bayesloop Documentation

Release 1.4

Christoph Mark

Sep 18, 2018

Contents

Probabilistic programming framework that facilitates objective model selection for time-varying parameter models.

See also:

If you want to contribute to the project or just browse the source code, visit the Github repository of *bayesloop*.

CHAPTER 1

Contents

1.1 Installation

The easiest way to install the latest release version of *bayesloop* is via `pip`:

```
pip install bayesloop
```

Alternatively, a zipped version can be downloaded here. The module is installed by calling `python setup.py install`.

1.1.1 Development version

The latest development version of *bayesloop* can be installed from the master branch using pip (requires git):

```
pip install git+https://github.com/christophmark/bayesloop
```

Alternatively, use this zipped version or clone the repository.

1.1.2 Dependencies

bayesloop is tested on Python 2.7, 3.5 and 3.6. It depends on NumPy, SciPy, SymPy, matplotlib, tqdm and dill. All except the last two are already included in the Anaconda distribution of Python. Windows users may also take advantage of pre-compiled binaries for all dependencies, which can be found at Christoph Gohlke's page.

1.1.3 Optional dependencies

bayesloop supports multiprocessing for computationally expensive analyses, based on the pathos module. The latest version can be obtained directly from GitHub using pip (requires git):

```
pip install git+https://github.com/uqfoundation/pathos
```

Note: Windows users need to install a C compiler *before* installing pathos. One possible solution for 64bit systems is to install Microsoft Visual C++ 2008 SP1 Redistributable Package (x64) and Microsoft Visual C++ Compiler for Python 2.7.

1.2 Tutorials

1.2.1 First steps with bayesloop

bayesloop models feature a two-level hierarchical structure: the low-level, observation model filters out measurement noise and provides the parameters, that one is interested in (volatility of stock prices, diffusion coefficient of particles, directional persistence of migrating cancer cells, rate of randomly occurring events, . . .). The observation model is, in most cases, given by a simple and well-known stochastic process: price changes are Gaussian-distributed, turning angles of moving cells follow a von-Mises distribution and the number of rare events within a given interval of time is Poisson-distributed. The aim of the observation model is to describe the measured data on a short time scale, while the parameters may change on longer time scales. The high-level, transition model describes *how* the parameters of the observation model change over time, i.e. whether there are abrupt parameter jumps or gradual variations. The transition model may itself depend on so-called hyper-parameters, for example the likelihood of parameter jumps, the magnitude of gradual parameter variations or the slope of a deterministic linear trend. The following tutorials show how to use the *bayesloop* module to infer both time-varying parameter values of the observation model as well as the hyper-parameter values of the transition model and compare different hypotheses about the parameter dynamics by approximating the model evidence, i.e. the probability of the measured data, given the observation model and transition model.

The first section of the tutorial introduces the main class of the module, `Study`, which enables fits of time-varying parameter models with fixed hyper-parameter values and the optimization of such hyper-parameters based on the model evidence. We provide a detailed description of how to import data, set the observation model and transition model, and perform the model fit. Finally, a plotting function to display the results is discussed briefly. This tutorial therefore provides the basis for later tutorials that discuss the extended classes `HyperStudy`, `ChangepointStudy` and `OnlineStudy`.

Study class

To start a new data study/analysis, create a new instance of the `Study` class:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        import bayesloop as bl

        S = bl.Study()
```

```
+ Created new study.
```

This object is central to an analysis conducted with *bayesloop*. It stores the data and further provides the methods to perform probabilistic inference on the models defined within the class, as described below.

Data import

In this first study, we use a simple, yet instructive example of heterogeneous time series, the annual number of coal mining accidents in the UK from 1851 to 1962. The data is imported as a NumPy array, together with corresponding timestamps. Note that setting timestamps is optional (if none are provided, timestamps are set to an integer sequence: 0, 1, 2,...).

```
In [2]: import numpy as np

        data = np.array([5, 4, 1, 0, 4, 3, 4, 0, 6, 3, 3, 4, 0, 2, 6, 3, 3, 5, 4, 5, 3, 1, 4,
                         4, 1, 5, 5, 3, 4, 2, 5, 2, 2, 3, 4, 2, 1, 3, 2, 2, 1, 1, 1, 1, 3, 0,
                         0, 1, 0, 1, 1, 0, 0, 3, 1, 0, 3, 2, 2, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0,
                         0, 2, 1, 0, 0, 0, 1, 1, 0, 2, 3, 3, 1, 1, 2, 1, 1, 1, 1, 2, 3, 3, 0,
                         0, 0, 1, 4, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0])

        S.load(data, timestamps=np.arange(1852, 1962))
```

```
+ Successfully imported array.
```

Note that this particular data set is also hard-coded into the `Study` class, for convenient testing:

```
In [3]: S.loadExampleData()
```

```
+ Successfully imported example data.
```

In case you have multiple observations for each time step, you may also provide the data in the form `np.array([[x1,y1,z1], [x2,y2,z2], ..., [xn,yn,zn]])`. Missing data points should be included as `np.nan`.

Observation model

The first step to create a probabilistic model to explain the data is to define the **observation model**, or **likelihood**. The observation model states the probability (density) of a data point at time t, given the parameter values at time t and possibly past data points. It therefore resembles the low-level model, in contrast to the transition model which describes how the parameters of the observation model change over time.

As coal mining disasters fortunately are rare events, we may model the number of accidents per year by a Poisson distribution. In *bayesloop*, this is done as follows:

```
In [4]: L = bl.observationModels.Poisson('accident_rate', bl.oint(0, 6, 1000))
        S.set(L)
```

```
+ Observation model: Poisson. Parameter(s): ['accident_rate']
```

We first define the observation model and provide two arguments: A name for the only parameter of the model, the `'accident_rate'`. We further have to provide discrete values for this parameter, as *bayesloop* computes all probability distributions on grids. As the Poisson distribution expects its parameter to be greater than zero, we choose an *open* interval between 0 and 6 with 1000 equally spaced values in between, by using the function `bl.oint()`. For closed intervals, one can also use `bl.cint()`, which acts exactly like the function `linspace` from NumPy. To avoid singularities in the probability values of the observation model, it is however recommended to use `bl.oint()` in most cases. Finally, we assign the defined observation model to our study instance with the method `set()`.

As the parameter boundaries depend on the data at hand, *bayesloop* will estimate appropriate parameter values, if one does not provide them:

```
In [5]: L = bl.observationModels.Poisson('accident_rate')
        S.set(L)
```

```
+ Estimated parameter interval for "accident_rate": [0.00749250749251, 7.49250749251] (1000 values).
+ Observation model: Poisson. Parameter(s): ['accident_rate']
```

Note that you can also use the following short form to define observation models: `L = bl.om.Poisson()`. All currently implemented observation models can be looked up in the API Docs or directly in `observationModels.py`. *bayesloop* further supports all probability distributions that are included in the scipy.stats as well as the sympy.stats module. See this tutorial for instructions on how to build custom observation models from arbitrary distributions.

In this example, the observation model only features a single parameter. If we wanted to model the annual number of accidents with a Gaussian distribution instead, we have to supply two parameter names (`mean` and `std`) and corresponding values:

```
L = bl.om.Gaussian('mean', bl.cint(0, 6, 200), 'std', bl.oint(0, 2, 200))
S.set(L)
```

Again, if we are not sure about parameter boundaries, we may assign `None` to one or all parameters, and *bayesloop* will estimate them:

```
L = bl.om.Gaussian('mean', None, 'std', bl.oint(0, 2, 200))
S.set(L)
```

The order has to remain `Name, Value, Name, Value, ...`, which is why we cannot simply omit the values and have to write `None` instead.

Transition model

As the dynamics of many real-world systems are the result of a multitude of underlying processes that act on different spatial and time scales, common statistical models with static parameters often miss important aspects of the systems' dynamics (see e.g. this article). *bayesloop* therefore calls for a second model, the **transition model**, which describes the temporal changes of the model parameters.

In this example, we assume that the accident rate itself may change gradually over time and choose a Gaussian random walk with the standard deviation $\sigma = 0.2$ as transition model. As for the observation model, we supply a unique name for hyper-parameter σ (named `sigma`) that describes the standard deviation of the parameter fluctuations and therefore the magnitude of changes. Again, we have to assign values for `sigma`, but only choose a single fixed value of 0.2, instead of a whole set of values. This single value can be optimized, by maximizing the model evidence, see here. To analyze and compare a set of different values, one may use an instance of a `HyperStudy` that is described in detail here. in this first example, we simply take the value of 0.2 as given. As the observation model may contain several parameters, we further have specify the parameter `accident_rate` as the target of this transition model.

```
In [6]: T = bl.transitionModels.GaussianRandomWalk('sigma', 0.2, target='accident_rate')
        S.set(T)
```

```
+ Transition model: Gaussian random walk. Hyper-Parameter(s): ['sigma']
```

Note that you can also use the following short form to define transition models: `T = bl.tm.GaussianRandomWalk()`. All currently implemented transition models can be looked up in the API Docs or directly in `transitionModels.py`.

Model fit

At this point, the hierarchical time series model for the coal mining data set is properly defined and we may continue to perform the model fit. *bayesloop* employs a forward-backward algorithm that is based on Hidden Markov models. It basically breaks down the high-dimensional inference problem of all time steps into many low-dimensional ones for each individual time step. The inference algorithm is implemented by the `fit` method:

```
In [7]: S.fit()
```

```
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -74.63801

    + Finished backward pass.
    + Computed mean parameter values.
```

By default, `fit` computes the so-called *smoothing distribution* of the model parameters for each time step. This distribution states the probability (density) of the parameter value at a time step t, given all past and future data points. All distributions have the same shape as the parameter grid, and are stored in `S.posteriorSequence` for further analysis. Additionally, the mean values of each distribution are stored in `S.posteriorMeanValues`, as point estimates. Finally, the (natural) logarithmic value of the model evidence, the probability of the data given the chosen model, is stored in `S.logEvidence` (more details on evidence values follow).

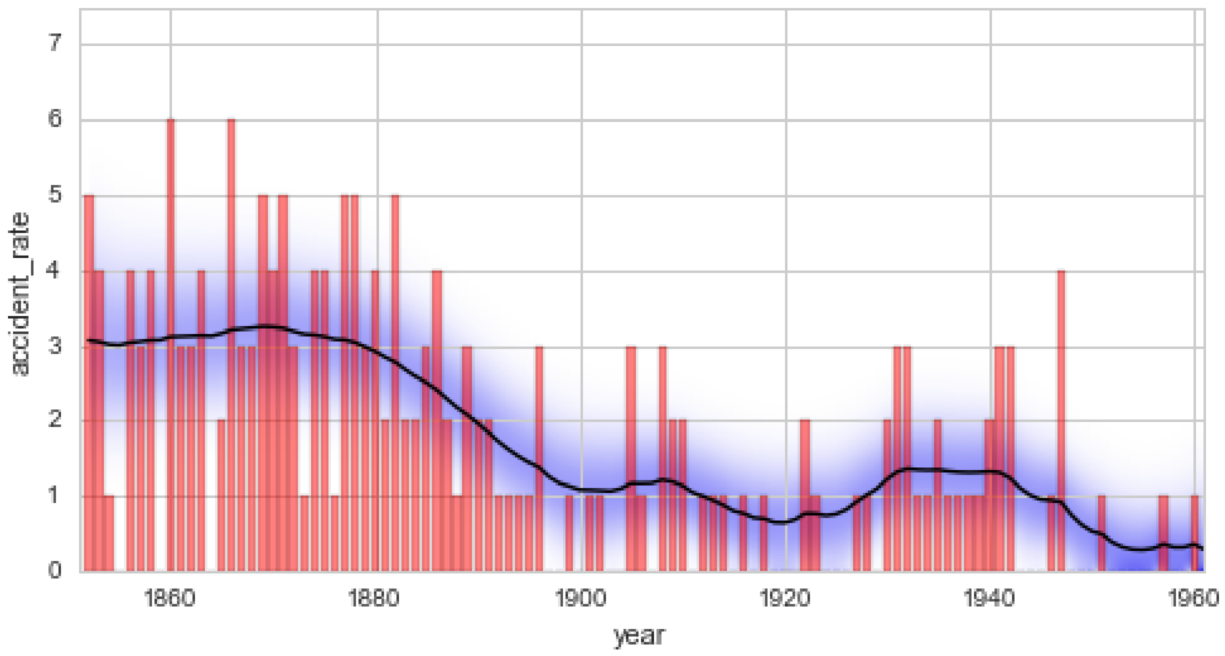
To simulate an on-line analysis, where at each step in time t, only past data points are available, one may provide the keyword-argument `forwardOnly=True`. In this case, only the *forward*-part of the algorithm in run. The resulting parameter distributions are called *filtering distributions*.

Plotting

To display the temporal evolution or the distribution of the model parameters at a certain time step, the `Study` class provides the method `plot`. If no time step is specified, the method displays the mean values together with the marginal distributions for one parameter of the model. The parameter to be plotted can be chosen by providing its name.

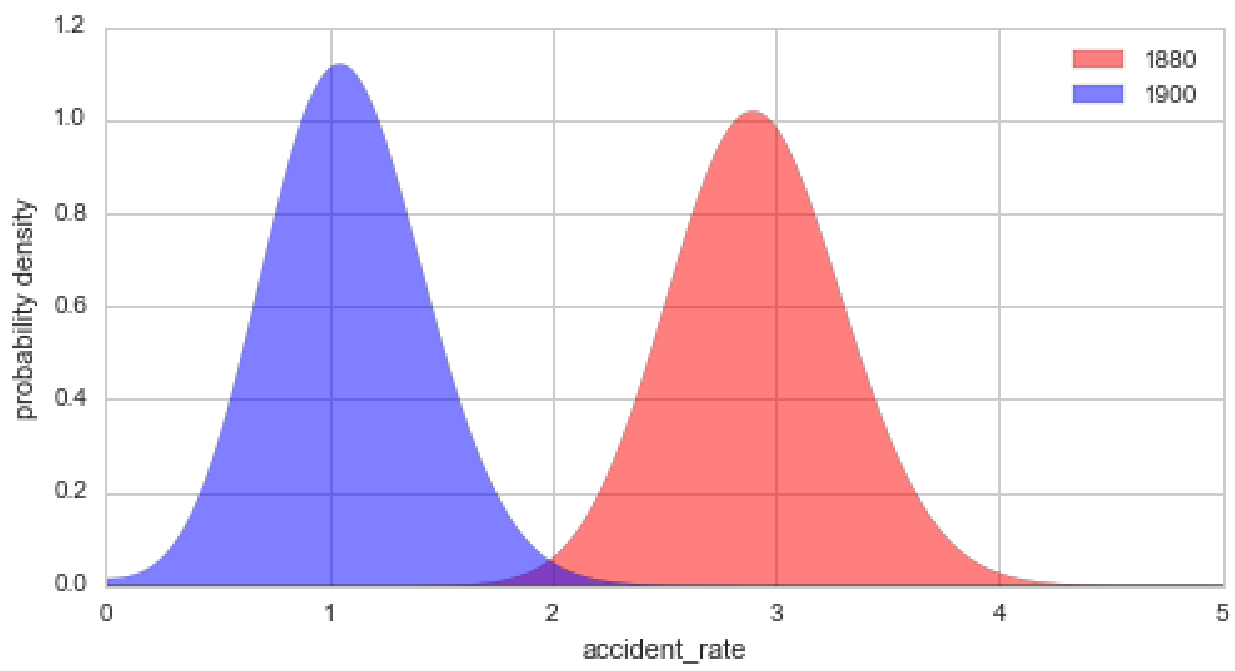
Here, we plot the original data (in red) together with the inferred disaster rate (mean value in black). The marginal parameter distribution is displayed as a blue overlay, by default with a gamma correction of $\gamma = 0.5$ to enhance relative differences in the width of the distribution (this behavior can be changed by the keyword argument `gamma`):

```
In [8]: plt.figure(figsize=(8, 4))

        # plot of raw data
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)

        # parameter plot
        S.plot('accident_rate')

        plt.xlim([1851, 1961])
        plt.xlabel('year');
```



From this first analysis, we may conclude that before 1880, an average of ≈ 3 accidents per year were recorded. This changes significantly between 1880 and 1900, when the accident-rate drops to ≈ 1 per year. We can also directly inspect the distribution of the accident rate at specific points in time, using the `plot` method with specified keyword argument `t`:

```
In [9]: plt.figure(figsize=(8, 4))

        S.plot('accident_rate', t=1880, facecolor='r', alpha=0.5, label='1880')
        S.plot('accident_rate', t=1900, facecolor='b', alpha=0.5, label='1900')

        plt.legend()
        plt.xlim([0, 5]);
```



Without the `plot=True` argument, this method only returns the parameter values (`r1`, `r2`, as specified when setting the observation model) as well as the corresponding probability values `p1` and `p2`. Note that the returned probability values are always normalized to 1, so that we may easily evaluate the probability of certain conditions, like the probability of an accident rate < 1 in the year 1900:

We can further evaluate the probability of certain conditions, for example the probability that the accident rate was < 1 in the year 1900, using the `eval` method:

```
In [10]: S.eval('accident_rate < 1', t=1900);
```

```
P(accident_rate < 1) = 0.42198256057
```

For further details on the evaluation of probabilities derived from a combination of inferred parameters (possibly from different `Study` instances), see this tutorial

Saving studies

As the `Study` class instance (above denoted by `S`) of a conducted analysis contains all information about the inferred parameter values, it may be convenient to store the entire instance `S` to file. This way, it can be loaded again later, for example to refine the study, create different plots or perform further analyses based on the obtained results. *bayesloop* provides two functions, `bl.save()` and `bl.load()` to store and retrieve existing studies:

```
bl.save('file.bl', S)

...

