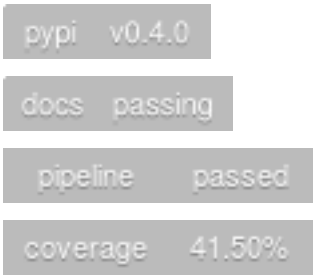
spux Documentation

Release 0.4.0

Scientific Computing Group at Eawag and SIS ID at ETH Zurich

Jun 18, 2019

Contents



SPUX - “Scalable Package for Uncertainty Quantification”.

SPUX is a modular framework for Bayesian inference and uncertainty quantification.
SPUX can be coupled with linear and nonlinear, deterministic and stochastic models.
SPUX supports model in any programming language (e.g. Python, R, Julia, C/C++, Fortran, Java).
SPUX scales effortlessly from serial run to parallel high performance computing clusters.
SPUX is application agnostic, with current examples in environmental dataset sciences.

SPUX is actively developed at Eawag, Swiss Federal Institute of Aquatic Science and Technology,
by researchers in the High Performance Scientific Computing Group, https://www.eawag.ch/sc.
For the scientific website of the SPUX project, please refer to https://eawag.ch/spux.

Documentation is available at https://spux.readthedocs.io.
Source code repository is available at https://gitlab.com/siam-sc/spux.

You are welcome to browse through the results gallery of the models already coupled to spux at
https://spux.readthedocs.io/en/stable/gallery.html.

This is free software, distributed under Apache (v2) License.

If you use this software, please cite (preprint available at http://arxiv.org/abs/1711.01410):

```
Šukys, J. and Kattwinkel, M.
"SPUX: Scalable Particle Markov Chain Monte Carlo
for uncertainty quantification in stochastic ecological models".
Advances in Parallel Computing - Parallel Computing is Everywhere,
IOS Press, (32), pp. 159-168, 2018.
```

CHAPTER 1

Contents

1.1 Introduction

SPUX stands for "Scalable Package for Uncertainty Quantification".

1.1.1 Summary

SPUX is a modular framework for Bayesian inference and uncertainty quantification in linear and nonlinear, deterministic and stochastic models. SPUX can be coupled to any model written in any programming language (e.g. Python, R, Julia, C/C++, Fortran, Java). SPUX scales effortlessly from serial runs on a personal computer to parallel high performance computing clusters. SPUX is application agnostic, with current examples available in the field of environmental data sciences.

In the near future, multi-level methods (e.g. ML(ET)PF, MLCV) will be included in SPUX to enable significant algorithmic acceleration of the inference and uncertainty quantification for models that support multiple resolution configurations.

An earlier prototype of spux was already described in a technical paper (preprint available at http://arxiv.org/abs/1711.01410):

```
Šukys, J. and Kattwinkel, M.
"SPUX: Scalable Particle Markov Chain Monte Carlo
for uncertainty quantification in stochastic ecological models".
Advances in Parallel Computing - Parallel Computing is Everywhere,
IOS Press, (32), pp. 159-168, 2018.
```

To give you a brief introduction regarding difference aspects of SPUX, we first begin with the mathematical concepts of the underlying scientific problem addressed by this framework.

1.1.2 Mathematical concepts

Here we briefly introduce mathematical concepts used in the Bayesian inference and uncertainty quantification.

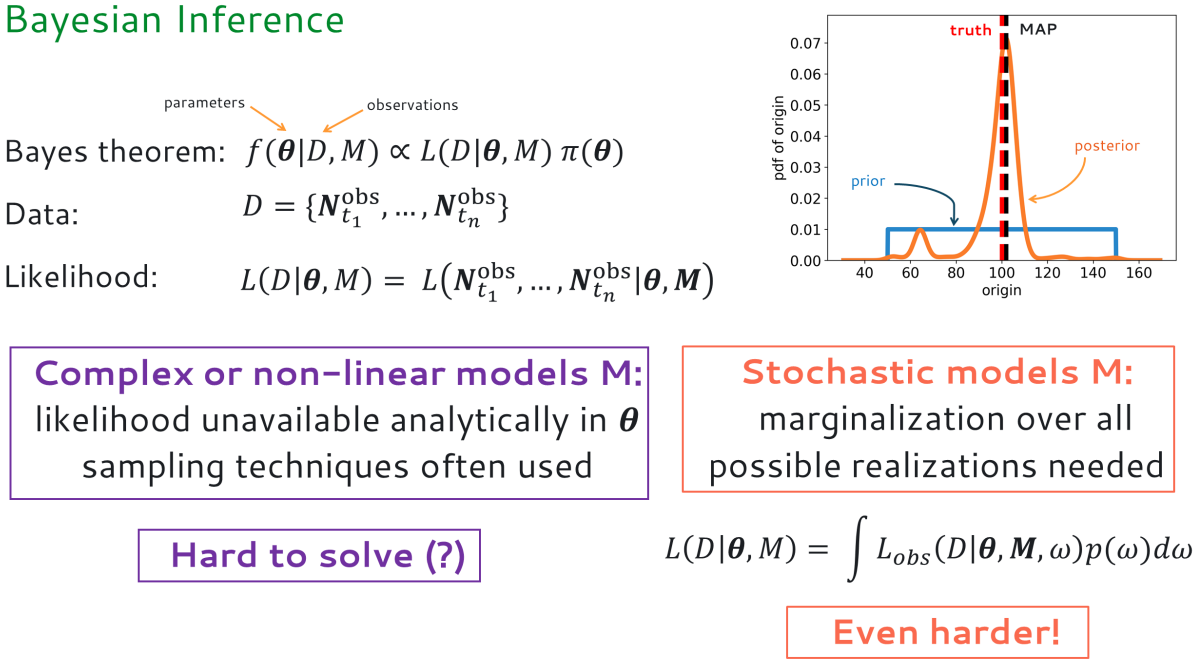


Fig. 1: Bayesian inference: updated the prior distribution of the model parameters to a posterior distribution given model observations (dataset). Bayesian theorem allows us to evaluate (or sample from) the posterior parameter distribution by computing the likelihood of the dataset given candidate model parameters.

1.1.3 Algorithms

1.2 Installation

Python package spux is available at PyPI repository: https://pypi.org/project/spux.

1.2.1 Main prerequisites

- Python, we recommend using Python 3.
  - in macOS, pre-installed
  - in Debian/Ubuntu, pre-installed
  - in Windows, follow instructions in https://www.python.org/downloads/windows/
  - to test, run in terminal: `python3 -V`
- Open MPI:
  - in macOS with Homebrew, in terminal: `brew open-mpi`
  - in Debian/Ubuntu with Apt, in terminal: `apt-get install openmpi-bin libopenmpi-dev`
  - in Windows, the parallel version of the framework has not been tested yet
- SPUX Package, in terminal:

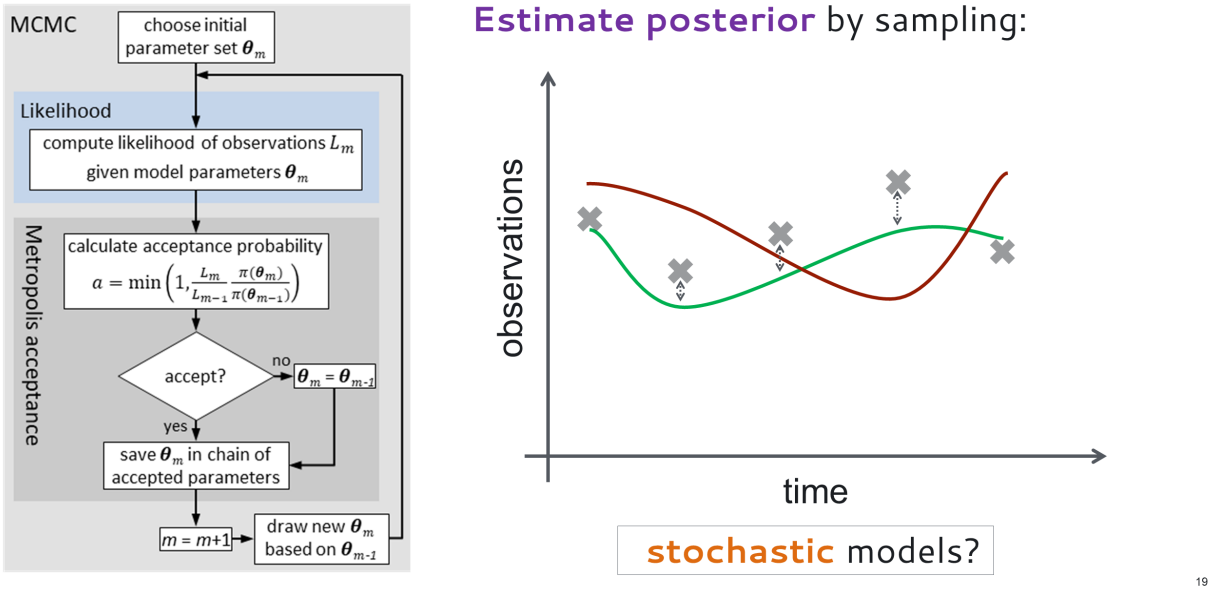


Fig. 2: Markov Chain Monte Carlo (MCMC) for deterministic models M - likelihood of model parameters can be estimated by simply executing the model.

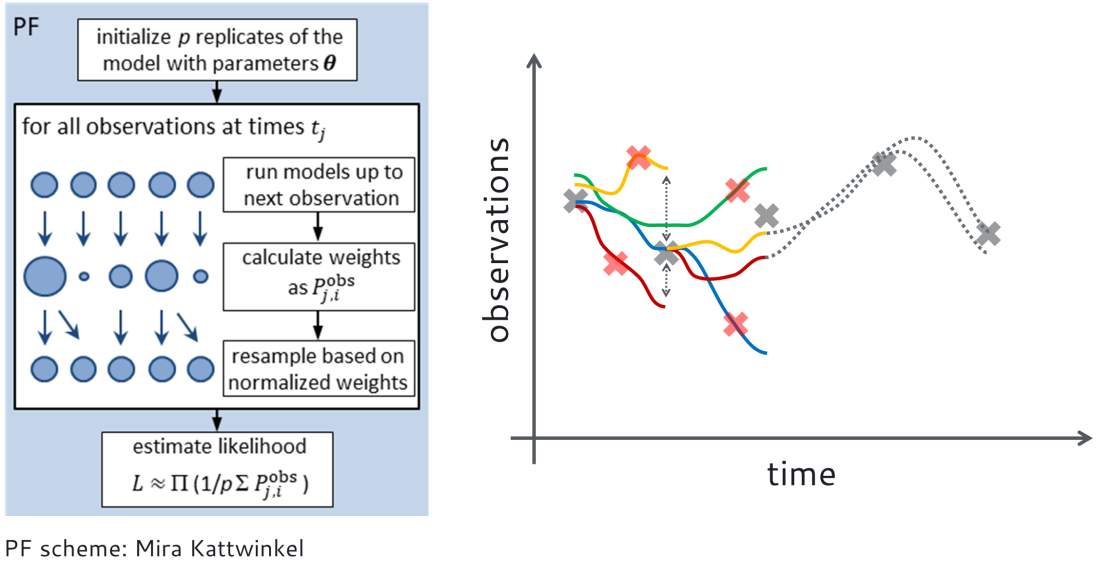


Fig. 3: Particle Filter (PF) of the stochastic model realizations in order to estimate the likelihood of the model parameters marginalized over all possible scenarios. Note that particles (stochastic model realizations) are adaptively filtered using dataset, making this algorithm very computationally efficient.

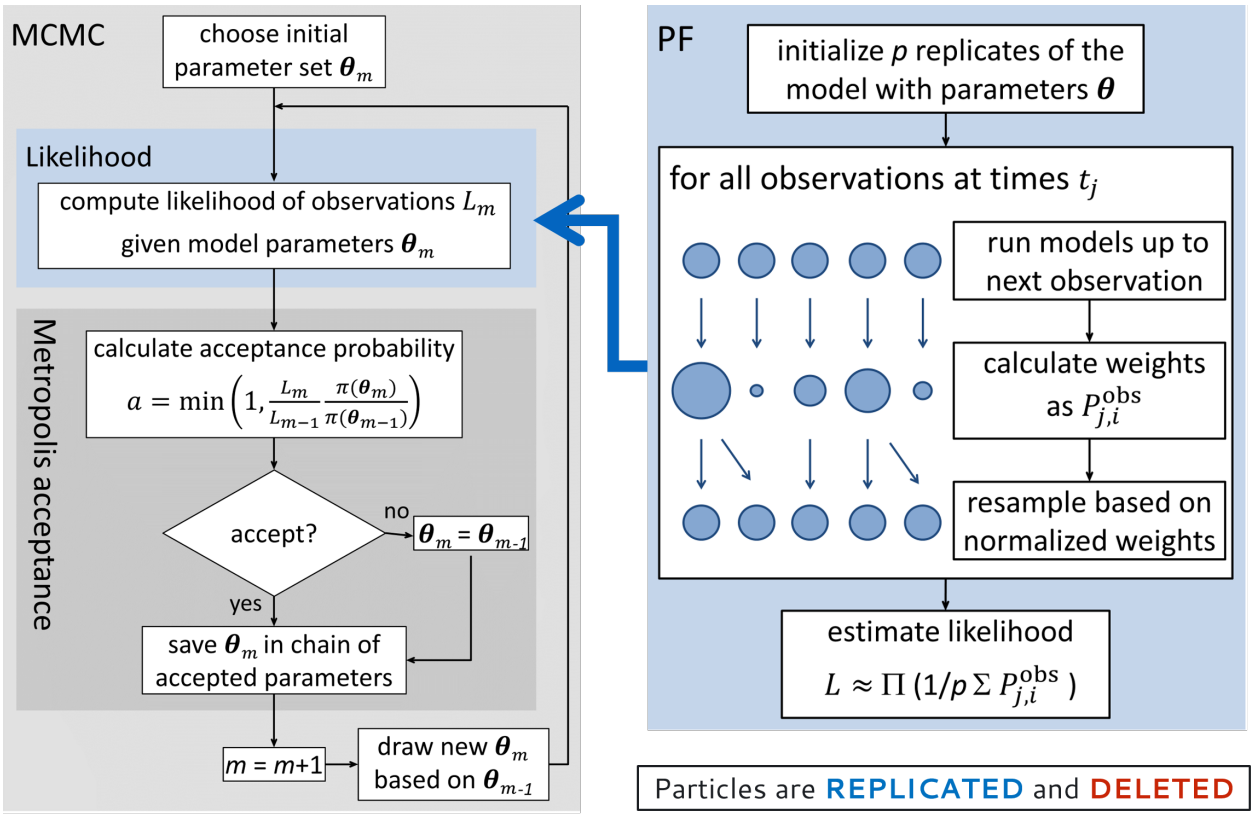


Fig. 4: Markov Chain Monte Carlo (MCMC) and Particle Filtering used in the SPUX framework. Notice, that in most of the real applications using SPUX, an adaptive ensemble affine invariant sampler (EMCEE) with multiple chains is used instead of the conventional single-chain Metropolis-Hastings MCMC.

```
$ pip3 install --user --upgrade pip
$ pip3 install --user spux
```

Remember, that if you reinstall or somehow else change your MPI library, you must reinstall `mpi4py` package by running in terminal:

```
$ pip3 install --user --upgrade pip
$ pip3 install --user --upgrade --force-reinstall --no-cache-dir mpi4py
```

1.2.2 Additional prerequisites

Depending on the programming language of your model, additional prerequisites might be needed:

- R driver: R package, and in terminal: `$ pip3 install --user rpy2`,
- Julia driver: Julia package `PyCall`, and in terminal: `$ pip3 install --user julia`,
- Fortran driver: Fortran compiler (if needed), and in terminal: `$ pip3 install --user ctypes`,
- C/C++ driver: C/C++ compiler (if needed), and Swig (http://www.swig.org/) code wrapper,
- Java driver: Java SDK, and in terminal: `$ pip3 install --user Jpype1`.

