is straightforward using `pstore.create_model()`. The `add_recharge` keyword argument allows the user to choose (default is True) whether recharge is automatically added to the model using the nearest precipitation and evaporation stations in the stresses library. Note that the x,y-coordinates of the stresses and oseries must be set in the metadata for each time series in order for this to work.

[21]:
```
ml = pstore.create_model("oseries1", add_recharge=True)
ml
```

[21]:
```
Model(oseries=oseries1, name=oseries1, constant=True, noisemodel=True)
```

4.2 Storing a model

The model that was created in the previous step is not automatically stored in the models library. Use `store.add_model()` to store the model. If the model already exists, an Exception is raised warning the user the model is already in the library. Use `overwrite=True` to add the model anyway.

Note: The model is stored without the time series. It is assumed the time series are already stored in the oseries or stresses libraries, making it redundant to store these again. When adding the model, the stored copy of the time series is compared to the version in the model to ensure these are the same. If not, an error is raised and the model cannot be stored. These validation options can be overridden, but that is only recommended for advanced users.

[22]:
```
ml.solve()
```

```
Fit report oseries1                Fit Statistics
================================================
nfev    22                     EVP           92.91
nobs    644                    R2             0.93
noise   True                   RMSE           0.11
tmin    1985-11-14 00:00:00    AIC        -3257.53
tmax    2015-06-28 00:00:00    BIC        -3235.20
freq    D                      Obj            2.02
warmup  3650 days 00:00:00     ___
solver  LeastSquares           Interp.          No

Parameters (5 optimized)
================================================
                optimal    stderr     initial  vary
recharge_A   686.247135   ±5.30%  215.674528  True
recharge_a   159.386040   ±5.01%   10.000000  True
recharge_f    -1.305359   ±4.04%   -1.000000  True
constant_d    27.920134   ±0.21%   27.900078  True
noise_alpha   49.911868  ±11.86%   15.000000  True
```

[23]:
```
pstore.add_model(ml)
```

[24]:
```
pstore.models
```

[24]:
```
['oseries1']
```

[25]:
```
pstore.oseries_models
```

[25]:
```
{'oseries1': ['oseries1']}
```

4.3 Loading a model

Loading a stored model is simple using `pstore.get_models("<name>")` or using the key-value interface for models: `pstore.models["<name>"]`.

The model is stored as a dictionary (see `ml.to_dict()`) without the time series data. The time series in the model are picked up based on the names of those series from the respective libraries (oseries or stresses).

[26]:
```
ml2 = pstore.get_models("oseries1")
ml2
```

[26]:
```
Model(oseries=oseries1, name=oseries1, constant=True, noisemodel=True)
```

4.4 Overview of models

An overview of the models is available through `pstore.models` which lists the names of all the models:

[27]:
```
pstore.models
```

[27]:
```
['oseries1']
```

4.5 Deleting models

Deleting the model is done with `pstore.del_models`:

[28]:
```
pstore.del_models("oseries1")
```

Checking to see if it was indeed deleted:

[29]:
```
pstore
```

[29]:
```
<PastaStore> my_first_project:
 - <PasConnector> 'my_first_connector': 1 oseries, 2 stresses, 0 models
```

[30]:
```
pstore.models
```

[30]:
```
[]
```

[31]:
```
pstore.oseries_models
```

[31]:
```
{}
```

5. Bulk operations

The following bulk operations are available: - `create_models`: create models for all or a selection of oseries in database - `solve_models`: solve all or selection of models in database - `model_results`: get results for all or selection of models in database. Requires the `art_tools` module!

Let's add some more data to the pystore to show how the bulk operations work.

[32]:
```
# oseries 2
o = pd.read_csv(os.path.join(datadir, "obs.csv"), index_col=0, parse_dates=True)
pstore.add_oseries(o, "oseries2", metadata={"x": 100000, "y": 400000})

# prec 2
s = pd.read_csv(os.path.join(datadir, "rain.csv"), index_col=0, parse_dates=True)
pstore.add_stress(s, "prec2", kind="prec", metadata={"x": 100000, "y": 400000})

# evap 2
s = pd.read_csv(os.path.join(datadir, "evap.csv"), index_col=0, parse_dates=True)
pstore.add_stress(s, "evap2", kind="evap", metadata={"x": 100000, "y": 400000})
```

Let's take a look at our `PastaStore`:

[33]:
```
pstore
```

[33]:
```
<PastaStore> my_first_project:
 - <PasConnector> 'my_first_connector': 2 oseries, 4 stresses, 0 models
```

Let's try using the bulk methods on our database. The `pstore.create_models_bulk()` method allows the user to get models for all or a selection of oseries in the database. Options include: - selecting specific oseries to create models for - automatically adding recharge based on nearest precipitation and evaporation stresses - solving the models - storing the models in the models library

Note: when using the progressbar, for a prettier result the pastas log level can be set to ERROR using: `ps.set_log_level("ERROR")` or `ps.logger.setLevel("ERROR")`.

[34]:
```
# to suppress most of the log messages
ps.logger.setLevel("ERROR")
```

[35]:
```
errors = pstore.create_models_bulk()
```

```
Bulk creation models: 100%|----| 2/2 [00:00<00:00, 12.09it/s]
```

To solve all or a selection of models use `pstore.solve_models()`. Options for this method include: - selecting models to solve - store results in models library - raise error (or not) when solving fails - print solve reports

[36]:
```
pstore
```

[36]:
```
<PastaStore> my_first_project:
 - <PasConnector> 'my_first_connector': 2 oseries, 4 stresses, 2 models
```

[37]:
```
pstore.solve_models(store_result=True, report=False)
```

```
Solving models: 100%|----| 2/2 [00:01<00:00,  1.73it/s]
```

Obtaining the model parameters and statistics is easy with `pstore.get_parameters()` and `pstore.get_statistics()`. Results can be obtained for all or a selection of models. The results are returned as DataFrames.

[38]:
```
params = pstore.get_parameters()
params
```

[38]:
```
          recharge_A  recharge_a  recharge_f  constant_d  noise_alpha
oseries2  607.978079  154.752928   -1.423119   28.094025    66.035700
oseries1  686.247135  159.386040   -1.305359   27.920134    49.911868
```

[39]:
```
stats = pstore.get_statistics(["evp", "rmse"])
stats
```

[39]:
```
                evp      rmse
oseries2  88.661133  0.124792
oseries1  92.913581  0.114420
```

6. Deleting databases

The `pastastore.util` submodule contains functions for deleting database contents:

[40]:
```
pst.util.delete_pastastore(pstore)
```

```
Deleting PasConnector database: 'my_first_connector' ...  Done!
```

[ ]:
```

```

2.7.2 PastaStore plot and map utilities

This notebook shows the `PastaStore` functionality for quickly plotting time series, or plotting metadata or models (time series locations) on a map.

Content

[1]:
```
import pandas as pd
import pastastore as pst
import pastas as ps

from pastastore.datasets import example_pastastore
```

[2]:
```
ps.logger.setLevel("ERROR")  # silence Pastas for this notebook
ps.show_versions()
```

```
Python version: 3.9.7
NumPy version: 1.21.2
Pandas version: 1.5.2
SciPy version: 1.10.0
Matplotlib version: 3.6.1
Numba version: 0.55.1
LMfit version: 1.0.3
Latexify version: Not Installed
Pastas version: 1.0.0b
```

Populate a PastaStore with some data

First we create a Connector and a PastaStore object and add some data to it. We're using the example dataset to show the PastaStores plot and map methods.

[3]:
```
# get the example pastastore
conn = pst.DictConnector("my_connector")
pstore = example_pastastore(conn)

# remove some example data because it's far away
pstore.del_oseries(["head_nb5", "head_mw"])
pstore.del_stress(["prec_nb5", "evap_nb5", "riv_nb5"])
pstore.del_stress(["prec_mw", "evap_mw", "extraction_2", "extraction_3"])
```

```
INFO:hydropandas.io.io_menyanthes:reading menyanthes file /home/david/Github/pastastore/
↪pastastore/../tests/data/MenyanthesTest.men
INFO:hydropandas.io.io_menyanthes:reading oseries -> Obsevation well
INFO:hydropandas.io.io_menyanthes:reading stress -> Evaporation
INFO:hydropandas.io.io_menyanthes:reading stress -> Air Pressure
INFO:hydropandas.io.io_menyanthes:reading stress -> Precipitation
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 1
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 2
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 3
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 4
```

Maps

PastaStore contains a `maps` attribute that exposes methods for spatially plotting data contained in our database. There are methods for plotting oseries, stress and model locations and there is also a method for plotting a single model and all the time series it contains. The following sections showcase each of these methods. But a map is not a map without some kind of background. The function `PastaStore.maps.add_background_map` allows you to add a background map to any axes object. The method is powered by contextily and basically allows users to access some of the great functionality provided by that package. Contextily is not a `pastastore` dependency but is obviously recommended, and necessary if you want to access the background maps. For a list of possible background maps, consult `PastaStore.maps._list_contextily_providers()` (see below). We'll be using a few different background map options in the plots below. The default is `OpenStreetMap.Mapnik`.

Background maps

[4]:
```
# DataFrame of all contextily map providers
providers_df = pd.DataFrame(pstore.maps._list_contextily_providers()).T
providers_df
```

```
/home/david/anaconda3/envs/artesia/lib/python3.9/site-packages/requests/__init__.py:102:␣
↪RequestsDependencyWarning: urllib3 (1.26.7) or chardet (5.1.0)/charset_normalizer (2.0.
↪0) doesn't match a supported version!
```

[4]:
```
                                                              url  \
OpenStreetMap.Mapnik  https://{s}.tile.openstreetmap.org/{z}/{x}/{y}...
OpenStreetMap.DE      https://{s}.tile.openstreetmap.de/{z}/{x}/{y}.png
OpenStreetMap.CH        https://tile.osm.ch/switzerland/{z}/{x}/{y}.png
OpenStreetMap.France  https://{s}.tile.openstreetmap.fr/osmfr/{z}/{x...
OpenStreetMap.HOT     https://{s}.tile.openstreetmap.fr/hot/{z}/{x}/...
...                                                               ...
Strava.All            https://heatmap-external-a.strava.com/tiles/al...
Strava.Ride           https://heatmap-external-a.strava.com/tiles/ri...
Strava.Run            https://heatmap-external-a.strava.com/tiles/ru...
Strava.Water          https://heatmap-external-a.strava.com/tiles/wa...
Strava.Winter         https://heatmap-external-a.strava.com/tiles/wi...

                     max_zoom  \
OpenStreetMap.Mapnik       19
OpenStreetMap.DE           18
OpenStreetMap.CH           18
OpenStreetMap.France       20
OpenStreetMap.HOT          19
...                       ...
Strava.All                 15
Strava.Ride                15
Strava.Run                 15
Strava.Water               15
Strava.Winter              15

                                                  html_attribution  \
OpenStreetMap.Mapnik  &copy; <a href="https://www.openstreetmap.org/...
OpenStreetMap.DE      &copy; <a href="https://www.openstreetmap.org/...
OpenStreetMap.CH      &copy; <a href="https://www.openstreetmap.org/...
OpenStreetMap.France  &copy; OpenStreetMap France | &copy; <a href="...
```

(continues on next page)

(continued from previous page)

```
OpenStreetMap.HOT     &copy; <a href="https://www.openstreetmap.org/...
...                                                                 ...
Strava.All            Map tiles by <a href="https://labs.strava.com/...
Strava.Ride           Map tiles by <a href="https://labs.strava.com/...
Strava.Run            Map tiles by <a href="https://labs.strava.com/...
Strava.Water          Map tiles by <a href="https://labs.strava.com/...
Strava.Winter         Map tiles by <a href="https://labs.strava.com/...

                                                    attribution  \
OpenStreetMap.Mapnik                    (C) OpenStreetMap contributors
OpenStreetMap.DE                        (C) OpenStreetMap contributors
OpenStreetMap.CH                        (C) OpenStreetMap contributors
OpenStreetMap.France  (C) OpenStreetMap France | (C) OpenStreetMap c...
OpenStreetMap.HOT     (C) OpenStreetMap contributors, Tiles style by...
...                                                                 ...
Strava.All            Map tiles by <a href="https://labs.strava.com/...
Strava.Ride           Map tiles by <a href="https://labs.strava.com/...
Strava.Run            Map tiles by <a href="https://labs.strava.com/...
Strava.Water          Map tiles by <a href="https://labs.strava.com/...
Strava.Winter         Map tiles by <a href="https://labs.strava.com/...

                                    name               bounds variant  \
OpenStreetMap.Mapnik  OpenStreetMap.Mapnik                  NaN     NaN
OpenStreetMap.DE          OpenStreetMap.DE                  NaN     NaN
OpenStreetMap.CH          OpenStreetMap.CH  [[45, 5], [48, 11]]     NaN
OpenStreetMap.France  OpenStreetMap.France                  NaN     NaN
OpenStreetMap.HOT        OpenStreetMap.HOT                  NaN     NaN
...                                    ...                  ...     ...
Strava.All                    Strava.All                    NaN     NaN
Strava.Ride                  Strava.Ride                    NaN     NaN
Strava.Run                    Strava.Run                    NaN     NaN
Strava.Water                 Strava.Water                   NaN     NaN
Strava.Winter               Strava.Winter                   NaN     NaN

                     apikey min_zoom subdomains  ... language format size  \
OpenStreetMap.Mapnik    NaN      NaN        NaN  ...      NaN    NaN  NaN
OpenStreetMap.DE        NaN      NaN        NaN  ...      NaN    NaN  NaN
OpenStreetMap.CH        NaN      NaN        NaN  ...      NaN    NaN  NaN
OpenStreetMap.France    NaN      NaN        NaN  ...      NaN    NaN  NaN
OpenStreetMap.HOT       NaN      NaN        NaN  ...      NaN    NaN  NaN
...                     ...      ...        ...  ...      ...    ...  ...
Strava.All              NaN      NaN        NaN  ...      NaN    NaN  NaN
Strava.Ride             NaN      NaN        NaN  ...      NaN    NaN  NaN
Strava.Run              NaN      NaN        NaN  ...      NaN    NaN  NaN
Strava.Water            NaN      NaN        NaN  ...      NaN    NaN  NaN
Strava.Winter           NaN      NaN        NaN  ...      NaN    NaN  NaN

                     time tilematrixset  tms detectRetina apiVersion  \
OpenStreetMap.Mapnik  NaN           NaN  NaN          NaN        NaN
OpenStreetMap.DE      NaN           NaN  NaN          NaN        NaN
OpenStreetMap.CH      NaN           NaN  NaN          NaN        NaN
OpenStreetMap.France  NaN           NaN  NaN          NaN        NaN
```

(continues on next page)

(continued from previous page)

```
OpenStreetMap.HOT     NaN           NaN  NaN          NaN        NaN
...                   ...           ...  ...          ...        ...
Strava.All            NaN           NaN  NaN          NaN        NaN
Strava.Ride           NaN           NaN  NaN          NaN        NaN
Strava.Run            NaN           NaN  NaN          NaN        NaN
Strava.Water          NaN           NaN  NaN          NaN        NaN
Strava.Winter         NaN           NaN  NaN          NaN        NaN

                     subscriptionKey timeStamp
OpenStreetMap.Mapnik             NaN       NaN
OpenStreetMap.DE                 NaN       NaN
OpenStreetMap.CH                 NaN       NaN
OpenStreetMap.France             NaN       NaN
OpenStreetMap.HOT                NaN       NaN
...                              ...       ...
Strava.All                       NaN       NaN
Strava.Ride                      NaN       NaN
Strava.Run                       NaN       NaN
Strava.Water                     NaN       NaN
Strava.Winter                    NaN       NaN

[233 rows x 35 columns]
```

[5]:
```
# list all
pstore.maps._list_contextily_providers().keys()
```

[5]:
```
dict_keys(['OpenStreetMap.Mapnik', 'OpenStreetMap.DE', 'OpenStreetMap.CH',
↪'OpenStreetMap.France', 'OpenStreetMap.HOT', 'OpenStreetMap.BZH', 'OpenStreetMap.
↪BlackAndWhite', 'OpenSeaMap', 'OPNVKarte', 'OpenTopoMap', 'OpenRailwayMap',
↪'OpenFireMap', 'SafeCast', 'Stadia.AlidadeSmooth', 'Stadia.AlidadeSmoothDark', 'Stadia.
↪OSMBright', 'Stadia.Outdoors', 'Thunderforest.OpenCycleMap', 'Thunderforest.Transport',
↪ 'Thunderforest.TransportDark', 'Thunderforest.SpinalMap', 'Thunderforest.Landscape',
↪'Thunderforest.Outdoors', 'Thunderforest.Pioneer', 'Thunderforest.MobileAtlas',
↪'Thunderforest.Neighbourhood', 'CyclOSM', 'Jawg.Streets', 'Jawg.Terrain', 'Jawg.Sunny',
↪ 'Jawg.Dark', 'Jawg.Light', 'Jawg.Matrix', 'MapBox', 'MapTiler.Streets', 'MapTiler.
↪Basic', 'MapTiler.Bright', 'MapTiler.Pastel', 'MapTiler.Positron', 'MapTiler.Hybrid',
↪'MapTiler.Toner', 'MapTiler.Topo', 'MapTiler.Voyager', 'MapTiler.Basic4326', 'MapTiler.
↪Outdoor', 'MapTiler.Topographique', 'MapTiler.Winter', 'MapTiler.Satellite', 'MapTiler.
↪Terrain', 'Stamen.Toner', 'Stamen.TonerBackground', 'Stamen.TonerHybrid', 'Stamen.
↪TonerLines', 'Stamen.TonerLabels', 'Stamen.TonerLite', 'Stamen.Watercolor', 'Stamen.
↪Terrain', 'Stamen.TerrainBackground', 'Stamen.TerrainLabels', 'Stamen.TopOSMRelief',
↪'Stamen.TopOSMFeatures', 'TomTom.Basic', 'TomTom.Hybrid', 'TomTom.Labels', 'Esri.
↪WorldStreetMap', 'Esri.DeLorme', 'Esri.WorldTopoMap', 'Esri.WorldImagery', 'Esri.
↪WorldTerrain', 'Esri.WorldShadedRelief', 'Esri.WorldPhysical', 'Esri.OceanBasemap',
↪'Esri.NatGeoWorldMap', 'Esri.WorldGrayCanvas', 'Esri.ArcticOceanBase', 'Esri.
↪ArcticOceanReference', 'Esri.AntarcticBasemap', 'OpenWeatherMap.Clouds',
↪'OpenWeatherMap.CloudsClassic', 'OpenWeatherMap.Precipitation', 'OpenWeatherMap.
↪PrecipitationClassic', 'OpenWeatherMap.Rain', 'OpenWeatherMap.RainClassic',
↪'OpenWeatherMap.Pressure', 'OpenWeatherMap.PressureContour', 'OpenWeatherMap.Wind',
↪'OpenWeatherMap.Temperature', 'OpenWeatherMap.Snow', 'HERE.normalDay', 'HERE.
↪normalDayCustom', 'HERE.normalDayGrey', 'HERE.normalDayMobile', 'HERE.
↪normalDayGreyMobile', 'HERE.normalDayTransit', 'HERE.normalDayTransitMobile', 'HERE.
```