S = bl.load('file.bl)
```

1.2.2 Model Selection

In Bayesian statistics, an objective model comparison is carried out by comparing the models' marginal likelihood. The likelihood function describes the probability (density) of the data, given the parameter values (and thereby the chosen model). By integrating out (marginalizing) the parameter values, one obtains the marginal likelihood (also called the *model evidence*), which directly measures the fitness of the model at hand. The model evidence represents a powerful tool for model selection, as it does not assume specific distributions (e.g. Student's t-test assumes Gaussian distributed variables) and automatically follows the principle of Occam's razor.

The forward-backward algorithm that *bayesloop* allows to approximate the model evidence based on the discretized parameter distributions. Details on this method in the context of Hidden Markov models can be found here.

This section aims at giving a very brief introduction of Bayes factors together with an example based on the coal mining data and further introduces combined transition models in *bayesloop*.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        import numpy as np
        import bayesloop as bl

        # prepare study for coal mining data
        S = bl.Study()
        S.loadExampleData()

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        S.set(L)
```

```
+ Created new study.
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
```

Bayes factors

Instead of interpreting the value of the marginal likelihood for a single model, it is common to compare two competing models/explanations M_1 and M_2 by evaluating the Bayes factor, here denoted as B_{12}. The Bayes factor is given by the ratio of the two marginal likelihood values:

$$B_{12} = \frac{p(D|M_1)}{p(D|M_2)}$$

where $p(D|M_i)$ states the probability of the data (marginal likelihood) given model M_i. A value of $B_{12} > 1$ therefore indicates that the data is better explained by model M_1, compared to M_2. More detailed information for the interpretation of the value of Bayes factors can be found here and in the references therein.

As a first example, we investigate whether the inferred disaster rate of the coal mining data set indeed should be modeled as a time-varying parameter (a constant rate is an equally valid hypothesis). We first fit the model using the `GaussianRandomWalk` transition model with a standard deviation of $\sigma = 0.2$ to determine the corresponding model evidence (on a log scale). Subsequently, we use the simpler `Static` transition model (assuming no change of the disaster rate over time) and compare the resulting model evidence by computing the Bayes factor. Note that for computational efficiency, the keyword argument `evidenceOnly=True` is used, which evaluates the model evidence, but does not store any results for plotting.

```
In [2]: # dynamic disaster rate
        T = bl.tm.GaussianRandomWalk('sigma', 0.2, target='accident_rate')
        S.set(T)
        S.fit(evidenceOnly=True)

        dynamicLogEvidence = S.log10Evidence

        #static disaster rate
        T = bl.tm.Static()
        S.set(T)
        S.fit(evidenceOnly=True)

        staticLogEvidence = S.log10Evidence

        # determine Bayes factor
        B = 10**(dynamicLogEvidence - staticLogEvidence)
        print '\nBayes factor: B =', B
```

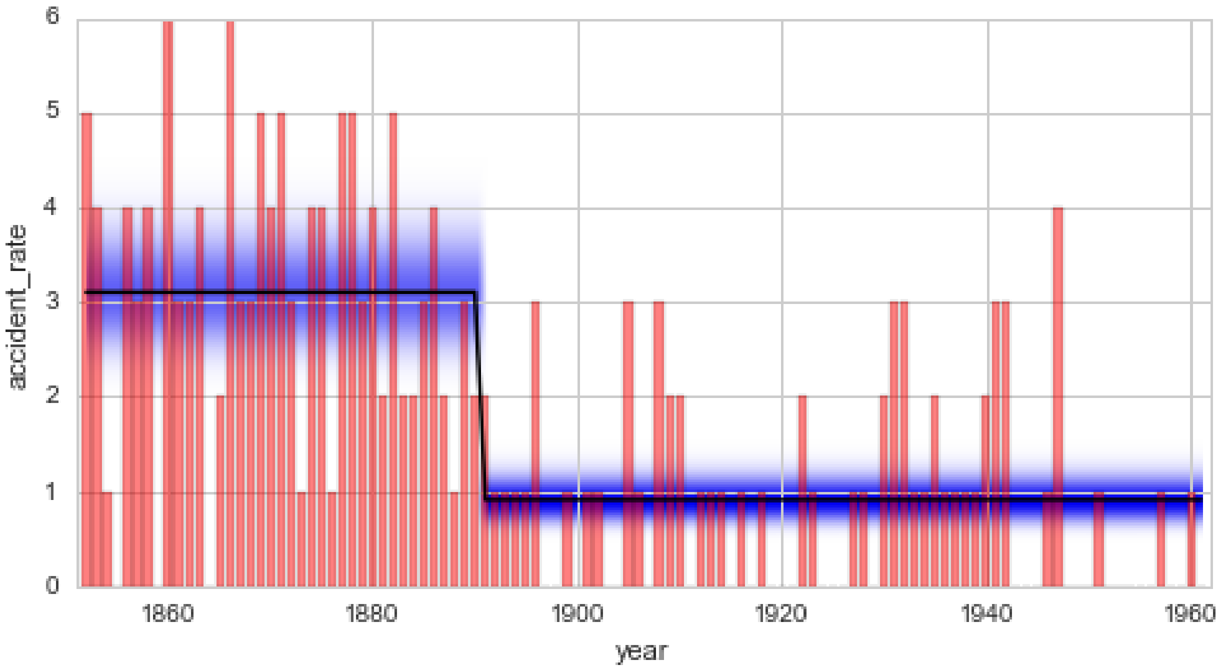
```
+ Transition model: Gaussian random walk. Hyper-Parameter(s): ['sigma']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -74.59055
+ Transition model: Static/constant parameter values. Hyper-Parameter(s): []
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -88.00564

Bayes factor: B = 2.60066520417e+13
```

The computed Bayes factor $B = 2.6 \cdot 10^{13}$ shows decisive support for the first hypothesis of a dynamic disaster rate.

While this analysis conducted above clearly rules for a time-varying rate, there may still exist a dynamic model that represents a better fit than the Gaussian random walk with $\sigma = 0.2$. A very simple dynamic model is given by the transition model `ChangePoint` that assumes an abrupt change of the disaster rate at a predefined point in time, we choose 1890 here. Note that the transition model `ChangePoint` does not need a `target` parameter, as it is applied to all parameters of an observation model. We perform a full fit in this case, as we want to provide a plot of the result.

```
In [3]: # dynamic disaster rate (change-point model)
        T = bl.tm.ChangePoint('tChange', 1890)
        S.set(T)
        S.fit()

        dynamicLogEvidence2 = S.log10Evidence

        # determine Bayes factor
        B = 10**(dynamicLogEvidence2 - dynamicLogEvidence)
        print '\nBayes factor: B =', B

        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Transition model: Change-point. Hyper-Parameter(s): ['tChange']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -74.41178

    + Finished backward pass.
    + Computed mean parameter values.

Bayes factor: B = 1.50929883739
```



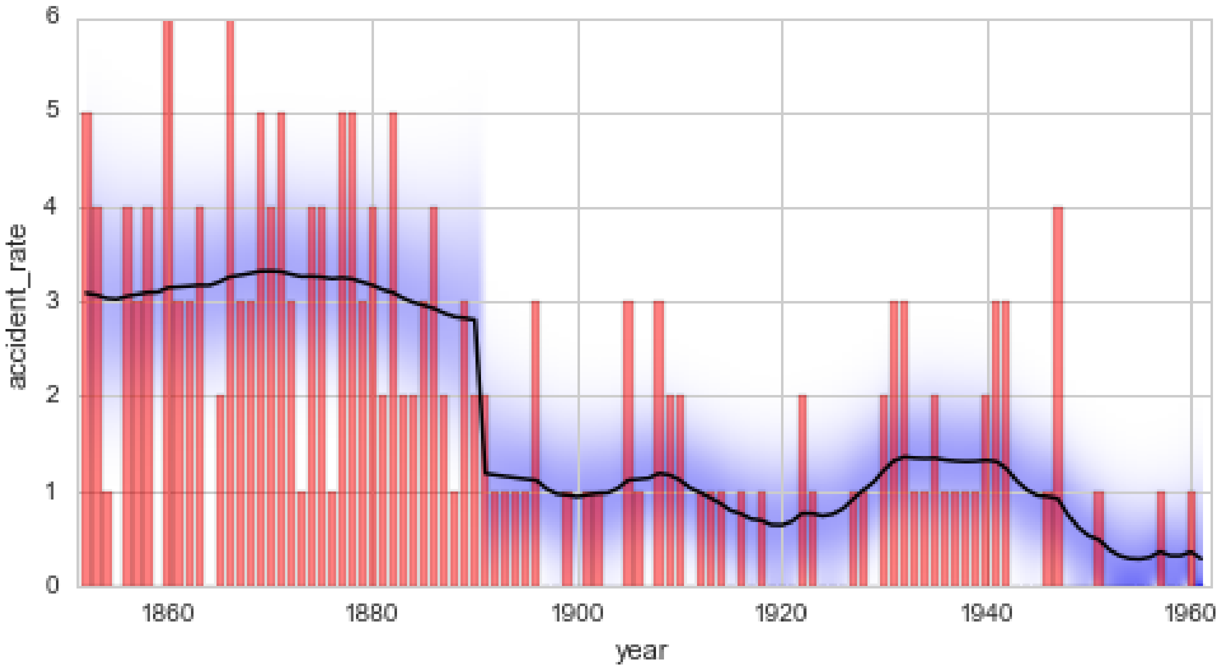
The Bayes factor shows support in favor of the new change-point model. There is a 50% increased probability that the data is generated by the change-point model, compared to the Gaussian random walk model. Some may however argue that the value of $B = 1.5$ indicates only very weak support and is not worth more than a bare mention. Based on the data at hand, no clear decision between the two models can be made. While the change-point model is favored because it is more restricted (the number of possible data sets that can be described by this model is much smaller than for the Gaussian random walk) and therefore *"simpler"*, it cannot capture fluctuations of the disaster rate before and after 1890 like the Gaussian random walk model does.

Combined transition models

The discussion above shows that depending on the data set, different transition models better explain different aspects of the data. For some data sets, a satisfactory transition model may only be found by combining several simple transition models. *bayesloop* provides a transition model class called `CombinedTransitionModel` that can be supplied with a sequence of transition models that are subsequently applied at every time step. Here, we combine the change-point model and the Gaussian random walk model to check whether the combined model yields a better explanation of the data, compared to the change-point model alone:

```
In [4]: # combined model (change-point model + Gaussian random walk)
        T = bl.tm.CombinedTransitionModel(bl.tm.ChangePoint('tChange', 1890),
                                          bl.tm.GaussianRandomWalk('sigma', 0.2, target='accident_rat
        S.set(T)
        S.fit()

        combinedLogEvidence = S.log10Evidence

        # determine Bayes factor
        B = 10**(combinedLogEvidence - dynamicLogEvidence2)
        print '\nBayes factor: B =', B
        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Transition model: Combined transition model. Hyper-Parameter(s): ['tChange', 'sigma']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -74.01460

    + Finished backward pass.
    + Computed mean parameter values.

Bayes factor: B = 2.49563205383
```



Again, the refined model is favored by a Bayes factor of $B = 2.5$.

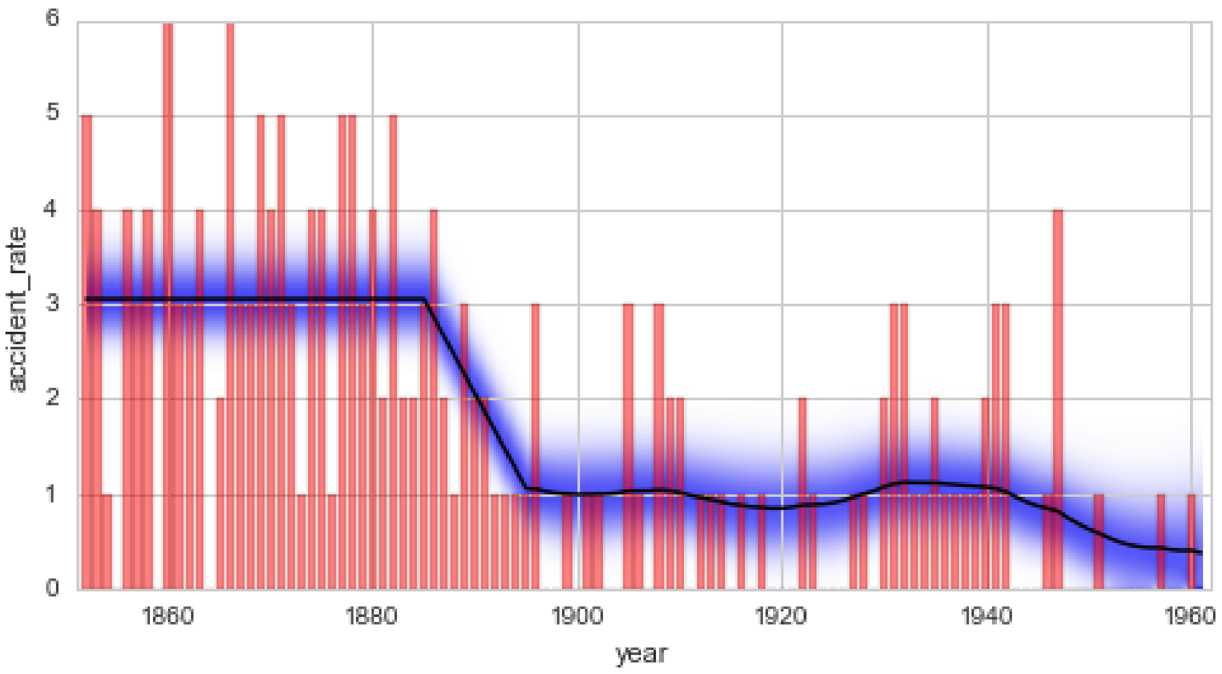
Serial transition model

The combined transition models introduced above substantially extend the number of different transition models. These transition models imply identical parameter dynamics for all time steps. In many applications, however, there exist so-called structural breaks when parameter dynamics exhibit a fundamental change. In contrast to an abrupt change of the parameter values in case of a change-point, a structural break indicates an abrupt change of the transition model at a specific point in time. The class `SerialTransitionModel` allows to define a sequence of transition models (including combined transition models) together with a sequence of time steps that denote the structural breaks. Each break-point is defined just like any other transition model, containing a unique name and a time stamp (or an array of possible time stamps, see this tutorial on change-point studies). Note, however, that the `BreakPoint` transition model class can only be used as a sub-model of an `SerialTransitionModel` instance.