If you additionally want the generated LaTeX report source files to be compiled to the PDF files, then you will need to have the `pdflatex` installed (the procedure varies depending on the OS and distribution and hence is not described here).

1.2.3 Stable release

To install spux, run this command in your terminal:

```
$ pip install spux
```

This is the preferred method to install spux, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

To update when a newer release becomes available, run in your terminal:

```
$ pip install --user --upgrade spux
```

1.2.4 Latest release

To install the latest development version of spux, run these commands in your terminal:

```
$ git clone --single-branch --branch test https://gitlab.com/siam-sc/spux.git
$ cd spux
$ python setup.py install --user
```

To update when a newer version of the `test` branch becomes available, run in your terminal (within the `spux` directory from above):

```
$ git pull
$ python setup.py install --user
```

If you use latest version of SPUX, please also refer to the latest version of documentation at: https://spux.readthedocs.io/en/latest/.

1.2.5 From sources

The sources for spux can be downloaded from the https://gitlab.com/siam-sc/spux.

You can also clone the public repository:

```
$ git clone https://gitlab.com/siam-sc/spux.git
```

1.3 Tutorial

This section guides you through a tutorial for an example model and usage pattern of SPUX. For peculiarities regarding the coupling your model with the SPUX framework, refer to *Customization*.

1.3.1 Overview

Commands should be executed in Python terminal, or inside a Python script, or in a Jupyter notebook. To learn how to write your own custom scripts and configure spux, first read through the rest of this section and take a look at the examples suite.

1.3.1.1 Editor

Try one of the following cross-platform editors (you can also use Vim or Emacs, of course):

- Spyder - similar UI as R,
- PyCharm - proprietary,
- VS Code - works very well with GitLab integration extensions - give it a try!

1.3.1.2 Model stochasticity

SPUX supports two types of models for the Bayesian inference:

- Deterministic - model evaluation is uniquely determined by the inputset and parameters,
- Stochastic - model evaluation is additionally driven by a random variable/process.

Bayesian inference of model parameters for deterministic models is often less difficult, since a simple so-called `Direct` likelihood can be used, which, for any given parameters, is analytically computed from the specified error model. Error model describes a probabilistic distribution for observational data, conditional on the true model evaluation (referred to as model prediction).

For stochastic models, in addition to uncertain model parameters, also the uncertain model evaluations (predictions, sometimes referred to as states) need to be inferred. To this end, the error model alone is often not sufficient to analytically compute the marginal likelihood of a given dataset and model parameters. Currently SPUX supports the Particle Filter approximation of such marginal likelihood, used in this tutorial.

As SPUX framework has a focus on the Bayesian inference in stochastic models, in the present tutorial we also focus on the stochastic models, with an example of a `Randomwalk`. `Straightwalk` is another educational model available in SPUX, which is simply a stripped down version (with the randomness eliminated) of the stochastic `Randomwalk`

model. To learn more about the `Straightwalk` and how to use SPUX with deterministic models, we recommend to read the tutorial below and then refer to analogous example scripts in the respective example directory at: examples/straightwalk/.

1.3.1.3 Replicate datasets

In some applications, multiple replicates of observational datasets are available, with each replicate dataset corresponding to the same (assumed to be unknown) model parameter values, but different independent stochastic model evaluations (for instance, w.r.t. the seed of the pseudo-random number generator.) Examples of such replicates could be independent datasets from several consecutive sufficiently separated time periods, or even several simultaneously collected measurements from independent experimental sites.

If this is the case, each dataset (and the respective model inputset) can be treated statistically independently but at the same time fully integrated into the SPUX framework by simply merging individual respective (direct or marginal) likelihoods into a single `Replicates` likelihood. For the sake of simplicity, this tutorial only considers a single dataset. To learn more about how to use SPUX with multiple independent replicate datasets, we recommend to read the tutorial below and then refer to analogous example scripts in the respective example directory at: examples/randomwalk-replicates/.

1.3.2 Randomwalk (serial)

Here we provide an elaborate description of the SPUX framework setup and simulation execution for an example of a simple Randomwalk model.

1.3.2.1 Model description

The model describes a stochastic one-dimensional walk on integers, with a prescribed (let's say, genetically) `stepsize`. Starting at the location given by the `origin` parameter, a randomwalk takes a random step of size `stepsize` either to the left or to the right, with direction distribution depending on the `drift` parameter. Inaccurate observations at several times of the randomwalk's position are available, with unknown observational error distribution, assumed to be Gaussian with zero mean and standard deviation given by the parameter `$\sigma$`.

All files required throughout this example (and some more) could be found in examples/randomwalk/, which we assume is the current working directory where commands are executed. This means that all `import module` statements will import the corresponding `module.py` script from this directory (or an already installed external Python module). All imports starting with `from spux import ...` import modules that are built-in in the spux module, and we use relative links starting with spux/ for a corresponding file in the GitLab repository.

The randomwalk model is a built-in module in spux and can be found at spux/models/randomwalk.py:

```
from scipy import stats
import numpy

from spux.models.model import Model
from spux.utils.annotate import annotate

class Randomwalk (Model):
    """Class for Randomwalk model."""

    # no need for sandboxing
    sandboxing = 0

    # construct model
    def __init__ (self, stepsize=1):
```

(continues on next page)

(continued from previous page)

```
        self.stepsize = stepsize

    # initialize model using specified 'inputset' and 'parameters'
    def init (self, inputset, parameters):
        """Initialize model using specified 'inputset' and 'parameters'."""

        # base class 'init (...)' method - OPTIONAL
        Model.init (self, inputset, parameters)

        self.position = parameters ["origin"]
        self.drift = parameters ["drift"]

        self.time = 0

    # run model up to specified 'time' and return the prediction
    def run (self, time):
        """Run model up to specified 'time' and return the prediction."""

        # base class 'run (...)' method - OPTIONAL
        Model.run (self, time)

        # pre-generate random variables for all steps
        steps = time - self.time
        distribution = stats.uniform (loc=-1, scale=2)
        rvs = distribution.rvs (steps, random_state=self.rng)

        # update position (e.g., perform walk)
        directions = numpy.where (rvs < self.drift, 1, -1)
        self.position += self.stepsize * numpy.sum (directions)

        # update time
        self.time = time

        # return results
        return annotate ([self.position], ['position'], time)
```

In the source code above, `Randomwalk` class has a constructor (note the underscores!) `__init__ (self, ...)`, which is called when constructing model by `model = Randomwalk (stepsize=1)`. The argument `self` is a pointer to the object itself, analogous to `this` in C/C++. Additional methods include:

- `init (self, inputset, parameters)` - initialize model with the specified `inputset` and `parameters`,
- `run (self, time)` - run model from the current time up to the specified `time`.

Note, that the "current time" in the above is set in `init (...)` or the previous call of `run (...)` and is handled differently in different models (in this example: simply saving it to `self.time`).

The `annotate (values, labels, time)` method packages model predictions stored in `values` to a `pandas.DataFrame` with the specified list of `labels` for the elements if the `values` and with the requested `time`.

1.3.2.2 SPUX configuration

Apart from the actual model, we also need to specify several auxiliary configuration files for observational datasets, statistical error model (i.e. a probabilistic distribution of the observations conditional on the specified model pre-

diction), and prior distribution of the parameters. Actual dataset files are located in the datasets directory at examples/randomwalk/datasets/.

The script to load the dataset into `pandas` DataFrames (a default container for dataset management in SPUX, see https://pandas.pydata.org) is located in examples/randomwalk/dataset.py.

```
import pandas
dataset = pandas.read_csv ('datasets/dataset.dat', sep=",", index_col=0)
```

Error model is defined in examples/randomwalk/error.py as an object with a method `distribution (prediction, parameters)` which returns a distribution of the model observations (dataset):

```
from scipy import stats
from spux.distributions.tensor import Tensor

# define an error model
class Error (object):

    # return an error model (distribution) for the specified prediction and parameters
    def distribution (self, prediction, parameters):

        # specify error distributions using stats.scipy for each observed variable␣
↪independently
        # available options (univariate): https://docs.scipy.org/doc/scipy/reference/
↪stats.html
        distributions = {}
        distributions ['position'] = stats.norm (prediction['position'], parameters[r'
↪$\sigma$'])

        # construct a joint distribution for a vector of independent parameters by␣
↪tensorization
        distribution = Tensor (distributions)

        return distribution

error = Error ()
```

Prior distribution is defined in examples/randomwalk/prior.py:

```
# specify prior distributions using stats.scipy for each parameter independently
# available options (univariate): https://docs.scipy.org/doc/scipy/reference/stats.
↪html

from scipy import stats

from spux.distributions.tensor import Tensor
from spux.utils import transforms

distributions = {}

distributions ['origin'] = stats.uniform (loc=50, scale=100)
distributions ['drift'] = stats.uniform (loc=-1, scale=2)
distributions [r'$\sigma$'] = stats.lognorm (**transforms.logmeanstd (logm=10,␣
↪logs=1))

# construct a joint distribution for a vector of independent parameters by␣
↪tensorization
prior = Tensor (distributions)
```

Within the context if this illustrative `Randomwalk` example, we also make use of the (optional) exact ("the truth") parameter values as well as the model predictions (without the observational noise) which are set at examples/randomwalk/exact.py:

```
# exact parameters
parameters = {}
parameters ['origin'] = 100
parameters ['drift'] = 0.2
parameters [r'$\sigma$'] = 10

# exact predictions
import os, pandas
filename = 'datasets/predictions.dat'
if os.path.exists (filename):
    predictions = pandas.read_csv (filename, sep=",", index_col=0)
else:
    predictions = None

# dictionary for exact parameters and predictions
exact = {'parameters' : parameters, 'predictions' : predictions}
```

The datasets (and the predictions) in the examples/randomwalk/datasets/ directory were generated using the above exact model parameters from examples/randomwalk/exact.py by the synthesis script based on the built-in SPUX data generation method: examples/randomwalk/script_synthesize.py

```
from spux.models.randomwalk import Randomwalk
from exact import exact
from error import error

model = Randomwalk ()
parameters = exact ['parameters']
steps = 1000
period = 20
times = range (period, period + steps, period)

sandbox = None

from spux.utils.seed import Seed
seed = Seed (2)

from spux.utils import synthesize
synthesize.generate (model, parameters, times, error, sandbox = sandbox, seed = seed)
```

We would also like to emphasize, that in the above scripts we generously use LaTeX syntax within labels for parameters, predictions, and observations. The benefit of such scrupulous naming will become evident from the generated plots within this tutorial, where all axes labels are LaTeX-formatted mathematical symbols. Notice, that for a LaTeX syntax to be supported in Python, one must prepend the string with the `r` letter (as in "raw").

In order to give you a better overview of the datasets, the error model, the prior distribution, and (optional) exact parameters values for a reference, consider running a preparation script examples/randomwalk/plot_config.py:

```
# generate config
import script
del script

# plotting class
from spux.plot.mpl import MatPlotLib
from exact import exact
```

(continues on next page)

(continued from previous page)

```
plot = MatPlotLib (exact = exact)

# plot dataset
plot.dataset ()

# plot marginal prior distributions
plot.priors ()

# plot marginal error model distributions
plot.errors ()

# generate report
from spux.report import generate
generate.report (authors = r'Jonas {\v S}ukys')
```

The above script plots model observations (datasets), marginal prior distributions of model parameters, and marginal error model distributions for specified model prediction and parameters and by default saves them in the `fig` directory under multiple file formats (PDF, EPS, SVG, PNG), additionally including a caption file `*.cap` containing the description of the figure contents:

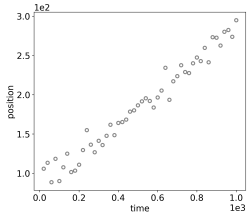


Fig. 5: Observational dataset.

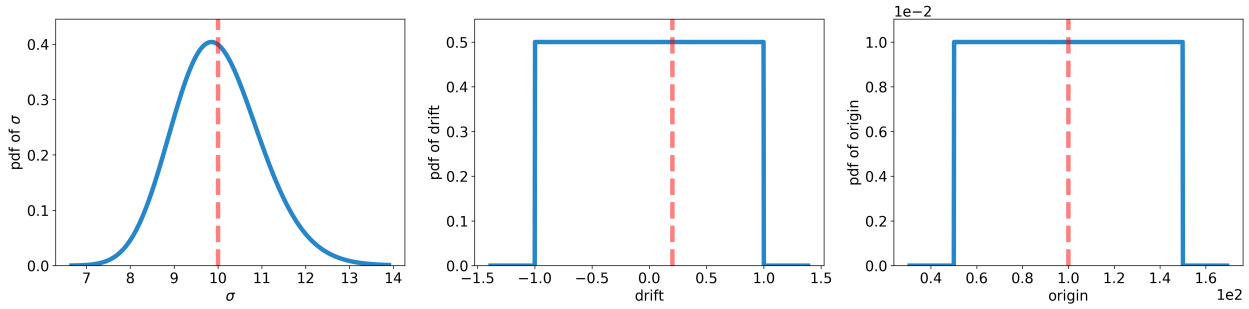


Fig. 6: Marginal distributions (prior).

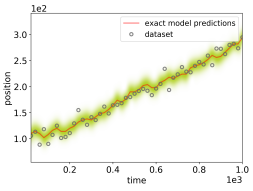


Fig. 7: Observational dataset and the associated error model, evaluated using exact model predictions and exact model parameters. The circles (or thick dots) indicate the dataset values, the thick solid line indicates the model predictions used in the error model. The shaded green regions indicate the density of the error model distribution.

In addition to the plots, an auxiliary `report` directory is created (can be changed by setting `reportdir` in `sampler.setup (...)`), including information regarding SPUX framework setup. Each report file in the `report` directory is saved using three different formats:

- `.dat` - a (cloudpickle) binary dump of the respective Python object or dictionary,
- `.txt` - a formatted ASCII table to be easily read directly,
- `.tex` - a formatted LaTeX table to be included in a LaTeX report.
- `.cap` - a caption with the table title and description of the table contents.

At the end of the `plot_config.py` script all generated plots are compiled into a LaTeX report under directory `latex`, where a PDF report is also generated if the `pdflatex` can be found in the system (which is not a SPUX dependency). The PDF report for this example can be downloaded here. The report is continuously overwritten with the newest plots and tables, and contains separate sections for the SPUX framework setup, and, as described in the following sections, the results of the inference run and the computational performance amd runtime profiling reports.