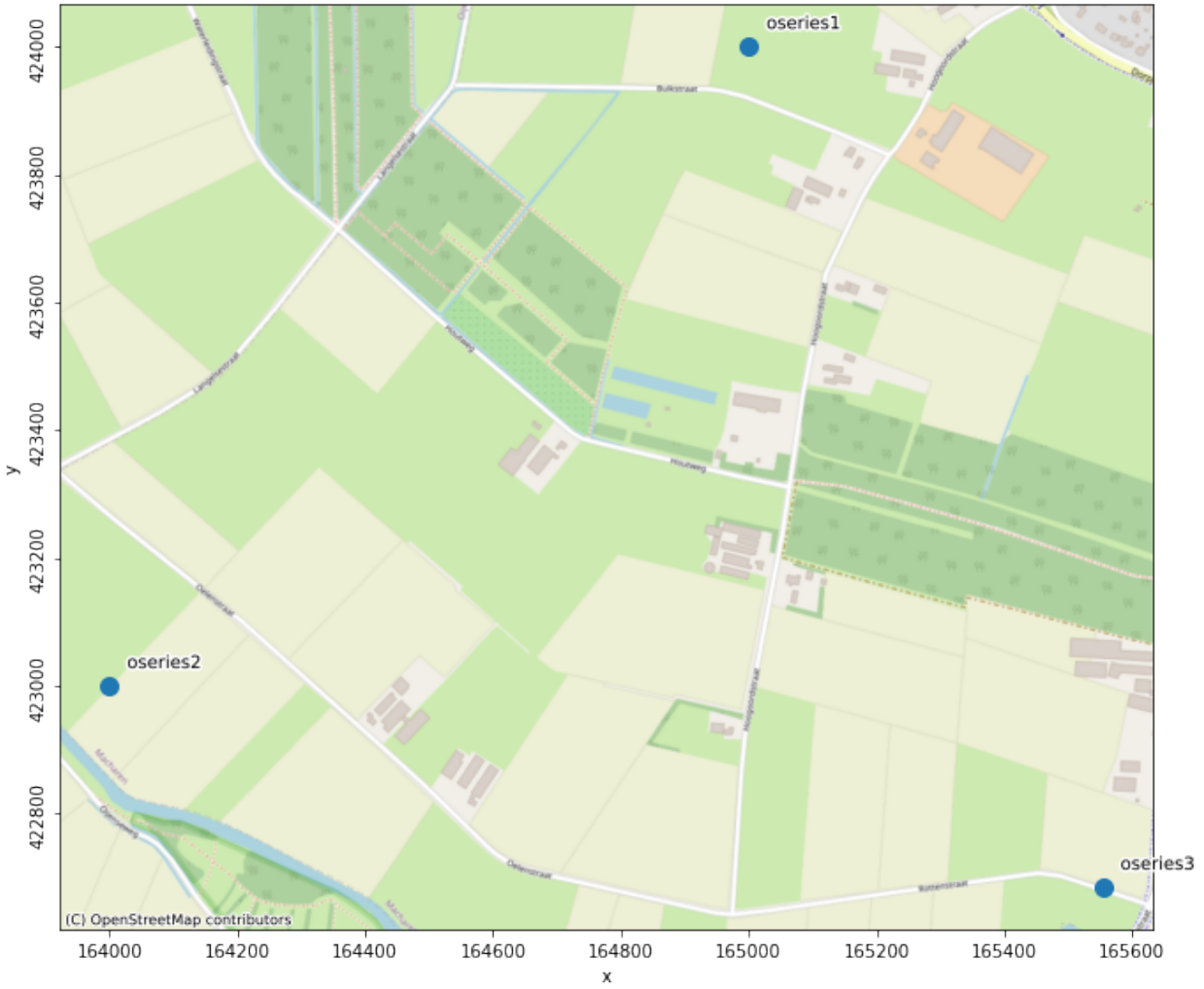
(continues on next page)

(continued from previous page)

```
↪normalDayTraffic', 'HERE.normalNight', 'HERE.normalNightMobile', 'HERE.normalNightGrey
↪', 'HERE.normalNightGreyMobile', 'HERE.normalNightTransit', 'HERE.
↪normalNightTransitMobile', 'HERE.reducedDay', 'HERE.reducedNight', 'HERE.basicMap',
↪'HERE.mapLabels', 'HERE.trafficFlow', 'HERE.carnavDayGrey', 'HERE.hybridDay', 'HERE.
↪hybridDayMobile', 'HERE.hybridDayTransit', 'HERE.hybridDayGrey', 'HERE.hybridDayTraffic
↪', 'HERE.pedestrianDay', 'HERE.pedestrianNight', 'HERE.satelliteDay', 'HERE.terrainDay
↪', 'HERE.terrainDayMobile', 'HEREv3.normalDay', 'HEREv3.normalDayCustom', 'HEREv3.
↪normalDayGrey', 'HEREv3.normalDayMobile', 'HEREv3.normalDayGreyMobile', 'HEREv3.
↪normalDayTransit', 'HEREv3.normalDayTransitMobile', 'HEREv3.normalNight', 'HEREv3.
↪normalNightMobile', 'HEREv3.normalNightGrey', 'HEREv3.normalNightGreyMobile', 'HEREv3.
↪normalNightTransit', 'HEREv3.normalNightTransitMobile', 'HEREv3.reducedDay', 'HEREv3.
↪reducedNight', 'HEREv3.basicMap', 'HEREv3.mapLabels', 'HEREv3.trafficFlow', 'HEREv3.
↪carnavDayGrey', 'HEREv3.hybridDay', 'HEREv3.hybridDayMobile', 'HEREv3.hybridDayTransit
↪', 'HEREv3.hybridDayGrey', 'HEREv3.pedestrianDay', 'HEREv3.pedestrianNight', 'HEREv3.
↪satelliteDay', 'HEREv3.terrainDay', 'HEREv3.terrainDayMobile', 'FreeMapSK', 'MtbMap',
↪'CartoDB.Positron', 'CartoDB.PositronNoLabels', 'CartoDB.PositronOnlyLabels', 'CartoDB.
↪DarkMatter', 'CartoDB.DarkMatterNoLabels', 'CartoDB.DarkMatterOnlyLabels', 'CartoDB.
↪Voyager', 'CartoDB.VoyagerNoLabels', 'CartoDB.VoyagerOnlyLabels', 'CartoDB.
↪VoyagerLabelsUnder', 'HikeBike.HikeBike', 'HikeBike.HillShading', 'BasemapAT.basemap',
↪'BasemapAT.grau', 'BasemapAT.overlay', 'BasemapAT.terrain', 'BasemapAT.surface',
↪'BasemapAT.highdpi', 'BasemapAT.orthofoto', 'nlmaps.standaard', 'nlmaps.pastel',
↪'nlmaps.grijs', 'nlmaps.water', 'nlmaps.luchtfoto', 'NASAGIBS.ModisTerraTrueColorCR',
↪'NASAGIBS.ModisTerraBands367CR', 'NASAGIBS.ViirsEarthAtNight2012', 'NASAGIBS.
↪ModisTerraLSTDay', 'NASAGIBS.ModisTerraSnowCover', 'NASAGIBS.ModisTerraAOD', 'NASAGIBS.
↪ModisTerraChlorophyll', 'NASAGIBS.ModisTerraBands721CR', 'NASAGIBS.ModisAquaTrueColorCR
↪', 'NASAGIBS.ModisAquaBands721CR', 'NASAGIBS.ViirsTrueColorCR', 'NASAGIBS.
↪BlueMarble3413', 'NASAGIBS.BlueMarble3031', 'NASAGIBS.BlueMarble', 'NLS', 'JusticeMap.
↪income', 'JusticeMap.americanIndian', 'JusticeMap.asian', 'JusticeMap.black',
↪'JusticeMap.hispanic', 'JusticeMap.multi', 'JusticeMap.nonWhite', 'JusticeMap.white',
↪'JusticeMap.plurality', 'GeoportailFrance.plan', 'GeoportailFrance.parcels',
↪'GeoportailFrance.orthos', 'OneMapSG.Default', 'OneMapSG.Night', 'OneMapSG.Original',
↪'OneMapSG.Grey', 'OneMapSG.LandLot', 'USGS.USTopo', 'USGS.USImagery', 'USGS.
↪USImageryTopo', 'WaymarkedTrails.hiking', 'WaymarkedTrails.cycling', 'WaymarkedTrails.
↪mtb', 'WaymarkedTrails.slopes', 'WaymarkedTrails.riding', 'WaymarkedTrails.skating',
↪'OpenAIP', 'OpenSnowMap.pistes', 'AzureMaps.MicrosoftImagery', 'AzureMaps.
↪MicrosoftBaseDarkGrey', 'AzureMaps.MicrosoftBaseRoad', 'AzureMaps.
↪MicrosoftBaseHybridRoad', 'AzureMaps.MicrosoftTerraMain', 'AzureMaps.
↪MicrosoftWeatherInfraredMain', 'AzureMaps.MicrosoftWeatherRadarMain',
↪'SwissFederalGeoportal.NationalMapColor', 'SwissFederalGeoportal.NationalMapGrey',
↪'SwissFederalGeoportal.SWISSIMAGE', 'SwissFederalGeoportal.JourneyThroughTime', 'Gaode.
↪Normal', 'Gaode.Satellite', 'Strava.All', 'Strava.Ride', 'Strava.Run', 'Strava.Water',
↪'Strava.Winter'])
```

Oseries locations

[6]:
```
# plot oseries locations
ax1 = pstore.maps.oseries(labels=True, s=100)
pstore.maps.add_background_map(ax1)
```



All stresses

[7]:
```
pstore.stresses
```

[7]:

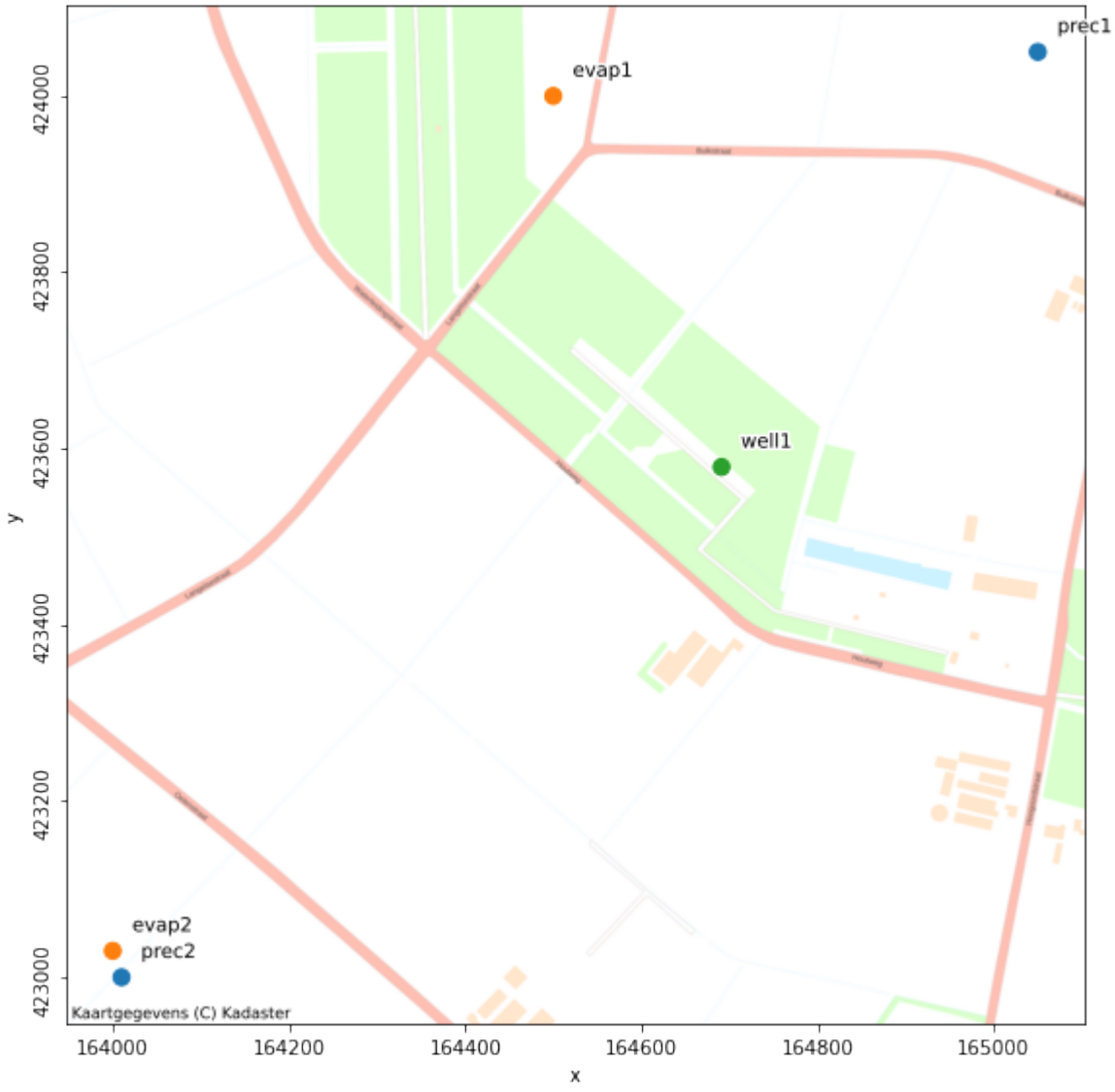
name	x	y	kind
prec1	165050.000000	424050.000000	prec
prec2	164010.000000	423000.000000	prec
evap1	164500.000000	424000.000000	evap
evap2	164000.000000	423030.000000	evap
well1	164691.000000	423579.000000	well
pressure_mw	123715.757915	397547.780543	pressure

(continues on next page)

(continued from previous page)

```
extraction_1   81985.384615  380070.307692      well
extraction_4   87111.000000  374334.000000      well
```

[8]:
```
# plot all stresses locations
ax2 = pstore.maps.stresses(names=["prec1", "evap1", "well1", "prec2", "evap2"])
pstore.maps.add_background_map(ax2, map_provider="nlmaps.pastel")
```

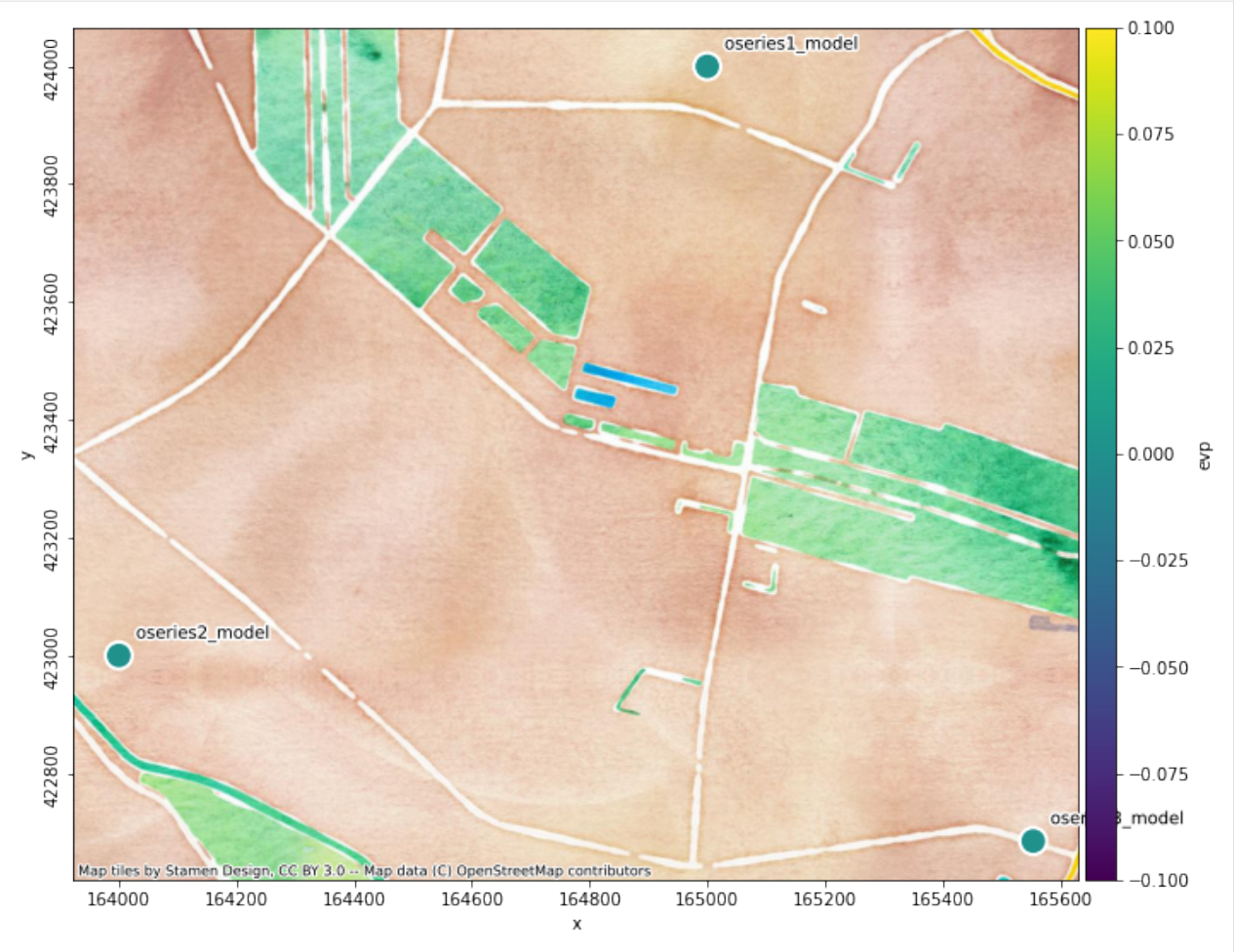


Model locations

[9]:
```
# create models to show
for o in pstore.oseries.index:
    ml = pstore.create_model(o, modelname=f"{o}_model", add_recharge=True)
    pstore.add_model(ml, overwrite=True)

# plot model location
ax3 = pstore.maps.models(color="C2", s=250, edgecolor="k")
pstore.maps.add_background_map(ax3, map_provider="nlmaps.standaard")
```

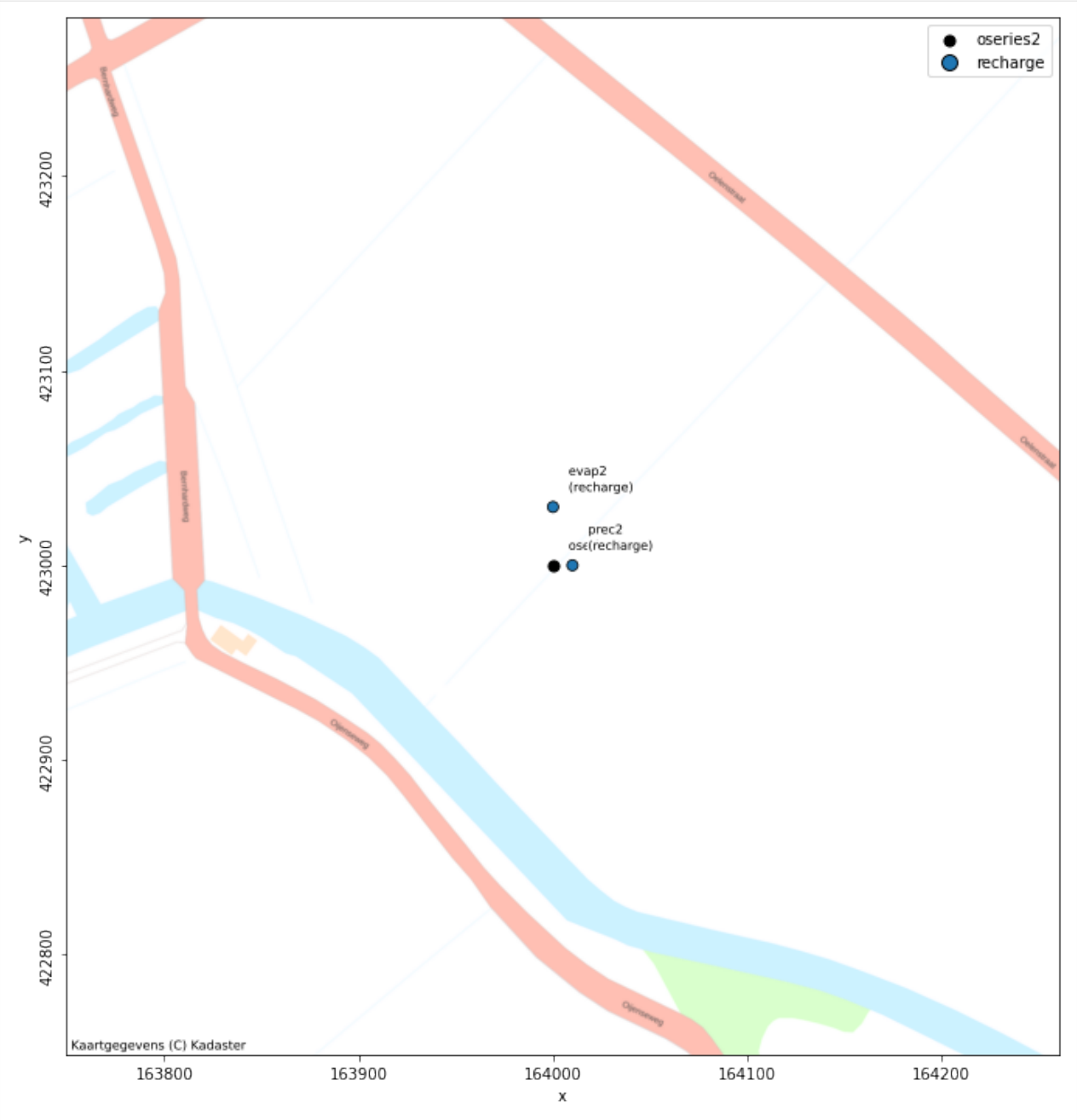
Model statistics

[10]:
```
# plot model evp on map
ax4 = pstore.maps.modelstat("evp", s=250, edgecolors="w", linewidths=2)
pstore.maps.add_background_map(ax4, map_provider="Stamen.Watercolor")
```