We use this new class of transition model to explore the idea that the number of coal mining disasters did not decrease instantaneously, but instead decreased linearly over the course of a few years. We assume a static disaster rate until 1885, followed by a deterministic decrease of 0.2 disasters per year (deterministic transition models are defined easily by custom Python functions, see below) until 1895. Finally, the disaster rate after the year 1895 is modeled by a Gaussian random walk of the disaster rate with a standard deviation of $\sigma = 0.1$. Note that we pass the transition models and the corresponding structural breaks (time steps) to the `SerialTransitionModel` in turns. While this order may increase readability, one can also pass the models first, followed by the corresponding time steps.

```
In [5]: # Definition of a linear decrease transition model.
        # The first argument of any determinsitic model must be the time stamp
        # and any hyper-parameters of the model are supplied as keyword-arguments.
        # The hyper-parameter value(s) is/are supplied as default values.
        def linear(t, slope=-0.2):
            return slope*t

        T = bl.tm.SerialTransitionModel(bl.tm.Static(),
                                        bl.tm.BreakPoint('t_1', 1885),
                                        bl.tm.Deterministic(linear, target='accident_rate'),
                                        bl.tm.BreakPoint('t_2', 1895),
                                        bl.tm.GaussianRandomWalk('sigma', 0.1, target='accident_rate
        S.set(T)
        S.fit()

        serialLogEvidence = S.log10Evidence

        # determine Bayes factor
        B = 10**(serialLogEvidence - combinedLogEvidence)
        print '\nBayes factor: B =', B

        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Transition model: Serial transition model. Hyper-Parameter(s): ['slope', 'sigma', 't_1', 't_2']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -72.93384

    + Finished backward pass.
    + Computed mean parameter values.

Bayes factor: B = 12.0436384646
```



The Bayes factor of the serial model compared to the combined model is determined to $B = 12.0$. This value indicates positive/strong evidence in favor of the serial model. Keep in mind, though, that the time steps of the structural breaks and the slope of the linear decrease are not inferred in this example, but are instead set by the user. The determined Bayes factor therefore relies on the assumption that these values are true, and the uncertainty tied to these values is therefore not reflected in the value of the Bayes factor. More elaborate studies that take into account the uncertainty tied to these hyper-parameters are introduced here and here.

The iterative approach of creating new hypotheses and comparing them to the best hypothesis currently available provides an objective and never-the-less straight-forward way of exploring new data sets. The subsequent tutorial provides methods to improve upon values for hyper-parameters that are *"blindly"* set by the user.

1.2.3 Optimization of hyper-parameters

The model evidence cannot only be used to compare different kinds of time series models, but also to optimize the hyper-parameters of a given transition model by maximizing its evidence value. The `Study` class of *bayesloop* contains a method `optimize` which relies on the `minimize` function of the `scipy.optimize` module. Since *bayesloop* has no gradient information about the hyper-parameters, the optimization routine is based on the COBYLA algorithm. The following two sections introduce the optimization of hyper-parameters using *bayesloop* and further describe how to selectively optimize specific hyper-parameters in nested transition models.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        import numpy as np
        import bayesloop as bl

        # prepare study for coal mining data
        S = bl.Study()
        S.loadExampleData()
        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        S.set(L)
```

```
+ Created new study.
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
```

Global optimization

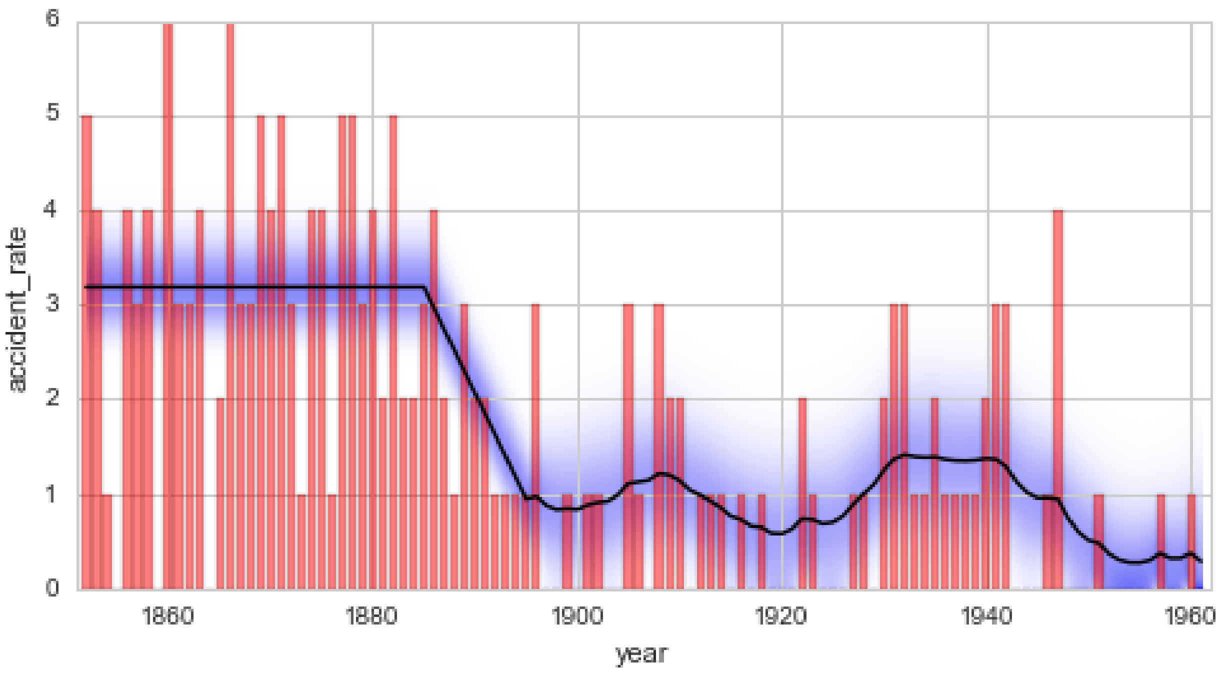
The `optimize` method supports all currently implemented transition models with continuous hyper-parameters, as well as combinations of multiple models. The change-point model as well as the serial transition model represent exceptions here, as their parameters `tChange` and `tBreak`, respectively, are discrete. These discrete parameters are ignored by the optimization routine. See the tutorial on change-point studies for further information on how to analyze structural breaks and change-points. By default, all continuous hyper-parameters of the transition model are optimized. *bayesloop* further allows to selectively optimize specific hyper-parameters, see *below*. The parameter values set by the user when defining the transition model are used as starting values. During optimization, only the log-evidence of the model is computed. When finished, a full fit is done to provide the parameter distributions and mean values for the optimal model setting.

We take up the coal mining example again, and stick with the serial transition model defined here. This time, however, we optimize the slope of the linear decrease from 1885 to 1895 and the magnitude of the fluctuations afterwards (i.e. the standard deviation of the Gaussian random walk):

```
In [2]: # define linear decrease transition model
        def linear(t, slope=-0.2):
            return slope*t

        T = bl.tm.SerialTransitionModel(bl.tm.Static(),
                                        bl.tm.BreakPoint('t_1', 1885),
                                        bl.tm.Deterministic(linear, target='accident_rate'),
                                        bl.tm.BreakPoint('t_2', 1895),
                                        bl.tm.GaussianRandomWalk('sigma', 0.1, target='accident_rate
        S.set(T)

        S.optimize()

        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Transition model: Serial transition model. Hyper-Parameter(s): ['slope', 'sigma', 't_1', 't_2']
+ Starting optimization...
  --> All model parameters are optimized (except change/break-points).
    + Log10-evidence: -72.93384 - Parameter values: [-0.2   0.1]
    + Log10-evidence: -96.81252 - Parameter values: [ 0.8   0.1]
    + Log10-evidence: -75.18192 - Parameter values: [-0.2   1.1]
    + Log10-evidence: -78.43877 - Parameter values: [-1.19559753  0.00626873]
    + Log10-evidence: -78.80509 - Parameter values: [-0.69779877  0.05313437]
    + Log10-evidence: -85.79404 - Parameter values: [ 0.04572939  0.05398839]
    + Log10-evidence: -72.76628 - Parameter values: [-0.21058883  0.34977565]
    + Log10-evidence: -73.72301 - Parameter values: [-0.33553394  0.34607154]
    + Log10-evidence: -74.02943 - Parameter values: [-0.08663732  0.3659319 ]
    + Log10-evidence: -73.17022 - Parameter values: [-0.14861308  0.35785378]
    + Log10-evidence: -72.79393 - Parameter values: [-0.21462789  0.38076353]
    + Log10-evidence: -72.92776 - Parameter values: [-0.27089564  0.33336413]
    + Log10-evidence: -72.76915 - Parameter values: [-0.24074224  0.34156989]
    + Log10-evidence: -72.76679 - Parameter values: [-0.20498306  0.33519087]
    + Log10-evidence: -72.78617 - Parameter values: [-0.20460701  0.35480089]
    + Log10-evidence: -72.75279 - Parameter values: [-0.21791489  0.347062  ]
    + Log10-evidence: -72.74424 - Parameter values: [-0.21953173  0.33941864]
    + Log10-evidence: -72.74031 - Parameter values: [-0.22648739  0.33586139]
    + Log10-evidence: -72.73350 - Parameter values: [-0.22642717  0.32804913]
    + Log10-evidence: -72.72784 - Parameter values: [-0.22747605  0.32030735]
    + Log10-evidence: -72.72743 - Parameter values: [-0.23183807  0.313826  ]
    + Log10-evidence: -72.72248 - Parameter values: [-0.22818709  0.31243706]
    + Log10-evidence: -72.71753 - Parameter values: [-0.22205348  0.30759827]
    + Log10-evidence: -72.72569 - Parameter values: [-0.21571672  0.31216781]
    + Log10-evidence: -72.71235 - Parameter values: [-0.22363971  0.2999485 ]
    + Log10-evidence: -72.70840 - Parameter values: [-0.22123414  0.29251557]
    + Log10-evidence: -72.70418 - Parameter values: [-0.22439552  0.28537129]
    + Log10-evidence: -72.70077 - Parameter values: [-0.22547625  0.2776339 ]
    + Log10-evidence: -72.70450 - Parameter values: [-0.23171754  0.2729348 ]
    + Log10-evidence: -72.70076 - Parameter values: [-0.21867277  0.27379362]
    + Log10-evidence: -72.69810 - Parameter values: [-0.22058074  0.27038503]
    + Log10-evidence: -72.69617 - Parameter values: [-0.2224925   0.26697858]
    + Log10-evidence: -72.69482 - Parameter values: [-0.22372428  0.26327162]
    + Log10-evidence: -72.69366 - Parameter values: [-0.22269705  0.25950286]
    + Log10-evidence: -72.69256 - Parameter values: [-0.22373849  0.255738  ]
    + Log10-evidence: -72.69157 - Parameter values: [-0.2233398   0.25185215]
    + Log10-evidence: -72.69093 - Parameter values: [-0.22465019  0.24817225]
    + Log10-evidence: -72.69026 - Parameter values: [-0.22230095  0.24505137]
    + Log10-evidence: -72.68984 - Parameter values: [-0.22155855  0.24121632]
    + Log10-evidence: -72.69200 - Parameter values: [-0.21792638  0.23977891]
    + Log10-evidence: -72.68976 - Parameter values: [-0.22524306  0.23991896]
    + Log10-evidence: -72.68944 - Parameter values: [-0.2249306   0.23799099]
    + Log10-evidence: -72.68913 - Parameter values: [-0.22449406  0.23608727]
    + Log10-evidence: -72.68891 - Parameter values: [-0.22415052  0.2341646 ]
    + Log10-evidence: -72.68891 - Parameter values: [-0.22465387  0.23227745]
    + Log10-evidence: -72.68873 - Parameter values: [-0.22367816  0.23223652]
    + Log10-evidence: -72.68886 - Parameter values: [-0.22184677  0.23155776]
    + Log10-evidence: -72.68877 - Parameter values: [-0.22341493  0.23317694]
    + Log10-evidence: -72.68870 - Parameter values: [-0.22323996  0.23202113]
    + Log10-evidence: -72.68870 - Parameter values: [-0.22291326  0.23165823]
    + Log10-evidence: -72.68875 - Parameter values: [-0.22250844  0.23193124]
    + Log10-evidence: -72.68867 - Parameter values: [-0.22322072  0.23127891]
    + Log10-evidence: -72.68866 - Parameter values: [-0.22344227  0.23084378]
    + Log10-evidence: -72.68864 - Parameter values: [-0.22319177  0.23042466]
    + Log10-evidence: -72.68863 - Parameter values: [-0.22300139  0.22997502]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22340318  0.22969756]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22359843  0.22925002]
    + Log10-evidence: -72.68864 - Parameter values: [-0.22362574  0.22979791]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22331126  0.22965818]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22321913  0.22961928]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22312139  0.22959815]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22319968  0.22966534]
    + Log10-evidence: -72.68862 - Parameter values: [-0.22319262  0.22952285]
    + Log10-evidence: -72.68861 - Parameter values: [-0.22320182  0.22942328]
    + Log10-evidence: -72.68861 - Parameter values: [-0.22319833  0.22932334]
    + Log10-evidence: -72.68861 - Parameter values: [-0.22324289  0.22923382]
    + Log10-evidence: -72.68861 - Parameter values: [-0.22322927  0.22913475]
    + Log10-evidence: -72.68861 - Parameter values: [-0.22322709  0.22903477]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22326426  0.22894194]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22322328  0.22885071]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22319605  0.22875449]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22322886  0.22866003]
```

```
    + Log10-evidence: -72.68860 - Parameter values: [-0.22321891  0.22856053]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22322983  0.22846113]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22316925  0.22855468]
    + Log10-evidence: -72.68860 - Parameter values: [-0.22324084  0.22846296]
+ Finished optimization.
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -72.68860

    + Finished backward pass.
    + Computed mean parameter values.
```



The optimal value for the standard deviation of the varying disaster rate is determined to be ≈ 0.23, the initial guess of $\sigma = 0.1$ is therefore too restrictive. The value of the slope is only optimized slightly, resulting in an optimal value of ≈ -0.22. The optimal hyper-parameter values are displayed in the output during optimization, but can also be inspected directly:

```
In [3]: print 'slope =', S.getHyperParameterValue('slope')
        print 'sigma =', S.getHyperParameterValue('sigma')
```

```
slope = -0.223240841019
sigma = 0.228462963962
```

Conditional optimization in nested transition models

The previous section introduced the `optimize` method of the `Study` class. By default, all (continuous) hyper-parameters of the chosen transition model are optimized. In some applications, however, only specific hyper-parameters may be subject to optimization. Therefore, a list of parameter names (or a single name) may be passed to `optimize`, specifying which parameters to optimize. Note that all hyper-parameters have to be given a unique name. An example for a (quite ridiculously) nested transition model is defined below. Note that the deterministic transition models are defined via `lambda` functions.