As already hinted in the report above, the main script examples/randomwalk/script.py, uses the above auxiliary scripts to configure SPUX:

```
# === Randomwalk model

# construct Randomwalk model
from spux.models.randomwalk import Randomwalk
model = Randomwalk (stepsize = 1)

# === SPUX

# LIKELIHOOD
# for marginalization in stochastic models, construct a Particle Filter likelihood
from spux.likelihoods.pf import PF
likelihood = PF (particles = [4, 8], threshold=-5)

# SAMPLER
# construct EMCEE sampler
from spux.samplers.emcee import EMCEE
sampler = EMCEE (chains = 8)

# ASSEMBLE ALL COMPONENTS
from error import error
from dataset import dataset
from prior import prior
likelihood.assign (model, error, dataset)
sampler.assign (likelihood, prior)
```

In this script, different additional components, such as model, likelihood, and sampler, are created. Afterwards, all these components are merged together by assigning according to the logical dependencies. In the future, SPUX will provide a `spux.utils.assign` module containing a `assign` function, which takes a list of components (in any order) as an argument, tries to "automagically" perform all needed assignments (assuming all components are directly derived from the respective SPUX component base classes) and returns the resulting top-level component (in this example, the `sampler`):

```
from spux.utils.assign import assign
components = [model, error, dataset, likelihood, prior]
sampler = assign (sampler, components)
```

The corresponding SPUX configuration table from `report/randomwalk_config.txt` reports the selected class for each SPUX component, together with the selected constructor arguments:

```
================================================================================================================
SPUX components configuration                                                                                  ␣
↪
================================================================================================================
Component  | Class      | Options                                                                              ␣
↪
---------- + ---------- + ------------------------------------------------------------------------------------
↪----------------------------------------
Model      | Randomwalk | stepsize=1                                                                           ␣
↪
Likelihood | PF         | particles=[4, 128], adaptive=True, accuracy=0.1, margin=0.
↪05, threshold=-5, factor=2, log=1, noresample=0
Sampler    | EMCEE      | chains=8, a=2.0, attempts=100, reset=10                                              ␣
↪
================================================================================================================
```

In the SPUX configuration above, in the constructor arguments of the `PF` marginal likelihood estimator, the number of particles, instead of being a fixed number, is set to a list indicating the minimum and maximum number of particles to be used adaptively, starting with the minumum number of particles and then iteratively taking into account the feedback from the empirical standard deviations of these estimates.

1.3.2.3 SPUX results

It is a good idea to keep this main `script.py` separate from the scripts that will actually run SPUX, in order to have flexibility for the later customization of output location, sampling duration, and the targeted hardware resources.

To achieve this, we import the main configuration script and initiate the execution of the framework in a separate script named (you will see later why such name) examples/randomwalk/script_serial.py:

```
# === SCRIPT

from script import sampler

# === SAMPLING

# SANDBOX
# use fast tmpfs
from spux.utils.sandbox import Sandbox
sandbox = Sandbox (path = None, target = '/dev/shm/spux-sandbox')

# SEED
from spux.utils.seed import Seed
seed = Seed (8)

# init executor
sampler.executor.init ()

# setup sampler
sampler.setup (sandbox = sandbox, verbosity = 1, seed = seed, lock = 50)
#sampler.setup (sandbox = sandbox, verbosity = 1, seed = seed, lock = 50, thin = 4)

# init sampler (use prior for generating starting values)
sampler.init ()

# generate samples from posterior distribution
sampler (12)
```

(continues on next page)

(continued from previous page)

```
# exit executor
sampler.executor.exit ()
```

Within `sampler.setup (...)`, various additional (optional) sampling options can be set, including:

- `sandbox` instance - target filesystem and directory for the SPUX sandbox,
- `verbosity` - [0 to infinity] limits the depth of verbosity in the hierarchy of SPUX components,
- `seed` instance - initial seed array for the pseudo random number generators,
- `lock` - batch index to lock sampler's feedback to likelihood (e.g. adaptive `PF`) [default: `None`],
- `thin` period - to save only every `thin`-th sample batch and information [default: `None`].

The adaptive number of particles in `PF` is meant to be used only during the initial burn-in period, after which, if the number of particles is not completely stationary, the adaptation should be locked. The optional `lock` argument in the `sampler.setup (...)` can be used to stop the adaptation of the number of particles in `PF` and lock the most recent value until the end of sampling.

The `sandbox` in this example script is configured to use the fast virtual node-local RAM-based Linux filesystem called `tmpfs`. Note, that the exact path to the local `tmpfs` might differ depending on the Linux distribution. For local testing, a user might prefer to temporarily switch to a conventional filesystem and set a different desired sandbox `path`, leaving `target=None`. For production runs on high performance computing clusters, we recommend either to use the default `tmpfs`, or, if the amount of system memory is a limiting factor, to set `target` to the scratch file system of the cluster. Node-local (not shared) scratch filesystems are preferable, due to their better performance and the lack of restrictions regarding any storage quotas. If target is a shared (hence not a node-local) filesystem (the worst case scenario, for development only), you can also set `path`, where a corresponding symlink pointing to the specified `target` will be created.

Regarding the auxiliary calls `sampler.executor.init ()` and `sampler.executor.exit ()`, you must have them, and in this particular order, i.e. wrapping every other call of the SPUX framework (apart from the calls in the main configurations script).

The main script then generates an `output/` directory (can be changed by setting `outputdir` in `sampler.setup (...)`) with files containing posterior samples and supporting information; multiple files of each type will be generated for each checkpoint, with the default period being 10 minutes:

- `samples-*.csv` - a CSV file containing comma-separated posterior samples of parameters,
- `samples-*.dat` - a binary file (`cloudpickle`) containing posterior samples of parameters,
- `infos-*.dat` - a binary file (`cloudpickle`) containing a `list` of supporting information,
- `pickup-*.dat` - a binary file (`cloudpickle`) containing a dictionary of sampler pickup information.

The supporting files `infos-*.dat` contain detailed information about each component in the hierarchical assignment structure specified by the main configuration script. In the particular example, when loaded (see following paragraphs) will contain a list of dictionaries for each draw of the posterior parameters from the sampler. For samplers supporting multiple MCMC chains, each draw provides as many samples as there are chains, and hence the list in the `infos-*.dat` will be shorter than the list of all posterior parameters by a factor of the number of chains. The structure of each element in the list of loaded `infos-*.dat` can be inferred from the corresponding info generation routines for each SPUX component (look for `info = {...}`).

The sampler pickup files `pickup-*.dat` contain all information needed to continue sampling past the respective checkpoint (will be discussed in detail in the following sections).

In addition to the `output` directory, the auxiliary `report` directory is also updated, appending information regarding computational environment (in `report/randomwalk_environment.txt`):

```
============================================================
Computational environment
============================================================
Descriptor          | Value
------------------- + ----------------------------------------
Timestamp           | 2019-05-23 00:58:01
Version             | 0.3.0
GIT branch (plain)  | dev_jonas
GIT revision        | f98e24ae81841a98a39d7d9e3b655ae6cdc97d4f
============================================================
```

required computational resources (in `report/randomwalk_resources.txt`):

```
=================================================================================================
Required computational resources                                                                 ␣
↪
=================================================================================================
Component  | Class      | Task       | Executor | manager | workers | resources |␣
↪cumulative
---------- + ---------- + ---------- + -------- + ------- + ------- + --------- + ----
↪------
Model      | Randomwalk | -          | -        |       0 |       1 |         1 |  ␣
↪     1
Likelihood | PF         | Randomwalk | Serial   |       0 |       1 |         1 |  ␣
↪     1
Sampler    | EMCEE      | PF         | Serial   |       0 |       1 |         1 |  ␣
↪     1
=================================================================================================
```

sampler setup (in `report/randomwalk_setup.txt`):

```
========================================
Setup argument list
========================================
seed | thin | lock (batch) | lock (sample)
---- + ---- + ------------ + -------------
[8]  | None | 100          | 800
========================================
```

the cumulative number of model evaluations in each SPUX component (in `report/randomwalk_evaluations.txt`):

```
=================================================
Number of model evaluations
=================================================
Component  | Class      | tasks | sizes | cumulative
---------- + ---------- + ----- + ----- + ----------
Model      | Randomwalk |     1 |     1 |          1
Likelihood | PF         |   128 |     1 |        128
Sampler    | EMCEE      |    1K |   128 |       128K
=================================================
```

and the structure of the information available in `infos-*.dat`:

```
=================================================================================================
SPUX infos structure                                                                             ␣
↪                                                                                                ␣
↪
```

(continues on next page)

(continued from previous page)

```
================================================================================================================================
Component  | Class | Fields                                                                                                    ␣
↪                                                                                                                              ␣
↪                        | Iterators for infos
---------- + ----- + ------------------------------------------------------------------------
↪--------------------------------------------------------------------------------------------
↪---------------------- + -------------------
Sampler    | EMCEE | proposes, infos, parameters, posteriors, accepts, likelihoods,␣
↪timing, priors, timings, index, feedbacks, feedback, resets                                                                   ␣
↪                          | 0 - 8
Likelihood | PF    | particles, successful, variances, sources, timing, MAP, timings,␣
↪redraw, weights, predictions_prior, estimates, variance, errors_prior, predictions,␣
↪traffic, avg_deviation | -
================================================================================================================================
```

If `self.sandboxing == 1` is set for the model, a `sandbox` directory is created (or a custom name, if custom sandbox is specified). This directory is populated with nested sandboxes for each sample, chain, replicate, likelihood, and model. For sandboxes in the `local` mode (for non-shared filesystems), nested directories are replaced by a single directory with a long name indicating the underlying hierarchy. Please also note, that sandboxes in `local` mode are tentative, meaning that they are only created once `self.sandbox ()` is executed for the first time. If `trace=True` is additionally specified in the `sampler.setup (...)`, this directory contains the stored sandboxes of all samplers, likelihoods and models, including all the generated results. However, these results would be easily accessible only if sandboxes are placed in a shared filesystem and a non-local mode is used.

In the future, SPUX will explicitly save the sandbox with a complete trace of the model output (independently of the value set to `trace`) for the approximated joint maximum a posteriori (MAP) model parameters and states under the directory specified in `sampler.setup (...)` by `MAPdir` with the default being `'MAP'`. In addition, there will be an option to save only a (possibly thinned) collection of posterior sandboxes by specifying `trace = 'posterior'` in `sampler.setup (...)`.

An example analysis script to load and visualize results from the `output/` directory is available at examples/randomwalk/plot_results.py:

```
# === load results

from spux.io import loader
samples, infos = loader.reconstruct ()

# === plot

# burnin sample batch
burnin = 70

# plotting class
from spux.plot.mpl import MatPlotLib
from exact import exact
plot = MatPlotLib (samples, infos, burnin = burnin, exact = exact)

# plot unsuccessful posteriors
plot.unsuccessfuls ()

# plot resets of stuck chains
plot.resets ()

# compute and report approximated maximum a posterior (MAP) parameters estimate
plot.MAP ()
```

(continues on next page)

(continued from previous page)

```
# plot samples
plot.parameters ()

# plot evolution of likelihoods
plot.likelihoods ()

# plot evolution of likelihood accuracies
plot.accuracies ()

# plot evolution of likelihood particles
plot.particles ()

# plot redraw rates
plot.redraw ()

# plot evolution of acceptances
plot.acceptances ()

# plot timestamps
plot.timestamps ()
timestamps = [ "evaluate", "routings", "wait"]
timestamps += [ "init", "init sync", "run", "run sync"]
timestamps += ["errors", "errors sync", "resample", "resample sync"]
plot.timestamps (keys = timestamps, suffix = '-cherrypicked')

# === remove burnin

samples, infos = loader.tail (samples, infos, batch = burnin)
plot = MatPlotLib (samples, infos, burnin = burnin, tail = burnin, exact = exact)
plot.MAP ()

# plot autocorrelations
plot.autocorrelations ()

# compute and report effective sample size (ESS)
plot.ESS ()

# plot marginal posterior distributions
plot.posteriors ()

# plot pairwise joint posterior distributions
plot.posteriors2d ()

# plot pairwise joint posterior distribution for selected parameter pairs
plot.posterior2d ('origin', 'drift')

# plot posterior model predictions including datasets
plot.predictions ()

# plot quantile-quantile comparison of the error and residual distributions
plot.QQ ()

# show metrics
plot.metrics ()

# generate report
```

(continues on next page)

(continued from previous page)

```
from spux.report import generate
generate.report (authors = r'Jonas {\v S}ukys')
```

This script uses some built-in plotting routines available in spux/plot/mpl.py module. However, the user is free to use only the loading parts and choose how to visualize the results using other established data visualization libraries, including the built-in visualization module `pandas.plotting` in `pandas` for the visualization of the `pandas.DataFrame` objects. Also check out NumFOCUS.

Note, that for runtime saving purposes, actual linked example scripts in repository usually could be setup for smaller computational resources (i.e. fewer particles, chains, samples, etc.), and hence the following example plots for SPUX results could differ from your versions (please check the corresponding entries in the provided configuration and setup tables above).

The above analysis script generates multiple plots of the results, but firstly it is most useful to look at the `unsuccessfuls` and `resets` plots.

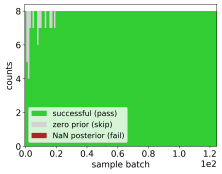


Fig. 8: Diagnostics of the posterior sampler, indicating the successes and failures of the likelihood estmation procedures. Legend: green - successfully passed, gray - estimation skipped due to (numerically) zero prior, red - estimation failure due to failed model simulatios and/or failed PF filtering.

Note, that skipped likelihoods are not scheduled in the executor and might result in a temporary (for that particular parameter proposal) parallelization imbalances due to the lack of tasks to process.

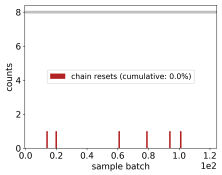


Fig. 9: The report for the number of resets (re-estimation of the marginal likelihood) for stuck Markov chains, including the cumulative percentage of resets relative to the total number of samples.

From the diagnostic plot above, we determine that the inference was (tentatively) successful: not many failed (NaN) or skipped (zero prior) likelihood evaluations, and a negligible amount of total chain resets (likelihood re-estimations).

The next most important plots are the `parameters` plot, reporting the sampling progress of all sampler chains,

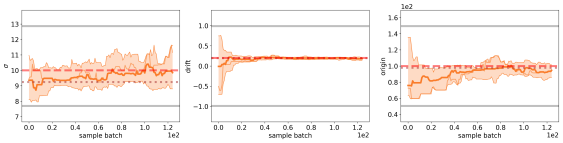


Fig. 10: Markov chain parameters samples. The solid lines indicate the median and the semi-transparent spreads indicate the 5% - 95% percentiles accross multiple concurrent chains of the sampler. An auxiliary semi-transparent line indicates an example of such chain. The thick semi-transparent gray lines indicate the interval containing centererd 99% mass of the respective prior distribution. The brown dotted line indicates the estimated maximum a postriori (MAP) parameters values. The red dashed line represents the exact parameter values.

and the progress of the corresponding likelihood, accuracies, particles, redraw, and acceptances plots providing diagnostic information of the algorithmic technicalilies within the `PF` likelihood estimation and Markov chain sampling:

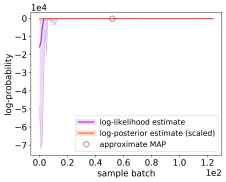


Fig. 11: Log-likelihood and (scaled [TODO: estimate evidence and remove scaling]) log-posterior estimates for the sampled model posterior parameters. The solid lines indicate the median and the semi-transparent spreads indicate the 10% - 90% percentiles accross multiple concurrent chains of the sampler. For log-likelihood, the estimates from the rejected proposed parameters are also taken into account.The brown "o" symbol indicates the posterior estimate at the approximate MAP parameters.

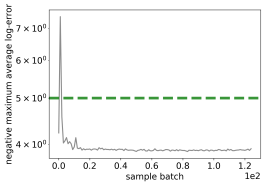


Fig. 12: Maximums of the average (over dataset snapshots) marginal observational log-errors accross multiple concurrent chains of the sampler. The dashed green line indicates the threshold set in the adaptive PF likelihood.