A single model and its time series

[11]:
```
# plot one model, oseries and stresses
ax5 = pstore.maps.model("oseries2_model", metadata_source="store", offset=250)
pstore.maps.add_background_map(ax5, map_provider="nlmaps.pastel")
```


oseries2
recharge
evap2
(recharge)
prec2
ose(recharge)
Kaartgegevens (C) Kadaster
423200
423100
423000
422900
422800
y
163800
163900
164000
164100
164200
x

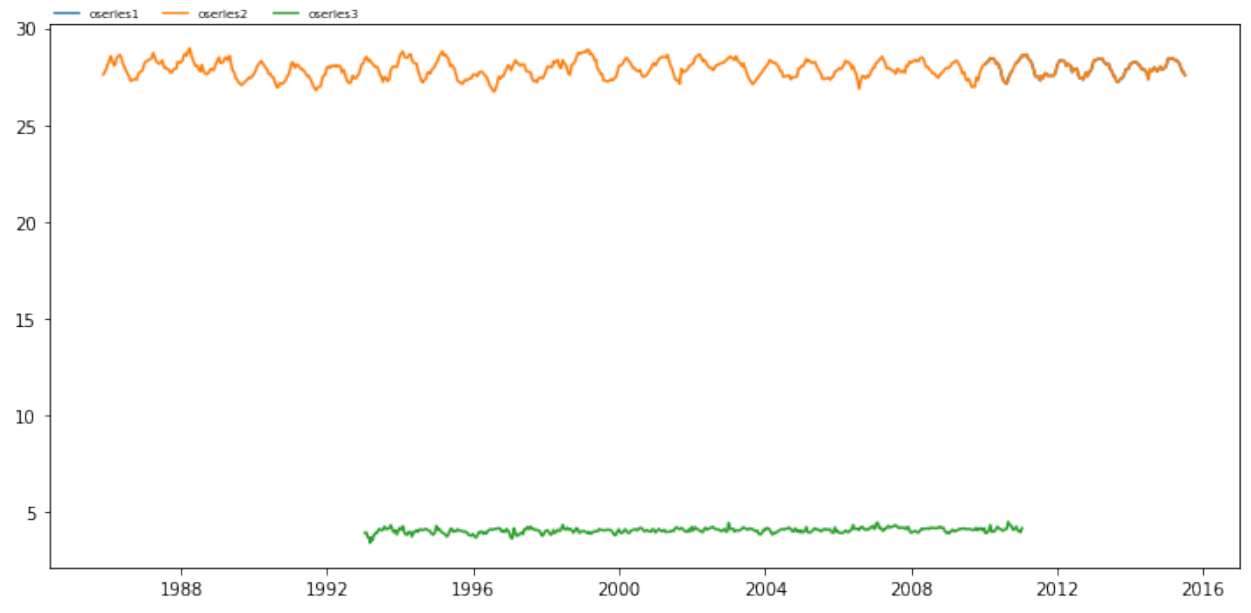
Plots

A PastaStore also has a `.plots` attribute that exposes methods for plotting time series or an overview of data availability. The examples below run through the different methods and how they work.

Plot oseries

[12]:
```
# plot oseries
ax6 = pstore.plots.oseries()
```

```
Get oseries: 100%|----| 3/3 [00:00<00:00, 11086.27it/s]
```

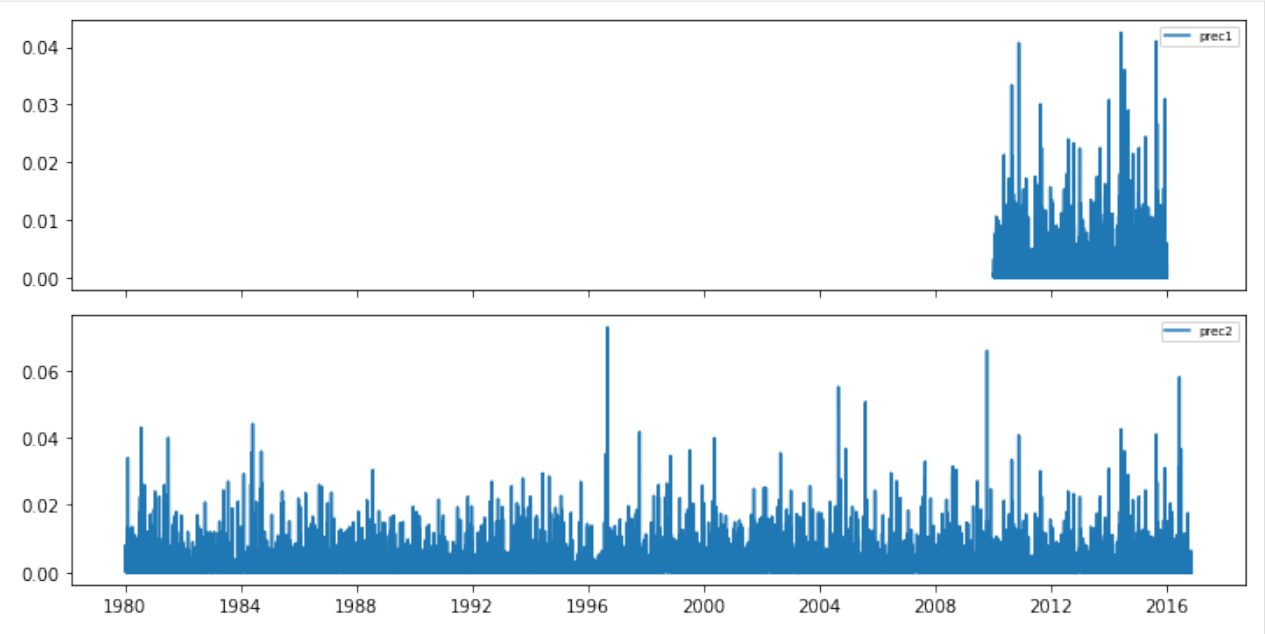


Plot stresses

When plotting stresses you can pass the `kind` argument to select only stresses of a particular kind. The `split` keyword argument allows you to plot each stress in a separate axis. Note that if there are more than 20 stresses, split is no longer supported.

Also, you can silence the progressbar by passing `progressbar=False`.

[13]:
```
# plot well stresses on separate axes
ax7 = pstore.plots.stresses(kind="prec", split=True, progressbar=False)
```

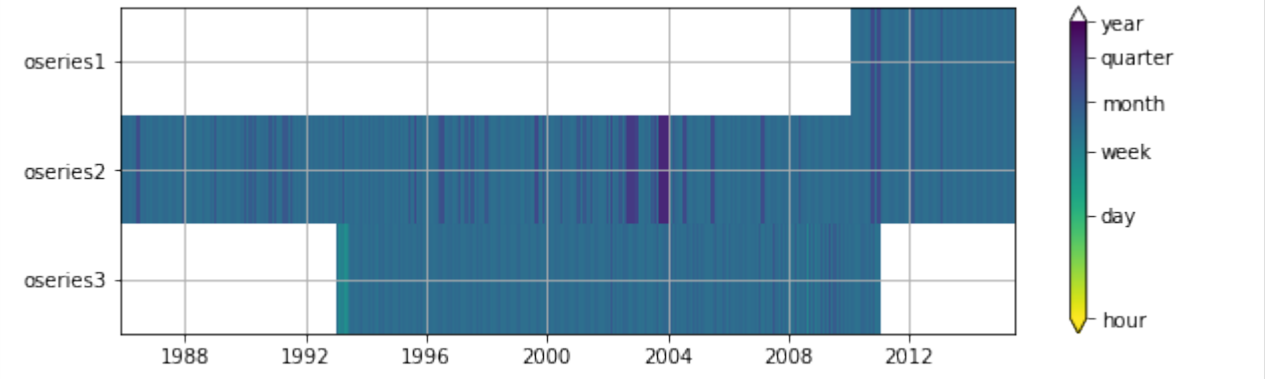


Data availability

Plotting data availability shows time period for which data is available and also the observation timestep. Below are three examples for oseries, all stresses, and on;y the evaporation stresses. The `set_yticks` keyword determines whether the names of the time series are used as yticks. This generally works fine if the number of time series isn't too large, but for large datasets, setting it to False is recommended.

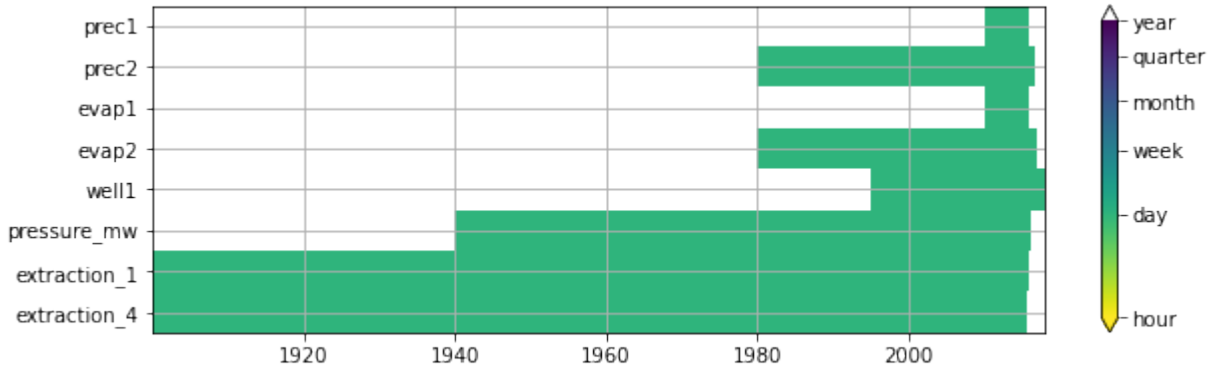
[14]:
```
# plot data availability for oseries
ax8 = pstore.plots.data_availability("oseries", set_yticks=True, figsize=(10, 3))
```

```
Get oseries: 100%|----| 3/3 [00:00<00:00, 6462.72it/s]
```



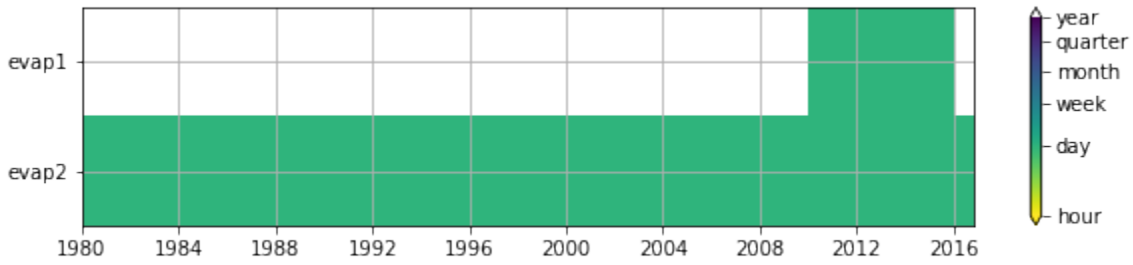
[15]:
```
# plot data availability for all stresses
ax9 = pstore.plots.data_availability("stresses", set_yticks=True, figsize=(10, 3))
```

```
Get stresses: 100%|----| 8/8 [00:00<00:00, 8085.41it/s]
```



[16]:
```
# plot data availability only stresses with kind="well"
ax10 = pstore.plots.data_availability(
    "stresses", kind="evap", set_yticks=True, figsize=(10, 2)
)
```

```
Get stresses: 100%|----| 2/2 [00:00<00:00, 4217.50it/s]
```



[ ]:

2.7.3 The PastaStore YAML interface

This notebook shows how Pastas models can be built from YAML files, using Pastastore.

Contents

[30]:
```
import os
import tempfile
from contextlib import contextmanager
from io import StringIO

import pastastore as pst
import yaml
```

[31]:
```
# create a temporary yaml file that is deleted after usage


@contextmanager
def tempyaml(yaml):
    temp = tempfile.NamedTemporaryFile(delete=False)
    temp.write(yaml.encode("utf-8"))
    temp.close()
    try:
        yield temp.name
    finally:
        os.unlink(temp.name)
```

Why YAML?

YAML, according to the official webpage is *"YAML is a human-friendly data serialization language for all programming languages"*. The file structure is similar to JSON (nested dictionaries) and therefore similar to the storage format for pastas Models, i.e. `.pas`-files.

So why develop a method for reading/writing pastas models to and from YAML files? The human-readability of the file structure in combination with leveraging tools in pastastore allow users to quickly build pastas Models using a mini-language, without having to explicitly program each line of code. When users are working with a lot of models with different model structures, the YAML files can provide a simple and convenient interface to structure this work, without having to search through lots of lines of code.

Whether it is useful to "program" the models in YAML or in normal Python/pastas code depends on the application or project. This feature was developed to give users an extra option that combines human-readable files with useful tools from the pastastore to quickly develop pastas models.

An example YAML file

A YAML file is text file that uses Python-style indentation to indicate nesting. The following shows the structure of a YAML file for defining a pastas model.

```
# comments are allowed, this is a pastas Model:

my_first_model:                 # model name
  oseries: head_nb1             # head time series name, obtained from pastastore
  stressmodels:                 # stressmodels dictionary
    recharge:                   # name of the recharge stressmodel
      class: RechargeModel      # type of pastas StressModel
      prec: prec1               # name of precipitation stress, obtained from␣
↪pastastore
      evap: evap1               # name of evaporation stress, obtained from pastastore
```

(continues on next page)

(continued from previous page)

```
      recharge: Linear            # pastas recharge type
      rfunc: Exponential          # rfunc
```

Reading this file converts it into a nested dictionary, as shown below. This dictionary can be used to (re-)construct pastas models, as is shown in the next sections.

[32]:
```
yaml_file = """
# comments are allowed, this is a pastas Model:

my_first_model:                   # model name
  oseries: head_nb1               # head time series name, obtained from pastastore
  stressmodels:                   # stressmodels dictionary
    recharge:                     # name of the recharge stressmodel
      class: RechargeModel        # type of pastas StressModel
      prec: prec1                 # name of precipitation stress, obtained from␣
↪pastastore
      evap: evap1                 # name of evaporation stress, obtained from pastastore
      recharge: Linear            # pastas recharge type
      rfunc: Exponential          # response function
"""

# load the file
d = yaml.load(StringIO(yaml_file), Loader=yaml.Loader)

# view the resulting dictionary
d
```

[32]:
```
{'my_first_model': {'oseries': 'head_nb1',
  'stressmodels': {'recharge': {'class': 'RechargeModel',
    'prec': 'prec1',
    'evap': 'evap1',
    'recharge': 'Linear',
    'rfunc': 'Exponential'}}}}
```

The PastaStore.yaml interface

The logic for reading/writing YAML files is accessed through the `PastaStore.yaml` interface. First we need a PastaStore and fill it with some data to showcase this. Load the example dataset from the PastaStore (included since version 0.8.0 (note, this data is only available if the pastastore repository was cloned and not if it was installed with `pip`).

[33]:
```
from pastastore.datasets import example_pastastore  # noqa: E402
```

[34]:
```
pstore = example_pastastore()
pstore
```

```
INFO:hydropandas.io.io_menyanthes:reading menyanthes file /home/david/Github/pastastore/
↪pastastore/../tests/data/MenyanthesTest.men
INFO:hydropandas.io.io_menyanthes:reading oseries -> Obsevation well
INFO:hydropandas.io.io_menyanthes:reading stress -> Evaporation
INFO:hydropandas.io.io_menyanthes:reading stress -> Air Pressure
INFO:hydropandas.io.io_menyanthes:reading stress -> Precipitation
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 1
```

(continues on next page)

(continued from previous page)

```
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 2
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 3
INFO:hydropandas.io.io_menyanthes:reading stress -> Extraction 4
```

[34]:
```
<PastaStore> example:
 - <DictConnector> 'my_db': 5 oseries, 15 stresses, 0 models
```

Let's check which oseries are available:

[35]:
```
pstore.oseries
```

[35]:
```
                x         y
name
oseries1  165000.0  424000.0
oseries2  164000.0  423000.0
oseries3  165554.0  422685.0
head_nb5  200000.0  450000.0
head_mw    85850.0  383362.0
```

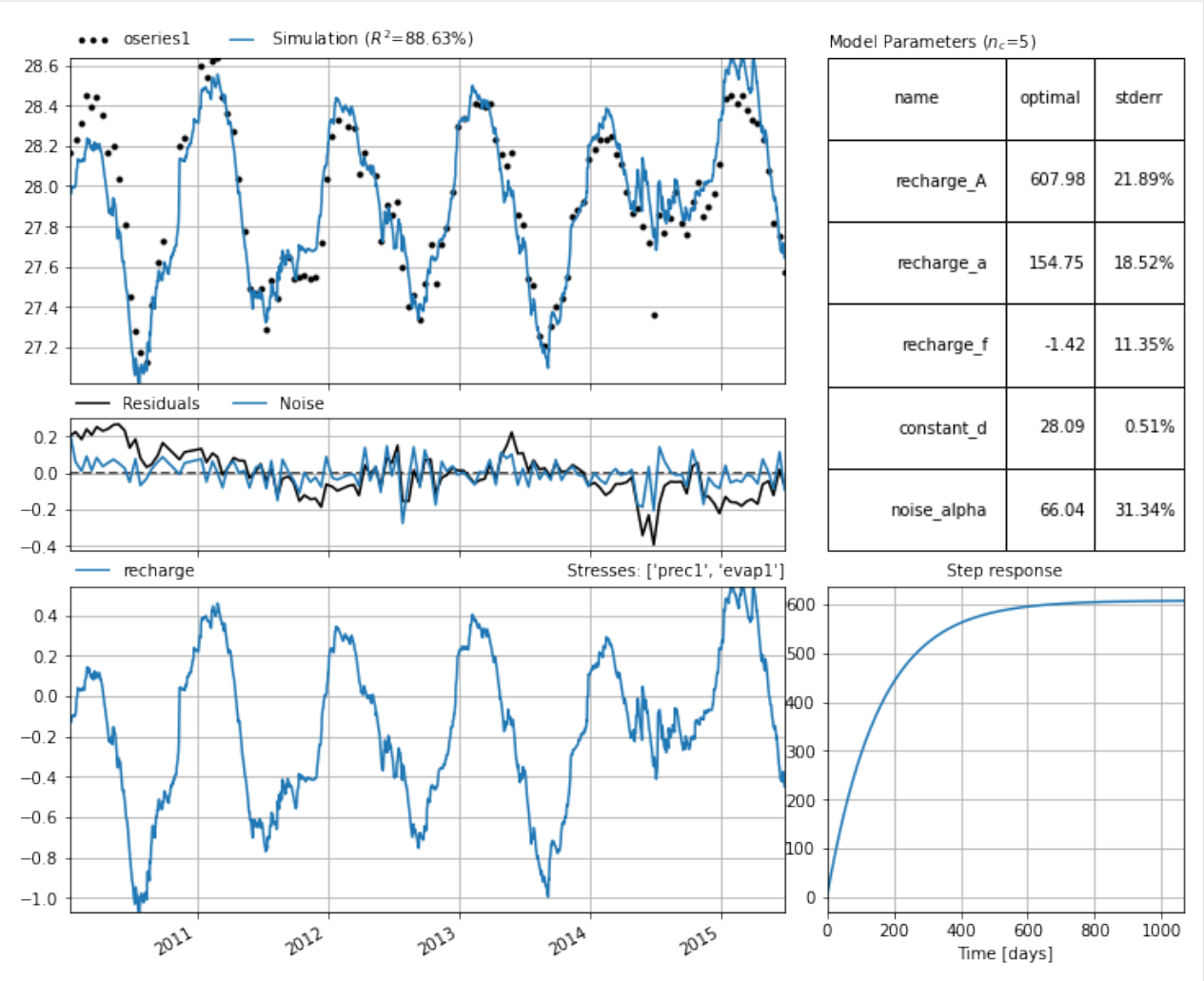
Building model(s) from a YAML file

[36]:
```
my_first_yaml = """
my_first_model:                  # model name
  oseries: oseries1              # head time series name, obtained from pastastore
  stressmodels:                  # stressmodels dictionary
    recharge:                    # name of the recharge stressmodel
      class: RechargeModel       # type of pastas StressModel
      prec: prec1                # name of precipitation stress, obtained from␣
↪pastastore
      evap: evap1                # name of evaporation stress, obtained from pastastore
      recharge: Linear           # pastas recharge type
      rfunc: Exponential         # response function
"""

with tempyaml(my_first_yaml) as f:
    ml = pstore.yaml.load(f)[0]  # returns a list

ml
```

```
INFO:pastastore.yaml_interface:Building model 'my_first_model' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
```

[36]:
```
Model(oseries=oseries1, name=my_first_model, constant=True, noisemodel=True)
```

[37]:
```
ml.solve(report=False)
ml.plots.results()
```



A YAML file can contain multiple models

[38]:
```
my_multi_model_yaml = """
my_first_model:                    # model name
  oseries: oseries1                # head time series name, obtained from pastastore
  stressmodels:                    # stressmodels dictionary
    recharge:                      # name of the recharge stressmodel
      class: RechargeModel         # type of pastas StressModel
      prec: prec1                  # name of precipitation stress, obtained from␣
↪pastastore
      evap: evap1                  # name of evaporation stress, obtained from pastastore
      recharge: Linear             # pastas recharge type
      rfunc: Exponential           # response function

my_second_model:                   # model name
  oseries: oseries1                # head time series name, obtained from pastastore
  stressmodels:                    # stressmodels dictionary
    recharge:                      # name of the recharge stressmodel
      class: RechargeModel         # type of pastas StressModel
      prec: prec1                  # name of precipitation stress, obtained from␣
```

(continues on next page)

(continued from previous page)

```
↪pastastore
      evap: evap1                # name of evaporation stress, obtained from pastastore
      recharge: FlexModel        # pastas recharge type
      rfunc: Exponential         # response function
"""
```

[39]:
```
with tempyaml(my_multi_model_yaml) as f:
    models = pstore.yaml.load(f)

models
```

```
INFO:pastastore.yaml_interface:Building model 'my_first_model' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:Building model 'my_second_model' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
```

[39]:
```
[Model(oseries=oseries1, name=my_first_model, constant=True, noisemodel=True),
 Model(oseries=oseries1, name=my_second_model, constant=True, noisemodel=True)]
```

Note that these models are not automatically added to the PastaStore. They are only created. To store them use `PastaStore.add_model`.