```
In [4]: T = bl.tm.SerialTransitionModel(bl.tm.CombinedTransitionModel(
                                            bl.tm.GaussianRandomWalk('early_sigma', 0.05, target='acc
                                            bl.tm.RegimeSwitch('pmin', -7)
                                            ),
                                        bl.tm.BreakPoint('first_break', 1885),
                                        bl.tm.Deterministic(lambda t, slope_1=-0.2: slope_1*t, target
                                        bl.tm.BreakPoint('second_break', 1895),
                                        bl.tm.CombinedTransitionModel(
                                            bl.tm.GaussianRandomWalk('late_sigma', 0.25, target='acc
                                            bl.tm.Deterministic(lambda t, slope_2=0.0: slope_2*t, ta
                                            )
                                        )
        S.set(T)
```

```
+ Transition model: Serial transition model. Hyper-Parameter(s): ['early_sigma', 'pmin', 'slope_1',
```

This transition model assumes a combination of gradual and abrupt changes until 1885, followed by a deterministic decrease of the annual disaster rate until 1895. Afterwards, the disaster rate is modeled by a combination of a decreasing trend and random fluctuations. Instead of discussing exactly how meaningful the proposed transition model really is, we focus on how to specify different (groups of) hyper-parameters that we might want to optimize.

All hyper-parameter names occur only once within the transition model and may simply be stated by their name: `S.optimize('pmin')`. Note that you may also pass a single or multiple hyper-parameter(s) as a list: `S.optimize(['pmin'])`, `S.optimize(['pmin', 'slope_2'])`. For deterministic models, the argument name also represents the hyper-parameter name:

```
In [5]: S.optimize(['slope_2'])
```

```
+ Starting optimization...
  --> Parameter(s) to optimize: ['slope_2']
    + Log10-evidence: -72.78352 - Parameter values: [ 0.]
    + Log10-evidence: -93.84882 - Parameter values: [ 1.]
    + Log10-evidence: -80.98325 - Parameter values: [-1.]
    + Log10-evidence: -85.81409 - Parameter values: [-0.5]
    + Log10-evidence: -82.83302 - Parameter values: [ 0.25]
    + Log10-evidence: -73.27797 - Parameter values: [-0.125]
    + Log10-evidence: -74.00209 - Parameter values: [ 0.0625]
    + Log10-evidence: -72.56560 - Parameter values: [-0.03125]
    + Log10-evidence: -72.59014 - Parameter values: [-0.0625]
    + Log10-evidence: -72.54880 - Parameter values: [-0.046875]
    + Log10-evidence: -72.59014 - Parameter values: [-0.0625]
    + Log10-evidence: -72.56237 - Parameter values: [-0.0546875]
    + Log10-evidence: -72.54744 - Parameter values: [-0.04296875]
    + Log10-evidence: -72.54976 - Parameter values: [-0.0390625]
    + Log10-evidence: -72.54814 - Parameter values: [-0.04101562]
    + Log10-evidence: -72.54744 - Parameter values: [-0.04394531]
    + Log10-evidence: -72.54752 - Parameter values: [-0.04443359]
    + Log10-evidence: -72.54742 - Parameter values: [-0.04370117]
    + Log10-evidence: -72.54741 - Parameter values: [-0.04345703]
    + Log10-evidence: -72.54742 - Parameter values: [-0.04321289]
    + Log10-evidence: -72.54741 - Parameter values: [-0.04335703]
    + Log10-evidence: -72.54741 - Parameter values: [-0.04355703]
+ Finished optimization.
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -72.54741
    + Finished backward pass.
    + Computed mean parameter values.
```

Although the optimization of hyper-parameters helps to objectify the choice of hyper-parameter values and may even be used to gain new insights into the dynamics of systems, optimization alone does not provide any measure of uncertainty tied to the obtained, optimal hyper-parameter value. The next tutorial discusses an approach to infer not only the time-varying parameter distributions, but also the distribution of hyper-parameters.

1.2.4 Hyper-study

The time series models built with the help of *bayesloop* are called hierarchical models, since the parameters of the observation model are in turn controlled by hyper-parameters that are possibly included in the transition model. In the previous section, we optimized two hyper-parameters of a serially combined transition model: the slope of the decrease in coal mining disasters from 1885 to 1895, and the magnitude of the fluctuations afterwards. While the optimization routine yields the most probable values of these hyper-parameters, one might also be interested in the uncertainty tied to these *"optimal"* values. *bayesloop* therefore provides the `HyperStudy` class that allows to compute the full distribution of hyper-parameters by defining a discrete grid of hyper-parameter values.

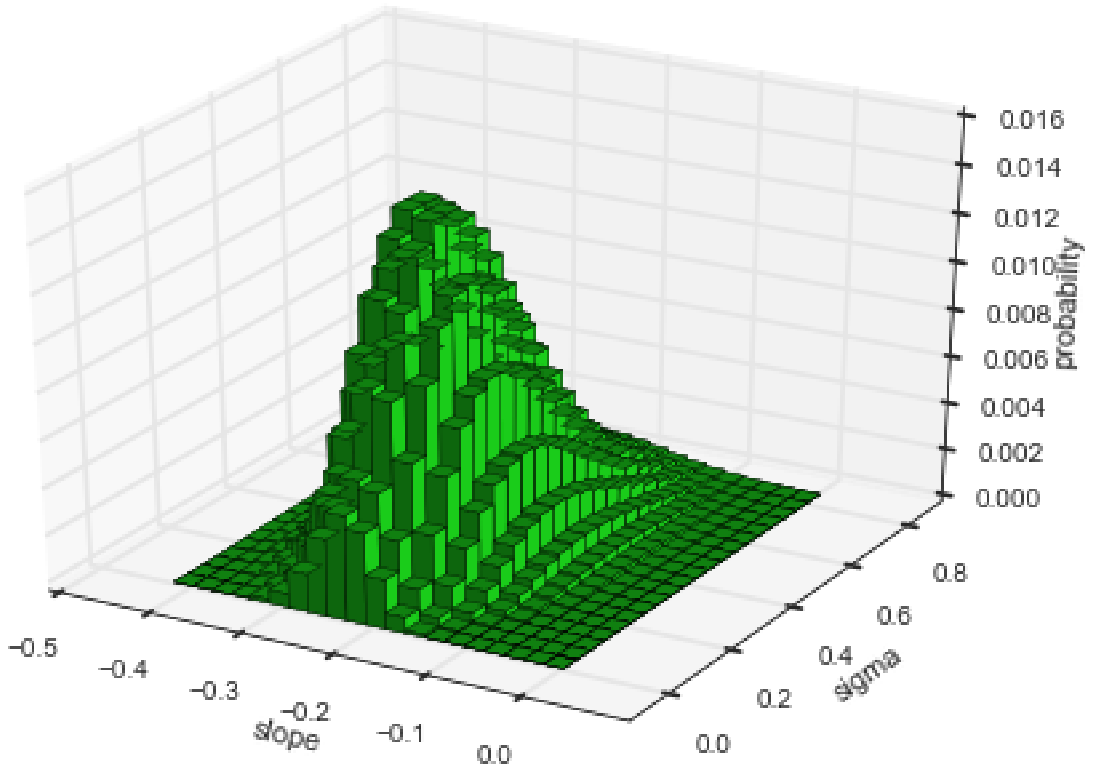
While the `HyperStudy` instance can be configured just like a standard `Study` instance, one may supply not only single hyper-parameter values, but also lists/arrays of values (*Note:* It will fall back to the standard fit method if only one combination of hyper-parameter values is supplied). Here, we test a range on hyper-parameter values by supplying regularly spaced hyper-parameter values using the `cint` function (one can of course also use similar functions like numpy.linspace). In the example below, we return to the serial transition model used here and compute a two-dimensional distribution of the two hyper-parameters `slope` (20 steps from -0.4 to 0.0) and `sigma` (20 steps from 0.0 to 0.8).

After running the fit-method for all value-tuples of the hyper-grid, the model evidence values of the individual fits are used as weights to compute weighted average parameter distributions. These average parameter distributions allow to assess the temporal evolution of the model parameters, explicitly considering the uncertainty of the hyper-parameters. However, in case one is only interested in the hyper-parameter distribution, setting the keyword-argument `evidenceOnly=True` of the `fit` method shortens the computation time but skips the evaluation of parameter distributions.

Finally, *bayesloop* provides several methods to plot the results of the hyper-study. To display the joint distribution of two hyper-parameters, choose `getJointHyperParameterDistribution` (or shorter: `getJHPD`). The method automatically computes the marginal distribution for the two specified hyper-parameters and returns three arrays, two for the hyper-parameters values and one with the corresponding probability values. Here, the first argument represents a list of two hyper-parameter names. If the keyword-argument `plot=True` is set, a visualization is created using the `bar3d` function from the mpl_toolkits.mplot3d module.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        import numpy as np
        import bayesloop as bl

        S = bl.HyperStudy()
        S.loadExampleData()

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))

        T = bl.tm.SerialTransitionModel(bl.tm.Static(),
                                        bl.tm.BreakPoint('t_1', 1885),
                                        bl.tm.Deterministic(lambda t, slope=bl.cint(-0.4, 0, 20): slo
                                                            target='accident_rate'),
                                        bl.tm.BreakPoint('t_2', 1895),
                                        bl.tm.GaussianRandomWalk('sigma', bl.cint(0, 0.8, 20),
                                                                 target='accident_rate'))
        S.set(L, T)

        S.fit()
        S.getJointHyperParameterDistribution(['slope', 'sigma'], plot=True, color=[0.1, 0.8, 0.1])

        plt.xlim([-0.5, 0.1])
        plt.ylim([-0.1, 0.9]);
```

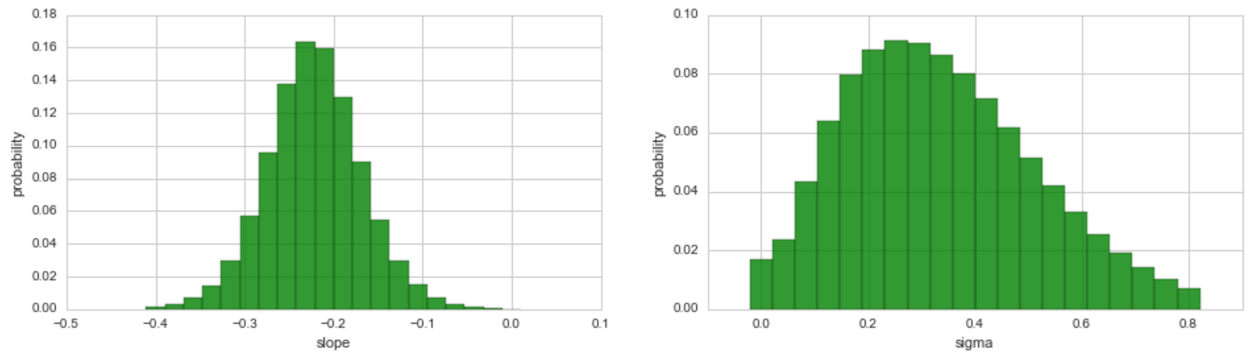
```
+ Created new study.
  --> Hyper-study
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Transition model: Serial transition model. Hyper-Parameter(s): ['slope', 'sigma', 't_1', 't_2']
+ Set hyper-prior(s): ['uniform', 'uniform', 'uniform', 'uniform']
+ Started new fit.
    + 400 analyses to run.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -73.47969
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```



It is important to note here, that the evidence value of $\approx 10^{-73.5}$ is smaller compared to the value of $\approx 10^{-72.7}$ obtained in a previous analysis here. There, we optimized the hyper-parameter values and assumed that these optimal values are not subject to uncertainty, therefore over-estimating the model evidence. In contrast, the hyper-study explicitly considers the uncertainty tied to the hyper-parameter values.

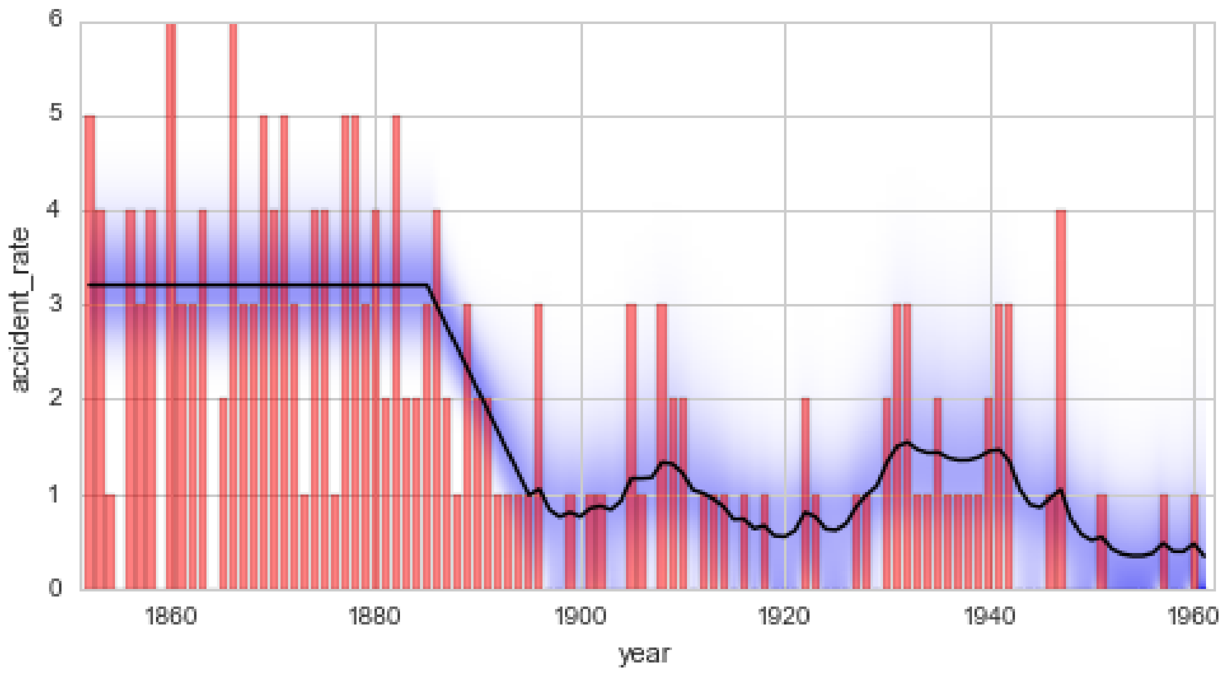
While the joint distribution of two hyper-parameters may uncover possible correlations between the two quantities, the 3D plot is often difficult to integrate into existing figures. To plot the marginal distribution of a single hyper-parameter in a simple 2D histogram/bar plot, use the `plot` method, just as for the parameters of the observation model:

```
In [2]: plt.figure(figsize=(16,4))
        plt.subplot(121)
        S.plot('slope', color='g', alpha=.8);

        plt.subplot(122)
        S.plot('sigma', color='g', alpha=.8);
```



Finally, the temporal evolution of the model parameter may be displayed using, again, the `plot` method:

```
In [3]: plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```



In principle, a `HyperStudy` could also be used to test different time stamps for change- and break-points. However, in a transition model with several change- or break-points, the correct order has to be maintained. Since a `HyperStudy` fits all possible combinations of hyper-parameter values, we need another type of study that takes care of the correct order of change-/break-points, the `ChangepointStudy` class. It is introduced in the next tutorial.