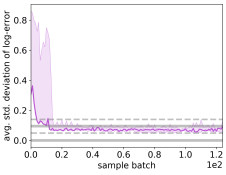


Fig. 13: Average (over dataset snapshots) standard deviations for the estimated marginal observational log-error using the PF. The semi-transparent spread indicates the range (minimum and maximumum) accross multiple concurrent chains of the sampler and the solid line indicates the value of the chain with the largest estimated log-likelihood. The solid thick gray line above the same line for a zero value reference indicates the specified accuracy and the dashed thick gray lines indicate the specified margins - all specified within the adaptive PF likelihood.

We estimate the burnin period to last for approximately 70 batches of EMCEE samples from multiple chains. We use this information to generate all subsequent plots with the initial bias already removed by setting `burnin=70` in the `reconstruct` method for loading results in the `plot_results.py` script above. We note, that `lock` sample batch reported in the `setup` table above must not exceed the selected `burnin` to ensure that the number of particles in `PF` likelihood is fixed throught the posterior sampling (and hence avoid any potential bias due to the adaptivity process). The resulting inference plots provide a detailed insight into posterior parameters and model predictions distributions, as well as the performance of the overall Bayesian inference.

In addition to the plots in the `fig` directory, the auxiliary `report` directory is also updated, appending information regarding the approximate maximum a posterior parameters values (in `report/randomwalk_MAP.txt`):

```
================================================
Maximum A Posteriori (MAP) estimate parameters
================================================
$\sigma$ | drift    | origin
-------- + -------- + --------
```

(continues on next page)

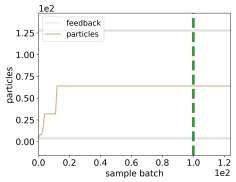


Fig. 14: The adaptavity of the number of particles in the PF likelihood. The brighter line indicates the feedback (recommendation) of the adaptation algorithm, and the darker line indicates the actual number of used particles. The semi-transparent thick gray lines indicate the limits for minimum and the maximum number of allowed particles in the PF likelihood.

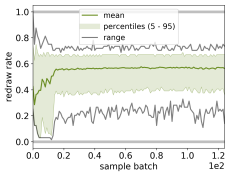


Fig. 15: Particle redraw rates (the fraction of surviving particles) in the PF likelihood estimator. The solid line indicates the mean, the semi-transparent spreads indicate the 5% - 95% percentiles, and the dotted lines indicate the range (minimum and maximum) accross multiple concurrent chains of the sampler.

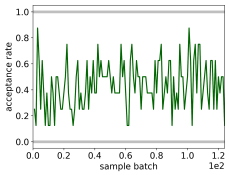


Fig. 16: Acceptance rate (accross multiple concurrent chains of the sampler) for the proposed parameters samples.

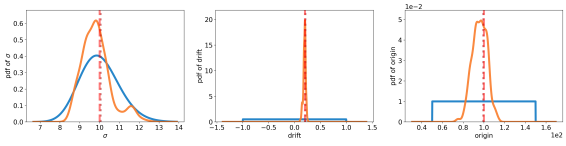


Fig. 17: Marginal posterior (orange) and prior (blue) distributions of model parameters. The brown dotted line indicates the estimated maximum a postriori (MAP) parameters values. The red dashed line represents the exact parameter values.

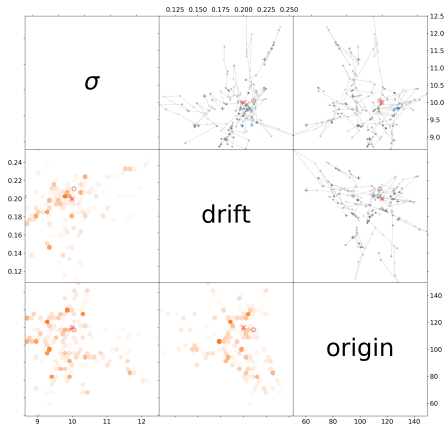


Fig. 18: Joint pairwise marginal posterior distribution of all model parameters, including the corresponding Markov chains from the sampler. Legend: thick semi-transparent gray lines - intervals containing centererd 99% mass of the respective prior distribution, blue “+” - initial parameters, brown “o” - approximate MAP parameters, red “x” - the exact parameters, thin semi-transparent gray lines and dots - concurrent chains, orange hexagons - histogram of the joint pairwise marginal posterior parameters samples.

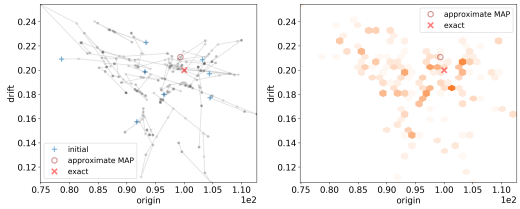


Fig. 19: Joint pairwise marginal posterior distribution of origin and drift, including the corresponding Markov chains from the sampler. Legend: thin semi-transparent gray lines and dots - concurrent chains, orange hexagons - histogram of the joint pairwise marginal posterior parameters samples, blue “+” - initial parameters, brown “o” - approximate MAP parameters, red “x” - the exact parameters.

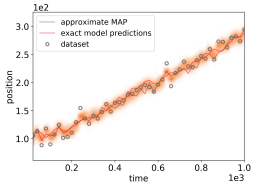


Fig. 20: Posterior distribution of model predictions for the observational dataset. The shaded orange regions indicate the log-density of the posterior model predictions distribution at the respective time points, the brown line indicates the approximate MAP model prediction., the red line represents the exact model prediction values.

Fig. 21: Quantile-quantile distribution comparison between the prediction residuals and the specified error model.

(continued from previous page)

```
9.24e+00 | 1.95e-01 | 9.78e+01
================================================
```

and various metrics for inference efficiency (in `report/randomwalk_metrics.txt`):

```
=================================================================================================================
Metrics for the inference efficiency.
↪
=================================================================================================================
Metric                                      | Value
↪
------------------------------------------- + ----------------------------------------
↪------------
Maximum A Posteriori (MAP) estimate         | batch:52, chain:2, sample:418, log-
↪posterior:-2.01e+02
Multivariate Effective Sample Size (mESS)   | not implemented
↪
Multivariate thin period                    | not implemented
↪
Univariate Effective Sample Size (ESS)      | 1 - 125 (across chains), with average 17
↪and sum 140
Univariate thin period                      | 1 - 71 (across chains), with mean 44
↪
=================================================================================================================
```

As with the `plot_config.py` script, at the end of the `plot_results.py` script all generated plots are compiled into a LaTeX report under directory `latex`.

1.3.2.4 SPUX performance

The table of inference runtimes is available in `report/randomwalk_runtimes.txt`:

```
================================================================
Runtimes (excl. checkpointer)
================================================================
Timer                               | Value
----------------------------------- + ---------------------------
Total runtime (excl. checkpointer)  | 2 hours 43 minutes 2 seconds
Average runtime per sample          | 0 hours 0 minutes 10 seconds
Equivalent serial runtime           | 2 hours 43 minutes 2 seconds
================================================================
```

Even without explicitly specifying `informative = 1` in `sampler.setup (...)`, the informative output is enable for the first and the last sample batches. This allows us to generate a simplified and a standard `timestamps` plots, providing an inisight into the runtimes profiles of the very last sample.

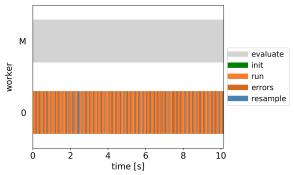


Fig. 22: Timestamps of key methods within a single estimation of the PF likelihood across all parallel workers.

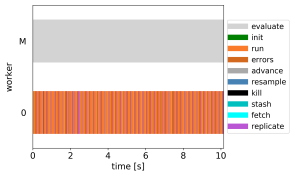


Fig. 23: Timestamps of key methods within a single estimation of the PF likelihood across all parallel workers.

1.3.2.5 Continue sampling

It is possible to continue the sampling process where it finished without any additional re-evaluation of the likelihoods, since all the needed information is already available. The easiest way to continue sampling is to increase the number of samples as needed and then execute the sampling script with the additional command line option `--continue`, for example:

```
$ python script_serial.py --continue
```

This automatically loads the most recent pickup file (see above).

An alternative and a more flexible way to manually customize the `sampler.setup (...)` method by providing

- sample batch `index` - from which to continue (usually incremented last sampled batch index),
- `feedback` - from the previous sample batch `index` (only for `PF` likelihood),

and the `sampler.init (...)` method by providing (for the sampele batch `index` specified above)

- `initial` parameters - respective loaded parameter samples for each sampler chain,
- `posteriors` - the posteriors of the above parameters.

An example of such advanced manual continuation script is available at examples/randomwalk/script_serial_continue.py.

1.3.2.6 Informative output

By default, the computation performance measurements and the additional `infos` from each SPUX component are incorporated into `infos-*.dat` for the first and the last sample batches. Alternatively, you can choose to enable or disable such informative output by setting `informative = 1` or `informative = 0` flag in the `sampler.setup (...)`. Note, however, that `informative = 1` results in an overhead for your inference, and is only advised for the non-production runs during the development stages. You can then use the sample script examples/randomwalk/plot_performance.py to generate plots for timings and scaling for each sample batch, in addition to the timings plots generated by `plot_results` above.

1.3.2.7 Profiling

If you would like to have more detailed information about the execution process, you can enable the profiling by setting `profile=1` in `sampler.sample (...)`. The profiling information after the execution of the framework will be saved in `output/profile.pstats`. You can then use the sample script examples/randomwalk/plot_profiler.py to generate a report and a callgraph plot, which will be saved under the `fig/` directory.

1.3.3 Randomwalk (parallel)

With minimal effort, the above example configuration could be parallelized either on a local machine or on high performance computing clusters.

Note, that NO MODIFICATIONS are needed for this particular `Randomwalk` model class. For a more detailed discussion for other user-specific models written not in pure Python, refer to *Customization*.

1.3.3.1 SPUX executors

To enable parallel execution, each SPUX component in the main configuration script `script.py` can be optionally assigned a parallel executor, specifying the number of parallel workers (for that particular component). In this example, we use a separate script examples/randomwalk/script_executors.py:

```
# === SCRIPT with SPUX components

from script import likelihood, sampler

# === EXECUTORS

# import required executor classes
from spux.executors.mpi4py.pool import Mpi4pyPool
from spux.executors.mpi4py.ensemble import Mpi4pyEnsemble

# create executors with the specified number of parallel workers and attach them
likelihood.attach (Mpi4pyEnsemble (workers=2))
sampler.attach (Mpi4pyPool (workers=2))

# display resources table
sampler.executor.table ()
```

The additional lines at the end of the script regarding the `table` estimate the needed computational resources, which are determined by the number of workers requested in each executor. Note, that a separate core is used for the manager process of each executor. The table with estimated computational resources can be printed to the terminal simply by executing this script (parallel inference is NOT launched at this point):

```
$ python script_executors.py
```

An example output of such table (for this particular configuration) is provided below, where the cumulative number of 31 cores are needed (bottom left cell, scroll horizontally to see everything).

```
=================================================================================================
Required computational resources                                                                 ␣
↪
=================================================================================================
Component  | Class      | Task       | Executor | manager | workers | resources |␣
↪cumulative
---------- + ---------- + ---------- + -------- + ------- + ------- + --------- + ----
↪------
Model      | Randomwalk | -          | -        |       0 |       1 |         1 |    ␣
↪     1
Likelihood | PF         | Randomwalk | Serial   |       0 |       1 |         1 |    ␣
↪     1
Sampler    | EMCEE      | PF         | Serial   |       0 |       1 |         1 |    ␣
↪     1
=================================================================================================
```

1.3.3.2 Launching parallel SPUX

To launch a parallel inference using SPUX with the attached parallel executors as above, we use a separate script for parallel runs which additionally initializes a parallel connector and passes it to the `sampler.executor.init`

(. . .), as given in examples/randomwalk/script_parallel.py:

```
# === CONNECTOR INITIALIZATION

from spux.executors.mpi4py.connectors import utils
connector = utils.select ('auto')

# === SAMPLER with attached executors

from script_executors import sampler

# === SAMPLING

# SANDBOX
# use fast tmpfs
from spux.utils.sandbox import Sandbox
sandbox = Sandbox (path = None, target = '/dev/shm/spux-sandbox')

# SEED
from spux.utils.seed import Seed
seed = Seed (8)

# init executor
sampler.executor.init (connector)

# setup sampler
sampler.setup (sandbox = sandbox, verbosity = 2, seed = seed, lock = 50, thin = 10)

# init sampler (use prior for generating starting values)
sampler.init ()

# generate samples from posterior distribution
sampler (12)

# exit executor
sampler.executor.exit ()
```

Note the initialization of the connector as the very first item in the script. We emphasize, that such ordering of the connector and remaining SPUX components is mandatory to ensure efficient startup.

Assuming you have MPI installed (see *Installation*), the above script can be executed with:

```
$ python script_parallel.py
```

Note, that this is the shortest way to run it, but not the best. In particular, if the simulation crashes, the backtrace of the crash source will be complicated and the parallel processes cleanup will not be as desired. Hence, please consider using a more extensive command for production runs:

```
$ mpiexec -n 1 python -m mpi4py script_parallel.py
```

Note, that any needed additionally required MPI processes will be spawned automatically according to the resource table above, hence for this configuration always use `-n 1`, independently of the workers in executors.

When `script_parallel.py` is executed, the computational resources table described in the preceding section is printed to the terminal and also written to `report/randomwalk_resources.txt`.

1.3.3.3 Remark on MPI libraries

Note, that different MPI libraries provide different implementations, which have different configuration capabilities and might not completely follow the MPI standard.

Here we try to provide a list of useful advices to address any issues that you could encounter.

For `OpenMPI` (launched with `mpirun` or `mpiexec`):

- Specify `--mca mpi_warn_on_fork 0` to avoid annoying warnings (only for `Spawn` connector).
- Specify `--mca pmix_server_max_wait 3600` and `--mca pmix_base_exchange_timeout 3600` to avoid connection failures (not relevant for `Legacy` connector).
- For local testing, specify `--oversubscribe` to use more MPI ranks than you have cores.

For `CrayMPI` (launched with `srun`):

- Specify `--hint=nomultithread` and `--cpu-bind=rank`.

1.3.3.4 Remark on executors

Note, that multiple serial (`Serial`) and parallel (`Pool`/`Ensemble`) executors can be freely selected (that is, serial or parallel) for every SPUX component (`Model`, `Likelihood`, `Sampler`, etc.).

This freedom to use individual parallel executors of arbitrary size for every SPUX component provides a lot of flexibility, but also leaves ample of space for computationally inefficient parallel configurations. Hence, for large production runs, we strongly advice to take the following guidelines into consideration (decreasing in priority):

- Allocate most workers to the executors of the outer-most SPUX component(s) (e.g. `sampler`).
- Avoid parallel executors with few workers (less than 4) - replace them with `Serial` executors.

SPUX will report the percentage of the number of manager cores w.r.t. the total number of all (manager and worker) cores. If the above guidelines are not properly implemented, this fraction could be larger than 20%, in which case SPUX will display an explicit inefficiency warning. Such warning will be skipped for jobs that requested not more than 8 cores in total, to filter out local debug and development runs, where parallelization efficiency is not of the highest importance.