[40]:
```
for ml in models:
    pstore.add_model(ml)
```

[41]:
```
pstore
```

[41]:
```
<PastaStore> example:
 - <DictConnector> 'my_db': 5 oseries, 15 stresses, 2 models
```

Writing model(s) to a YAML file

Writing an existing model to a YAML file is done with `PastaStore.yaml.export_model()`. The resulting YAML file contains a lot more information as all model information is stored in the file, similar to saving a model as `.pas`-file with `ml.to_file()`. It can be useful to take a look at this file as a template for writing your own YAML files.

[42]:
```
pstore.yaml.export_model(ml)
```

The YAML file can be simplified with the `minimal_yaml` keyword argument.

Warning: Using the `minimal_yaml=True` option can lead to a different model than the one being exported as certain important model settings might have been removed in the resulting YAML file. Use with care!

[43]:
```
ml.name = ml.name + "_minimal"
pstore.yaml.export_model(ml, minimal_yaml=True)
```

Additionally, the `use_nearest` option fills in "`nearest <n> <kind>`" instead of the names of the time series, filling in <n> and <kind> where possible. This option is only used when `minimal_yaml=True`.

Warning: This option does not check whether the time series are actually nearest, it simply fills in "nearest" for all stresses and fills in "kind" where possible.

[44]:
```
ml.name = ml.name + "_nearest"
pstore.yaml.export_model(ml, minimal_yaml=True, use_nearest=True)
```

The models can als be written to a single YAML-file using `PastaStore.yaml.export_models()`. The `split=False` kwarg forces all models to be written to the same file.

[45]:
```
pstore.yaml.export_models(models=models, split=False)
```

"Nearest" options for time series

The YAML file format introduces some useful features that leverage useful tools in PastaStore. Instead of explicitly defining the time series to use for a particular stressmodel, there is a `nearest` option. Note that this requires the metadata of the time series in the PastaStore to be properly defined, with `x` and `y` coordinates for all time series.

First let's revisit the first example YAML file, but this time use the "nearest" option to select the precipitation and evaporation time series. After nearest the `kind` identifier is supplied to tell the PastaStore which types of stresses to consider when looking for the nearest one.

[46]:
```
nearest_yaml = """
my_first_model:                  # model name
  oseries: oseries1              # head time series name, obtained from pastastore
  stressmodels:                  # stressmodels dictionary
    recharge:                    # name of the recharge stressmodel
      class: RechargeModel       # type of pastas StressModel
      prec: nearest prec         # nearest stress with kind="prec" obtained from␣
↪pastastore
      evap: nearest evap         # nearest stress with kind="evap" obtained from␣
↪pastastore
      recharge: Linear           # pastas recharge type
      rfunc: Exponential         # response function
"""

with tempyaml(nearest_yaml) as f:
    ml = pstore.yaml.load(f)[0]  # returns a list

ml
```

```
INFO:pastastore.yaml_interface:Building model 'my_first_model' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='prec': 'prec1'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='evap': 'evap1'
```

[46]:
```
Model(oseries=oseries1, name=my_first_model, constant=True, noisemodel=True)
```

The nearest option is parsed depending on the type of stressmodel. Generally, the form is `nearest <kind>`, but for the RechargeModel, just providing `nearest` will assume the kind is `kind="prec"` or `kind="evap"`.

For WellModel, the number of nearest stresses can be passed as well, e.g. `nearest <n> <kind>`.

The following examples illustrate this:

[47]:
```
full_nearest_yaml = """
nearest_model_1:                 # model name
  oseries: head_nb5              # head time series name, obtained from pastastore
  stressmodels:                  # stressmodels dictionary
    recharge:                    # name of the recharge stressmodel
      class: RechargeModel       # type of pastas stressmodel
      prec: nearest              # nearest stress with kind="prec" obtained from␣
```

(continues on next page)

(continued from previous page)

```
↪pastastore
      evap: nearest evap          # nearest stress with kind="evap" obtained from␣
↪pastastore
      recharge: Linear            # pastas recharge type
      rfunc: Exponential          # response function
    river:                        # name for river stressmodel
      class: StressModel          # type of pastas stressmodel
      stress: nearest riv         # nearest stress with kind="riv" obtained from␣
↪pastastore
      rfunc: One                  # response function
      settings: level             # time series settings


nearest_model_2:
  oseries: head_mw
  stressmodels:
    recharge:
      class: RechargeModel
      prec: nearest
      evap: nearest evap
      recharge: Linear
      rfunc: Exponential
    wells:
      class: WellModel
      stress: nearest 2 well
      rfunc: HantushWellModel
      up: False
"""
```

[48]:
```
pstore.oseries
```

[48]:
```
                 x         y
name
oseries1  165000.0  424000.0
oseries2  164000.0  423000.0
oseries3  165554.0  422685.0
head_nb5  200000.0  450000.0
head_mw    85850.0  383362.0
```

[49]:
```
with tempyaml(full_nearest_yaml) as f:
    models = pstore.yaml.load(f)  # returns a list
```

```
INFO:pastastore.yaml_interface:Building model 'nearest_model_1' for oseries 'head_nb5'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='prec': 'prec_nb5'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='evap': 'evap_nb5'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'river'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='riv': riv_nb5
INFO:pastastore.yaml_interface:Building model 'nearest_model_2' for oseries 'head_mw'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='prec': 'prec_mw'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='evap': 'evap_mw'
```

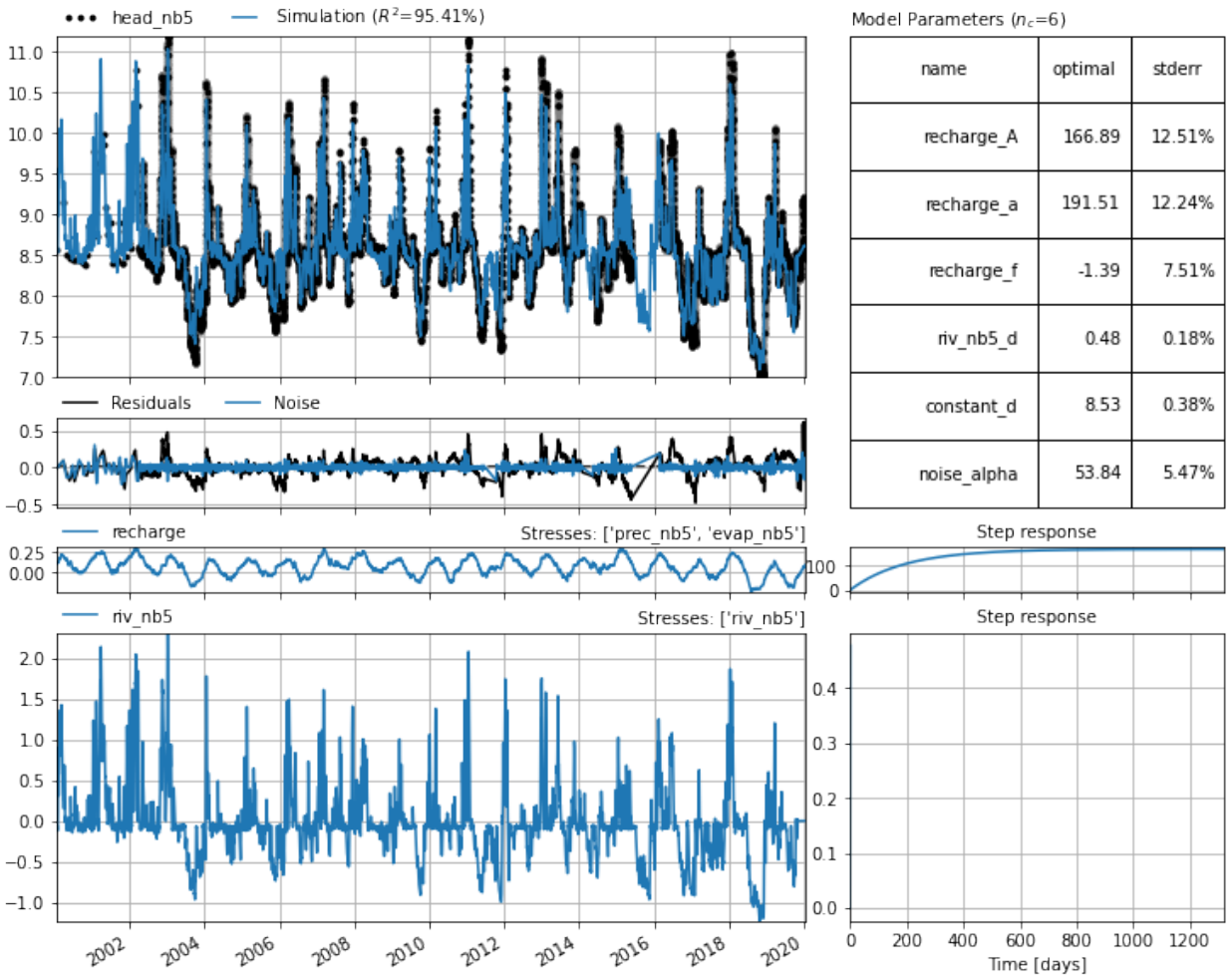
(continues on next page)

(continued from previous page)

```
INFO:pastastore.yaml_interface:| parsing stressmodel: 'wells'
INFO:pastastore.yaml_interface:  | using 2 nearest stress(es) with kind='well': [
↪'extraction_2' 'extraction_3']
```

[50]:
```
ml = models[0]
```

[51]:
```
ml.solve(report=False)
ml.plots.results()
```



name	optimal	stderr
recharge_A	166.89	12.51%
recharge_a	191.51	12.24%
recharge_f	-1.39	7.51%
riv_nb5_d	0.48	0.18%
constant_d	8.53	0.38%
noise_alpha	53.84	5.47%

Defaults

The Pastastore YAML interface adds some additional defaults as compared to pastas. These defaults allow the user to only provide only certain information in a YAML file in order to construct a model. These defaults are determined based on commonly used options. It should be noted that these defaults are not necessarily appropriate in all situations, and it is highly recommended to try different models with different options. These defaults are therefore implemented to facilitate building models, but should not be deemed holy.

The YAML interface mostly uses the Pastas defaults, but adds some additional logic for stressmodels. When default settings implemented in the YAML interface are implemented, this is logged to the console.

- **RechargeModel**:
  - If stressmodel name is one of "rch", "rech", "recharge", or "rechargemodel", assume stressmodel type is RechargeModel.
  - If no "prec" or "evap" keys are provided for RechargeModel, use the "nearest" option.
  - Default rfunc for RechargeModel is "Exponential".
  - prec: accepts `nearest` or `nearest <kind>`, if only `nearest` is provided, stresses in PastaStore must be labelled with kind="prec"
  - evap: accepts `nearest` or `nearest <kind>`, if only `nearest` is provided, stresses in PastaStore must be labelled with kind="evap"
- **StressModel**:
  - If no "stressmodel" key is contained in dictionary, assume stressmodel type is StressModel
  - Default rfunc for StressModel is "Gamma".
  - stress: accepts `nearest` or `nearest <kind>`, if only "nearest" is provided, uses whichever stress is nearest.
- **WellModel**:
  - Default rfunc for WellModel is "HantushWellModel".
  - If "up" is not provided, assume up=False, i.e. positive discharge time series indicates pumping.
  - stress: accepts `nearest`, `nearest <n>` and `nearest <n> <kind>`, where n is the number of wells to add. If kind is not passed, stresses must be labelled with kind="well" in PastaStore. If n is not passed, assumes n=1.

This is the shortest possible YAML file for a model with recharge, that makes use of all of the defaults for RechargeModel:

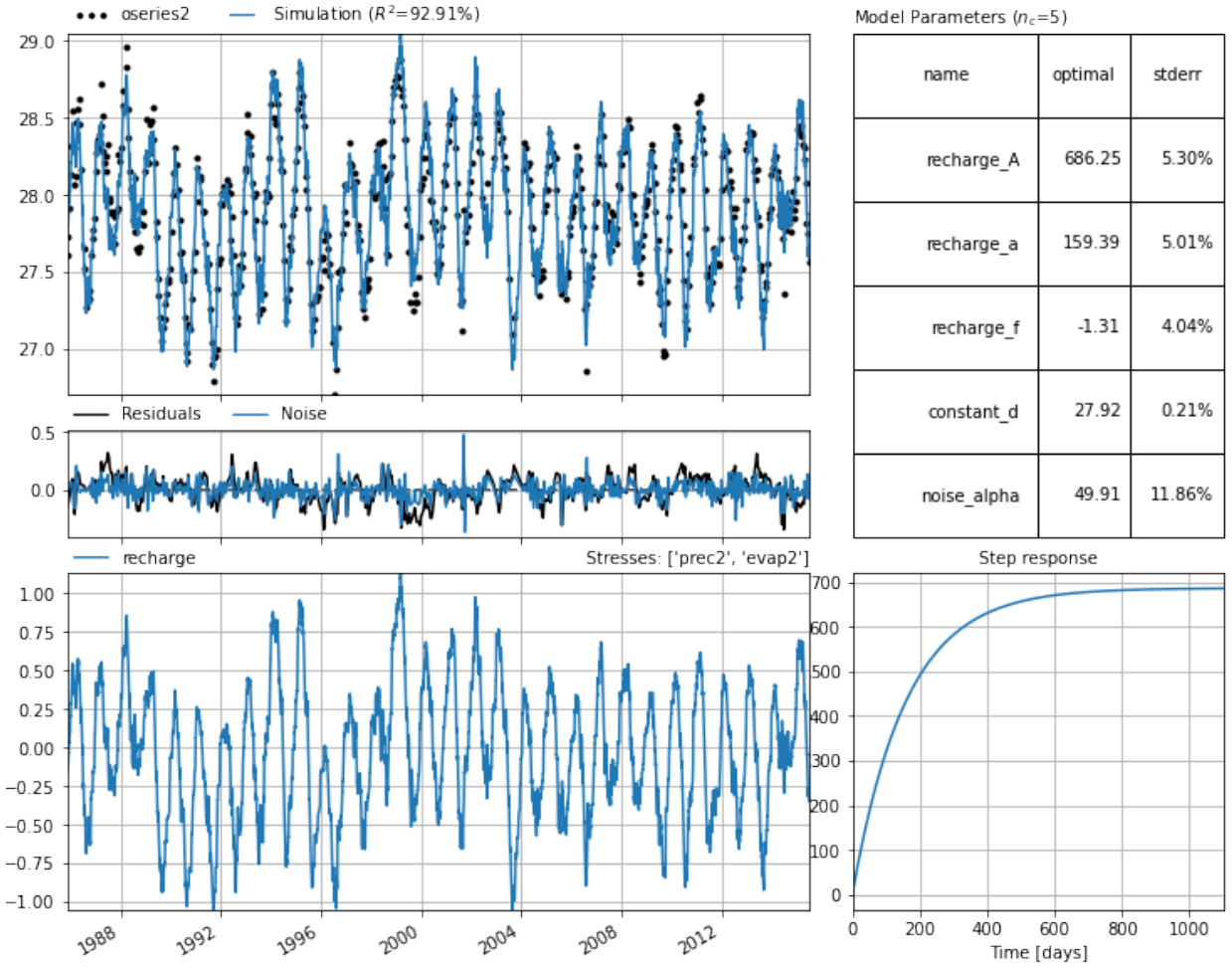
[52]:
```
minimal_yaml = """
ml_minimal:
  oseries: oseries2
  stressmodels:
    recharge:
"""
```

Note that the YAML load method recognizes the stressmodel name "recharge" and assumes the type of stress model should be RechargeModel. Additionally note the defaults as no other information is provided. - prec –> nearest stress with kind="prec" - evap –> nearest stress with kind="evap" - recharge –> Linear - rfunc –> Exponential

[53]:
```
with tempyaml(minimal_yaml) as f:
    ml = pstore.yaml.load(f)[0]  # returns a list
```

```
INFO:pastastore.yaml_interface:Building model 'ml_minimal' for oseries 'oseries2'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:| assuming RechargeModel based on stressmodel name.
INFO:pastastore.yaml_interface:  | using nearest stress with kind='prec': 'prec2'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='evap': 'evap2'
INFO:pastastore.yaml_interface:  | no 'rfunc' provided, using 'Exponential'
INFO:pastastore.yaml_interface:  | no 'recharge' type provided, using 'Linear'
```

[54]:
```
ml.solve(report=False)
ml.plots.results()
```



Model Parameters (n_c=5)

name	optimal	stderr
recharge_A	686.25	5.30%
recharge_a	159.39	5.01%
recharge_f	-1.31	4.04%
constant_d	27.92	0.21%
noise_alpha	49.91	11.86%

More examples

[55]:
```
yaml_examples = """
# Pastas YAML example file
# ------------------------

# 1. Explicitly provide oseries, stresses names rfunc and
#    recharge type.

ml_explicit:
  settings:
    freq: D
  oseries: oseries1
  stressmodels:
    recharge:
      class: RechargeModel
      prec: prec1
      evap: evap1
```

(continues on next page)

(continued from previous page)

```
      rfunc: Exponential
      recharge: Linear

# 2. Provide oseries, stresses names but use defaults for
#    other settings:

ml_stresses:
  oseries: oseries1
  stressmodels:
    recharge:
      prec: prec1
      evap: evap1

# 3. Use "nearest" to obtain nearest precipitation and evaporation
#    time series. Requires x, y data to be present in oseries and
#    stresses metadata.

ml_nearest:
  oseries: oseries1
  stressmodels:
    recharge:
      prec: nearest prec
      evap: nearest
"""
```

[56]:
```
with tempyaml(yaml_examples) as f:
    models = pstore.yaml.load(f)  # returns a list
```

```
INFO:pastastore.yaml_interface:Building model 'ml_explicit' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:Building model 'ml_stresses' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:| assuming RechargeModel based on stressmodel name.
INFO:pastastore.yaml_interface:  | no 'rfunc' provided, using 'Exponential'
INFO:pastastore.yaml_interface:  | no 'recharge' type provided, using 'Linear'
INFO:pastastore.yaml_interface:Building model 'ml_nearest' for oseries 'oseries1'
INFO:pastastore.yaml_interface:| parsing stressmodel: 'recharge'
INFO:pastastore.yaml_interface:| assuming RechargeModel based on stressmodel name.
INFO:pastastore.yaml_interface:  | using nearest stress with kind='prec': 'prec1'
INFO:pastastore.yaml_interface:  | using nearest stress with kind='evap': 'evap1'
INFO:pastastore.yaml_interface:  | no 'rfunc' provided, using 'Exponential'
INFO:pastastore.yaml_interface:  | no 'recharge' type provided, using 'Linear'
```

The first and last models are identical, except for the name obviously. The second one is also the same, but is not shown below.