1.2.5 Change-point study

Change point analysis or change detection deals with abrupt changes in statistical properties of time series. *bayesloop* includes two types abrupt changes: an abrupt change in parameter values is modeled by the transition model `Changepoint`. In contrast to this change in value, the transition model itself may change at specific points in time, which we will refer to as *structural breaks*. These structural changes are implemented using the `SerialTransitionModel` class. The following two sections introduce the `ChangepointStudy` class and describe its usage to analyze both change-points and structural breaks in time series data.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        import bayesloop as bl
        import numpy as np
```

Analyzing abrupt changes of parameter values

The `ChangepointStudy` class represents an extention of the `HyperStudy` introduced above and provides an easy-to-use interface to conduct change-point studies. By calling the `fit` method, this type of study first analyzes the defined transition model and detects all instances of the `Changepoint` class. Instead of directly using all combinations of predefined change-points provided by the user, it then computes a list of all valid combinations of change-point times and fits them (basically preventing the double-counting of change-point times due to reversed order). With Bayesian evidence as an objective fitness measure, this type of study can be used to answer the general question of when changes have happened. Furthermore, we may compute a distribution of change-point times to assess the (un-)certainty of these points in time.

Note: For simple change-point analyses with only one change-/break-point or multiple change-/break-points which do not “overlap”, using the `HyperStudy` class is sufficient, too.

Analysis of a single change-point

In a first example, we assume a single change-point in our data set of coal mining disasters. Using the `ChangepointStudy`, we iterate over all possible time steps using the change-point model. After processing all time steps, the probability distribution of the change-point as well as an average model are computed. This study may also be carried out using MCMC methods, see e.g. the PyMC tutorial for comparison.

The change-point study is set up as shown below. Note that we supply the value `'all'` to the change-point hyper-parameter `'tChange'`, indicating that all possible time steps are considered as a change-point. One could of course also provide a list of possible candidate time steps.

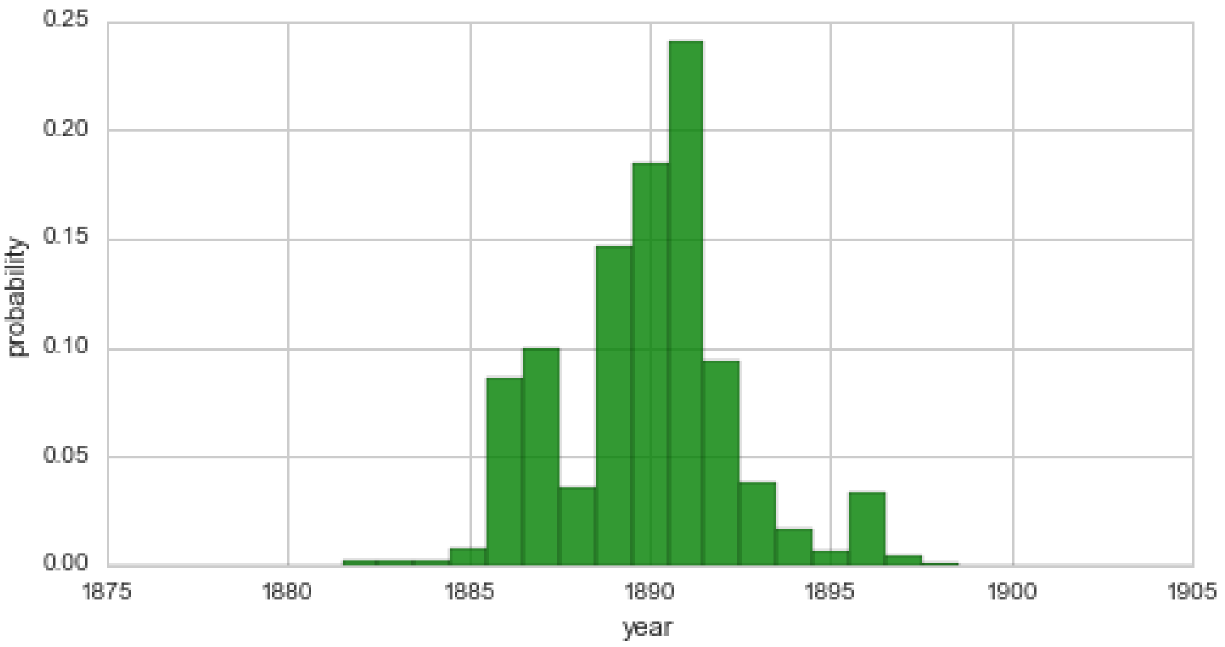
```
In [2]: S = bl.ChangepointStudy()
        S.loadExampleData()

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        T = bl.tm.ChangePoint('change_point', 'all')

        S.set(L, T)
        S.fit()
```

```
+ Created new study.
  --> Hyper-study
  --> Change-point analysis
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Transition model: Change-point. Hyper-Parameter(s): ['change_point']
+ Detected 1 change-point(s) in transition model: ['change_point']
+ Set hyper-prior(s): ['uniform']
+ Started new fit.
    + 109 analyses to run.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -75.71555
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```

After all fits are completed, we can plot the change-point distribution, by using the usual `plot` method. We may further use the `eval` method to draw quantitative conclusions from the change-point distribution:

```
In [3]: plt.figure(figsize=(8, 4))

        S.plot('change_point', color='g', alpha=.8)

        plt.xlim([1875, 1905])
        plt.xlabel('year')


        p1 = S.eval('change_point < 1887')
        p2 = S.eval('change_point > 1893')

        # compute probability of change-point between 1887 and 1893
        print('Probability of change-point between 1887 and 1893: {}'.format(1.-p1-p2))
```

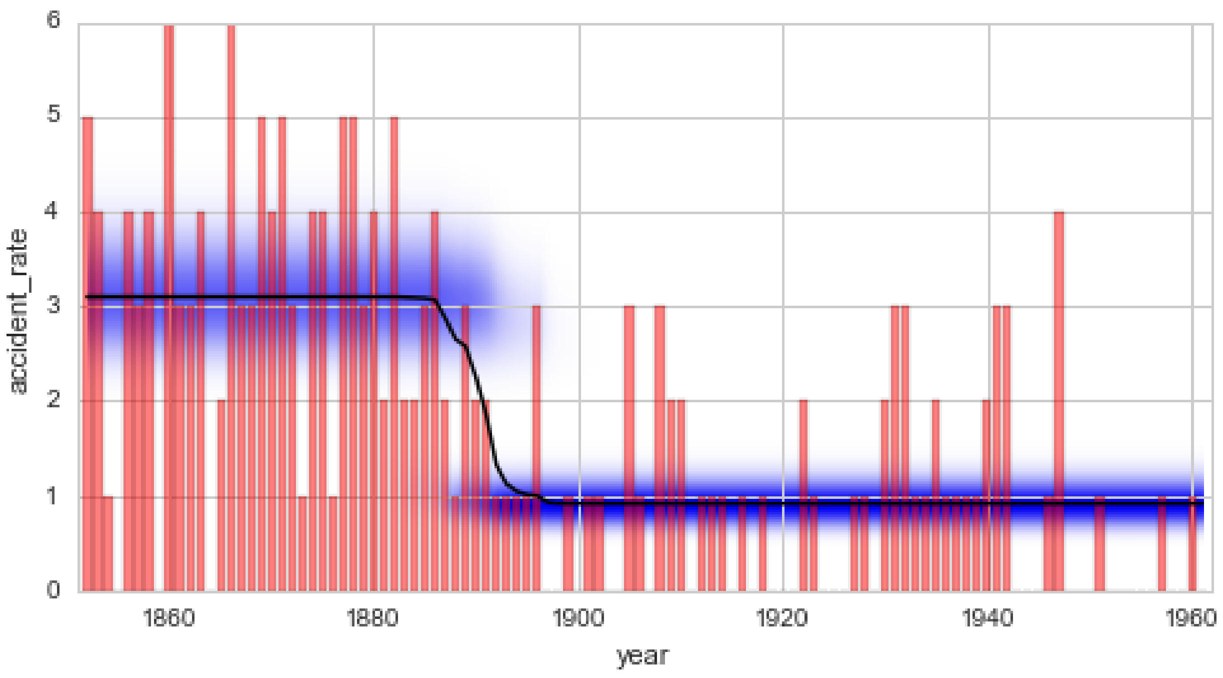
```
P(change_point < 1887) = 0.099250764569
P(change_point > 1893) = 0.0616571951525
Probability of change-point between 1887 and 1893: 0.839092040278
```



From this distribution, we may conclude that a change in safety conditions of coal mines in the UK happened during the seven-year interval from 1887 to 1893 with a probability of $\approx 84\%$.

bayesloop further weighs all fitted models by their probability from the change-point distribution and subsequently adds them up, resulting in an average model, which is stored in `S.posteriorSequence` and `S.posteriorMeanValues`. Additionally, the log-evidence of the average model is set by the weighted sum of all log-evidence values. These results can be plotted as before, using `plot`:

```
In [4]: plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```



Exploring possible change-points

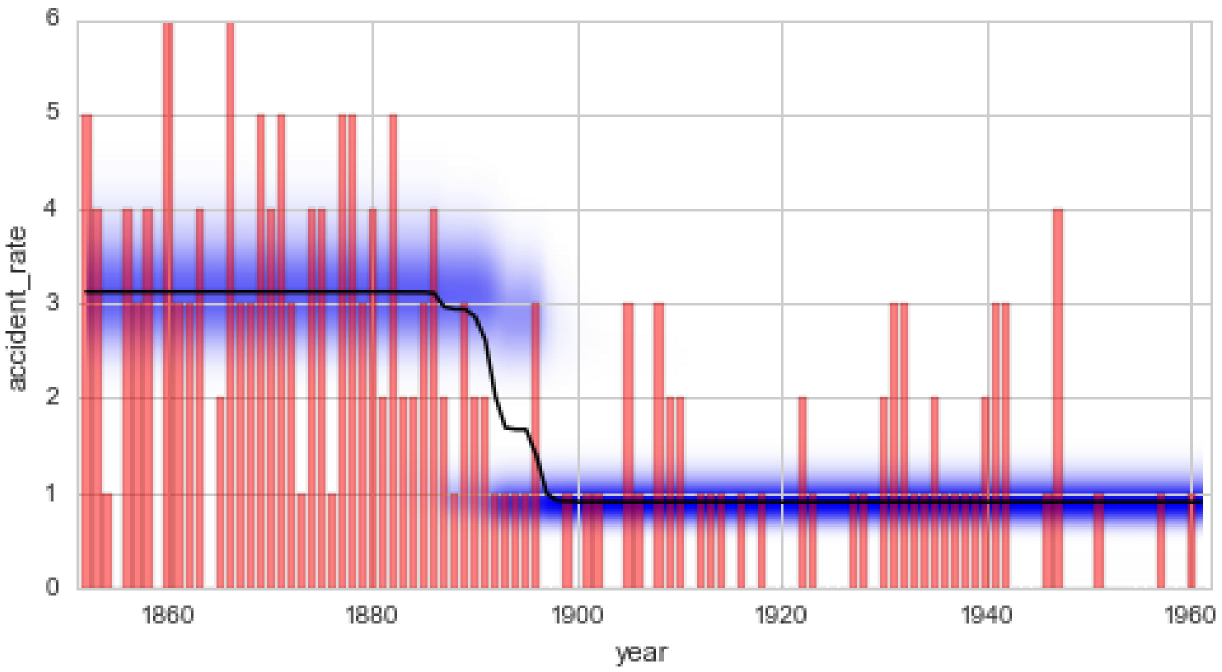
The change-point study described above explicitly assumes the existence of a single change-point in the data set. Without any prior knowledge about the data, however, this assumption can rarely be made with certainty as the number of potential change-points is often unknown.

In order to *explore* possible change-points without prior knowledge, *bayesloop* includes the transition model `RegimeSwitch`, which assigns a minimal probability (specified on a log10-scale by `log10pMin`, relative to the probability value of a flat distribution) to all parameter values on the parameter grid at every time step. This model allows for abrupt parameter changes only, and neglects gradually varying parameter dynamics. Note that no `ChangepointStudy` is needed for this kind of analysis, the *"standard"* `Study` class is sufficient.

For the coal mining example, the results from the regime-switching model with a minimal probability of 10^{-7} resemble the average model of the change-point study:

```
In [5]: S = bl.Study()
        S.loadExampleData()

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        T = bl.tm.RegimeSwitch('log10pMin', -7)

        S.set(L, T)
        S.fit()

        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Created new study.
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Transition model: Regime-switching model. Hyper-Parameter(s): ['log10pMin']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -80.63781

    + Finished backward pass.
    + Computed mean parameter values.
```



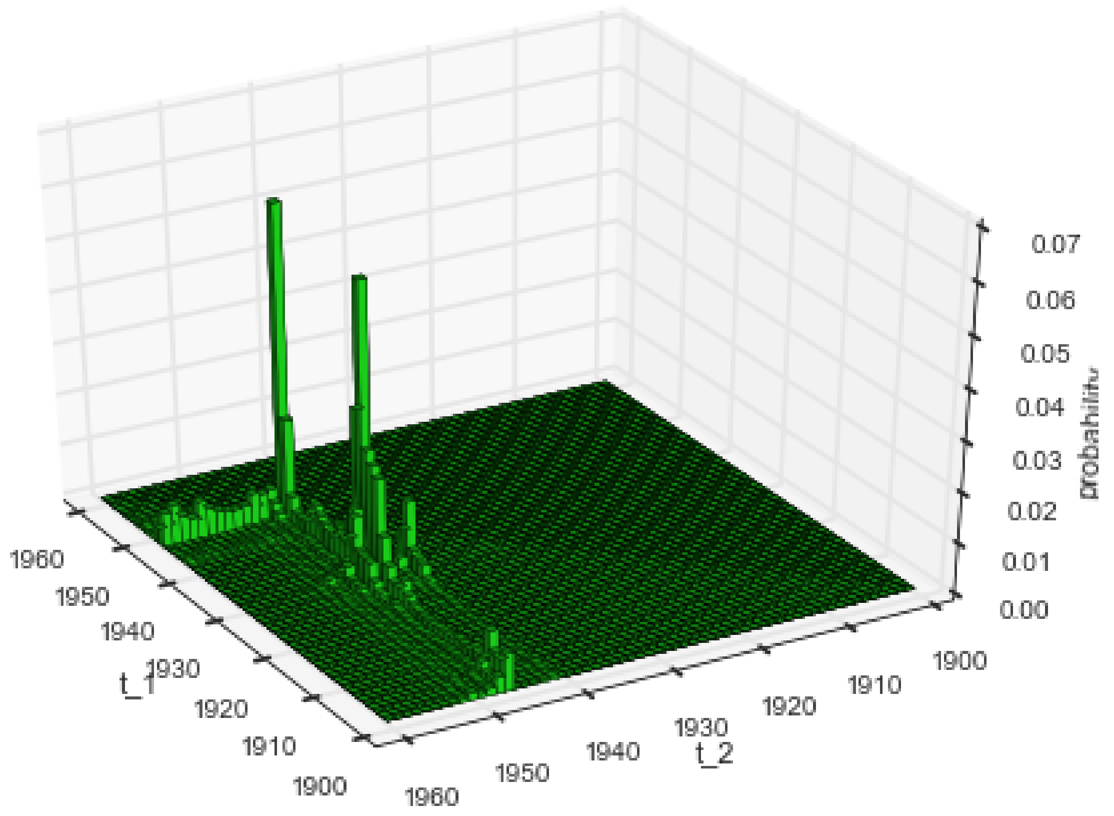
Analysis of multiple change-points

Suppose the regime-switching process introduced above indicates two distinct change-points in a data set. In this case, the `ChangepointStudy` class can be used together with a `CombinedTransitionModel` to perform a comprehensive analysis assuming two change-points. The combined transition model here simply consists of two instances of the `ChangePoint` model. We use the example below to investigate possible change-points in the disaster rate after the significant decrease at the end of the 19th century, i.e. restricting the data set to the time after 1900. Note that the `ChangepointStudy` will only consider combinations of change-points that are in ascending temporal order, i.e. the second change-point must occur after the first one.

After all fits are done, the resulting joint change-point distribution can be plotted using the `getJointHyperParameterDistribution` (`getJHPD`) method (similar to plotting the joint distribution of a hyper-study).

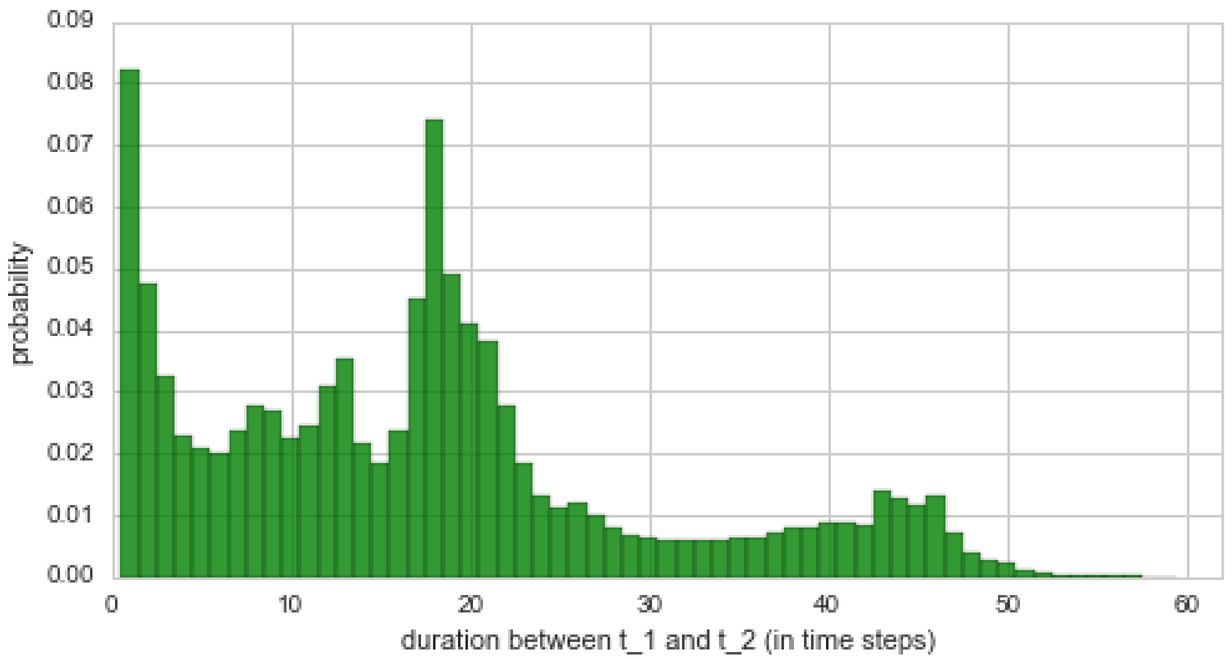
```
In [6]: S = bl.ChangepointStudy()
        S.loadExampleData()