1.3.3.5 Remark on connectors

The default way of initializing a connector is given by

```
from spux.executors.mpi4py.connectors import utils
connector = utils.select ('auto')
```

checks how many MPI ranks are available in `MPI_COMM_WORLD` intra-communicator and automatically selects `Spawn` connector if only one rank is found (and all other workers need to be spawned.)

Since some HPC systems do not allow dynamically spawning new MPI processes, to run SPUX in parallel on 21 cores with MPI, simply execute:

```
$ mpiexec -n 21 python -m mpi4py script_parallel.py
```

Note, that `-n 21` MUST match the total amount of required resources (bottom right cell) reported by running examples/randomwalk/script_executors.py: and stored in `report/randomwalk_resources.txt`.

In this case, more than one MPI rank is available in `MPI_COMM_WORLD` intra-communicator, and hence the `Split` connector is automatically selected.

Some HPC systems not only disallow dynamically spawning new MPI processes, but also do not support modern dynamical MPI process management features such as inter-communicator creation between two existing MPI intra-communicators using `MPI_Comm_accept ()` and `MPI_Comm_connect ()` methods. In this case, you will simply need to use a legacy `Legacy` connector instead of the `Split` connector. Any such specific connector can be also specified manually either by replacing `auto` with `spawn`/`split`/`legacy` in the connector initialization, or by specifying such connector name as a command line argument to the `script_parallel.py`, for example:

```
$ mpiexec -n 21 python -m mpi4py script_parallel.py --connector legacy
```

1.3.3.6 Remark on replicates

For parallel runs with replicate datasets (e.g. `randomwalk-replicates` example), the `Replicates` likelihood performs guided load balancing by evaluating the lengths of the associated datasets together and sorting likelihood evaluations, taking into account adaptive number of particles in the `PF` likelihood as well, if used. Higher priorities are assigned to the likelihoods with longer datasets and large number of particles (if applicable), and lower priorities are assigned to the likelihoods with shorter datasets and smaller number of particles (if applicable). If needed, one can disable such behavior by setting `sort=0` in the constructor `Replicates (...)`. We warn, however, that depending on the executor configuration, disabling sorting (and hence reverting to non-guided load balancing) might lead to inefficient parallelization.

1.3.3.7 Performance progress

The performance plots of the last sample from a parallel simulation are also generated with `plot_results.py`. Additional highly technical parallel performance plots can be generated by executing `plot_performance.py`, provided that `informative = 1` was used in `sampler.setup (...)`, and provide a summarized insight into the computation and communication balance within the parallel `PF` resampling stages, over the evolution of the entire sampling process.

As with the `plot_config.py` and `plot_results.py` scripts, at the end of the `plot_performance.py` script all generated plots are compiled into a LaTeX report under directory `latex`.

1.3.3.8 Parallel scaling

The template for post-processing results from multiple independent SPUX runs using a different cumulative amount of processor cores and generating the strong scaling plot is available in examples/randomwalk/plot_scaling.py.

1.3.3.9 Profiling (parallel)

Currently no special scripts for parallel profiling are packaged with SPUX.

1.4 Customization

This section discusses the peculiarities of coupling your model with the framework (the most common use case of SPUX), including some guidelines on writing a custom posterior sampler or a custom likelihood module and using it within SPUX. You are welcome to browse through the results of the models already coupled to spux in the *Gallery*.

1.4.1 Adding a model

In the most common use scenario of SPUX, you will want to implement your own Python model class, which will wrap your existing application to the SPUX framework. To avoid any confusion between these two concepts, we try to refer to your existing application (in any pogramming language) as the "application", and to the Python class coupling your application to the SPUX framework as the "model". While reading these instructions, make sure to take a look at the base Model class at spux/models/model.py. There, extensive comments describe the requirements and many additional helper variables available within all model methods from the base model class.

You will need to create a new file with your model class derived from this base `Model` class. To have an idea what is required, take a look at the code for the Randomwalk model used in *Tutorial*.

1.4.1.1 Model test script

To test any modifications or new additions to the model class, we recommend to use a dataset synthesis script for debugging purposes. We recommend to adapt a respective `script_synthesize.py` script from any of these examples:

- examples/strightwalk/script_synthesize.py for a deterministic model,
- examples/randomwalk/script_synthesize.py for a stochastic model,
- examples/randomwalk-replicates/script_synthesize.py as above, but with replicate datasets.

As explained in *Tutorial*, these scripts only need the exact parameters to be specified. No need for exact model predictions - these will be generated. At this point, we also suggest to start with no error model, that is `error = None`. The error model will become relevant only once the model is fully implemented, and you will switch from dataset synthesis using `script_synthesize.py` to running complete Bayesian inference using SPUX as described in *Tutorial*. Note, that even if the synthetically generated dataset(s) (that is, by specifying the error model) are usually scientifically irrelevant (since you want to perform the Bayesian inference using the real dataset(s)), the inference using such dataset(s) still provides an invaluable resource for making sure the correctness of your implementation.

If your model is (or is intended to become) stochastic and you will be using the `PF` likelihood, we do recommend to start with resampling disabled by setting `noresample=1` in its constructor. This will skip the resampling procedures that require properly functioning `save/load/state` methods in your model, allowing you to fully develop your model's `init` and `run` methods first. Make sure to remove `noresample=1` once you move to the implementation and testing of the `save/load/state` methods.

An alternative option is to start with a deterministic version of your model first (provided such version is possible), and to use the `Direct` likelihood, which does not perform any resampling and does not use `save/load/state` methods of your model.

1.4.1.2 Model execution control

In your custom model class (inherited from the base `Model` class), you will need to modify two mandatory methods:

- `init (self, inputset, parameters)` - initialize model for the specified inputset and parameters
- `run (self, time)` - run model until the specified time and return the prediction

In the method declarations above, the arguments have the following meanings:

- `inputset` - an optional arbitrary object specified in the `likelihood` (default is `None`)
- `parameters` - a `pandas.Dataframe` object with model parameters (as in the `prior` of the `sampler`)
- `time` - an entry from the `index` of the `pandas.Dataframe` object `dataset` specified in the `likelihood`

Within these two main methods, you have two alternative options to execute your actual application:

- Basic (easy but less efficient) - run application execution command with arguments in a shell,
- Advanced (an efficient SPUX'onic way) - call application methods from Python using "drivers".

The "basic" method is very similar to the way you are most probably already executing your application, and hence is recommended as the starting option for the first coupling of your application to the SPUX framework. In particular, the application execution command (name it "mymodel") with some additional prescribed command line arguments (name them "arg1" and "arg2"), is often called by

```
$ mymodel arg1 arg2
```

To achieve exactly the same behavior from within any method of your SPUX model class (`init (...)`, `run (...)`, etc.), including the isolation of the application to the required sandbox directory if the sandboxing is enabled (see below), you can simply use the built-in convenience method `shell`:

```
code = self.shell (r'mymodel arg1 arg2')
```

Note the `r` in front of the string to make sure all speciad characters are specified properly. Internally it uses Python's `subprocess` module, and returns the exit code of your application.

The "advanced" method allows to control the execution stages of your application directly from within the Python model methods, avoiding the unnecessary overhead of application initialization and finalization inbetween consecutive calls of the model's `run (...)` method. The computational efficiency gains are particularly large for a long time series datasets, where `run (...)` needs to be called multiple times, and for models with small stochastic volatility (including deterministic models), where model states change infrequently (or never) inbetween consecutive `run (...)` calls.

Taking into account the additional requirements for respective model drivers (see *Installation*), you can also start directly from the advanced model execution control template corresponding to the required programming language:

- Python: spux/models/straightwalk.py spux/models/randomwalk.py, examples/hydro/hydro.py,
- Fortran: examples/superflex/superflex.py,
- Java: spux/models/ibm.py.

1.4.1.3 Model scope variables

Any model instance has the following internal variables (some different for each instance) available in all methods:

- `self.sandbox ()` - a path to an isolated sandbox directory (if `self.sandboxing == 1`),
- `self.verbosity` - a integer indicating verbosity level for `print ()` intensity management,
- `self.seed ()` - a list containing all hierarchical seeds,
- `self.seed.cumulative ()` - a (large) integer seed obtained by combining all hierarchical seeds,
- `self.rng` - a `numpy.random.RandomState` instance for `random_state` in `scipy.stats` distributions.

The detailed usage of these methods is described in the following sections.

1.4.1.4 Model sandboxing

Sandboxing is enabled by default and a default sandbox is created under `sandbox`. From within any method of the model, the sandbox path can be retrieved by executing `self.sandbox ()`. If certain common files need to be present in every model sandbox, consider creating and populating a template sandbox directory, for instance named

`input`, and specifying a custom sandbox by `sandbox = Sandbox (template = 'input')` in `sampler.setup (...)`. The contents of the template sandbox are always automatically copied (using efficient local caching) to the actual isolated sandbox. and are accessible under sandbox path retrieved using the same instruction as before, i.e. `self.sandbox ()`. During the model development and debugging, we do recommend to use `trace = 1` in `sampler.setup (...)` to be able to inspect each sandbox.

1.4.1.5 Model stochasticity

Note, that `self.seed ()`, `self.seed.cumulative ()` and `self.rng` change for EACH call of `self.run ()`. Make sure that your underlying model is properly configured to implement such frequent updates in the seeding of the random number generator.

1.4.1.6 Initial model state

For some models that do not have a clear starting state, there are basically only two alternative choices regarding the implementation of the `init (...)` method:

- perform computationally expensive "warmup" simulations to obtain (hopefully) valid model states,
- given a prior initial states distribution, infer the posterior initial states distribution.

For the latter choice, the hydrological example is in particularly interesting, since the initial model state is stochastic (to be inferred using Particle Filter). For an example usage of such setup, please refer to the hydrological example at: examples/hydro. In short, a probabilistic prior distribution is provided for the initial model state in examples/hydro/initial.py, which is then filtered by the `PF` likelihood to infer the posterior distribution of initial model states that are consistent to a respective dataset.

1.4.1.7 Auxiliary predictions

The `run (self, time)` method of the model returns annotated model predictions. For the sake of simplicity and to keep the amount of data manageable, only a list or an array of scalars is allowed to be included for such annotation. The full state of some complex models, for instance, in computational fluid dynamics, consists of large multi-dimensional arrays instead of just a couple of scalar values. The suggested strategy is to cherry-pick a handful of the most important scalar values (at the most important array locations) and use them for annotation. This will be sufficient for some simple plots of posterior predictions. However, the error model might still require the full multi-dimensional array for the evaluation of the observation likelihood given some multi-dimensional dataset. To accommodate this, assign any extracted large arbitrary Python objects to the `auxiliary` argument in the `annotate (...)` call. By doing so, the `predictions` in the error model's `distribution (...)` method will instead be a dictionary containing `predictions ['scalars']` as a `pandas.DataFrame` formed from the provided scalars, and `predictions ['auxiliary']` as a arbitrary Python object assigned by the model. This auxiliary object will be accessible only in the error model, and will be discarded immidiately afterwards.

1.4.1.8 Inputsets for models

If `Replicates` likelihood is used to incorporate different observations provided by multiple (replicate) datasets, some models might also require different inputset configurations for each such dataset. For instance, each dataset might require a specific starting time and value of the model. These inputset configurations are not allowed to be set in the model constructor, since this would result in identical inputs across all replicates (which is fine only if there are no different dataset replicates). Instead, the `inputsets` argument provides complementary information for each replicate by passing a respective `inputset` to the model's `init (...)` method (see description above). For an example usage of this setup, please refer to the hydrological example at: examples/hydro.

1.4.1.9 Model state serialization

The `PF` likelihood estimator for stochastic models requires your model to have a capability of being cloned, which in SPUX is based on the concept of the model "state" serialization to a binary stream (array). If you model is written in pure Python or R and you are NOT using the sandbox for any files relevant to your model state, then the required model state serialization functionality from the model's base class is already sufficient.

However, if you save some part of your model state in the sandbox with snapshot-dependent filenames, or if your model is not written in pure Python or R, you will need to specify custom methods for model serialization into its binary representation (state) and a corresponding de-serialization:

- `save (self)` - save and return a `bytearray` representing the current state of the model,
- `load (self, state)` - load model using the `bytearray` representing its required state.

The corresponding helper methods `save (obj)` and `load (state)` are available in the `spux.utils.serialize` module, and are suggested to be used to serialize and de-serialize any arbitrary Python object.

If any files relevant to the model state are saved in the sandbox, the full state of the `save` method must also include the sandbox state, which is obtained by calling the `self.sandbox.save (...)` method of the sandbox. This functionality is already implemented in the base `Model` class (de-)serialization methods, as long as the list of relevant files (otherwise all sandbox files will be included) is specified in the `statefiles` argument of the `Sandbox` constructor. Note, that the files specified in the `statefiles` list do not necessarily need to exist in the initial template sandbox directory, since they might be dynamically generated during the `init (...)` and `run (...)` methods of the model.

If the filenames of the sandbox state files depend on the snapshot, then they cannot be statically specified in the sandbox constructor, and a custom model (de-)serialization methods `save` and `load` need to be implemented. In such case, the list of relevant state files (otherwise all sandbox files will be included) needs to be specified in the `files` argument of the `self.sandbox.save (...)` method. The obtained sandbox state can then be combined with any additional required model instance fields (for instance, `self.time`) in a dictionary and then passed to the serializer:

```
state = {}
files = ['relevant_file_1', 'relevant_file_2']
state ['sandbox'] = self.sandbox.save (files)
state ['model'] = self.time
state = serialize.save (state)
```

The corresponding model `load` method must then extract the relevant model and sandbox states from the full serialized model state and write back the correponding files into the new sandbox:

```
state = serialize.load (state)
self.sandbox.load (state ['sandbox'])
self.time = state ['model']
```

If your model is not written in Python or R, then for some of the other most common programming languages, SPUX contains built-in driver modules in spux/drivers, which can be used to implement the above model state saving and loading routines quickly and efficiently. We recommend to look at the provided example codes in examples/.

1.4.1.10 Serialization test script

To test your custom implementation of the model state (de-)serialization using `save ()` and `load ()` routines, we recommend to use a clone testing script for debugging purposes. We recommend to adapt the example `script_clone.py` script from examples/randomwalk/script_clone.py. The script runs the specified model up to the specified clone time and makes a clone of the original model by saving its state. Then, a second model is created by loading the saved state of the original model and both models are run using the same RNG and seed up to the

specified compare time. If the `save ()` and `load ()` methods work as expected, the predictions of both models must be identical.

1.4.2 SPUX executors

Any set of independent tasks within any of the SPUX components can be executed in parallel using built-in SPUX executors. As described in the tutorial, the default exectutor is a `Serial` executor.

Currently, the parallel executors of each type ("pool" and "ensemble") are implemented in SPUX using MPI. Different types of executors support different functionality and are usually meant to be used in different SPUX components:

- "pool" - dynamically executes a set of independent tasks; changes in task "states" are discarded,
- "ensemble" - statically executes a set of independent tasks in an ensemble (keeping task "states").

The "pool" type executor can be used by calling its `map (...)` method and passing one of the three sets of arguments:

- a callable object (for instance, a function) and a list of arguments for the evaluations,
- a list of callable objects (with an optional list of common fixed arguments),
- a list of callable objects and a corresponding list of arguments for the evaluations.

The "ensemble" type executor does not accept a list of tasks directly, but requires and instance of an `Ensemble` class. Currently the only implemented ensemble class is for an ensemble of SPUX models (to be used in the `PF` likelihood), available at spux/likelihoods/ensemble.py.