[57]:
```
pst.util.compare_models(models[0], models[-1], detailed_comparison=True)
```

[57]:
```
                                      model 0             model 1  \
name:                               ml_explicit          ml_nearest
- settings: tmin                           None                None
- settings: tmax                           None                None
- settings: freq                              D                   D
```

(continues on next page)

(continued from previous page)

```
- settings: warmup               3650 days 00:00:00   3650 days 00:00:00
- settings: time_offset             0 days 00:00:00      0 days 00:00:00
- settings: noise                                True                 True
- settings: solver                               None                 None
- settings: fit_constant                         True                 True
oseries: series_original                         True                 True
oseries: series_series                           True                 True
stressmodel: 'recharge'                     recharge             recharge
- rfunc                                  Exponential          Exponential
- time series: 'prec1'                         prec1                prec1
  - prec1 settings: freq                           D                    D
  - prec1 settings: sample_up                  bfill                bfill
  - prec1 settings: sample_down                 mean                 mean
  - prec1 settings: fill_nan                     0.0                  0.0
  - prec1 settings: fill_before                 mean                 mean
  - prec1 settings: fill_after                  mean                 mean
  - prec1 settings: tmin          2010-01-01 00:00:00  2010-01-01 00:00:00
  - prec1 settings: tmax          2015-12-31 00:00:00  2015-12-31 00:00:00
  - prec1 settings: time_offset      0 days 00:00:00      0 days 00:00:00
  - prec1: series_original                       True                 True
  - prec1: series                                True                 True
- time series: 'evap1'                         evap1                evap1
  - evap1 settings: freq                           D                    D
  - evap1 settings: sample_up                  bfill                bfill
  - evap1 settings: sample_down                 mean                 mean
  - evap1 settings: fill_nan               interpolate          interpolate
  - evap1 settings: fill_before                 mean                 mean
  - evap1 settings: fill_after                  mean                 mean
  - evap1 settings: tmin          2010-01-01 00:00:00  2010-01-01 00:00:00
  - evap1 settings: tmax          2015-12-31 00:00:00  2015-12-31 00:00:00
  - evap1 settings: time_offset      0 days 00:00:00      0 days 00:00:00
  - evap1: series_original                       True                 True
  - evap1: series                                True                 True
param: recharge_A (init)                  211.567577           211.567577
param: recharge_A (opt)                          NaN                  NaN
param: recharge_a (init)                        10.0                 10.0
param: recharge_a (opt)                          NaN                  NaN
param: recharge_f (init)                        -1.0                 -1.0
param: recharge_f (opt)                          NaN                  NaN
param: constant_d (init)                  27.927937            27.927937
param: constant_d (opt)                          NaN                  NaN
param: noise_alpha (init)                       14.0                 14.0
param: noise_alpha (opt)                         NaN                  NaN

                                 comparison
name:                                 False
- settings: tmin                       True
- settings: tmax                       True
- settings: freq                       True
- settings: warmup                     True
- settings: time_offset                True
- settings: noise                      True
```

(continues on next page)

(continued from previous page)

```
- settings: solver                    True
- settings: fit_constant              True
oseries: series_original              True
oseries: series_series                True
stressmodel: 'recharge'               True
- rfunc                               True
- time series: 'prec1'                True
  - prec1 settings: freq               True
  - prec1 settings: sample_up          True
  - prec1 settings: sample_down        True
  - prec1 settings: fill_nan           True
  - prec1 settings: fill_before        True
  - prec1 settings: fill_after         True
  - prec1 settings: tmin               True
  - prec1 settings: tmax               True
  - prec1 settings: time_offset        True
  - prec1: series_original             True
  - prec1: series                      True
- time series: 'evap1'                True
  - evap1 settings: freq               True
  - evap1 settings: sample_up          True
  - evap1 settings: sample_down        True
  - evap1 settings: fill_nan           True
  - evap1 settings: fill_before        True
  - evap1 settings: fill_after         True
  - evap1 settings: tmin               True
  - evap1 settings: tmax               True
  - evap1 settings: time_offset        True
  - evap1: series_original             True
  - evap1: series                      True
param: recharge_A (init)               True
param: recharge_A (opt)                True
param: recharge_a (init)               True
param: recharge_a (opt)                True
param: recharge_f (init)               True
param: recharge_f (opt)                True
param: constant_d (init)               True
param: constant_d (opt)                True
param: noise_alpha (init)              True
param: noise_alpha (opt)               True
```

Clean up the written YAML files.

[58]:
```
for f in [fi for fi in os.listdir(".") if fi.endswith(".yaml")]:
    os.remove(f)
```

[ ]:
```
```

CHAPTER

THREE

USER GUIDE

This section describes the functionality of the module in more detail.

3.1 Connector objects

The structure and some background on the different types of Connectors is detailed below.

Each connector makes a distinction between the following datasets:

- observation time series (the series to be simulated)
- stresses time series (the forcing series on the system)
- models (the time series models)

3.1.1 In-memory

The *DictConnector* is a very simple object that stores all data and models in dictionaries. The data is stored in-memory and not on disk and is therefore not persistent, i.e. you cannot pick up where you left off last time. Once you exit Python your data is lost. For small projects, this connector can be useful as it is extremely simple and fast.

3.1.2 Pas-files

The *PasConnector* is an object that stores Pastas time series and models on disk as pas-files. These are JSON files (with a .pas extension) and make use of Pastas methods to store models on disk. There is no compression of files and the files are stored in directories on the harddrive which means all files are human-readable. The advantage of this Connector is that no external dependencies are required. The downside of this storage method is that it takes up more diskspace and is slower than the other Connectors.

The PasConnector uses the following structure:

```
+-- directory
|   +-- sub-directories (i.e. oseries, stresses, models)
|   |   +-- pas-files... (i.e. individual time series or models)
```

The data is stored within these sub-directories. Observations and stresses tim eseries are stored as JSON files. Models are stored as JSON as well but *do not* contain the time series themselves. These are picked up from the other directories when the model is loaded from the database.

3.1.3 ArcticDB

Note: this Connector uses ArcticDB the next-generation version of Arctic. Requires arcticdb Python package.

The *ArcticDBConnector* is an object that creates a local database. This can be an existing or a new database. For each of the datasets a collection or library is created. These are named using the following convention: *<database name>.<library name>*.

The ArcticDB implementation uses the following structure:

```
+-- database
|   +-- libraries (i.e. oseries, stresses, models)
|   |   +-- items... (i.e. individual time series or models)
```

The data is stored within these libraries. Observations and stresses time series are stored as pandas.DataFrames. Models are stored as pickled dictionaries and *do not* contain the time series themselves. These are picked up from the other libraries when the model is loaded from the database.

3.1.4 Arctic

Note: this Connector is not actively tested!

The *ArcticConnector* is an object that creates a connection with a MongoDB database. This can be an existing or a new database. For each of the datasets a collection or library is created. These are named using the following convention: *<database name>.<library name>*.

The Arctic implementation uses the following structure:

```
+-- database
|   +-- collections or libraries (i.e. oseries, stresses, models)
|   |   +-- documents... (i.e. individual time series or models)
```

The data is stored within these libraries. Observations and stresses time series are stored as pandas.DataFrames. Models are stored in JSON (actually binary JSON) and *do not* contain the time series themselves. These are picked up from the other libraries when the model is loaded from the database.

3.1.5 Pystore

Note: this Connector is not actively tested!

The *PystoreConnector* is an object that links to a location on disk. This can either be an existing or a new Pystore. A new store is created with collections (or libraries) that hold the different datasets:

- observation time series
- stresses time series
- models

The Pystores have the following structure:

```
+-- store
|   +-- collections or libraries... (i.e. oseries, stresses, models)
|   |   +-- items... (i.e. individual time series or models)
```

The time series data is stored as Dask DataFrames which can be easily converted to pandas DataFrames. The models are stored as JSON (not including the time series) in the metadata file belonging to an item. The actual data in the

item is an empty DataFrame serving as a placeholder. This slightly 'hacky' design allows the models to be saved in a PyStore. The time series are picked up from their respective stores when the model is loaded from disk.

3.1.6 Custom Connectors

It should be relatively straightforward to write your own custom connector object. The *Base* submodule contains the *BaseConnector* class that defines which methods and properties *must* be defined. The *ConnectorUtil* mix-in class contains some general methods that are used by each connector. Each Connector object should inherit from these two classes.

The *BaseConnector* class also shows the expected call signature for each method. Following the same call signature should ensure that your new connector works directly with *PastaStore*. Extra keyword arguments can be added in the custom class.

Below is a small snippet showing a custom Connector class:

```
class MyCustomConnector(BaseConnector, ConnectorUtil):
    """Must override each method and property in BaseConnector, e.g."""

    def _get_item(self, name, progressbar=False):
        # your code here for getting an item from your database
        pass
```

3.2 PastaStore object

The *PastaStore* object is essentially a class for working with time series and pastas Models. A Connector has to be passed to the object which manages the retrieval and storage of data.

Methods are available for the following tasks:

- Calculating distances between locations of time series, i.e. getting the nearest time series to a location:

```
# get 3 nearest oseries
store.get_nearest_oseries("my_oseries", n=3)

# get nearest precipitation series
store.get_nearest_stress("my_oseries", kind="prec")
```

- Creating pastas Models, optionally adding a recharge stressmodel:

```
# create model
ml = store.create_model("my_oseries", add_recharge=True)
```

Bulk operations are also provided for:

- Creating and storing pastas Models:

```
# create models and store in database
store.create_models(add_recharge=True, store=True)
```

- Optimizing pastas Models and storing the results:

```
# solve models and store result in database
store.solve_models(ignore_solver_errors=True, store_result=True)
```

3.3 Utilities

The *pastastore.util* submodule contains useful functions, i.e. for deleting databases, connector objects, and PastaStore objects, emptying a library of all its contents or copying all data to a new database:

- *pastastore.util.delete_pastastore()*
- pastastore.util.delete_dict_connector()
- pastastore.util.delete_pas_connector()
- *pastastore.util.delete_pystore_connector()*
- *pastastore.util.delete_arctic_connector()*
- *pastastore.util.delete_arcticdb_connector()*
- *pastastore.util.copy_database()*

It also contains a method for making a detailed comparison between two pastas.Models:

- *pastastore.util.compare_models()*

CHAPTER

FOUR

API DOCUMENTATION

4.1 Connectors

4.1.1 Base

class pastastore.base.**BaseConnector**

Base Connector class.

Class holds base logic for dealing with time series and Pastas Models. Create your own Connector to a data source by writing a a class that inherits from this BaseConnector. Your class has to override each abstractmethod and abstractproperty.

CHECK_MODEL_SERIES_VALUES = True

USE_PASTAS_VALIDATE_SERIES = True

abstract _add_item(*libname: str*, *item: DataFrame | Series | Dict*, *name: str*, *metadata: Dict | None = None*, *overwrite: bool = False*) → None

Internal method to add item for both time series and pastas.Models.

Must be overriden by subclass.

Parameters

- **libname** (*str*) – name of library to add item to
- **item** (*FrameorSeriesUnion or dict*) – item to add
- **name** (*str*) – name of the item
- **metadata** (*dict, optional*) – dictionary containing metadata, by default None

_add_oseries_model_links(*onam: str*, *mlnames: str | List[str]*)

Add model name to stored list of models per oseries.

Parameters

- **onam** (*str*) – name of oseries
- **mlnames** (*Union[str, List[str]]*) – model name or list of model names for an oseries with name onam.

_add_series(*libname: str*, *series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*, *validate: bool | None = None*, *overwrite: bool = False*) → None

Internal method to add series to database.

Parameters

- **libname** (*str*) – name of the library to add the series to
- **series** (*pandas.Series or pandas.DataFrame*) – data to add
- **name** (*str*) – name of the time series
- **metadata** (*dict, optional*) – dictionary containing metadata, by default None
- **validate** (*bool, optional*) – use pastas to validate series, default is None, which will use the USE_PASTAS_VALIDATE_SERIES value (default is True).
- **overwrite** (*bool, optional*) – overwrite existing dataset with the same name, by default False

Raises
ItemInLibraryException – if overwrite is False and name is already in the database

static _clear_cache(*libname: str*) → None

Clear cached property.

_default_library_names = ['oseries', 'stresses', 'models', 'oseries_models']

abstract _del_item(*libname: str*, *name: str*) → None

Internal method to delete items (series or models).

Must be overriden by subclass.

Parameters

- **libname** (*str*) – name of library to delete item from
- **name** (*str*) – name of item to delete

_del_oseries_model_link(*onam*, *mlnam*)

Delete model name from stored list of models per oseries.

Parameters

- **onam** (*str*) – name of oseries
- **mlnam** (*str*) – name of model

_get_all_oseries_model_links()

Get all model names per oseries in dictionary.

Returns
links – dictionary with oseries names as keys and lists of model names as values

Return type
dict

abstract _get_item(*libname: str*, *name: str*) → DataFrame | Series | Dict

Internal method to get item (series or pastas.Models).

Must be overriden by subclass.

Parameters

- **libname** (*str*) – name of library
- **name** (*str*) – name of item

Returns
item – item (time series or pastas.Model)

Return type
FrameorSeriesUnion or dict

abstract _get_library(*libname: str*)

Get library handle.

Must be overriden by subclass.

Parameters
libname (*str*) – name of the library

Returns
lib – handle to the library

Return type
Any

abstract _get_metadata(*libname: str*, *name: str*) → Dict

Internal method to get metadata.

Must be overriden by subclass.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
metadata – dictionary containing metadata

Return type
dict

_get_series(*libname: str*, *names: list | str*, *progressbar: bool = True*, *squeeze: bool = True*) → DataFrame | Series

Internal method to get time series.

Parameters

- **libname** (*str*) – name of the library
- **names** (*str or list of str*) – names of the time series to load
- **progressbar** (*bool, optional*) – show progressbar, by default True
- **squeeze** (*bool, optional*) – if True return DataFrame or Series instead of dictionary for single entry

Returns
either returns time series as pandas.DataFrame or dictionary containing the time series.

Return type
pandas.DataFrame or dict of pandas.DataFrames

_iter_series(*libname: str*, *names: List[str] | None = None*)

Internal method iterate over time series in library.

Parameters

- **libname** (*str*) – name of library (e.g. 'oseries' or 'stresses')
- **names** (*Optional[List[str]], optional*) – list of names, by default None, which defaults to all stored series

Yields
pandas.Series or pandas.DataFrame – time series contained in library

property _modelnames_cache

List of model names.

static _parse_series_input(*series: DataFrame | Series*, *metadata: Dict | None = None*) → Tuple[DataFrame | Series, Dict | None]

Internal method to parse series input.

Parameters

- **series** (*FrameorSeriesUnion,*) – series object to parse
- **metadata** (*dict, optional*) – metadata dictionary or None, by default None

Returns
series, metadata – time series as pandas.Series or DataFrame and optionally metadata dictionary

Return type
FrameorSeriesUnion, Optional[Dict]

_pastas_validate(*validate*)

Whether to validate time series.

Parameters
validate (*bool, NoneType*) – value of validate keyword argument

Returns
b – return global or local setting (True or False)

Return type
bool

_update_all_oseries_model_links()

Add all model names to oseries metadata dictionaries.

Used for old PastaStore versions, where relationship between oseries and models was not stored. If there are any models in the database and if the oseries_models library is empty, loops through all models to determine which oseries each model belongs to.

_update_series(*libname: str*, *series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*, *validate: bool | None = None*) → None

Internal method to update time series.

Parameters

- **libname** (*str*) – name of library
- **series** (*FrameorSeriesUnion*) – time series containing update values
- **name** (*str*) – name of the time series to update
- **metadata** (*Optional[dict], optional*) – optionally provide metadata dictionary which will also update the current stored metadata dictionary, by default None
- **validate** (*bool, optional*) – use pastas to validate series, default is None, which will use the USE_PASTAS_VALIDATE_SERIES value (default is True).

_upsert_series(*libname: str*, *series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*, *validate: bool | None = None*) → None

Update or insert series depending on whether it exists in store.

Parameters

- **libname** (*str*) – name of library
- **series** (*FrameorSeriesUnion*) – time series to update/insert
- **name** (*str*) – name of the time series
- **metadata** (*Optional[dict], optional*) – metadata dictionary, by default None
- **validate** (*bool, optional*) – use pastas to validate series, default is None, which will use the USE_PASTAS_VALIDATE_SERIES value (default is True).

add_model(*ml: Model | dict*, *overwrite: bool = False*, *validate_metadata: bool = False*) → None

Add model to the database.

Parameters

- **ml** (*pastas.Model or dict*) – pastas Model or dictionary to add to the database
- **overwrite** (*bool, optional*) – if True, overwrite existing model, by default False
- **validate_metadata** – remove unsupported characters from metadata dictionary keys
- **optional** (*bool*) – remove unsupported characters from metadata dictionary keys

Raises

- **TypeError** – if model is not pastas.Model or dict
- *ItemInLibraryException* – if overwrite is False and model is already in the database

add_oseries(*series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*, *validate: bool | None = None*, *overwrite: bool = False*) → None

Add oseries to the database.

Parameters

- **series** (*pandas.Series or pandas.DataFrame*) – data to add
- **name** (*str*) – name of the time series
- **metadata** (*dict, optional*) – dictionary containing metadata, by default None.
- **validate** (*bool, optional*) – use pastas to validate series, default is None, which will use the USE_PASTAS_VALIDATE_SERIES value (default is True).
- **overwrite** (*bool, optional*) – overwrite existing dataset with the same name, by default False

add_stress(*series: DataFrame | Series*, *name: str*, *kind: str*, *metadata: dict | None = None*, *validate: bool | None = None*, *overwrite: bool = False*) → None

Add stress to the database.

Parameters

- **series** (*pandas.Series or pandas.DataFrame*) – data to add, if pastas.Timeseries is passed, series_orignal and metadata is stored in database
- **name** (*str*) – name of the time series
- **kind** (*str*) – category to identify type of stress, this label is added to the metadata dictionary.
- **metadata** (*dict, optional*) – dictionary containing metadata, by default None.
- **validate** (*bool, optional*) – use pastas to validate series, default is True

- **overwrite** (*bool, optional*) – overwrite existing dataset with the same name, by default False

del_models(*names: list | str*) → None

Delete model(s) from the database.

Parameters
names (*str or list of str*) – name(s) of the model to delete

del_oseries(*names: list | str*, *remove_models: bool = False*)

Delete oseries from the database.

Parameters

- **names** (*str or list of str*) – name(s) of the oseries to delete
- **remove_models** (*bool, optional*) – also delete models for deleted oseries, default is False

del_stress(*names: list | str*)

Delete stress from the database.

Parameters
names (*str or list of str*) – name(s) of the stress to delete

empty_library(*libname: str*, *prompt: bool = True*, *progressbar: bool = True*)

Empty library of all its contents.

Parameters

- **libname** (*str*) – name of the library
- **prompt** (*bool, optional*) – prompt user for input before deleting contents, by default True. Default answer is "n", user must enter 'y' to delete contents
- **progressbar** (*bool, optional*) – show progressbar, by default True

get_metadata(*libname: str*, *names: list | str*, *progressbar: bool = False*, *as_frame: bool = True*, *squeeze: bool = True*) → dict | DataFrame

Read metadata from database.

Parameters

- **libname** (*str*) – name of the library containing the dataset
- **names** (*str or list of str*) – names of the datasets for which to read the metadata
- **squeeze** (*bool, optional*) – if True return dict instead of list of dict for single entry

Returns
returns metadata dictionary or DataFrame of metadata

Return type
dict or pandas.DataFrame

get_models(*names: list | str*, *return_dict: bool = False*, *progressbar: bool = False*, *squeeze: bool = True*, *update_ts_settings: bool = False*) → Model | list

Load models from database.

Parameters

- **names** (*str or list of str*) – names of the models to load

- **return_dict** (*bool, optional*) – return model dictionary instead of pastas.Model (much faster for obtaining parameters, for example)
- **progressbar** (*bool, optional*) – show progressbar, by default False
- **squeeze** (*bool, optional*) – if True return Model instead of list of Models for single entry
- **update_ts_settings** (*bool, optional*) – update time series settings based on time series in store. overwrites stored tmin/tmax in model.

Returns
return pastas model, or list of models if multiple names were passed

Return type
pastas.Model or list of pastas.Model

get_oseries(*names: list | str*, *return_metadata: bool = False*, *progressbar: bool = False*, *squeeze: bool = True*) → DataFrame | Series | Dict | List | None

Get oseries from database.