        mask = S.rawTimestamps > 1900 # select timestamps greater than the year 1900
        S.rawTimestamps = S.rawTimestamps[mask]
        S.rawData = S.rawData[mask]

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        T = bl.tm.CombinedTransitionModel(bl.tm.ChangePoint('t_1', 'all'),
                                          bl.tm.ChangePoint('t_2', 'all'))

        S.set(L, T)
        S.fit()

        S.getJHPD(['t_1', 't_2'], plot=True, color=[0.1, 0.8, 0.1])
        plt.xlim([1899, 1962])
        plt.ylim([1899, 1962])

        # set proper view-point
        ax = plt.gca()
        ax.view_init(elev=35, azim=150)
```

```
+ Created new study.
  --> Hyper-study
  --> Change-point analysis
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Transition model: Combined transition model. Hyper-Parameter(s): ['t_1', 't_2']
+ Detected 2 change-point(s) in transition model: ['t_1', 't_2']
+ Set hyper-prior(s): ['uniform', 'uniform']
+ Started new fit.
    + 1770 analyses to run.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -34.85781
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```



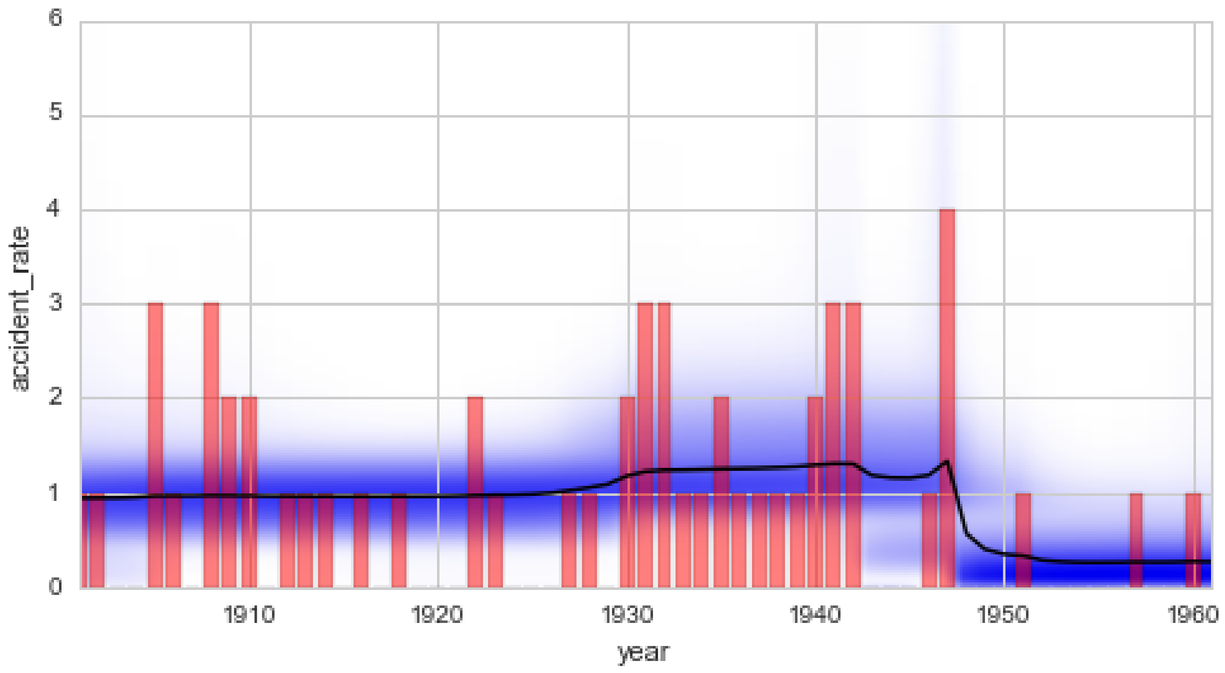
Instead of focusing on the exact time steps of the change-points, some applications may call for the analysis of the time interval between two change-points. The `ChangepointStudy` class provides a method called `getDurationDistribution` which computes the probabilities of different time intervals between two change-points and optionally plots them in a bar graph. Based on the example above, the resulting *"duration distribution"* is shown below:

```
In [7]: plt.figure(figsize=(8, 4))
        S.getDurationDistribution(['t_1', 't_2'], plot=True, color='g', alpha=.8)
        plt.xlim([0, 62]);
```



Finally, we plot the averaged parameter evolution of the two-change-point model. From the duration distribution as well as from the temporal evolution of the inferred disaster rate we may conclude that there is indeed a time period with a slightly increased disaster rate, which begins in ≈ 1930 and ends in ≈ 1945. The duration distribution underlines this finding with high probability values for durations up to 20 years.

```
In [8]: plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('accident_rate')
        plt.xlim([1901, 1961])
        plt.xlabel('year');
```



Analyzing structural breaks in time series models

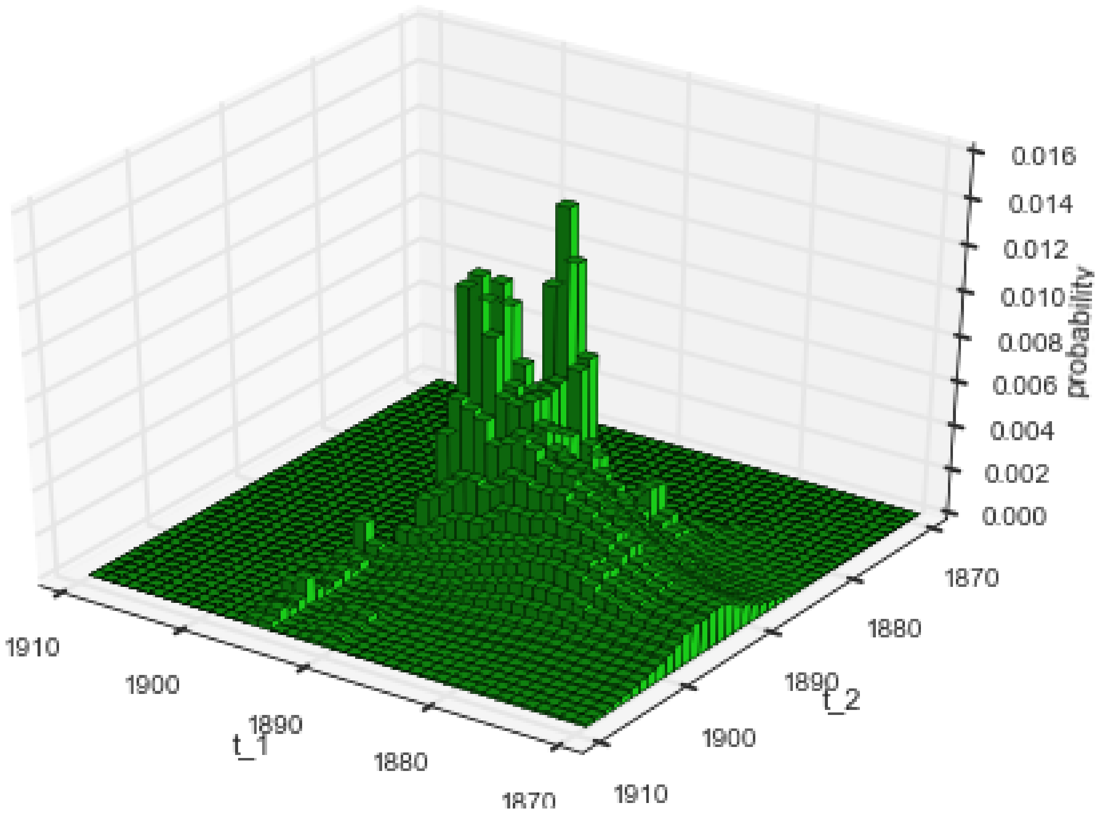
In contrast to a pure change in parameter value, the whole type of parameter dynamics may change at a given point in time. We will use the term *structural break* to describe such events. We have already investigated so-called serial transition models that describe parameter dynamics which change from time to time. While these serial transition models are a re-occurring topic in this tutorial (see here, here and here), the times at which the structural breaks happen have - up to this point - always been user-defined and fixed. This restriction can be lifted by the `ChangepointStudy` class. If a `SerialTransitionModel` is defined within the change-point study, all structural breaks will be treated as variables and the `fit` method will iterate over all possible combinations, just as with *"normal"* change-points.

In this section, our goal is to build a model to determine how long it took for the disaster rate to decrease from ≈ 3 disasters per year to only ≈ 1 per year. This kind of study may generally be applied to assess the effectivity of policies like safety regulations. Here, we devise a simple serial transition model that consists of three phases to describe the change in the annual number of coal mining disasters: in the first and the last phase, we assume a constant disaster rate, while the intermediate phase is modeled by a linear decrease of the disaster rate. By providing multiple values for the slope of the intermediate phase, we can combine the advantages of both hyper- and change-point study in order to consider the uncertainty of the change-points *and* the uncertainty of the slope of the intermediate phase. We restrict the data set to the years from 1870 to 1910, and assume an improvement of the disaster rate by 0 to 2.0 disasters per year.

***Note:** The following analysis consists of ~25000 individual model fits. It may take several minutes to complete.*

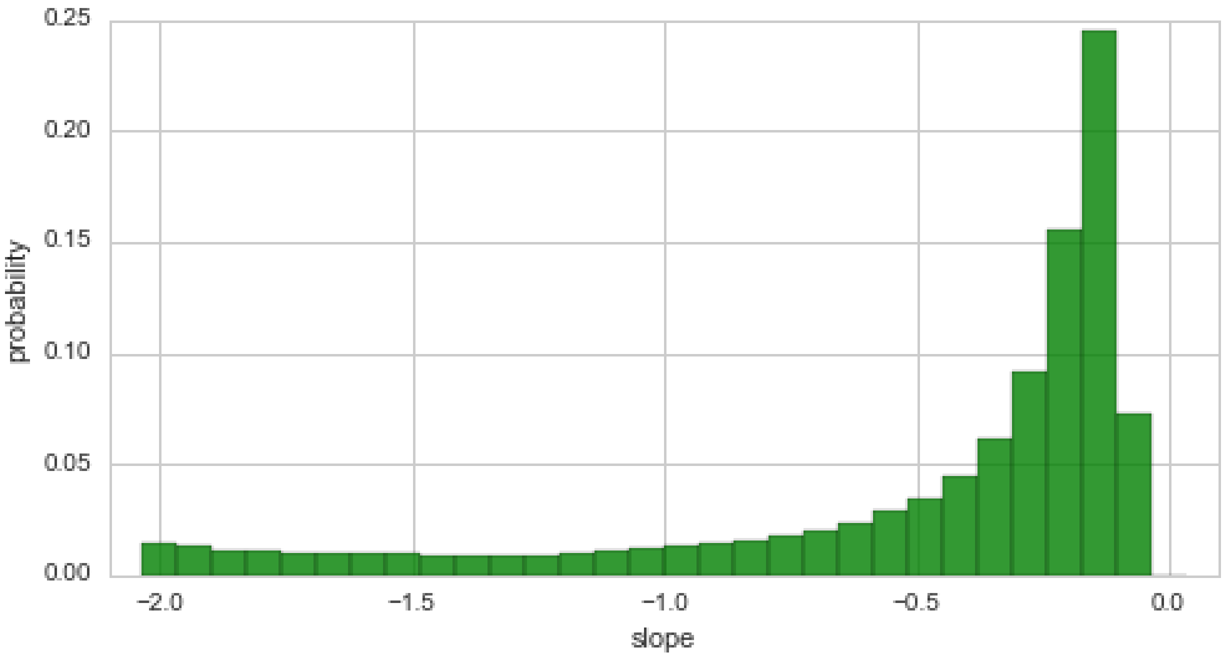
```
In [9]: S.loadExampleData()

        mask = (S.rawTimestamps >= 1870)*(S.rawTimestamps <= 1910) # restrict analysis to 1870-1910
        S.rawTimestamps = S.rawTimestamps[mask]
        S.rawData = S.rawData[mask]

        T = bl.tm.SerialTransitionModel(bl.tm.Static(),
                                        bl.tm.BreakPoint('t_1', 'all'),
                                        bl.tm.Deterministic(lambda t, slope=np.linspace(-2.0, 0.0, 3
                                                            target='accident_rate'),
                                        bl.tm.BreakPoint('t_2', 'all'),
                                        bl.tm.Static()
                                        )

        S.set(T)
        S.fit()

        S.getJHPD(['t_1', 't_2'], plot=True, color=[0.1, 0.8, 0.1])
        plt.xlim([1869, 1911])
        plt.ylim([1869, 1911])

        # set proper view-point
        ax = plt.gca()
        ax.view_init(elev=35, azim=125)
```

```
+ Successfully imported example data.
+ Transition model: Serial transition model. Hyper-Parameter(s): ['slope', 't_1', 't_2']
+ Detected 2 break-point(s) in transition model: ['t_1', 't_2']
+ Set hyper-prior(s): ['uniform', 'uniform', 'uniform']
+ Started new fit.
    + 23400 analyses to run.
    ! WARNING: Posterior distribution contains only zeros, check parameter boundaries!
      Stopping inference process. Setting model evidence to zero.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -30.63948
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```



Apart from this rather non-intuitive joint distribution of the two structural break times, we can also plot the marginal distribution of the inferred slope of the intermediate phase:

```
In [10]: plt.figure(figsize=(8,4))
         S.plot('slope', color='g', alpha=.8)
         plt.xlim([-2.1, 0.1]);
```



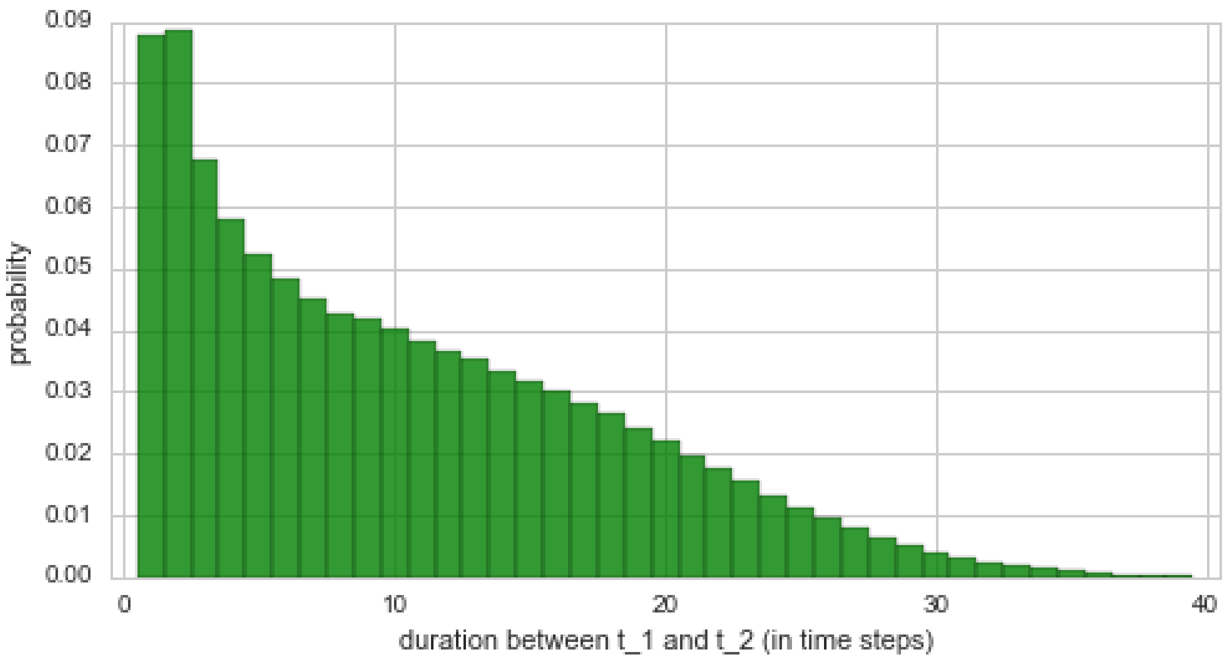
The plot above indicates that the decrease of the disaster rate is indeed a gradual process, as high probabilities are assigned to rather small slopes with an absolute value < 0.5. However, there is still a significant probability of slopes with an absolute value that is larger than 0.5:

```
In [11]: S.eval('abs(slope) > 0.5');
```

```
P(abs(slope) > 0.5) = 0.29265010301
```

More intuitive than the slope is the duration between the two structural breaks. This period of time directly measures the time it takes for the disaster rate to decrease from three to one disaster per year. The plot below shows the distribution for this period of time:

```
In [12]: plt.figure(figsize=(8,4))
         d, p = S.getDurationDistribution(['t_1', 't_2'], plot=True, color='g', alpha=.8)
         plt.xlim([-0.5, 40.5]);
```



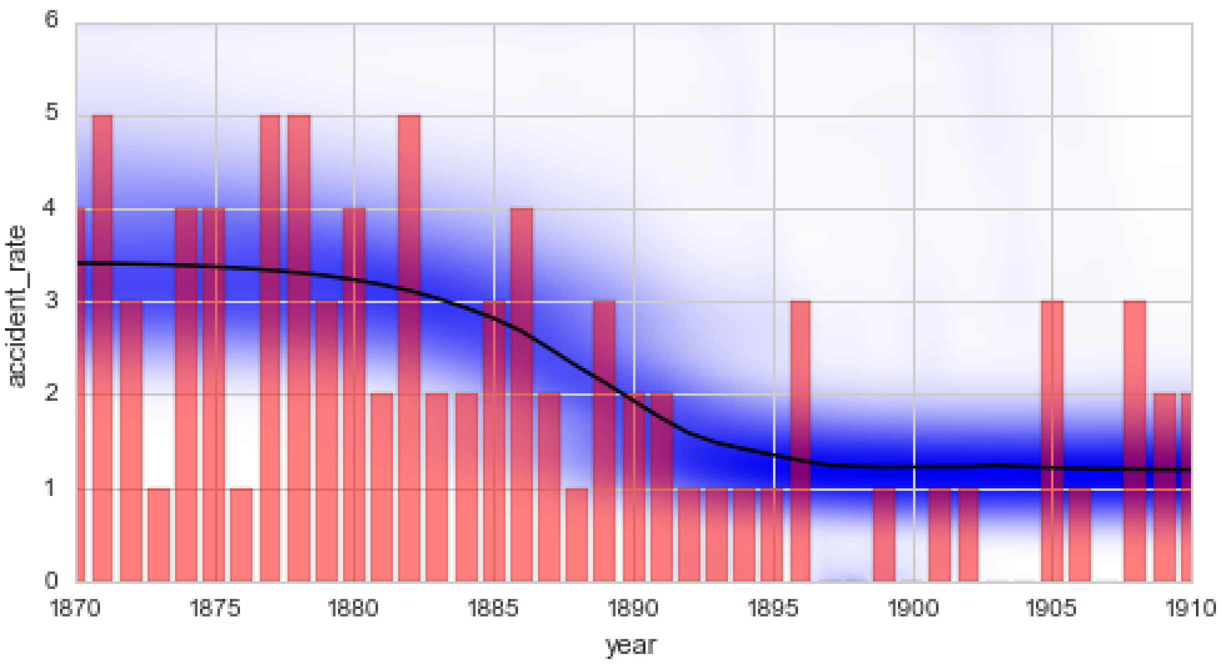
From this plot, we see that the period between the two structural breaks cannot be inferred with great accuracy. The accuracy may be improved by extending the simple serial model used in this example or by incorporating more data points (only 40 data points have been used here). We may still conclude that this analysis indicates an intermediate phase of improvement that is shorter than 15 years, with a probability of $\approx 70\%$:

```
In [13]: np.sum(p[np.abs(d) < 15])
```

```
Out[13]: 0.71545710930129347
```

The results from the break-point analysis are further illustrated by the temporal evolution of the inferred disaster rate. Below, the average model of the complete analysis is used to display the inferred changes in the disaster rate:

```
In [14]: plt.figure(figsize=(8, 4))
         plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
         S.plot('accident_rate')
         plt.xlim([1870, 1910])
         plt.xlabel('year');
```



Finally, we want to stress the difference between change-points and break-points again: If the `BreakPoint` transition model is used within the `SerialTransitionModel` class, different transition models will be specified before and after the break-point, but the parameter values will not be *reset* at the break-point. In contrast, the `ChangePoint` transition model can also be used in a `SerialTransitionModel` (not shown in this tutorial). In addition to the different parameter dynamics before and after the change-point, this case allows for an abrupt change of the parameter values. Finally, if the `ChangePoint` model is used *without* a `SerialTransitionModel`, the parameter dynamics will stay the same before and after the change-point, but an abrupt change of the parameter values is expected.

1.2.6 Online study

All study types of *bayesloop* introduced so far are used for retrospective data analysis, i.e. the complete data set is already available at the time of the analysis. Many applications, however, from algorithmic trading to the monitoring of heart function or blood sugar levels call for on-line analysis methods that can take into account new information as it arrives from external sources. For this purpose, *bayesloop* provides the class `OnlineStudy`, which enables the inference of time-varying parameters in a sequential fashion, much like a particle filter. In contrast to particle filters, the `OnlineStudy` can account for different *scenarios* of parameter dynamics (i.e. different transition models) and can apply on-line model selection to objectively determine which scenario is more likely to describe the current data point, or all past data points.

In this case, we avoid constructing some artificial usage example and directly point the reader at this case study on stock market fluctuations. In this detailed example, we investigate the intra-day price fluctuations of the exchange-traded fund SPY. Based on two different transition models, one for *normal* market function and a second one for *chaotic* market fluctuations, we identify price corrections that are induced by news announcements of economic indicators.

1.2.7 Prior distributions

One important aspect of Bayesian inference has not yet been discussed in this tutorial: prior distributions. In Bayesian statistics, one has to provide probability (density) values for every possible parameter value *before* taking into account the data at hand. This prior distribution thus reflects all *prior* knowledge of the system that is to be investigated. In the case that no prior knowledge is available, a *non-informative* prior in the form of the so-called Jeffreys prior allows to minimize the effect of the prior on the results. The next two sub-sections discuss how one can set custom prior distributions for the parameters of the observation model and for hyper-parameters in a hyper-study or change-point study.

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        import numpy as np
        import bayesloop as bl

        # prepare study for coal mining data
        S = bl.Study()
        S.loadExampleData()
```

```
+ Created new study.
+ Successfully imported example data.
```

Parameter prior

bayesloop employs a forward-backward algorithm that is based on Hidden Markov models. This inference algorithm iteratively produces a parameter distribution for each time step, but it has to start these iterations from a specified probability distribution - the parameter prior. All built-in observation models already have a predefined prior, stored in the attribute `prior`. Here, the prior distribution is stored as a Python function that takes as many arguments as there are parameters in the observation model. The prior distributions can be looked up directly within `observationModels.py`. For the `Poisson` model discussed in this tutorial, the default prior distribution is defined in a method called `jeffreys` as

```
def jeffreys(x):
    return np.sqrt(1. / x)
```

corresponding to the non-informative Jeffreys prior, $p(\lambda) \propto 1/\sqrt{\lambda}$. This type of prior can also be determined automatically for arbitrary user-defined observation models, see here.

Prior functions and arrays

To change the predefined prior of a given observation model, one can add the keyword argument `prior` when defining an observation model. There are different ways of defining a parameter prior in *bayesloop*: If `prior=None` is set, *bayesloop* will assign equal probability to all parameter values, resulting in a uniform prior distribution within the specified parameter boundaries. One can also directly supply a Numpy array with prior probability (density) values. The shape of the array must match the shape of the parameter grid! Another way to define a custom prior is to provide a function that takes exactly as many arguments as there are parameters in the defined observation model. *bayesloop* will then evaluate the function for all parameter values and assign the corresponding probability values.

Note: In all of the cases described above, *bayesloop* will re-normalize the provided prior values, so they do not need to be passed in a normalized form. Below, we describe the possibility of using probability distributions from the SymPy stats module as prior distributions, which are not re-normalized by *bayesloop*.

Next, we illustrate the difference between the Jeffreys prior and a flat, uniform prior with a very simple inference example: We fit the coal mining example data set using the `Poisson` observation model and further assume the rate parameter to be static:

```
In [2]: # we assume a static rate parameter for simplicity
        S.set(bl.tm.Static())

        print 'Fit with built-in Jeffreys prior:'
        S.set(bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000)))
        S.fit()
        jeffreys_mean = S.getParameterMeanValues('accident_rate')[0]
        print('-----\n')

        print 'Fit with custom flat prior:'
        S.set(bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000),
                                             prior=lambda x: 1.))
        # alternatives: prior=None, prior=np.ones(1000)
        S.fit()
        flat_mean = S.getParameterMeanValues('accident_rate')[0]
```

```
+ Transition model: Static/constant parameter values. Hyper-Parameter(s): []
Fit with built-in Jeffreys prior:
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -88.00564

    + Finished backward pass.
    + Computed mean parameter values.
-----

Fit with custom flat prior:
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Started new fit:
    + Formatted data.
    + Set prior (function): <lambda>. Values have been re-normalized.

    + Finished forward pass.
    + Log10-evidence: -87.98915

    + Finished backward pass.
    + Computed mean parameter values.
```

First note that the model evidence indeed slightly changes due to the different choices of the parameter prior. Second, one may notice that the posterior mean value of the flat-prior-fit does not exactly match the arithmetic mean of the data. This small deviation shows that a flat/uniform prior is not completely non-informative for a Poisson model! The fit using the Jeffreys prior, however, succeeds in reproducing the *frequentist* estimate, i.e. the arithmetic mean:

```
In [3]: print('arithmetic mean     = {}'.format(np.mean(S.rawData)))
        print('flat-prior mean     = {}'.format(flat_mean))
        print('Jeffreys prior mean = {}'.format(jeffreys_mean))
```

```
arithmetic mean     = 1.69090909091
flat-prior mean     = 1.7
Jeffreys prior mean = 1.69090909091
```

SymPy prior

The second option is based on the SymPy module that introduces symbolic mathematics to Python. Its sub-module sympy.stats covers a wide range of discrete and continuous random variables. The keyword argument `prior` also accepts a list of `sympy.stats` random variables, one for each parameter (if there is only one parameter, the list can be omitted). The multiplicative joint probability density of these random variables is then used as the prior distribution. The following example defines an exponential prior for the `Poisson` model, favoring small values of the rate parameter:

```
In [4]: import sympy.stats
        S.set(bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000),
                                             prior=sympy.stats.Exponential('expon', 1)))
        S.fit()
```

```
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Started new fit:
    + Formatted data.
    + Set prior (sympy): exp(-x)

    + Finished forward pass.
    + Log10-evidence: -87.94640

    + Finished backward pass.
    + Computed mean parameter values.
```

Note that one needs to assign a name to each `sympy.stats` variable. In this case, the output of *bayesloop* shows the mathematical formula that defines the prior. This is possible because of the symbolic representation of the prior by `SymPy`.

Note: The support interval of a prior distribution defined via SymPy can deviate from the parameter interval specified in *bayesloop*. In the example above, we specified the parameter interval]0, 6[, while the exponential prior has the support]0, ∞[. SymPy priors are not re-normalized with respect to the specified parameter interval. Be aware that the resulting model evidence value will only be correct if no parameter values outside of the parameter boundaries gain significant probability values. In most cases, one can simply check whether the parameter distribution has sufficiently *fallen off* at the parameter boundaries.