Given an `ensemble` instance as above, the "ensemble" type executor can be used by issuing a sequence of executor method calls for control of the ensemble initialization, iterative execution of multiple stages for all tasks, and (optional) resampling:

- `connect (ensemble, indices)` - initialize ensemble with tasks enumerated by the specified indices,
- `call (method, args)` - call a specified (as a string) `method` of each task and return the results,
- `resample (indices)` - resample tasks according to the specified indices (clone and/or delete),
- `disconnect ()` - finalize ensemble and discard any changes in the task "states".

Inbetween the `connect` and `disconnect`, the ensemble executor methods `call` and `resample` can be called multiple times, each time advancing all tasks to the next execution stage and performing any needed resampling of tasks. During the resampling, tasks are allowed to be cloned (duplicate indices) and deleted (missing indices). In the resampling call, load re-balancing across the resulting resampled ensemble is performed.

1.4.3 Parallel models

Most probably you have already noticed, that in the tutorial, no parallel executor is attached to the model object. This is because our implementation Randomwalk model does not support parallelization. However, a custom user model might be very computationally expensive and need further parallelization.

1.4.3.1 Parallelize serial model

Provided that the content of the pure Python model `init (...)` and/or `run (...)` methods can be split into a list of independent computationally intensive tasks, one could attach a `spux.executors.mpi4py.pool` executor to the model. The instruction how to make use of the `Mpi4pyPool` executor are provided in the preceding sections. A more complicated option to potentially achieve a better performance is to attach a `spux.executors.mpi4py.ensemble` executor to the model by splitting the model into independent sub-models and treating all of them as an

ensemble of sub-models. The instruction how to make use of the `Mpi4pyEnsemple` executor are provided in the preceding sections as well.

For more information, take a look at the corresponding documentation files in *Reference*.

1.4.3.2 Parallel model executor

In some cases, a custom user model might be either already parallelized, or the model might be written in another programming language rather than Python. SPUX framework does support such models too.

The easiest, but also the least efficient way to run parallel models is to rely on model evolution through file system (saving model states as files in sandboxes), and simply calling `os.system ('myparallelmodel <some args ...>')` from within the `self.init (...)` and `self.run (...)`. This requires sufficient allocated computation resources such that all processes from both the SPUX framework and the parallel model run on separate cores.

An alternative way is a bit more complicated and only minimally itrusive, and can be used provided that MPI is used for model parallelization. In particular, one can attach the built-in parallel MPI model executor from spux/executors/mpi4py/mpimodel.py:

```
from spux.executors.mpi4py.mpimodel import Mpi4pyModel
```

With the `Mpi4pyModel (workers=<workers>)` executor attached to the model, in the model `init (...)` and `run (...)` methods, the call `self.executor.connect (command)` returns an MPI inter-communicator connected to the parallel workers, each executing the provided shell command `command` in parallel, analogous to the manual launch of an MPI:

```
$ mpiexec -n <workers> command
```

You can use this manager-side MPI inter-communicator to workers for simulation control, parameters specification, predictions retrieval, and saving/loading of the model state, completely circumventing the need for any unnecessary filesytem related operations.

1.4.3.3 Model communicators

The connection procedure from the parallel workers (model) to the manager depends on the selected connector (see explanation in the *Tutorial*) as described below. Independently of the selected connector, in each parallel worker you will have an access to a corresponding inter-communicator with the manager. You can use this worker-side MPI inter-communicator to manager for simulation control, parameters acquisition, predictions reporting, and saving/loading of the model state.

For `spawn` connector, on the workers (model) side, you have access to an inter-communicator with the manager returned by calling `MPI_Comm_get_parent ()`. Within the model, the standard `MPI_COMM_WORLD` MPI intra-communicator is available, as in a normal MPI run. Currently `spawn` is the only fully supported connector for parallel models using MPI.

For more information, take a look at the corresponding documentation files in *Reference*.

1.4.4 Adding a distribution

The easiest way to specify a multivariate distribution is to use a tensor spux/distributions/tensor.py of selected univariate distributions from the `scipy.stats` module; see an example in *Tutorial*.

An example of how to have a joint parameters distribution with correlations, possibly by selecting a multivariate distribution from the `scipy.stats` module, can be found in spux/distributions/multivariate.py.

1.4.5 Adding an error

One custom scenario would be when the observations (both the predictions or the dataset) need to be transformed before the density of the distribution can be evaluated. To support this, one can provide a custom `transform (self, observations, parameters)` method in the error class which performs the required transformations using the (optional) specified parameters and returns the result.

For an example, look at the `error.py` in examples/hydro/error.py.

1.4.6 Setting variable types

Sometimes either some of the model parameters or some of the model predictions are represented by integers instead of floating point numbers. This is often the case if the quantity of interest represents a count of some occurrences, or a discrete categorical class.

By default, all parameters and predictions are assumed to be of type `float`. However, optional respective `parameters.types` and/or `predictions.types` files can be provided in the implementations of the `prior.py/error.py/<model>.py` scripts and in the constructor of the plotting class `MatPlotLib`.

The format of the files requires to specify two columns, with entries listing `<variable name>` and `<variable type>`, for instance:

```
prey int
prey_kFood double
```

The `parameters.types` file is used in the examples/IBM_2species example to round integer valued parameters in `prior.py`, `error.py` and `ibm.py`, since inconsistencies can arise depending on the type of sampler.

The `predictions.types` file is used in the plotting routines in the constructor of the `MatPlotLib` class. This is useful, for example, to plot the error distributions for integer-valued observations (model predictions or collected dataset).

Please refer to examples/IBM_2species for a specific usage example of these files.

1.4.7 Adding a sampler

In order to add a custom sampler (in addition to existing samplers, e.g. MCMC and EMCEE), you need to derive a new sampler class the base `Sampler` class in `spux.samplers.sampler`. For the source code of the sampler base class, please refer to spux/samplers/sampler.py.

In particular, the new sampler class must have the following methods:

- `__init__ (self, ...)` - constructor to set sampler properties (can be skipped),
- `init (self, ...)` - sampler initialization routine (can be empty),
- `pickup (self)` - return sampler pickup information (e.g. a dictionary with needed entries),
- `draw (self, sandbox, seed)` - return samples as `pandas.Dataframe` and the associated `info`.

In most use cases, sampler will need to have an assigned likelihood (which is assigned in the base class method `sampler.assign (...)`). The `init (...)` method can be used to initialize as many likelihoods as the number of needed concurrent chains (assuming `self.chains` is set in the constructor):

```
self.likelihoods = [copy (self.likelihood) for index in range (self.chains)]
```

The following code can be used to properly set up likelihoods in `draw (...)`:

```
for chain, likelihood in enumerate (self.likelihoods):
    label = 'C%05d' % chain
    chain_sandbox = sandbox.spawn (label) if self.sandboxing else None
    chain_seed = seed.spawn (chain, name=label)
    likelihood.setup (chain_sandbox, self.verbosity - 2, chain_seed, self.informative,
↪ self.trace, self._feedback)
```

Finally, the list of all likelihoods can be passed to the executor for evaluation, together with the list of corresponding parameters for each of them:

```
results, timings = self.executor.map (likelihoods, parameters)
```

Note, that to allow maximum flexibility, the `self.executor` is avaible within both `init (...)` and `draw (...)` methods.

For an example implementation, please refer to the source code of the EMCEE sampler at spux/samplers/emcee.py.

Work in progress.

1.4.8 Adding a likelihood

Work in progress.

1.5 Reference

1.5.1 spux package

SPUX: Scalable Package for Uncertainty Quantification in X

1.5.1.1 Subpackages

1.5.1.1.1 spux.distributions package

1.5.1.1.1.1 Submodules

1.5.1.1.1.2 spux.distributions.distribution module

class spux.distributions.distribution.**Distribution**
Bases: object

draw(*rng*)
Draw a random vector using the provided random state ‘rng’.

intervals(*alpha=0.99*)
Return intervals for the specified centered probability mass.

Intervals are returned for each parameter.

logmpdf(*label*, *parameter*)
Return marginal log-PDF for the specified parameter.

logpdf (*parameters*)
Base method to be overloaded to evaluate the logarithm of the (joint) prob. distr. function of parameters.

'parameters' are assumed to be of a pandas.DataFrame type

mpdf (*label*, *parameter*)
Return marginal PDF for the specified parameter.

pdf (*parameters*)
Base method to be overloaded to evaluate the (joint) prob. distr. function of parameters.

'parameters' are assumed to be of a pandas.DataFrame type

1.5.1.1.1.3 spux.distributions.multivariate module

class spux.distributions.multivariate.**Multivariate** (*distribution*, *labels*, *marginals=None*)
Bases: *spux.distributions.distribution.Distribution*

draw (*rng*)
Draw a random vector using the provided random state 'rng'.

intervals (*alpha=0.99*)
Return intervals for the specified centered probability mass.

logmpdf (*label*, *parameter*)
Return marginal log-PDF for the specified parameter.

logpdf (*parameters*)
Evaluate the logarithm of the (joint) prob. distr. function of (covariate) parameters.

'parameters' are assumed to be of a pandas.DataFrame type

mpdf (*label*, *parameter*)
Return marginal PDF for the specified parameter.

pdf (*parameters*)
Evaluate the (joint) prob. distr. function of (covariate) parameters.

'parameters' are assumed to be of a pandas.DataFrame type

1.5.1.1.1.4 spux.distributions.tensor module

class spux.distributions.tensor.**Tensor** (*distributions*, *types_of_keys=None*)
Bases: *spux.distributions.distribution.Distribution*

draw (*rng*)
Draw a random vector using the provided random state 'rng'.

intervals (*alpha=0.99*)
Return intervals for the specified centered probability mass.

logmpdf (*label*, *parameter*)
Return marginal log-PDF for the specified parameter.

logpdf (*parameters*)
Evaluate the logarithm of the (joint) prob. distr. function of the tensorized, i.e. assuming independence, random variables 'parameters'.

'parameters' are assumed to be of a pandas.Series type

mpdf(*label*, *parameter*)
Return marginal PDF for the specified parameter.

pdf(*parameters*)
Evaluate the (joint) prob. distr. function of the tensorized, i.e. assuming independence, random variables 'parameters'.

'parameters' are assumed to be of a pandas.Series type

1.5.1.1.2 spux.drivers package

1.5.1.1.2.1 Submodules

1.5.1.1.2.2 spux.drivers.java module

class spux.drivers.java.**Java**(*jvmpath=None*, *classpath=None*, *jvmargs=''*, *cwrank=-1*)
Bases: object

Convenience wrapper for Python Java bindings.

WARNING: due to underlying Python Java bindings library limitations, you cannot run a single Python process that uses this driver at least twice but with different Java classpaths. The subsequent classpaths won't correctly load.

get_class(*name*)
Return the java class 'name' from loaded java jar

jpype = None

classmethod load(*state*)
Take 'state' (of the model) from save(), i.e. as numpy uint8 array, and return 'buff' in byte to be passed to the java user code

classmethod save(*buff*)
Return 'state' (of the model) as numpy uint8 array when 'buff' (the state of the model from the user code) is in binary format

started_in = {}

classmethod state(*size*)
Return 'state' as jByteArray of given 'size'

1.5.1.1.3 spux.executors package

1.5.1.1.3.1 Subpackages

1.5.1.1.3.2 spux.executors.balancers package

1.5.1.1.3.3 Submodules

1.5.1.1.3.4 spux.executors.balancers.adaptive module

class spux.executors.balancers.adaptive.**Adaptive**
Bases: *spux.executors.balancers.balancer.Balancer*

Derived class to establish particle routings.

ensembles (*indices*, *workers*)
Initially distribute particles to ensembles according to how many 'workers' are available.

routings (*ensembles*, *indices*)
Compute routings of particles from current particle 'ensembles' and specified 'indices'

1.5.1.1.3.5 spux.executors.balancers.balancer module

class spux.executors.balancers.balancer.**Balancer**
Bases: object

Base class for balancing network traffic due to killing and cloning (resampling) of particles.

sources (*routings*)
Compute sources for the particles to be resampled according to the specified routings.

traffic (*routings*)
Compute network traffic (moves, copies, etc.) from routing of particles.

verbosity = 0

1.5.1.1.3.6 spux.executors.mpi4py package

1.5.1.1.3.7 Subpackages

1.5.1.1.3.8 spux.executors.mpi4py.connectors package

1.5.1.1.3.9 Submodules

1.5.1.1.3.10 spux.executors.mpi4py.connectors.legacy module

class spux.executors.mpi4py.connectors.legacy.**Legacy**(*verbosity=0*)
Bases: *spux.executors.mpi4py.connectors.split.Split*

Class to establish workers MPI processes when dealing with legacy MPI implementations.

static accept (*remote_leader*, *verbosity*)
Establish connection on manager side.

bootup (*contract*, *task*, *resource*, *root*, *verbosity*)
Inter-connect manager with the number of requested workers by returning leader rank.

static connect (*remote_leader*, *peers*)
Establish connection on worker side.

static disconnect (*workers*, *verbosity*)
Interrupt connection.

static shutdown (*port*, *verbosity*)
Finalize connector.

1.5.1.1.3.11 spux.executors.mpi4py.connectors.spawn module

class spux.executors.mpi4py.connectors.spawn.**Spawn**(*verbosity=0*)
Bases: object

Class to establish workers MPI processes by spawning of new processes.

static accept(*port*, *verbosity*)
Establish connection.

barrier()

bootup(*contract*, *task*, *resource*, *root=0*, *verbosity=0*)
Return means of inter-communication along a possible hierarchy of processes.

static connect(*port*, *peers*)
Establish connection on worker side.

static disconnect(*workers*, *verbosity*)
Interrupt connection.

init(*resources*)

static shutdown(*port*, *verbosity*)
Finalize connector.

1.5.1.1.3.12 spux.executors.mpi4py.connectors.split module

class spux.executors.mpi4py.connectors.split.**Split**(*verbosity=0*)
Bases: object

Class to establish workers MPI processes by using server/client mode through ports.

static accept(*port*, *verbosity*)
Establish connection on manager side.

barrier()
Split workers from manager, split workers into pools, wait for tasks.

bootup(*contract*, *task*, *resource*, *root*, *verbosity*)
Inter-connect manager with the number of requested workers by returning a port.

static connect(*port*, *peers*)
Establish connection on worker side.

static disconnect(*workers*, *verbosity*)
Interrupt connection.

init(*resources*)
Initialization for manager: bcast resources and split away from pool slots.

static shutdown(*port*, *verbosity*)
Finalize connector.

split()
Split workers recursively into several pools of workers.

spux.executors.mpi4py.connectors.split.**get_ranks**(*resource*, *root=None*, *manager=0*)
Compute worker ranks for the current level of resources

1.5.1.1.3.13 spux.executors.mpi4py.connectors.utils module

spux.executors.mpi4py.connectors.utils.**select**(*name='auto'*, *verbosity=0*)
Automatically select the connector, or specify it manually by its name.

spux.executors.mpi4py.connectors.utils.**universe_address**()
Return rank in MPI COMM_WORLD

1.5.1.1.3.14 spux.executors.mpi4py.connectors.worker module

spux.executors.mpi4py.connectors.worker.**universe_address**()
Return rank in MPI COMM_WORLD

1.5.1.1.3.15 Submodules

1.5.1.1.3.16 spux.executors.mpi4py.ensemble module

1.5.1.1.3.17 spux.executors.mpi4py.ensemble_contract module

1.5.1.1.3.18 spux.executors.mpi4py.ensemble_resample module

1.5.1.1.3.19 spux.executors.mpi4py.model module

1.5.1.1.3.20 spux.executors.mpi4py.pool module

1.5.1.1.3.21 spux.executors.mpi4py.pool_contract module

1.5.1.1.3.22 Submodules

1.5.1.1.3.23 spux.executors.executor module

1.5.1.1.3.24 spux.executors.serial module

1.5.1.1.4 spux.io package

1.5.1.1.4.1 Submodules

1.5.1.1.4.2 spux.io.checkpointer module

class spux.io.checkpointer.**Checkpointer**(*period*)
Bases: object

check(*force=0*)
Return timestamp.

init(*verbosity*)

1.5.1.1.4.3 spux.io.dumper module

1.5.1.1.4.4 spux.io.formatter module

spux.io.formatter.**compactify**(*resources*)
Improve format of 'resources' dictionary.

spux.io.formatter.**intf**(*number*, *table=1*, *empty=0*, *bar=0*)
Integer format with multipliers K, M, etc.

spux.io.formatter.**plain**(*name*)
Filter to remove special characters from strings to be used as filenames.

spux.io.formatter.**timestamp**(*time*, *precise=False*, *expand=False*)
Formats time in seconds into a string of (years and days - only if needed), hours and minutes.