Parameters

- **names** (*str or list of str*) – names of the oseries to load
- **return_metadata** (*bool, optional*) – return metadata as dictionary or list of dictionaries, default is False
- **progressbar** (*bool, optional*) – show progressbar, by default False
- **squeeze** (*bool, optional*) – if True return DataFrame or Series instead of dictionary for single entry

Returns

- **oseries** (*pandas.DataFrame or dict of DataFrames*) – returns time series as DataFrame or dictionary of DataFrames if multiple names were passed
- **metadata** (*dict or list of dict*) – metadata for each oseries, only returned if return_metadata=True

get_stresses(*names: list | str*, *return_metadata: bool = False*, *progressbar: bool = False*, *squeeze: bool = True*) → DataFrame | Series | Dict | List | None

Get stresses from database.

Parameters

- **names** (*str or list of str*) – names of the stresses to load
- **return_metadata** (*bool, optional*) – return metadata as dictionary or list of dictionaries, default is False
- **progressbar** (*bool, optional*) – show progressbar, by default False
- **squeeze** (*bool, optional*) – if True return DataFrame or Series instead of dictionary for single entry

Returns

- **stresses** (*pandas.DataFrame or dict of DataFrames*) – returns time series as DataFrame or dictionary of DataFrames if multiple names were passed
- **metadata** (*dict or list of dict*) – metadata for each stress, only returned if return_metadata=True

iter_models(*modelnames: List[str] | None = None*, *return_dict: bool = False*)

Iterate over models in library.

Parameters

- **modelnames** (*Optional[List[str]], optional*) – list of models to iterate over, by default None which uses all models
- **return_dict** (*bool, optional*) – if True, return model as dictionary, by default False, which returns a pastas.Model.

Yields

pastas.Model or dict – time series model

iter_oseries(*names: List[str] | None = None*)

Iterate over oseries in library.

Parameters

names (*Optional[List[str]], optional*) – list of oseries names, by default None, which defaults to all stored series

Yields

pandas.Series or pandas.DataFrame – oseries contained in library

iter_stresses(*names: List[str] | None = None*)

Iterate over stresses in library.

Parameters

names (*Optional[List[str]], optional*) – list of stresses names, by default None, which defaults to all stored series

Yields

pandas.Series or pandas.DataFrame – stresses contained in library

abstract property model_names

List of model names.

Property must be overriden by subclass.

property n_models

property n_oseries

property n_stresses

property oseries

Dataframe with overview of oseries.

property oseries_models

List of model names per oseries.

Returns

d – dictionary with oseries names as keys and list of model names as values

Return type

dict

abstract property oseries_names

List of oseries names.

Property must be overriden by subclass.

set_check_model_series_values(*b: bool*)

Turn CHECK_MODEL_SERIES_VALUES option on (True) or off (False).

The default option is on (it is highly recommended to keep it that way). When turned on, the model time series (ml.oseries._series_original, and stressmodel.stress._series_original) values are checked against the stored copies in the database. If these do not match, an error is raised, and the model is not added to the database. This guarantees the stored model will be identical after loading from the database. This check is somewhat computationally expensive, which is why it can be turned on or off.

Parameters

b (*bool*) – boolean indicating whether option should be turned on (True) or off (False). Option is on by default.

set_use_pastas_validate_series(*b: bool*)

Turn USE_PASTAS_VALIDATE_SERIES option on (True) or off (False).

This will use pastas.validate_oseries() or pastas.validate_stresses() to test the time series. If they do not meet the criteria, an error is raised. Turning this option off will allow the user to store any time series but this will mean that time series models cannot be made from stored time series directly and will have to be modified before building the models. This in turn will mean that storing the models will not work as the stored time series copy is checked against the time series in the model to check if they are equal.

Note: this option requires pastas>=0.23.0, otherwise it is turned off.

Parameters

b (*bool*) – boolean indicating whether option should be turned on (True) or off (False). Option is on by default.

property stresses

Dataframe with overview of stresses.

abstract property stresses_names

List of stresses names.

Property must be overriden by subclass.

update_metadata(*libname: str*, *name: str*, *metadata: dict*) → None

Update metadata.

Note: also retrieves and stores time series as updating only metadata is not supported for some Connectors.

Parameters

- **libname** (*str*) – name of library
- **name** (*str*) – name of the item for which to update metadata
- **metadata** (*dict*) – metadata dictionary that will be used to update the stored metadata

update_oseries(*series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*) → None

Update oseries values.

Parameters

- **series** (*FrameorSeriesUnion*) – time series to update stored oseries with
- **name** (*str*) – name of the oseries to update
- **metadata** (*Optional[dict], optional*) – optionally provide metadata, which will update the stored metadata dictionary, by default None

update_stress(*series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*) → None

Update stresses values.

Note: the 'kind' attribute of a stress cannot be updated! To update the 'kind' delete and add the stress again.

Parameters

- **series** (*FrameorSeriesUnion*) – time series to update stored stress with
- **name** (*str*) – name of the stress to update
- **metadata** (*Optional[dict], optional*) – optionally provide metadata, which will update the stored metadata dictionary, by default None

upsert_oseries(*series: DataFrame | Series*, *name: str*, *metadata: dict | None = None*) → None

Update or insert oseries values depending on whether it exists.

Parameters

- **series** (*FrameorSeriesUnion*) – time series to update/insert
- **name** (*str*) – name of the oseries
- **metadata** (*Optional[dict], optional*) – optionally provide metadata, which will update the stored metadata dictionary if it exists, by default None

upsert_stress(*series: DataFrame | Series*, *name: str*, *kind: str*, *metadata: dict | None = None*) → None

Update or insert stress values depending on whether it exists.

Parameters

- **series** (*FrameorSeriesUnion*) – time series to update/insert
- **name** (*str*) – name of the stress
- **metadata** (*Optional[dict], optional*) – optionally provide metadata, which will update the stored metadata dictionary if it exists, by default None

class pastastore.base.**ConnectorUtil**

Mix-in class for general Connector helper functions.

Only for internal methods, and not methods that are related to CRUD operations on database.

static _check_model_series_names_for_store(*ml*)

_check_oseries_in_store(*ml: Model | dict*)

Internal method, check if Model oseries are contained in PastaStore.

Parameters
ml (*Union[ps.Model, dict]*) – pastas Model

_check_stresses_in_store(*ml: Model | dict*)

Internal method, check if stresses time series are contained in PastaStore.

Parameters
ml (*Union[ps.Model, dict]*) – pastas Model

static _check_stressmodels_supported(*ml*)

_get_model_orphans()

Get models whose oseries no longer exist in database.

Returns
dictionary with oseries names as keys and lists of model names as values

Return type
dict

`static _meta_list_to_frame(metalist: list, names: list)`

Convert list of metadata dictionaries to DataFrame.

Parameters

- **metalist** (*list*) – list of metadata dictionaries
- **names** (*list*) – list of names corresponding to data in metalist

Returns
DataFrame containing overview of metadata

Return type
pandas.DataFrame

`static _metadata_from_json(fjson: str)`

Load metadata dictionary from JSON.

Parameters
fjson (*str*) – path to file

Returns
meta – dictionary containing metadata

Return type
dict

`_models_to_archive(archive, names=None, progressbar=True)`

Internal method for writing pastas.Model to zipfile.

Parameters

- **archive** (*zipfile.ZipFile*) – reference to an archive to write data to
- **names** (*str or list of str, optional*) – names of the models to write to archive, by default None, which writes all models to archive
- **progressbar** (*bool, optional*) – show progressbar, by default True

`_parse_model_dict(mdict: dict, update_ts_settings: bool = False)`

Internal method to parse dictionary describing pastas models.

Parameters

- **mdict** (*dict*) – dictionary describing pastas.Model
- **update_ts_settings** (*bool, optional*) – update stored tmin and tmax in time series settings based on time series loaded from store.

Returns
ml – time series analysis model

Return type
pastas.Model

`_parse_names(names: list | str | None = None, libname: str | None = 'oseries') → list`

Internal method to parse names kwarg, returns iterable with name(s).

Parameters

- **names** (*Union[list, str], optional*) – str or list of str or None or 'all' (last two options retrieves all names)

- **libname** (*str, optional*) – name of library, default is 'oseries'

Returns
list of names

Return type
list

static _series_from_json(*fjson: str*)

Load time series from JSON.

Parameters
fjson (*str*) – path to file

Returns
s – DataFrame containing time series

Return type
pd.DataFrame

_series_to_archive(*archive*, *libname: str*, *names: list | str | None = None*, *progressbar: bool = True*)

Internal method for writing DataFrame or Series to zipfile.

Parameters

- **archive** (*zipfile.ZipFile*) – reference to an archive to write data to
- **libname** (*str*) – name of the library to write to zipfile
- **names** (*str or list of str, optional*) – names of the time series to write to archive, by default None, which writes all time series to archive
- **progressbar** (*bool, optional*) – show progressbar, by default True

static _set_series_name(*series*, *name*)

Set series name to match user defined name in store.

Parameters

- **series** (*pandas.Series or pandas.DataFrame*) – set name for this time series
- **name** (*str*) – name of the time series (used in the pastastore)

_stored_metadata_to_json(*libname: str*, *names: list | str | None = None*, *squeeze: bool = True*, *progressbar: bool = False*)

Write metadata from stored series to JSON.

Parameters

- **libname** (*str*) – library containing series
- **names** (*Optional[Union[list, str]], optional*) – names to parse, by default None
- **squeeze** (*bool, optional*) – return single entry as json string instead of list, by default True
- **progressbar** (*bool, optional*) – show progressbar, by default False

Returns
files – list of json string

Return type
list or str

_stored_series_to_json(*libname: str*, *names: list | str | None = None*, *squeeze: bool = True*, *progressbar: bool = False*)

Write stored series to JSON.

Parameters

- **libname** (*str*) – library name
- **names** (*Optional[Union[list, str]], optional*) – names of series, by default None
- **squeeze** (*bool, optional*) – return single entry as json string instead of list, by default True
- **progressbar** (*bool, optional*) – show progressbar, by default False

Returns

files – list of series converted to JSON string or single string if single entry is returned and squeeze is True

Return type

list or str

static _validate_input_series(*series*)

check if series is pandas.DataFrame or pandas.Series.

Parameters

series (*object*) – object to validate

Raises

TypeError – if object is not of type pandas.DataFrame or pandas.Series

class pastastore.base.**ModelAccessor**(*conn*)

Object for managing access to stored models.

Provides dict-like access to models (i.e. PastaStore.models["model1"]), or allows adding models to the PastaStore using dict-like assignment (i.e. PastaStore.models["model1"] = ml), and it can serve as an iterator (i.e. [ml for ml in pstore.models]).

random()

4.1.2 DictConnector

class pastastore.**DictConnector**(*name: str = 'pastas_db'*)

Bases: *BaseConnector*, *ConnectorUtil*

_add_item(*libname: str*, *item: DataFrame | Series | Dict*, *name: str*, *metadata: Dict | None = None*, ***_*) → None

Internal method to add item (time series or models).

Parameters

- **libname** (*str*) – name of library
- **item** (*FrameorSeriesUnion*) – pandas.Series or pandas.DataFrame containing data
- **name** (*str*) – name of the item
- **metadata** (*dict, optional*) – dictionary containing metadata, by default None

_del_item(*libname: str*, *name: str*) → None

Internal method to delete items (series or models).

Parameters

- **libname** (*str*) – name of library to delete item from
- **name** (*str*) – name of item to delete

_get_item(*libname: str*, *name: str*) → DataFrame | Series | Dict

Internal method to retrieve item from pystore library.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
item – time series or model dictionary

Return type
Union[FrameorSeriesUnion, Dict]

_get_library(*libname: str*)

Get reference to dictionary holding data.

Parameters
libname (*str*) – name of the library

Returns
lib – library handle

Return type
dict

_get_metadata(*libname: str*, *name: str*) → dict

Internal method to read metadata.

Parameters

- **libname** (*str*) – name of the library the series are in ("oseries" or "stresses")
- **name** (*str*) – name of item to load metadata for

Returns
imeta – dictionary containing metadata

Return type
dict

property model_names

List of model names.

property oseries_names

List of oseries names.

property oseries_with_models

List of oseries with models.

property stresses_names

List of stresses names.

4.1.3 PasConnector

class pastastore.**PasConnector**(*name: str*, *path: str*)

Bases: *BaseConnector*, *ConnectorUtil*

_add_item(*libname: str*, *item: DataFrame | Series | Dict*, *name: str*, *metadata: Dict | None = None*, **_) → None

Internal method to add item (time series or models).

Parameters

- **libname** (*str*) – name of library
- **item** (*FrameorSeriesUnion*) – pandas.Series or pandas.DataFrame containing data
- **name** (*str*) – name of the item
- **metadata** (*dict, optional*) – dictionary containing metadata, by default None

_del_item(*libname: str*, *name: str*) → None

Internal method to delete items (series or models).

Parameters

- **libname** (*str*) – name of library to delete item from
- **name** (*str*) – name of item to delete

_get_item(*libname: str*, *name: str*) → DataFrame | Series | Dict

Internal method to retrieve item.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
item – time series or model dictionary

Return type
Union[FrameorSeriesUnion, Dict]

_get_library(*libname: str*)

Get path to directory holding data.

Parameters
libname (*str*) – name of the library

Returns
lib – path to library

Return type
str

_get_metadata(*libname: str*, *name: str*) → dict

Internal method to read metadata.

Parameters

- **libname** (*str*) – name of the library the series are in ("oseries" or "stresses")
- **name** (*str*) – name of item to load metadata for

Returns
imeta – dictionary containing metadata

Return type
dict

_initialize() → None

Internal method to initialize the libraries.

property model_names

List of model names.

property oseries_names

List of oseries names.

property oseries_with_models

List of oseries with models.

property stresses_names

List of stresses names.

4.1.4 ArcticDBConnector

class pastastore.**ArcticDBConnector**(*name: str*, *uri: str*)

Bases: *BaseConnector*, *ConnectorUtil*

_abc_impl = <_abc._abc_data object>

_add_item(*libname: str*, *item: DataFrame | Series | Dict*, *name: str*, *metadata: Dict | None = None*, **_) → None

Internal method to add item to library (time series or model).

Parameters

- **libname** (*str*) – name of the library
- **item** (*Union[FrameorSeriesUnion, Dict]*) – item to add, either time series or pastas.Model as dictionary
- **name** (*str*) – name of the item
- **metadata** (*Optional[Dict], optional*) – dictionary containing metadata, by default None

_del_item(*libname: str*, *name: str*) → None

Internal method to delete items (series or models).

Parameters

- **libname** (*str*) – name of library to delete item from
- **name** (*str*) – name of item to delete

_get_item(*libname: str*, *name: str*) → DataFrame | Series | Dict

Internal method to retrieve item from library.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
item – time series or model dictionary

Return type
Union[FrameorSeriesUnion, Dict]

`_get_library`(*libname: str*)

Get ArcticDB library handle.

Parameters
libname (*str*) – name of the library

Returns
lib – handle to the library

Return type
arcticdb.Library handle

`_get_metadata`(*libname: str*, *name: str*) → dict

Internal method to retrieve metadata for an item.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
dictionary containing metadata

Return type
dict

`_initialize`() → None

Internal method to initalize the libraries.

`_library_name`(*libname: str*) → str

Internal method to get full library name according to ArcticDB.

`conn_type = 'arcticdb'`

`property model_names`

List of model names.

Returns
list of models in library

Return type
list

`property oseries_names`

List of oseries names.

Returns
list of oseries in library

Return type
list

`property oseries_with_models`

List of oseries with models.

property stresses_names

List of stresses names.

Returns
list of stresses in library

Return type
list

4.1.5 ArcticConnector

class pastastore.**ArcticConnector**(*name: str*, *connstr: str*)

Bases: *BaseConnector*, *ConnectorUtil*

_abc_impl = <_abc._abc_data object>

_add_item(*libname: str*, *item: DataFrame | Series | Dict*, *name: str*, *metadata: Dict | None = None*, **_) → None

Internal method to add item to library (time series or model).

Parameters

- **libname** (*str*) – name of the library
- **item** (*Union[FrameorSeriesUnion, Dict]*) – item to add, either time series or pastas.Model as dictionary
- **name** (*str*) – name of the item
- **metadata** (*Optional[Dict], optional*) – dictionary containing metadata, by default None

_del_item(*libname: str*, *name: str*) → None

Internal method to delete items (series or models).

Parameters

- **libname** (*str*) – name of library to delete item from
- **name** (*str*) – name of item to delete

_get_item(*libname: str*, *name: str*) → DataFrame | Series | Dict

Internal method to retrieve item from library.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
item – time series or model dictionary

Return type
Union[FrameorSeriesUnion, Dict]

_get_library(*libname: str*)

Get Arctic library handle.

Parameters
libname (*str*) – name of the library

Returns
lib – handle to the library

Return type
arctic.Library handle

`_get_metadata`(*libname: str*, *name: str*) → dict

Internal method to retrieve metadata for an item.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
dictionary containing metadata

Return type
dict

`_initialize`() → None

Internal method to initalize the libraries.

`_library_name`(*libname: str*) → str

Internal method to get full library name according to Arctic.

`conn_type = 'arctic'`

`property model_names`

List of model names.

Returns
list of models in library

Return type
list

`property oseries_names`

List of oseries names.

Returns
list of oseries in library

Return type
list

`property oseries_with_models`

List of oseries with models.

`property stresses_names`

List of stresses names.

Returns
list of stresses in library

Return type
list

4.1.6 PystoreConnector

class pastastore.**PystoreConnector**(*name: str*, *path: str*)

Bases: *BaseConnector*, *ConnectorUtil*

_abc_impl = <_abc._abc_data object>

_add_item(*libname: str*, *item: DataFrame | Series | Dict*, *name: str*, *metadata: Dict | None = None*, *overwrite: bool = False*) → None

Internal method to add item to library (time series or model).

Parameters

- **libname** (*str*) – name of the library
- **item** (*Union[FrameorSeriesUnion, Dict]*) – item to add, either time series or pastas.Model as dictionary
- **name** (*str*) – name of the item
- **metadata** (*Optional[Dict], optional*) – dictionary containing metadata, by default None
- **overwrite** (*bool, optional*) – overwrite item if it already exists, by default False.

_del_item(*libname: str*, *name: str*) → None

Internal method to delete data from the store.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item to delete

_get_item(*libname: str*, *name: str*) → DataFrame | Series | Dict

Internal method to retrieve item from pystore library.

Parameters

- **libname** (*str*) – name of the library
- **name** (*str*) – name of the item

Returns
item – time series or model dictionary

Return type
Union[FrameorSeriesUnion, Dict]

_get_library(*libname: str*)

Get Pystore library handle.

Parameters
libname (*str*) – name of the library

Returns
handle to the library

Return type
Pystore.Collection handle

_get_metadata(*libname: str*, *name: str*) → dict

Internal method to read metadata from pystore.

Parameters

- **libname** (*str*) – name of the library the series are in ("oseries" or "stresses")
- **name** (*str*) – name of item to load metadata for

Returns
imeta – dictionary containing metadata

Return type
dict

_initialize() → None

Internal method to initalize the libraries (stores).

conn_type = 'pystore'

property model_names

List of model names.

Returns
list of models in library

Return type
list

property oseries_names

List of oseries names.

Returns
list of oseries in library

Return type
list

property oseries_with_models

List of oseries with models.

property stresses_names

List of stresses names.

Returns
list of stresses in library

Return type
list

4.2 PastaStore

class pastastore.store.**PastaStore**(*connector:* BaseConnector | *None* = *None*, *name: str* | *None* = *None*)

PastaStore object for managing pastas time series and models.

Requires a Connector object to provide the interface to the database. Different Connectors are available, e.g.:

- PasConnector for storing all data as .pas (JSON) files on disk (recommended)
- DictConenctor for storing all data in dictionaries (in-memory)

- ArcticConnector for saving data to MongoDB using the Arctic module
- PystoreConnector for saving data to disk using the Pystore module

Parameters

- **connector** (*Connector object*) – object that provides the interface to the database, e.g. ArcticConnector (see pastastore.connectors)
- **name** (*str, optional*) – name of the PastaStore, by default takes the name of the Connector object

add_recharge(*ml: Model*, *rfunc=None*, *recharge=None*, *recharge_name: str = 'recharge'*) → None

Add recharge to a pastas model.

Uses closest precipitation and evaporation time series in database. These are assumed to be labeled with kind = 'prec' or 'evap'.

Parameters

- **ml** (*pastas.Model*) – pastas.Model object
- **rfunc** (*pastas.rfunc, optional*) – response function to use for recharge in model, by default None which uses ps.Exponential() (for different response functions, see pastas documentation)
- **recharge** (*ps.RechargeModel*) – recharge model to use, default is ps.rch.Linear()
- **recharge_name** (*str*) – name of the RechargeModel

apply(*libname*, *func*, *names=None*, *progressbar=True*)

Apply function to items in library.