1.5.1.1.4.5 spux.io.loader module

1.5.1.1.4.6 spux.io.parameters module

spux.io.parameters.**load**(*filename*, *names=None*, *dtypes=None*)

spux.io.parameters.**save**(*data*, *filename*, *delimiter='\t'*)

1.5.1.1.4.7 spux.io.report module

spux.io.report.**report**(*instance*, *method*, *extras={}*)
Report name, method, root, sandbox, and any specified extras provided verbosity is enabled.

1.5.1.1.5 spux.likelihoods package

1.5.1.1.5.1 Submodules

1.5.1.1.5.2 spux.likelihoods.direct module

1.5.1.1.5.3 spux.likelihoods.ensemble module

1.5.1.1.5.4 spux.likelihoods.likelihood module

1.5.1.1.5.5 spux.likelihoods.pf module

1.5.1.1.5.6 spux.likelihoods.replicates module

1.5.1.1.6 spux.models package

1.5.1.1.6.1 Submodules

1.5.1.1.6.2 spux.models.ibm module

1.5.1.1.6.3 spux.models.model module

1.5.1.1.6.4 spux.models.randomwalk module

1.5.1.1.6.5 spux.models.randomwalk_numba module

1.5.1.1.6.6 spux.models.straightwalk module

1.5.1.1.7 spux.plot package

1.5.1.1.7.1 Submodules

1.5.1.1.7.2 spux.plot.mpl module

1.5.1.1.7.3 spux.plot.mpl_palette_pf module

1.5.1.1.7.4 spux.plot.mpl_utils module

spux.plot.mpl_utils.**brighten**(*color*, *factor=0.7*)
Create a new solid color which is slighly brighter.

spux.plot.mpl_utils.**figname**(*save*, *figpath='fig'*, *suffix=''*, *extension='pdf'*)
Generate figure name using the format 'figpath/pwd_suffix.extension'.

1.5.1.1.7.5 spux.plot.profile module

spux.plot.profile.**callgraph**(*pstatsfile*, *root=None*, *threshold=10*, *outputdir='fig'*)
Generate callgraph from profile stats using gprof2dot.

spux.plot.profile.**figname**(*figpath='fig'*, *suffix=''*, *extension='pdf'*)
Generate figure name using the format 'figpath/pwd_suffix.extension'.

spux.plot.profile.**report**(*pstatsfile*, *outputdir='fig'*)
Export profile information into a text file.

1.5.1.1.8 spux.processes package

1.5.1.1.8.1 Submodules

1.5.1.1.8.2 spux.processes.ornsteinuhlenbeck module

class spux.processes.ornsteinuhlenbeck.**OrnsteinUhlenbeck**(*tau*)
Bases: object

Class for Ornstein-Uhlenbeck process.

evaluate(*t*, *rng*)
Evaluate Ornstein-Uhlenbeck process at time 't'.

init(*t*, *xi*)

1.5.1.1.8.3 spux.processes.precipitation module

class spux.processes.precipitation.**Precipitation**(*g*, *a*, *b*, *c*, *xi_1*, *xi_2*)
Bases: object

Class for Precipitation process.

evaluate(*xi*)

inverse(*x*)

1.5.1.1.8.4 spux.processes.wastewater module

class spux.processes.wastewater.**Wastewater**(*zeta*, *chi*)
Bases: object

Class for waste water process.

evaluate(*t*)

1.5.1.1.9 spux.report package

1.5.1.1.9.1 Submodules

1.5.1.1.9.2 spux.report.generate module

1.5.1.1.10 spux.samplers package

1.5.1.1.10.1 Submodules

1.5.1.1.10.2 spux.samplers.emcee module

1.5.1.1.10.3 spux.samplers.forecast module

1.5.1.1.10.4 spux.samplers.mcmc module

1.5.1.1.10.5 spux.samplers.sampler module

1.5.1.1.11 spux.utils package

1.5.1.1.11.1 Submodules

1.5.1.1.11.2 spux.utils.annotate module

spux.utils.annotate.**annotate**(*data*, *labels*, *time*, *auxiliary=None*)
Annotate data array with the given labels (with an option for auxiliary information).

Optional auxiliary object of any type can be provided and will be passed to the error model, but will not be stored in the corresponding 'info' as the model prediction, and hence will be node-local. In the error model, the 'prediction' will then be a dictionary of the form: {'scalars' : predictions, 'auxiliary' : auxiliary}. This is useful for large non-scalar model outputs, such as vectors or multi-dimensional arrays (e.g. xarray's). Any additional (i.e outside the error model) access of such auxiliary information is not supported. As such data is often very large, the recommended option is to keep the trace of all sandboxes and perform additional a posteriori post-processing.

1.5.1.1.11.3 spux.utils.assign module

1.5.1.1.11.4 spux.utils.debug_inparallel module

1.5.1.1.11.5 spux.utils.environment module

1.5.1.1.11.6 spux.utils.evaluations module

spux.utils.evaluations.**construct**(*instance*, *tasks*)
Recursively construct model evaluations report.

1.5.1.1.11.7 spux.utils.progress module

class spux.utils.progress.**Progress**(*prefix*, *steps*, *length=20*, *caption='Progress: '*)
Bases: object

Class for update'able progress bar for the command line.

finalize()

increment(*diff=1*)

init()

message(*message*)

reset()

update(*step*)

1.5.1.1.11.8 spux.utils.sandbox module

1.5.1.1.11.9 spux.utils.seed module

class spux.utils.seed.**Seed**(*seed=0*, *name='root'*)
Bases: object

Class to generate independent seeds for random number generators.

cumulative()
Get cumulative seed.

spawn(*seed*, *name='noname'*)
Spawn new seed based on current state (append lists).

spux.utils.seed.**inc**(*a*)

spux.utils.seed.**pair**(*a*, *b*)
Construct (pair) two seeds from one [Szudzik].

1.5.1.1.11.10 spux.utils.serialize module

1.5.1.1.11.11 spux.utils.setup module

1.5.1.1.11.12 spux.utils.shell module

spux.utils.shell.**execute**(*command*, *directory=None*, *verbosity=1*, *executable=None*)
Execute an application command (including any arguments) in a command line shell.

1.5.1.1.11.13 spux.utils.synthesize module

1.5.1.1.11.14 spux.utils.testing module

1.5.1.1.11.15 spux.utils.timer module

class spux.utils.timer.**Timer**
Bases: object

current(*format=0*)

pause()

start()

timestamp()

1.5.1.1.11.16 spux.utils.timing module

class spux.utils.timing.**Timing**
Bases: object

start(*name*)

time(*name*)

1.5.1.1.11.17 spux.utils.transforms module

spux.utils.transforms.**flatten**(*dictlist*)
Flatten a list of dictionaries into a dictionary.

spux.utils.transforms.**logmeanstd**(*logm*, *logs*)
Return the needed quantities to construct stats.lognorm with a specified mean (logm) and standard deviation (logs).

According to documentation at: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.lognorm.html

spux.utils.transforms.**numpify**(*dictionary*)
Convert dict with integer keys to numpy array - assume user plays nice.

spux.utils.transforms.**pandify**(*dictionary*)
Convert dict with integer keys and pandas DataFrame rows to pandas DataFrame - assume user plays nice.

spux.utils.transforms.**rounding**(*method*)
A decorator to map method arguments from float to integer by rounding.

1.5.1.1.11.18 spux.utils.traverse module

spux.utils.traverse.**components**(*root, includes=['Model', 'Likelihood', 'Sampler']*)
Auto-magically generate a table for all SPUX components.

spux.utils.traverse.**infos**(*info*)
Auto-magically generate a table for info tructure.

1.6 Gallery

Here we provide a gallery containing selected example results of several applications (a non-exhaustive list) using SPUX framework for Bayesian inference and uncertainty quantification.

1.6.1 Randomwalk

Domain: demo.
Authors: Jonas Šukys (Eawag, Switzerland).
Model: one-dimensional random walk (built-in).
Language: Python.
Cluster: EULER (ETH Zurich, Switzerland) - 129 cores.

A simple one-dimensional randomwalk with uncertain origin, drift, and the observation error.

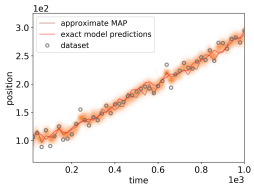


Fig. 24: Posterior distribution of model predictions for the observational dataset. The shaded orange regions indicate the log-density of the posterior model predictions distribution at the respective time points, the brown line indicates the approximate MAP model prediction., the red line represents the exact model prediction values.

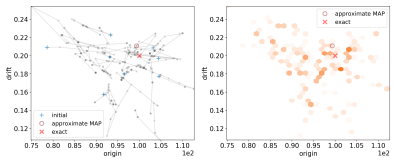


Fig. 25: Joint pairwise marginal posterior distribution of origin and drift, including the corresponding Markov chains from the sampler. Legend: thin semi-transparent gray lines and dots - concurrent chains, orange hexagons - histogram of the joint pairwise marginal posterior parameters samples, blue “+” - initial parameters, brown “o” - approximate MAP parameters, red “x” - the exact parameters.

1.6.2 Linear bucket

Domain: hydrology.
Authors: Andreas Scheidegger (Eawag, Switzerland).
Model: linear bucket model with stochastic forcing.
Language: R, with `rpy2` bindings to Python.

Work in progress.

1.6.3 Stochastic inputs

Domain: hydrology.
Authors: Jonas Šukys (Eawag, Switzerland).
Model: hydrological model with stochastic inputs (built-in).
Language: Python, with `numba` compiled C code for computationally expensive parts.
Cluster: EULER (ETH Zurich, Switzerland) - 1121 cores.

Publication: Del Giudice, D. et al., (2016) "Describing the catchment-averaged precipitation as a stochastic process improves parameter and input estimation; Water Resources Research. John Wiley & Sons, Ltd, 52(4), pp. 3162–3186. doi: 10.1002/2015WR017871.

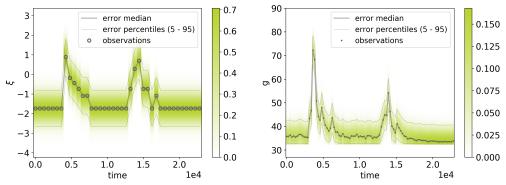


Fig. 26: The first dataset and the associated heteroscedastic error model for the input (precipitation) and the output (discharge) measurements.

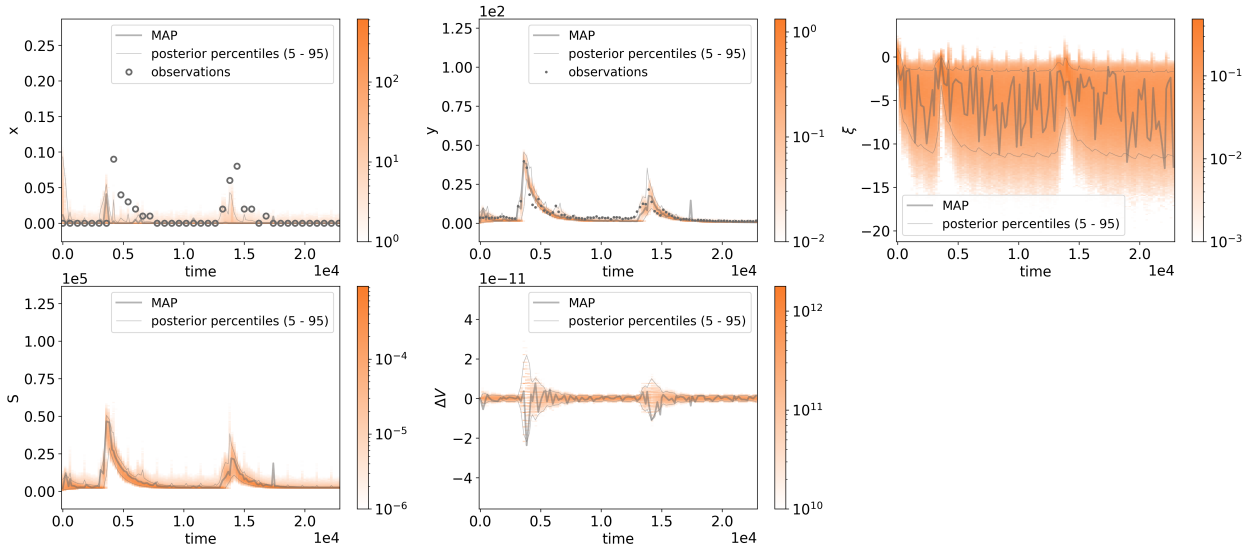


Fig. 27: Plots of the posterior distribution of model predictions for an observational dataset above, including auxiliary posterior distributions for rainfall potential `$\xi$`, reservoir level `S`, and the water volume discrepancy `$\Delta V$`.

Work in progress.

1.6.4 Stochastic parameters

Domain: hydrology.
Authors: Marco Bacci, Jonas Šukys (Eawag, Switzerland).
Model: hydrological model with stochastic time-dependent parameters (Superflex).

Language: Fortran, with `ctypes` bindings of the compiled Fortran model library to Python.

Work in progress.

1.6.5 Prey-Predator

Domain: aquatic ecology.
Authors: Jonas Šukys, Nele Schuwirth, Peter Reichert (Eawag, Switzerland), Mira Kattwinkel (University of Koblenz-Landau, Germany).
Model: prey-predator model using stochastic individual based model with synthetic dataset (IBM-Bugs).
Language: Java, with `JPype` bindings to Python.
Cluster: EULER (ETH Zurich, Switzerland) - up to 1000 cores.

Publication (preprint available at http://arxiv.org/abs/1711.01410):

```
Šukys, J. and Kattwinkel, M.
"SPUX: Scalable Particle Markov Chain Monte Carlo
for uncertainty quantification in stochastic ecological models".
Advances in Parallel Computing – Parallel Computing is Everywhere,
IOS Press, (32), pp. 159–168, 2018.
```

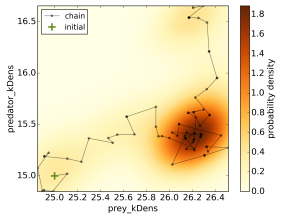


Fig. 28: Marginal posterior distribution of `prey_kDens` and `predator_kDens` parameters, including the corresponding MCMC chain from the sampler. Legend: green "+" - initial parameters.

Work in progress.