Supported libraries are oseries, stresses, and models.

Parameters

- **libname** (*str*) – library name, supports "oseries", "stresses" and "models"
- **func** (*callable*) – function that accepts items from one of the supported libraries as input
- **names** (*str, list of str, optional*) – apply function to these names, by default None which loops over all stored items in library
- **progressbar** (*bool, optional*) – show progressbar, by default True

Returns
dict of results of func, with names as keys and results as values

Return type
dict

create_model(*name: str*, *modelname: str | None = None*, *add_recharge: bool = True*, *recharge_name: str = 'recharge'*) → Model

Create a pastas Model.

Parameters

- **name** (*str*) – name of the oseries to create a model for
- **modelname** (*str, optional*) – name of the model, default is None, which uses oseries name
- **add_recharge** (*bool, optional*) – add recharge to the model by looking for the closest precipitation and evaporation time series in the stresses library, by default True

- **recharge_name** (*str*) – name of the RechargeModel

Returns
model for the oseries

Return type
pastas.Model

Raises

- **KeyError** – if data is stored as dataframe and no column is provided
- **ValueError** – if time series is empty

create_models_bulk(*oseries: list | str | None = None*, *add_recharge: bool = True*, *solve: bool = False*, *store_models: bool = True*, *ignore_errors: bool = False*, *progressbar: bool = True*, ***kwargs*) → Tuple[dict, dict] | dict

Bulk creation of pastas models.

Parameters

- **oseries** (*list of str, optional*) – names of oseries to create models for, by default None, which creates models for all oseries
- **add_recharge** (*bool, optional*) – add recharge to the models based on closest precipitation and evaporation time series, by default True
- **solve** (*bool, optional*) – solve the model, by default False
- **store_models** (*bool, optional*) – if False, return a list of models, by default True, which will store the models in the database.
- **ignore_errors** (*bool, optional*) – ignore errors while creating models, by default False
- **progressbar** (*bool, optional*) – show progressbar, by default True

Returns

- **models** (*dict, if return_models is True*) – dictionary of models
- **errors** (*list, always returned*) – list of model names that could not be created

export_model_series_to_csv(*names: list | str | None = None*, *exportdir: str = '.'*, *exportmeta: bool = True*)

Export model time series to csv files.

Parameters

- **names** (*Optional[Union[list, str]], optional*) – names of models to export, by default None, which uses retrieves all models from database
- **exportdir** (*str, optional*) – directory to export csv files to, default is current directory
- **exportmeta** (*bool, optional*) – export metadata for all time series as csv file, default is True

classmethod from_zip(*fname: str*, *conn: BaseConnector | None = None*, *storename: str | None = None*, *progressbar: bool = True*)

Load PastaStore from zipfile.

Parameters

- **fname** (*str*) – pathname of zipfile

- **conn** (*Connector object, optional*) – connector for storing loaded data, default is None which creates a DictConnector. This Connector does not store data on disk.
- **storename** (*str, optional*) – name of the PastaStore, by default None, which defaults to the name of the Connector.
- **progressbar** (*bool, optional*) – show progressbar, by default True

Returns
return PastaStore containing data from zipfile

Return type
pastastore.PastaStore

get_distances(*oseries: list | str | None = None*, *stresses: list | str | None = None*, *kind: List[str] | str | None = None*) → DataFrame | Series

Method to obtain the distances in meters between the oseries and stresses.

Parameters

- **oseries** (*str or list of str*) – name(s) of the oseries
- **stresses** (*str or list of str*) – name(s) of the stresses
- **kind** (*str, list of str*) – string or list of strings representing which kind(s) of stresses to consider

Returns
distances – Pandas DataFrame with the distances between the oseries (index) and the stresses (columns).

Return type
pandas.DataFrame

get_model_timeseries_names(*modelnames: list | str | None = None*, *dropna: bool = True*, *progressbar: bool = True*) → DataFrame | Series

Get time series names contained in model.

Parameters

- **modelnames** (*Optional[Union[list, str]], optional*) – list or name of models to get time series names for, by default None which will use all modelnames
- **dropna** (*bool, optional*) – drop stresses from table if stress is not included in any model, by default True
- **progressbar** (*bool, optional*) – show progressbar, by default True

Returns
structure – returns DataFrame with oseries name per model, and a flag indicating whether a stress is contained within a time series model.

Return type
pandas.DataFrame

get_nearest_oseries(*names: list | str | None = None*, *n: int = 1*, *maxdist: float | None = None*) → DataFrame | Series

Method to obtain the nearest (n) oseries.

Parameters

- **names** (*str or list of str*) – string or list of strings with the name(s) of the oseries
- **n** (*int*) – number of oseries to obtain

- **maxdist** (*float, optional*) – maximum distance to consider

Returns
list with the names of the oseries.

Return type
oseries

get_nearest_stresses(*oseries: list | str | None = None*, *stresses: list | str | None = None*, *kind: list | str | None = None*, *n: int = 1*, *maxdist: float | None = None*) → DataFrame | Series

Method to obtain the nearest (n) stresses of a specific kind.

Parameters

- **oseries** (*str*) – string with the name of the oseries
- **stresses** (*str or list of str*) – string with the name of the stresses
- **kind** (*str, list of str, optional*) – string or list of str with the name of the kind(s) of stresses to consider
- **n** (*int*) – number of stresses to obtain
- **maxdist** (*float, optional*) – maximum distance to consider

Returns
list with the names of the stresses.

Return type
stresses

get_oseries_distances(*names: list | str | None = None*) → DataFrame | Series

Method to obtain the distances in meters between the oseries.

Parameters
names (*str or list of str*) – names of the oseries to calculate distances between

Returns
distances – Pandas DataFrame with the distances between the oseries

Return type
pandas.DataFrame

get_parameters(*parameters: List[str] | None = None*, *modelnames: List[str] | None = None*, *param_value: str | None = 'optimal'*, *progressbar: bool | None = False*, *ignore_errors: bool | None = False*) → DataFrame | Series

Get model parameters. NaN-values are returned when the parameters are not present in the model or the model is not optimized.

Parameters

- **parameters** (*list of str, optional*) – names of the parameters, by default None which uses all parameters from each model
- **modelnames** (*str or list of str, optional*) – name(s) of model(s), by default None in which case all models are used
- **param_value** (*str, optional*) – which column to use from the model parameters dataframe, by default "optimal" which retrieves the optimized parameters.
- **progressbar** (*bool, optional*) – show progressbar, default is False
- **ignore_errors** (*bool, optional*) – ignore errors when True, i.e. when non-existent model is encountered in modelnames, by default False

Returns
p – DataFrame containing the parameters (columns) per model (rows)

Return type
pandas.DataFrame

`get_signatures`(*signatures=None*, *names=None*, *libname='oseries'*, *progressbar=False*, *ignore_errors=False*)

Get groundwater signatures. NaN-values are returned when the signature could not be computed.

Parameters

- **signatures** (*list of str, optional*) – list of groundwater signatures to compute, if None all groundwater signatures in ps.stats.signatures.__all__ are used, by default None
- **names** (*str, list of str, or None, optional*) – names of the time series, by default None which uses all the time series in the library
- **libname** (*str*) – name of the library containing the time series ('oseries' or 'stresses'), by default "oseries"
- **progressbar** (*bool, optional*) – show progressbar, by default False
- **ignore_errors** (*bool, optional*) – ignore errors when True, i.e. when non-existent timeseries is encountered in names, by default False

Returns
signatures_df – DataFrame containing the signatures (columns) per time series (rows)

Return type
pandas.DataFrame

`get_statistics`(*statistics: str | List[str]*, *modelnames: List[str] | None = None*, *progressbar: bool | None = False*, *ignore_errors: bool | None = False*, ***kwargs*) → DataFrame | Series

Get model statistics.

Parameters

- **statistics** (*str or list of str*) – statistic or list of statistics to calculate, e.g. ["evp", "rsq", "rmse"], for a full list see *pastas.modelstats.Statistics.ops*.
- **modelnames** (*list of str, optional*) – modelnames to calculates statistics for, by default None, which uses all models in the store
- **progressbar** (*bool, optional*) – show progressbar, by default False
- **ignore_errors** (*bool, optional*) – ignore errors when True, i.e. when trying to calculate statistics for non-existent model in modelnames, default is False
- ****kwargs** – any arguments that can be passed to the methods for calculating statistics

Returns
s

Return type
pandas.DataFrame

`get_tmin_tmax`(*libname*, *names=None*, *progressbar=False*)

Get tmin and tmax for time series.

Parameters

- **libname** (*str*) – name of the library containing the time series ('oseries' or 'stresses')

- **names** (*str, list of str, or None, optional*) – names of the time series, by default None which uses all the time series in the library
- **progressbar** (*bool, optional*) – show progressbar, by default False

Returns
tmintmax – Dataframe containing tmin and tmax per time series

Return type
pd.dataframe

model_results(*mls: Model | list | str | None = None*, *progressbar: bool = True*)

Get pastas model results.

Parameters

- **mls** (*list of str, optional*) – list of model names, by default None which means results for all models will be calculated
- **progressbar** (*bool, optional*) – show progressbar, by default True

Returns
results – dataframe containing parameters and other statistics for each model

Return type
pd.DataFrame

Raises
ModuleNotFoundError – if the art_tools module is not available

search(*libname: str*, *s: list | str | None = None*, *case_sensitive: bool = True*, *sort=True*)

Search for names of time series or models starting with *s*.

Parameters

- **libname** (*str*) – name of the library to search in
- **s** (*str, lst*) – find names with part of this string or strings in list
- **case_sensitive** (*bool, optional*) – whether search should be case sensitive, by default True
- **sort** (*bool, optional*) – sort list of names

Returns
matches – list of names that match search result

Return type
list

solve_models(*mls: Model | list | str | None = None*, *report: bool = False*, *ignore_solve_errors: bool = False*, *store_result: bool = True*, *progressbar: bool = True*, ***kwargs*) → None

Solves the models in the store.

Parameters

- **mls** (*list of str, optional*) – list of model names, if None all models in the pastastore are solved.
- **report** (*boolean, optional*) – determines if a report is printed when the model is solved, default is False
- **ignore_solve_errors** (*boolean, optional*) – if True, errors emerging from the solve method are ignored, default is False which will raise an exception when a model cannot be optimized

- **store_result** (*bool, optional*) – if True save optimized models, default is True
- **progressbar** (*bool, optional*) – show progressbar, default is True
- ****kwargs** – arguments are passed to the solve method.

to_zip(*fname: str*, *overwrite=False*, *progressbar: bool = True*)

Write data to zipfile.

Parameters

- **fname** (*str*) – name of zipfile
- **overwrite** (*bool, optional*) – if True, overwrite existing file
- **progressbar** (*bool, optional*) – show progressbar, by default True

4.3 Plots

class pastastore.plotting.**Plots**(*pstore*)

Plot class for Pastastore.

Allows plotting of time series and data availability.

static _data_availability(*series*, *names=None*, *intervals=None*, *ignore=('second', 'minute', '14 days')*, *ax=None*, *cax=None*, *normtype='log'*, *cmap='viridis_r'*, *set_yticks=False*, *figsize=(10, 8)*, *dropna=True*, ***kwargs*)

Plot the data-availability for a list of time series.

Parameters

- **libname** (*list of pandas.Series*) – list of series to plot data availability for
- **names** (*list, optional*) – specify names of series, default is None in which case names will be taken from series themselves.
- **kind** (*str, optional*) – if library is stresses, kind can be specified to obtain only stresses of a specific kind
- **intervals** (*dict, optional*) – A dict with frequencies as keys and number of seconds as values
- **ignore** (*list, optional*) – A list with frequencies in intervals to ignore
- **ax** (*matplotlib Axes, optional*) – pass axes object to plot data availability on existing figure. by default None, in which case a new figure is created
- **cax** (*matplotlib Axes, optional*) – pass object axes to plot the colorbar on. by default None, which gives default Maptlotlib behavior
- **normtype** (*str, optional*) – Determines the type of color normalisations, default is 'log'
- **cmap** (*str, optional*) – A reference to a matplotlib colormap
- **set_yticks** (*bool, optional*) – Set the names of the series as yticks
- **figsize** (*tuple, optional*) – The size of the new figure in inches (h,v)
- **progressbar** (*bool*) – Show progressbar
- **dropna** (*bool*) – Do not show NaNs as available data

- **kwargs** (*dict, optional*) – Extra arguments are passed to matplotlib.pyplot.subplots()

Returns
ax – The axes in which the data-availability is plotted

Return type
matplotlib Axes

_timeseries(*libname*, *names=None*, *ax=None*, *split=False*, *figsize=(10, 5)*, *progressbar=True*, *show_legend=True*, *labelfunc=None*, *legend_kwargs=None*, ***kwargs*)

Internal method to plot time series from pastastore.

Parameters

- **libname** (*str*) – name of the library to obtain time series from (oseries or stresses)
- **names** (*list of str, optional*) – list of time series names to plot, by default None
- **ax** (*matplotlib.Axes, optional*) – pass axes object to plot on existing axes, by default None, which creates a new figure
- **split** (*bool, optional*) – create a separate subplot for each time series, by default False. A maximum of 20 time series is supported when split=True.
- **figsize** (*tuple, optional*) – figure size, by default (10, 5)
- **progressbar** (*bool, optional*) – show progressbar when loading time series from store, by default True
- **show_legend** (*bool, optional*) – show legend, default is True.
- **labelfunc** (*callable, optional*) – function to create custom labels, function should take name of time series as input
- **legend_kwargs** (*dict, optional*) – additional arguments to pass to legend

Returns
ax – axes handle

Return type
matplotlib.Axes

Raises
ValueError – split=True is only supported if there are less than 20 time series to plot.

compare_models(*modelnames*, *ax=None*, ***kwargs*)

cumulative_hist(*statistic='rsq'*, *modelnames=None*, *extend=False*, *ax=None*, *figsize=(6, 6)*, *label=None*, *legend=True*)

Plot a cumulative step histogram for a model statistic.

Parameters

- **statistic** (*str*) – name of the statistic, e.g. "evp" or "rmse", by default "rsq"
- **modelnames** (*list of str, optional*) – modelnames to plot statistic for, by default None, which uses all models in the store
- **extend** (*bool, optional*) – force extend the stats Series with a dummy value to move the horizontal line outside figure bounds. If True the results are skewed a bit, especially if number of models is low.
- **ax** (*matplotlib.Axes, optional*) – axes to plot histogram, by default None which creates an Axes

- **figsize** (*tuple, optional*) – figure size, by default (6,6)
- **label** (*str, optional*) – label for the legend, by default None, which shows the number of models
- **legend** (*bool, optional*) – show legend, by default True

Returns
ax – The axes in which the cumulative histogram is plotted

Return type
matplotlib Axes

data_availability(*libname*, *names=None*, *kind=None*, *intervals=None*, *ignore=('second', 'minute', '14 days')*, *ax=None*, *cax=None*, *normtype='log'*, *cmap='viridis_r'*, *set_yticks=False*, *figsize=(10, 8)*, *progressbar=True*, *dropna=True*, ***kwargs*)

Plot the data-availability for multiple time series in pastastore.

Parameters

- **libname** (*str*) – name of library to get time series from (oseries or stresses)
- **names** (*list, optional*) – specify names in a list to plot data availability for certain time series
- **kind** (*str, optional*) – if library is stresses, kind can be specified to obtain only stresses of a specific kind
- **intervals** (*dict, optional*) – A dict with frequencies as keys and number of seconds as values
- **ignore** (*list, optional*) – A list with frequencies in intervals to ignore
- **ax** (*matplotlib Axes, optional*) – pass axes object to plot data availability on existing figure. by default None, in which case a new figure is created
- **cax** (*matplotlib Axes, optional*) – pass object axes to plot the colorbar on. by default None, which gives default Maptlotlib behavior
- **normtype** (*str, optional*) – Determines the type of color normalisations, default is 'log'
- **cmap** (*str, optional*) – A reference to a matplotlib colormap
- **set_yticks** (*bool, optional*) – Set the names of the series as yticks
- **figsize** (*tuple, optional*) – The size of the new figure in inches (h,v)
- **progressbar** (*bool*) – Show progressbar
- **dropna** (*bool*) – Do not show NaNs as available data
- **kwargs** (*dict, optional*) – Extra arguments are passed to matplotlib.pyplot.subplots()

Returns
ax – The axes in which the data-availability is plotted

Return type
matplotlib Axes

oseries(*names=None*, *ax=None*, *split=False*, *figsize=(10, 5)*, *show_legend=True*, *labelfunc=None*, *legend_kwargs=None*, ***kwargs*)

Plot oseries.

Parameters

- **names** (*list of str, optional*) – list of oseries names to plot, by default None, which loads all oseries from store
- **ax** (*matplotlib.Axes, optional*) – pass axes object to plot oseries on existing figure, by default None, in which case a new figure is created
- **split** (*bool, optional*) – create a separate subplot for each time series, by default False. A maximum of 20 time series is supported when split=True.
- **figsize** (*tuple, optional*) – figure size, by default (10, 5)
- **show_legend** (*bool, optional*) – show legend, default is True.
- **labelfunc** (*callable, optional*) – function to create custom labels, function should take name of time series as input
- **legend_kwargs** (*dict, optional*) – additional arguments to pass to legend

Returns
ax – axes handle

Return type
matplotlib.Axes

stresses(*names=None*, *kind=None*, *ax=None*, *split=False*, *figsize=(10, 5)*, *show_legend=True*, *labelfunc=None*, *legend_kwargs=None*, ***kwargs*)

Plot stresses.

Parameters

- **names** (*list of str, optional*) – list of oseries names to plot, by default None, which loads all oseries from store
- **kind** (*str, optional*) – only plot stresses of a certain kind, by default None, which includes all stresses
- **ax** (*matplotlib.Axes, optional*) – pass axes object to plot oseries on existing figure, by default None, in which case a new figure is created
- **split** (*bool, optional*) – create a separate subplot for each time series, by default False. A maximum of 20 time series is supported when split=True.
- **figsize** (*tuple, optional*) – figure size, by default (10, 5)
- **show_legend** (*bool, optional*) – show legend, default is True.
- **labelfunc** (*callable, optional*) – function to create custom labels, function should take name of time series as input
- **legend_kwargs** (*dict, optional*) – additional arguments to pass to legend

Returns
ax – axes handle

Return type
matplotlib.Axes

4.4 Maps

class pastastore.plotting.**Maps**(*pstore*)

Map Class for PastaStore.

Allows plotting locations and model statistics on maps.

4.4.1 Usage

Example usage of the maps methods: :

>> > ax = pstore.maps.oseries() # plot oseries locations >> > pstore.maps.add_background_map(ax) # add background map

static **_list_contextily_providers**()

List contextily providers.

Taken from contextily notebooks.

Returns

providers – dictionary containing all providers. See keys for names that can be passed as map_provider arguments.

Return type

dict

static **_plotmap_dataframe**(*df*, *x='x'*, *y='y'*, *column=None*, *colorbar=True*, *ax=None*, *figsize=(10, 8)*, ***kwargs*)

Internal method for plotting dataframe with point locations.

Can be called directly for more control over plot characteristics.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame containing coordinates and data to plot, with index providing names for each location
- **x** (*str, optional*) – name of the column with x - coordinate data, by default "x"
- **y** (*str, optional*) – name of the column with y - coordinate data, by default "y"
- **column** (*str, optional*) – name of the column containing data used for determining the color of each point, by default None (all one color)
- **colorbar** (*bool, optional*) – show colorbar, only if column is provided, by default True
- **ax** (*matplotlib Axes*) – axes handle to plot dataframe, optional, default is None which creates a new figure
- **figsize** (*tuple, optional*) – figure size, by default(10, 8)
- ****kwargs** – dictionary containing keyword arguments for ax.scatter, by default None

Returns

- **ax** (*matplotlib.Axes*) – axes object, returned if ax is None
- **sc** (*scatter handle*) – scatter plot handle, returned if ax is not None

static add_background_map(*ax*, *proj='epsg:28992'*, *map_provider='OpenStreetMap.Mapnik'*, ***kwargs*)

Add background map to axes using contextily.

Parameters

- **ax** (*matplotlib.Axes*) – axes to add background map to
- **map_provider** (*str, optional*) – name of map provider, see *contextily.providers* for options. Default is 'OpenStreetMap.Mapnik'
- **proj** (*pyproj.Proj or str, optional*) – projection for background map, default is 'epsg:28992' (RD Amersfoort, a projection for the Netherlands)

static add_labels(*df*, *ax*, *adjust=False*, *objects=None*, ***kwargs*)

Add labels to points on plot.