1.6.6 River invertebrates

Domain: aquatic ecology.
Authors: Marco Bacci, Nele Schuwirth, Peter Reichert, Jonas Šukys (Eawag, Switzerland) Mira Kattwinkel (U Koblenz-Landau, Germany).
Model: river invertebrates mesocosm modeling using stochastic IBMs (IBM-Bugs).
Model: Java, with `JPype` bindings to Python.
Cluster: EULER (ETH Zurich, Switzerland) - 736 cores.

Work in progress.

1.6.7 DATALAKES

Domain: hydrology and data science.

Authors: Artur Safin, Jonas Šukys (Eawag, Switzerland).
Model: DATALAKES - a scalable UQ framework for predicting lake dynamics (MITgcm).
Language: Fortran, with `ctypes` bindings of the compiled Fortran model library to Python.
Cluster: Daint (Swiss Supercomputing Center (CSCS), Switzerland).

Work in progress.

1.6.8 In-stream herbicides

Domain: aquatic ecology.
Authors: Peter Reichert, Fabrizio Fenizia, Lorenz Ammann, Jonas Šukys (Eawag, Switzerland).
Model: in-stream herbicide concentration dynamics (Superflex).
Language: Fortran, with `ctypes` bindings of the compiled Fortran model library to Python.

Work in progress.

1.6.9 Urban hydrology

Domain: urban hydrology.
Authors: Joao Leitao, Andreas Scheidegger, Jörg Rieckermann, Jonas Šukys.
Model: urban hydrologic model (SWMM).
Language: C, with `Swig` wrapper for Python.

Work in progress.

1.6.10 Solar dynamo

Domain: physics and data science

BISTOM - calibration of the solar dynamo simulations.

Work in progress.

1.7 Contributing

The source code is available at the GitLab repository: https://gitlab.com/siam-sc/spux.
Contributions are welcome, and they are greatly appreciated!
Every little bit helps, and credit will always be given.

1.7.1 The SPUX'onic way

When contributing, please always make an effort to adhere the SPUX'onic coding style and ethics:

- **think (at least) twice about proper variable names:**

  - avoid abbreviations and slang,
  - prioritize descriptive single-word variables to lengthy "sentence"-variables,
- use docstrings for each class and method you implement,
- place technical methods into `spux.utils` or `spux.io`,
- always use the `verbosity` level filter for any (carefully formatted, of course) output to console,
- always clean up (remove debug code and unnecessary comments) before merging to test branch.

1.7.2 Types of contributions

You can contribute in many ways, listed in the following paragraphs.

1.7.2.1 Report bugs

Report bugs at https://gitlab.com/siam-sc/spux/issues

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

1.7.2.2 Fix bugs

Look through the GitLab issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

1.7.2.3 Implement features

Look through the GitLab issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

1.7.2.4 Write documentation

SPUX could always benefit from more documentation, whether as part of the official spux docs, in docstrings, or even on the web in blog posts, articles, and such.

To contribute to the official spux docs, checkout the the spux repository and look through all the files listed in the `MANIFEST.in` file. The auto-generated documentation from all these files is placed under the `docs/` directory, where static files such as plots, tables, etc. are located under `docs/_static` (please respect the current directory structure). When adding your new contributions to the documentation, please follow the current style and make sure that:

- Code snippet uses the code block (instead of a standard paragraph text).
- Code snippet has no remaining programmer comments, notes, or legacy code.
- The line lengths of the code snippet do not protrude beyond the right margin of the paragraph.

1.7.2.5 Submit feedback

The best way to send feedback is to file an issue at https://gitlab.com/siam-sc/spux/issues

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Contributions are very welcome and will make the framework better for you and other users.

1.7.3 Get started!

Ready to contribute? Here's how to set up *spux* for local development.

1. Fork the *spux* repo on GitLab.
2. Clone your fork locally:

```
$ git clone git@gitlab.com:siam-sc/spux.git
$ cd spux/
```

3. (Optional) Install virtualenv and virtualenvwrapper (and source paths, if needed):

```
$ pip install virtualenvwrapper
```

4. (Optional) Install your local copy into a virtualenv:

```
$ mkvirtualenv spux
$ workon spux
```

5. Set up your fork for local development (use *–user* at the end if needed):

```
$ pip install -r requirements_dev.txt
$ python setup.py develop
```

6. Create a branch for local development (name it dev_username for private development branch):

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

   You can run tests within your environment:

```
$ flake8 spux tests examples
$ pytest -m "not mpi" tests
$ mpiexec -n 1 --oversubscribe pytest -m "mpi" tests
```

   Or, alternatively (a slower approach), using tox to include testing with Python and MPI versions:

```
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

7. Commit your changes and push your branch to GitLab:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Make sure all tests pass in the GitLab-CI as well.
9. Submit a merge request (e.g. to the "test" branch) through the GitLab website.
10. Maintainers: review merge request and activate "Merge automatically when pipeline succeeds".

1.7.4 Merge requests

Before you submit a merge request, check that it meets these guidelines:

1. The merge request should include tests.
2. If the merge request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The merge request should work for Python 3.7.

1.7.5 Tips

- To run only non-MPI tests:

```
$ pytest -m "not mpi" tests
```

  or only short module tests, excluding long integration tests:

```
$ pytest -m "not mpi and not integration" tests
```

  or only long integration tests using MPI:

```
$ pytest -m "mpi and integration" tests
```

- To run tests from a single file:

```
$ pytest tests/test_spux.py
```

  or a single test function:

```
$ pytest tests/test_spux.py::test_imports
```

- To add dependency, edit appropriate `*requirements` variable in the `setup.py` file and re-run:

```
$ python setup.py develop
```

  Check if this requirement should be also included in the `requirements_dev.txt` file.

1.7.6 Deploying

A reminder for the maintainers on how to deploy.

- **Make sure all issues on GitLab associated with this release milestone are:**
  - either fixed and closed with changes merged into the `test` branch,
  - or re-assigned to future release milestones.
- **Review documentation and make sure all examples and statements are up to date:**
  - run `make docs_html` in the terminal and check generated html pages carefully,

  - check all source code snippets that use specific line numbers and fix them,
  - check if additional examples, results, or publications should be added for the gallery,
  - check if additional contributions should be added in the credits.
- Verify all filenames listed in `MANIFEST.in`, including all needed package directories.
- Merge the release version of the code to the `release` branch, make sure all tests pass.
- **Make sure all your changes are COMMITTED (!), including:**
  - an entry in `HISTORY.rst`,
  - (optionally) the development status change in `setup.py` (see here for options).
- Make sure you have `texlive-science`, `latexmk`, and `image-magick` installed for PDF documentation.
- Make sure your working branch is `release`.

Then run in the terminal:

```
$ pip install -U -r requirements_rtd.txt
$ make docs
$ make clean
$ bumpversion patch # possible: major / minor / patch; might need --allow-dirty
$ git push
$ git push --tags
```

Afterwards, GitLab-CI will automatically deploy the release to PyPI and ReadTheDocs if tests pass. Then merge the `release` branch into the `master` and `test` branches.

1.8 Parallelization

Here we briefly describe parallelization algorithms, including communication patterns, load balancing strategies, and experimental results for parallelization profiling and scaling.

1.8.1 Communication patterns

1.8.2 Profiling and scaling

1.9 Credits

1.9.1 Development Lead

- Jonas Šukys <jonas.sukys@eawag.ch>
- Marco Bacci <marco.bacci@eawag.ch>

1.9.2 Contributors

- Uwe Schmitt <uwe.schmitt@id.ethz.ch>
- Mikołaj Rybiński <mikolaj.rybinski@id.ethz.ch>
- Andreas Scheidegger <andreas.scheidegger@eawag.ch>

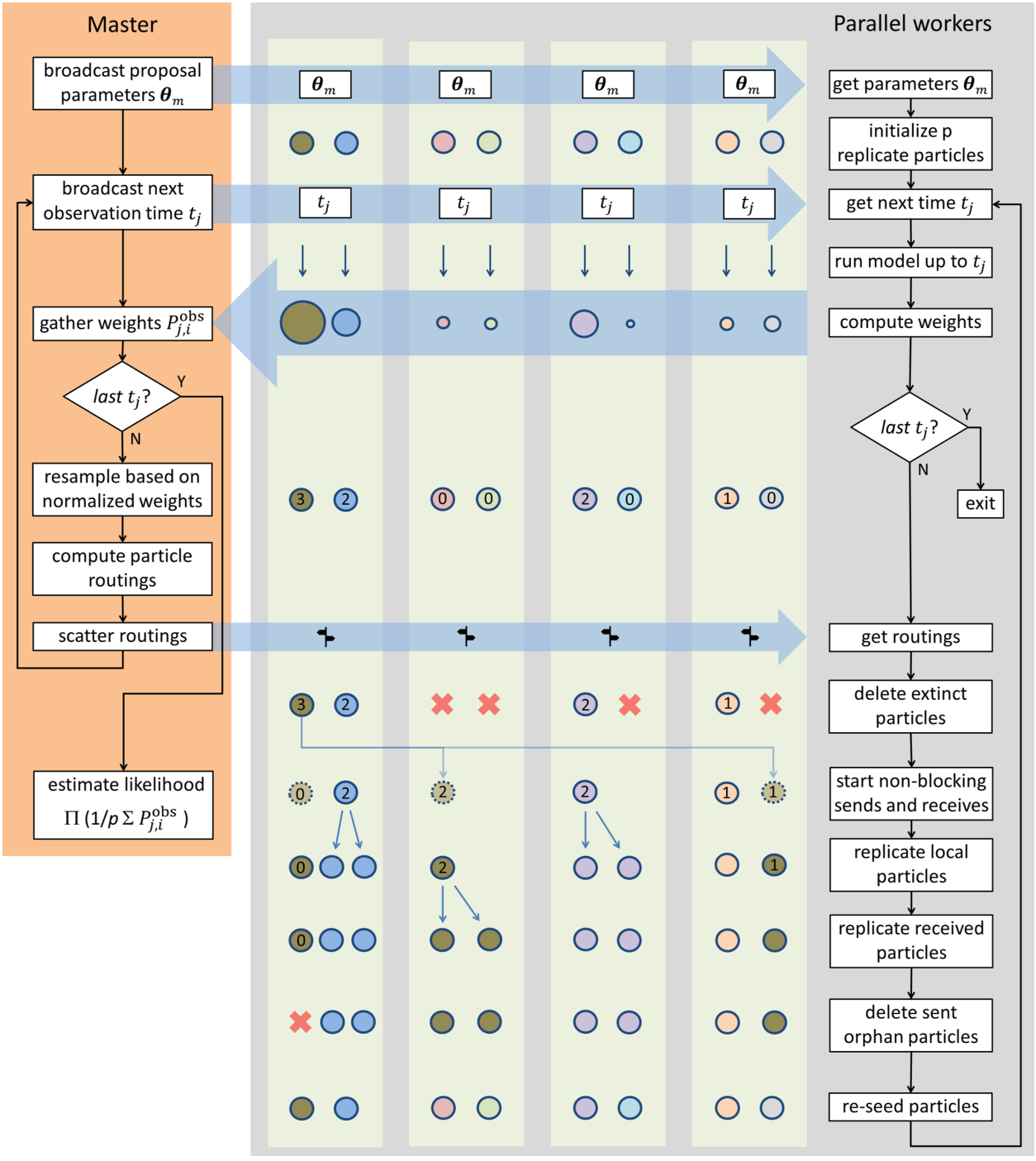


Fig. 29: Adaptive parallel resampling algorithm used in the SPUX framework.

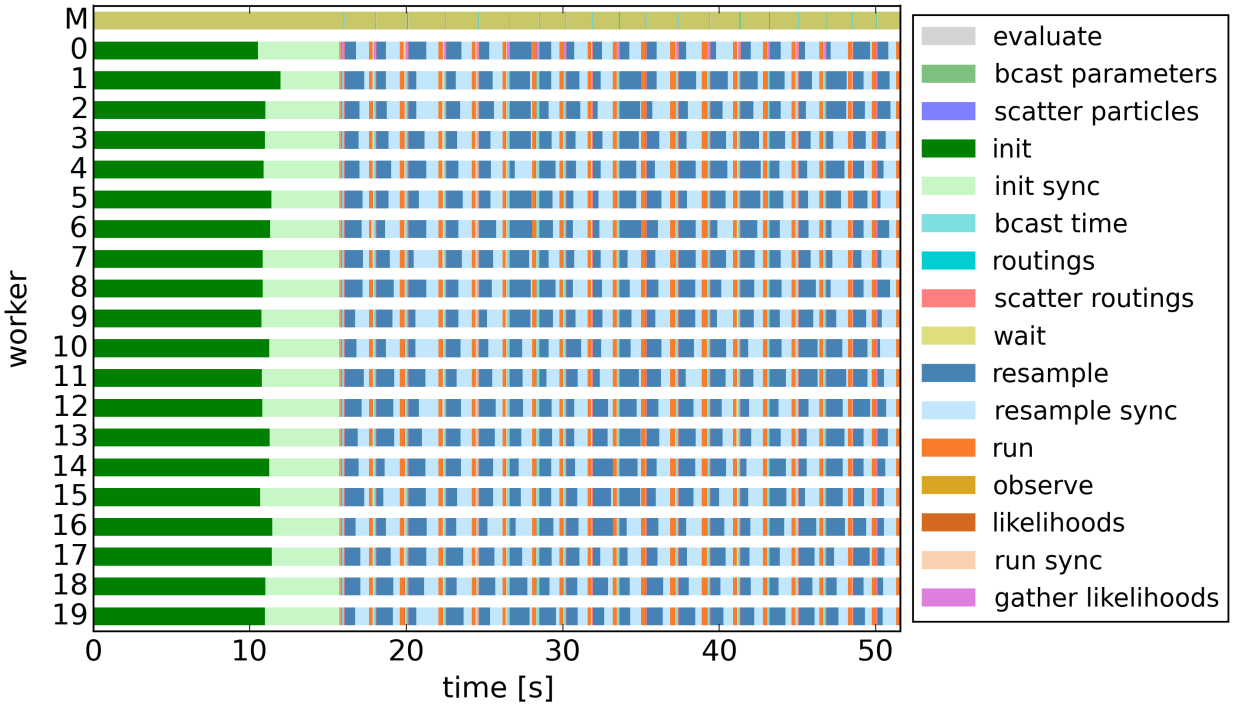


Fig. 30: Timestamps of within a single Particle Filter likelihood evaluation.

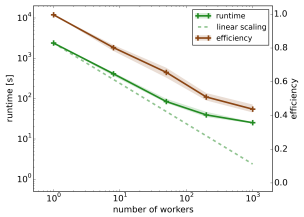


Fig. 31: Strong parallel scaling on Euler supercomputer at ETH Zurich, Switzerland.

- Artur Safin <artur.safin@eawag.ch>
- Mira Kattwinkel <kattwinkel-mira@uni-landau.de>
- Marco Dal Molin <marco.dalmolin@eawag.ch>
- Simone Ulzega <ulzg@zhaw.ch>
- Peter Reichert <peter.reichert@eawag.ch>

1.9.3 Acknowledgments

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage template.

1.10 History

1.10.1 0.4.0 (2019-06-12)

- Improvements in sandboxing, built-in serialization, report generation, plotting, thinning, support for legacy connector, and improvements in inference continuation procedure.

1.10.2 0.3.0 (2019-04-10)

- Many leaps forward: improvements in applications, local sandboxes, plotting, and many more.

1.10.3 0.2.1 (2019-03-06)

- Initial release for the spux project kickoff.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

T

U

V

W