Uses dataframe index to label points.

Parameters

- **df** (*pd.DataFrame*) – DataFrame containing x, y - data. Index is used as label
- **ax** (*matplotlib.Axes*) – axes object to label points on
- **adjust** (*bool*) – automated smart label placement using adjustText
- **objects** (*list of matplotlib objects*) – use to avoid labels overlapping markers
- ****kwargs** – keyword arguments to ax.annotate or adjusttext

model(*ml*, *label=True*, *metadata_source='model'*, *offset=0.0*, *ax=None*, *figsize=(10, 10)*, *backgroundmap=False*)

Plot oseries and stresses from one model on a map.

Parameters

- **ml** (*str or pastas.Model*) – pastas model or name of pastas model to plot on map
- **label** (*bool, optional, default is True*) – add labels to points on map
- **metadata_source** (*str, optional*) – whether to obtain metadata from model Time-series or from metadata in pastastore("store"), default is "model"
- **offset** (*float, optional*) – add offset to current extent of model time series, useful for zooming out around models
- **ax** (*matplotlib.Axes, optional*) – axes handle, if not provided a new figure is created.
- **figsize** (*tuple, optional*) – figsize, default is (10, 10)
- **backgroundmap** (*bool, optional*) – if True, add background map (default CRS is EPSG:28992) with default tiles by OpenStreetMap.Mapnik. Default option is False.

Returns

ax – axis handle of the resulting figure

Return type

axes object

See also:

`self.add_background_map`

models(*labels=True*, *adjust=False*, *figsize=(10, 8)*, *backgroundmap=False*, ***kwargs*)

Plot model locations on map.

Parameters

- **labels** (*bool, optional*) – label models, by default True
- **adjust** (*bool, optional*) – automated smart label placement using adjustText, by default False
- **ax** (*matplotlib.Axes, optional*) – axes handle, if not provided a new figure is created.
- **figsize** (*tuple, optional*) – figure size, by default(10, 8)
- **backgroundmap** (*bool, optional*) – if True, add background map (default CRS is EPSG:28992) with default tiles by OpenStreetMap.Mapnik. Default option is False.

Returns

ax – axes object

Return type

matplotlib.Axes

See also:

`self.add_background_map`

modelstat(*statistic*, *modelnames=None*, *label=True*, *adjust=False*, *cmap='viridis'*, *norm=None*, *vmin=None*, *vmax=None*, *figsize=(10, 8)*, *backgroundmap=False*, ***kwargs*)

Plot model statistic on map.

Parameters

- **statistic** (*str*) – name of the statistic, e.g. "evp" or "aic"
- **modelnames** (*list of str, optional*) – list of modelnames to include
- **label** (*bool, optional*) – label points, by default True
- **adjust** (*bool, optional*) – automated smart label placement using adjustText, by default False
- **cmap** (*str or colormap, optional*) – (name of) the colormap, by default "viridis"
- **norm** (*norm, optional*) – normalization for colorbar, by default None
- **vmin** (*float, optional*) – vmin for colorbar, by default None
- **vmax** (*float, optional*) – vmax for colorbar, by default None
- **ax** (*matplotlib.Axes, optional*) – axes handle, if not provided a new figure is created.
- **figsize** (*tuple, optional*) – figuresize, by default(10, 8)
- **backgroundmap** (*bool, optional*) – if True, add background map (default CRS is EPSG:28992) with default tiles by OpenStreetMap.Mapnik. Default option is False.

Returns

ax – axes object

Return type

matplotlib.Axes

See also:

`self.add_background_map`

oseries(*names=None*, *extent=None*, *labels=True*, *adjust=False*, *figsize=(10, 8)*, *backgroundmap=False*, *label_kwargs=None*, ***kwargs*)

Plot oseries locations on map.

Parameters

- **names** (*list, optional*) – oseries names, by default None which plots all oseries locations
- **extent** (*list of float, optional*) – plot only oseries within extent [xmin, xmax, ymin, ymax]
- **labels** (*bool or str, optional*) – label models, by default True, if passed as "grouped", only the first label for each x,y-location is shown.
- **adjust** (*bool, optional*) – automated smart label placement using adjustText, by default False
- **figsize** (*tuple, optional*) – figure size, by default(10, 8)
- **backgroundmap** (*bool, optional*) – if True, add background map (default CRS is EPSG:28992) with default tiles by OpenStreetMap.Mapnik. Default option is False.
- **label_kwargs** (*dict, optional*) – dictionary with keyword arguments to pass to add_labels method

Returns

ax – axes object

Return type

matplotlib.Axes

See also:

`self.add_background_map`

stresses(*names=None*, *kind=None*, *extent=None*, *labels=True*, *adjust=False*, *figsize=(10, 8)*, *backgroundmap=False*, *label_kwargs=None*, ***kwargs*)

Plot stresses locations on map.

Parameters

- **names** (*list of str, optional*) – list of names to plot
- **kind** (*str, optional*) – if passed, only plot stresses of a specific kind, default is None which plots all stresses.
- **extent** (*list of float, optional*) – plot only stresses within extent [xmin, xmax, ymin, ymax]
- **labels** (*bool, optional*) – label models, by default True
- **adjust** (*bool, optional*) – automated smart label placement using adjustText, by default False
- **ax** (*matplotlib.Axes, optional*) – axes handle, if not provided a new figure is created.
- **figsize** (*tuple, optional*) – figure size, by default(10, 8)
- **backgroundmap** (*bool, optional*) – if True, add background map (default CRS is EPSG:28992) with default tiles by OpenStreetMap.Mapnik. Default option is False.
- **label_kwargs** (*dict, optional*) – dictionary with keyword arguments to pass to add_labels method

Returns
ax – axes object

Return type
matplotlib.Axes

See also:

`self.add_background_map`

stresslinks(*kinds=None*, *model_names=None*, *color_lines=False*, *alpha=0.4*, *ax=None*, *figsize=(10, 8)*, *legend=True*, *labels=False*, *adjust=False*, *backgroundmap=False*)

Create a map linking models with their stresses.

Parameters

- **kinds** (*list, optional*) – kinds of stresses to plot, defaults to None, which selects all kinds.
- **model_names** (*list, optional*) – list of model names to plot, substrings of model names are also accepted, defaults to None, which selects all models.
- **color_lines** (*bool, optional*) – if True, connecting lines have the same colors as the stresses, defaults to False, which uses a black line.
- **alpha** (*float, optional*) – alpha value for the connecting lines, defaults to 0.4.
- **ax** (*matplotlib.Axes, optional*) – axes handle, if not provided a new figure is created.
- **figsize** (*tuple, optional*) – figure size, by default (10, 8)
- **legend** (*bool, optional*) – create a legend for all unique kinds, defaults to True.
- **labels** (*bool, optional*) – add labels for stresses and oseries, defaults to False.
- **adjust** (*bool, optional*) – automated smart label placement using adjustText, by default False
- **backgroundmap** (*bool, optional*) – if True, add background map (default CRS is EPSG:28992) with default tiles by OpenStreetMap.Mapnik. Default option is False.

Returns
ax – axis handle of the resulting figure

Return type
axes object

See also:

`self.add_background_map`

4.5 Yaml

class pastastore.yaml_interface.**PastastoreYAML**(*pstore*)

Class for reading/writing Pastas models in YAML format.

This class provides a more human-readable form of Pastas models in comparison to Pastas default .pas (JSON) files. The goal is to provide users with a simple mini-language to quickly build/test different model structures. A PastaStore is required as input, which contains existing models or time series required to build new models. This class also introduces some shortcuts to simplify building models. Shortcuts include the option to pass 'nearest' as the name of a stress, which will automatically select the closest stress of a particular type. Other shortcuts

include certain default options when certain information is not listed in the YAML file, that will work well in many cases.

4.5.1 Usage

Instantiate the PastastoreYAML class:

```
pyaml = PastastoreYAML(pstore)
```

Export a Pastas model to a YAML file:

```
pyaml.export_model_to_yaml(ml)
```

Load a Pastas model from a YAML file:

```
models = pyaml.load_yaml("my_first_model.yaml")
```

Example YAML file using 'nearest':

```
my_first_model:  # this is the name of the model
  oseries: "oseries1"  # name of oseries stored in PastaStore
  stressmodels:
      recharge:  # recognized as RechargeModel by name
        prec: "nearest"  # use nearest stress with kind="prec"
        evap: "EV24_DEELEN"  # specific station
      river:
        stress: "nearest riv"  # nearest stress with kind="riv"
      wells:
        stress: "nearest 3"  # nearest 3 stresses with kind="well"
        stressmodel: WellModel  # provide StressModel type
```

static export_model(*ml: Model | dict*, *outdir: str | None = '.'*, *minimal_yaml: bool | None = False*, *use_nearest: bool | None = False*)

Write single pastas model to YAML file.

Parameters

- **ml** (*ps.Model or dict*) – pastas model instance or dictionary representing a pastas model
- **outdir** (*str, optional*) – path to output directory, by default "." (current directory)
- **minimal_yaml** (*bool, optional*) – reduce yaml file to include the minimum amount of information that will still construct a model. Users are warned, using this option does not guarantee the same model will be constructed as the one that was exported! Default is False.
- **use_nearest** (*bool, optional*) – if True, replaces time series with "nearest <kind>", filling in kind where possible. Warning! This does not check whether the time series are actually the nearest ones! Only used when minimal_yaml=True. Default is False.

export_models(*models: List[Model] | List[Dict] | None = None*, *modelnames: List[str] | str | None = None*, *outdir: str | None = '.'*, *minimal_yaml: bool | None = False*, *use_nearest: bool | None = False*, *split: bool | None = True*, *filename: str | None = 'pastas_models.yaml'*)

Export (stored) models to yaml file(s).

Parameters

- **models** (*list of ps.Model or dict, optional*) – pastas Models to write to yaml file(s), if not provided, uses modelnames to collect stored models to export.
- **modelnames** (*list of str, optional*) – list of model names to export, by default None, which uses all stored models.
- **outdir** (*str, optional*) – path to output directory, by default "." (current directory)
- **minimal_yaml** (*bool, optional*) – reduce yaml file to include the minimum amount of information that will still construct a model. Users are warned, using this option does not guarantee the same model will be constructed as the one that was exported! Default is False.
- **use_nearest** (*bool, optional*) – if True, replaces time series with "nearest <kind>", filling in kind where possible. Warning! This does not check whether the time series are actually the nearest ones! Only used when minimal_yaml=True. Default is False.
- **split** (*bool, optional*) – if True, split into separate yaml files, otherwise store all in the same file. The model names are used as file names.
- **filename** (*str, optional*) – filename for YAML file, only used if *split=False*

export_stored_models_per_oseries(*oseries: List[str] | str | None = None*, *outdir: str | None = '.'*, *minimal_yaml: bool | None = False*, *use_nearest: bool | None = False*)

Export store models grouped per oseries (location) to YAML file(s).

Note: The oseries names are used as file names.

Parameters

- **oseries** (*list of str, optional*) – list of oseries (location) names, by default None, which uses all stored oseries for which there are models.
- **outdir** (*str, optional*) – path to output directory, by default "." (current directory)
- **minimal_yaml** (*bool, optional*) – reduce yaml file to include the minimum amount of information that will still construct a model. Users are warned, using this option does not guarantee the same model will be constructed as the one that was exported! Default is False.
- **use_nearest** (*bool, optional*) – if True, replaces time series with "nearest <kind>", filling in kind where possible. Warning! This does not check whether the time series are actually the nearest ones! Only used when minimal_yaml=True. Default is False.

load(*fyaml: str*) → List[Model]

Load Pastas YAML file.

Note: currently supports RechargeModel, StressModel and WellModel.

Parameters
fyaml (*str*) – path to file

Returns
models – list containing pastas model(s)

Return type
list

Raises

- **ValueError** – if insufficient information is provided to construct pastas model
- **NotImplementedError** – if unsupported stressmodel is encountered

pastastore.yaml_interface.**reduce_to_minimal_dict**(*d*, *keys=None*)

Reduce pastas model dictionary to a minimal form.

This minimal form strives to keep the minimal information that still allows a model to be constructed. Users are warned, reducing a model dictionary with this function can lead to a different model than the original!

Parameters

- **d** (*dict*) – pastas model in dictionary form
- **keys** (*list, optional*) – list of keys to keep, by default None, which defaults to: ["name", "oseries", "settings", "tmin", "tmax", "noise", "stressmodels", "rfunc", "stress", "prec", "evap", "stressmodel"]

pastastore.yaml_interface.**replace_ts_with_name**(*d*, *nearest=False*)

Replace time series dict with its name in pastas model dict.

Parameters

- **d** (*dict*) – pastas model dictionary
- **nearest** (*bool, optional*) – replace time series with "nearest" option. Warning, this does not check whether the time series are actually the nearest ones!

4.6 Util

exception pastastore.util.**ItemInLibraryException**

pastastore.util.**compare_models**(*ml1*, *ml2*, *stats=None*, *detailed_comparison=False*)

Compare two Pastas models.

Parameters

- **ml1** (*pastas.Model*) – first model to compare
- **ml2** (*pastas.Model*) – second model to compare
- **stats** (*list of str, optional*) – if provided compare these model statistics
- **detailed_comparison** (*bool, optional*) – if True return DataFrame containing comparison details, by default False which returns True if models are equivalent or False if they are not

Returns

returns True if models are equivalent when detailed_comparison=True else returns DataFrame containing comparison details.

Return type

bool or pd.DataFrame

pastastore.util.**copy_database**(*conn1*, *conn2*, *libraries: List[str] | None = None*, *overwrite: bool = False*, *progressbar: bool = False*) → None

Copy libraries from one database to another.

Parameters

- **conn1** (*pastastore.*Connector*) – source Connector containing link to current database containing data
- **conn2** (*pastastore.*Connector*) – destination Connector with link to database to which you want to copy

- **libraries** (*Optional[List[str]], optional*) – list of str containing names of libraries to copy, by default None, which copies all libraries: ['oseries', 'stresses', 'models']
- **overwrite** (*bool, optional*) – overwrite data in destination database, by default False
- **progressbar** (*bool, optional*) – show progressbars, by default False

Raises

ValueError – if library name is not understood

pastastore.util.**delete_arctic_connector**(*conn=None*, *connstr: str | None = None*, *name: str | None = None*, *libraries: List[str] | None = None*) → None

Delete libraries from arctic database.

Parameters

- **conn** (pastastore.ArcticConnector) – ArcticConnector object
- **connstr** (*str, optional*) – connection string to the database
- **name** (*str, optional*) – name of the database
- **libraries** (*Optional[List[str]], optional*) – list of library names to delete, by default None which deletes all libraries

pastastore.util.**delete_arcticdb_connector**(*conn=None*, *uri: str | None = None*, *name: str | None = None*, *libraries: List[str] | None = None*) → None

Delete libraries from arcticDB database.

Parameters

- **conn** (pastastore.ArcticDBConnector) – ArcticDBConnector object
- **uri** (*str, optional*) – uri connection string to the database
- **name** (*str, optional*) – name of the database
- **libraries** (*Optional[List[str]], optional*) – list of library names to delete, by default None which deletes all libraries

pastastore.util.**delete_pastastore**(*pstore*, *libraries: List[str] | None = None*) → None

Delete libraries from PastaStore.

Note: This deletes the original PastaStore object. To access data that has not been deleted, it is recommended to create a new PastaStore object with the same Connector settings. This also creates new empty libraries if they were deleted.

Parameters

- **pstore** (*pastastore.PastaStore*) – PastaStore object to delete (from)
- **libraries** (*Optional[List[str]], optional*) – list of library names to delete, by default None which deletes all libraries

Raises

TypeError – when Connector type is not recognized

pastastore.util.**delete_pystore_connector**(*conn=None*, *path: str | None = None*, *name: str | None = None*, *libraries: List[str] | None = None*) → None

Delete libraries from pystore.

Parameters

- **conn** (PystoreConnector, *optional*) – PystoreConnector object
- **path** (*str, optional*) – path to pystore
- **name** (*str, optional*) – name of the pystore
- **libraries** (*Optional[List[str]], optional*) – list of library names to delete, by default None which deletes all libraries

pastastore.util.**frontiers_aic_select**(*pstore*, *modelnames: List[str] | None = None*, *oseries: List[str] | None = None*, *full_output: bool = False*) → DataFrame

Select the best model structure based on the minimum AIC.

As proposed by Brakenhoff et al. 2022 [bra_2022].

Parameters

- **pstore** (*pastastore.PastaStore*) – reference to a PastaStore
- **modelnames** (*list of str*) – list of model names (that pass reliability criteria)
- **oseries** (*list of oseries*) – list of locations for which to select models, note that this uses all models associated with a specific location.
- **full_output** (*bool, optional*) – if set to True, returns a DataFrame including all models per location and their AIC values

Returns

DataFrame with selected best model per location based on the AIC, or a DataFrame containing statistics for each of the models per location

Return type

pandas.DataFrame

References

Bakker, M.: Application of Time Series Analysis to Estimate Drawdown From Multiple Well Fields. Front. Earth Sci., 14 June 2022 doi:10.3389/feart.2022.907609

pastastore.util.**frontiers_checks**(*pstore*, *modelnames: List[str] | None = None*, *oseries: List[str] | None = None*, *check1_rsq: bool = True*, *check1_threshold: float = 0.7*, *check2_autocor: bool = True*, *check2_test: str = 'runs'*, *check2_pvalue: float = 0.05*, *check3_tmem: bool = True*, *check3_cutoff: float = 0.95*, *check4_gain: bool = True*, *check5_parambounds: bool = False*, *csv_dir: str | None = None*) → DataFrame

Check models in a PastaStore to see if they pass reliability criteria.

The reliability criteria are taken from Brakenhoff et al. 2022 [bra_2022]. These criteria were applied in a region with recharge, river levels and pumping wells as stresses. This is by no means an exhaustive list of reliability criteria but might serve as a reasonable starting point for model diagnostic checking.

Parameters

- **pstore** (*pastastore.PastaStore*) – reference to a PastaStore
- **modelnames** (*list of str, optional*) – list of model names to consider, if None checks 'oseries', if both are None, all stored models will be checked
- **oseries** (*list of str, optional*) – list of oseries to consider, corresponding models will be picked up from pastastore. If None, uses all stored models are checked.

- **check1** (*bool, optional*) – check if model fit is above a threshold of the coefficient of determination R^2 , by default True
- **check1_threshold** (*float, optional*) – threshold of the R^2 fit statistic, by default 0.7
- **check2** (*bool, optional*) – check if the noise of the model has autocorrelation with statistical test, by default True
- **check2_test** (*str, optional*) – statistical test for autocorrelation. Available options are Runs test "runs", Stoffer-Toloi "stoffer" or "both", by default "runs"
- **check2_pvalue** (*float, optional*) – p-value for the statistical test to define the confindence interval, by default 0.05
- **check3** (*bool, optional*) – check if the length of the response time is within the calibration period, by default True
- **check3_cutoff** (*float, optional*) – the cutoff of the response time, by default 0.95
- **check4** (*bool, optional*) – check if the uncertainty of the gain, by default True
- **check5** (*bool, optional*) – check if parameters hit parameter bounds, by default False
- **csv_dir** (*string, optional*) – directory to store CSV file with overview of checks for every model, by default None which will not store results

Returns
df – dataFrame with all models and whether or not they pass the reliability checks

Return type
pandas.DataFrame

References

Brakenhoff, D.A., Vonk M.A., Collenteur, R.A., van Baar, M., Bakker, M.: Application of Time Series Analysis to Estimate Drawdown From Multiple Well Fields. Front. Earth Sci., 14 June 2022 doi:10.3389/feart.2022.907609

pastastore.util.**validate_names**(*s: str | None = None*, *d: dict | None = None*, *replace_space: str | None = '_'*, *deletechars: str | None = None*, ***kwargs*) → str | Dict

Remove invalid characters from string or dictionary keys.

Parameters

- **s** (*str, optional*) – remove invalid characters from string
- **d** (*dict, optional*) – remove invalid characters from keys from dictionary
- **replace_space** (*str, optional*) – replace spaces by this character, by default "_"
- **deletechars** (*str, optional*) – a string combining invalid characters, by default None

Returns
string or dict with invalid characters removed

Return type
str, dict

CHAPTER

FIVE

INDICES AND TABLES

- genindex

BIBLIOGRAPHY

[bra_2022] Brakenhoff, D.A., Vonk M.A., Collenteur, R.A., van Baar, M.,
[bra_2022]

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V