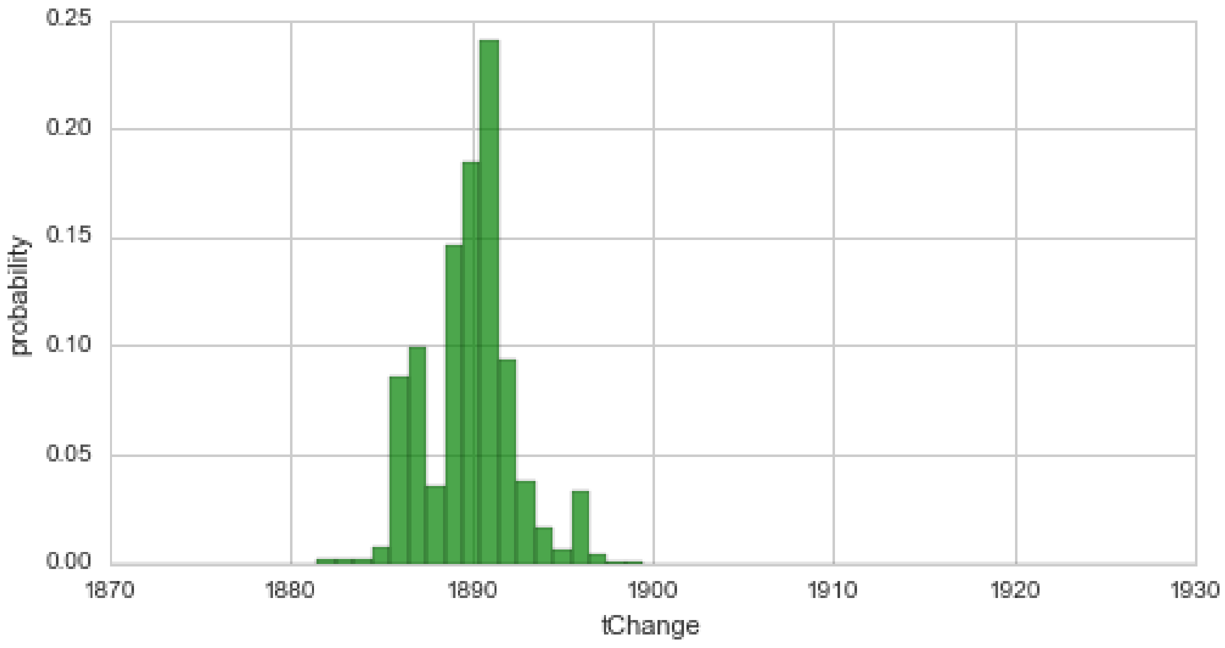
Hyper-parameter priors

As shown before, hyper-studies and change-point studies can be used to determine the full distribution of hyper-parameters (the parameters of the transition model). As for the time-varying parameters of the observation model, one might have prior knowledge about the values of certain hyper-parameters that can be included into the study to refine the resulting distribution of these hyper-parameters. Hyper-parameter priors can be defined just as regular priors, either by an arbitrary function or by a list of `sympy.stats` random variables.

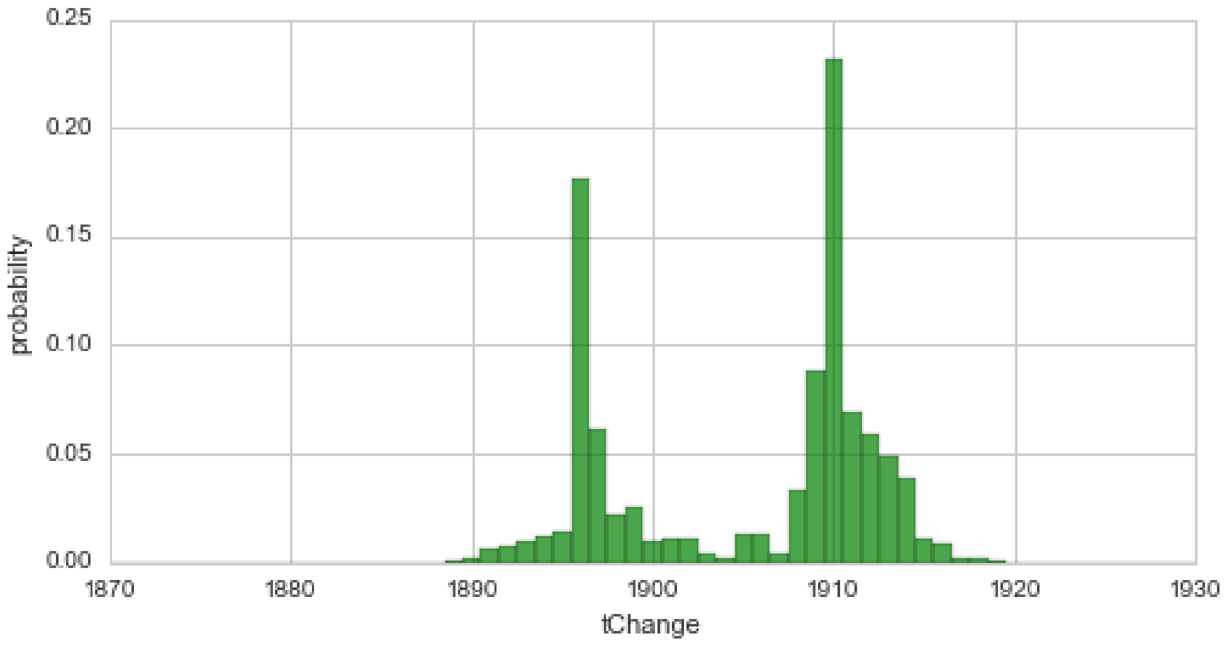
In a first example, we return to the simple change-point model of the coal-mining data set and perform to fits of the change-point: first, we specify no hyper-prior for the time step of our change-point, assuming equal probability for each year in our data set. Second, we define a Normal distribution around the year 1920 with a (rather unrealistic) standard deviation of 5 years as the hyper-prior using a SymPy random variable. For both fits, we plot the change-point distribution to show the differences induced by the different priors:

```
In [5]: print 'Fit with flat hyper-prior:'
        S = bl.ChangepointStudy()
        S.loadExampleData()

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        T = bl.tm.ChangePoint('tChange', 'all')

        S.set(L, T)
        S.fit()

        plt.figure(figsize=(8,4))
        S.plot('tChange', facecolor='g', alpha=0.7)
        plt.xlim([1870, 1930])
        plt.show()
        print('-----\n')

        print 'Fit with custom normal prior:'
        T = bl.tm.ChangePoint('tChange', 'all', prior=sympy.stats.Normal('norm', 1920, 5))
        S.set(T)
        S.fit()

        plt.figure(figsize=(8,4))
        S.plot('tChange', facecolor='g', alpha=0.7)
        plt.xlim([1870, 1930]);
```

```
Fit with flat hyper-prior:
+ Created new study.
  --> Hyper-study
  --> Change-point analysis
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Transition model: Change-point. Hyper-Parameter(s): ['tChange']
+ Detected 1 change-point(s) in transition model: ['tChange']
+ Set hyper-prior(s): ['uniform']
+ Started new fit.
    + 109 analyses to run.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -75.71555
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```



```
-----

Fit with custom normal prior:
+ Transition model: Change-point. Hyper-Parameter(s): ['tChange']
+ Detected 1 change-point(s) in transition model: ['tChange']
+ Set hyper-prior(s): ['sqrt(2)*exp(-(x - 1920)**2/50)/(10*sqrt(pi))']
+ Started new fit.
    + 109 analyses to run.
    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -80.50692
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```



Since we used a quite narrow prior (containing a lot of information) in the second case, the resulting distribution is strongly shifted towards the prior. The following example revisits the two break-point-model from here and a linear decrease with a varying slope as a hyper-parameter. Here, we define a Gaussian prior for the slope hyper-parameter, which is centered around the value -0.2 with a standard deviation of 0.4, via a lambda-function. For simplification, we set the break-points to fixed years.

```
In [6]: S = bl.HyperStudy()
        S.loadExampleData()

        L = bl.om.Poisson('accident_rate', bl.oint(0, 6, 1000))
        T = bl.tm.SerialTransitionModel(bl.tm.Static(),
                                        bl.tm.BreakPoint('t_1', 1880),
                                        bl.tm.Deterministic(lambda t, slope=np.linspace(-2.0, 0.0, 3
                                                            target='accident_rate',
                                                            prior=lambda slope: np.exp(-0.5*((slope -
                                        bl.tm.BreakPoint('t_2', 1900),
                                        bl.tm.Static()
                                        )

        S.set(L, T)
        S.fit()
```

```
+ Created new study.
  --> Hyper-study
+ Successfully imported example data.
+ Observation model: Poisson. Parameter(s): ['accident_rate']
+ Transition model: Serial transition model. Hyper-Parameter(s): ['slope', 't_1', 't_2']
+ Set hyper-prior(s): ['<lambda> (re-normalized)', 'uniform', 'uniform']
+ Started new fit.
```

```
    + 30 analyses to run.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -74.84129
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```

Finally, note that you can mix SymPy- and function-based hyper-priors for nested transition models.

1.2.8 Custom observation models

While *bayesloop* provides a number of observation models like `Poisson` or `AR1`, many applications call for different distributions, possibly with some parameters set to fixed values (e.g. with a mean value set to zero). The sympy.stats and the scipy.stats modules include a large number of continuous as well as discrete probability distributions. The observation model classes `SciPy` and `SymPy` allow to create observation models to be used in *bayesloop* studies on-the-fly, just by passing the desired `scipy.stats` distribution (and setting values for fixed parameters, if necessary), or by providing a `sympy.stats` random variable, respectively. Note that these classes can only be used to model statistically independent observations.

In cases where neither `scipy.stats` nor `sympy.stats` provide the needed model, one can further define a custom observation model by stating a likelihood function in terms of arbitrary NumPy functions, using the `NumPy` class.

Sympy.stats random variables

The SymPy module introduces symbolic mathematics to Python. Its sub-module sympy.stats covers a wide range of discrete and continuous random variables. In the following, we re-define the observation model of the coal mining study `S` defined above, but this time use the `sympy.stats` version of the Poisson distribution:

```
In [1]: import bayesloop as bl
        import numpy as np
        import sympy.stats
        from sympy import Symbol

        rate = Symbol('lambda', positive=True)
        poisson = sympy.stats.Poisson('poisson', rate)

        L = bl.om.SymPy(poisson, 'lambda', bl.oint(0, 6, 1000))
```

```
    + Trying to determine Jeffreys prior. This might take a moment...
    + Successfully determined Jeffreys prior: 1/sqrt(lambda). Will use corresponding lambda function
```

First, we specify the only parameter of the Poisson distribution (denoted λ) symbolically as a positive real number. Note that providing the keyword argument `positive=True` is important for SymPy to define the Poisson distribution correctly (not setting the keyword argument correctly results in a error). Having defined the parameter, a random variable based on the Poisson distribution is defined. This random variable is then passed to the `SymPy` class of the *bayesloop* observation models. Just as for the built-in observation models of *bayesloop*, one has to specify the parameter names and values (in this case, `lambda` is the only parameter).

Note that upon creating an instance of the observation model, *bayesloop* automatically determines the correct Jeffreys prior for the Poisson model:

$$p(\lambda) \propto 1/\sqrt{\lambda}$$

This calculation is done symbolically and therefore represents an important advantage of using the `SymPy` module within *bayesloop*. This behavior can be turned off using the keyword argument `determineJeffreysPrior`, in case one wants to use a flat parameter prior instead or in the case that the automatic determination of the prior takes too long:

```
M = bl.om.SymPy(poisson, 'lambda', bl.oint(0, 6, 1000), determineJeffreysPrior=False)
```

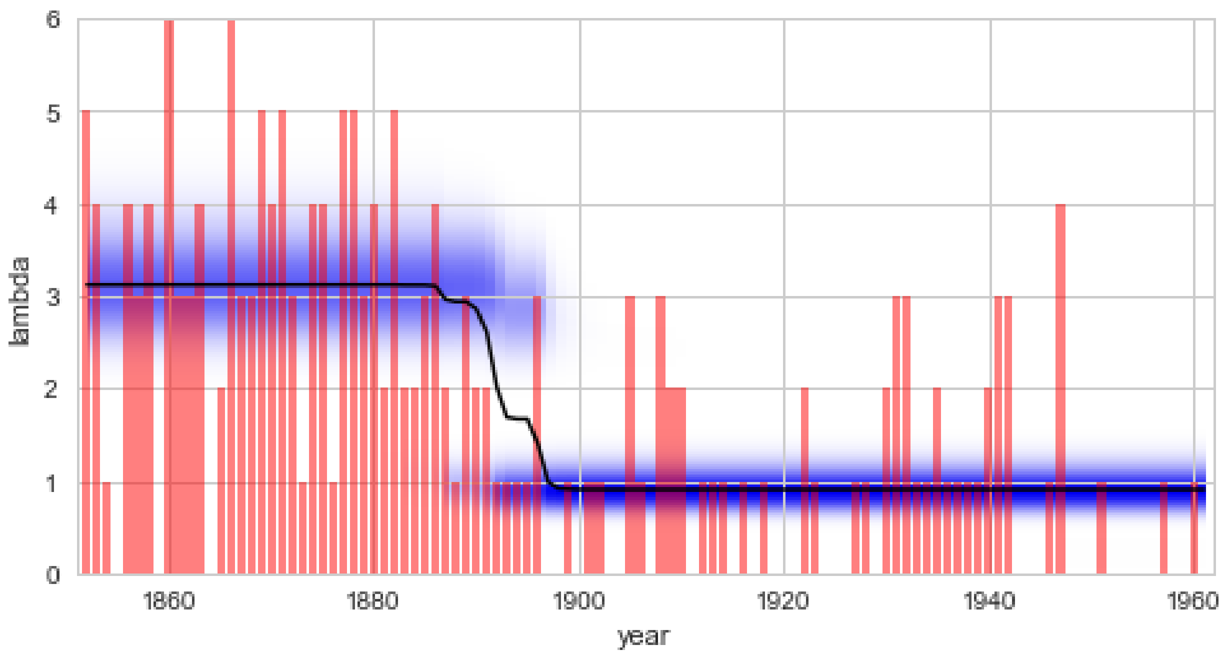
Alternatively, you can of course provide a custom prior via the keyword argument `prior`. This will switch off the automatic determination of the Jeffreys prior as well:

```
M = bl.om.SymPy(poisson, 'lambda', bl.oint(0, 6, 1000), prior=lambda x: 1/x)
```

See also this tutorial for further information on prior distributions. Having defined the observation model, it can be used for any type of study introduced above. Here, we reproduce the result of the regime-switching example we discussed before. We find that the parameter distributions as well as the model evidence is identical - as expected:

```
In [2]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling

        S = bl.Study()
        S.loadExampleData()

        T = bl.tm.RegimeSwitch('log10pMin', -7)

        S.set(L, T)
        S.fit()

        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('lambda')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Created new study.
+ Successfully imported example data.
+ Observation model: poisson. Parameter(s): ('lambda',)
+ Transition model: Regime-switching model. Hyper-Parameter(s): ['log10pMin']
+ Started new fit:
    + Formatted data.
    + Set prior (function): <lambda>. Values have been re-normalized.


    + Finished forward pass.
    + Log10-evidence: -80.63781


    + Finished backward pass.
    + Computed mean parameter values.
```



Finally, it is important to note that the `SymPy` module can also be used to create random variables for which some parameters have user-defined fixed values. The following example creates a normally distributed random variable with a fixed mean value $\mu = 4$, leaving only the standard deviation as a free parameter of the resulting observation model (which is assigned the parameter interval]0, 3[):

```
mu = 4
std = Symbol('stdev', positive=True)

normal = sympy.stats.Normal('normal', mu, std)
L = bl.om.SymPy(normal, 'stdev', bl.oint(0, 3, 1000))
```

Scipy.stats probability distributions

We continue by describing the use of probability distributions of the `scipy.stats` module. Before we show some usage examples, it is important to note here that `scipy.stats` does not use the canonical parameter names for probability distributions. Instead, all continuous distributions have two parameters denoted `loc` (for shifting the distribution) and `scale` (for scaling the distribution). Discrete distributions only support `loc`. While some distributions may have additional parameters, `loc` and `scale` often take the role of known parameters, like *mean* and *standard deviation* in case of the normal distribution. In `scipy.stats`, you do not have to set `loc` or `scale`, as they have default values `loc=0` and `scale=1`. In *bayesloop*, however, you will have to provide values for these parameters, if you want either of them to be fixed and not treated as a variable.

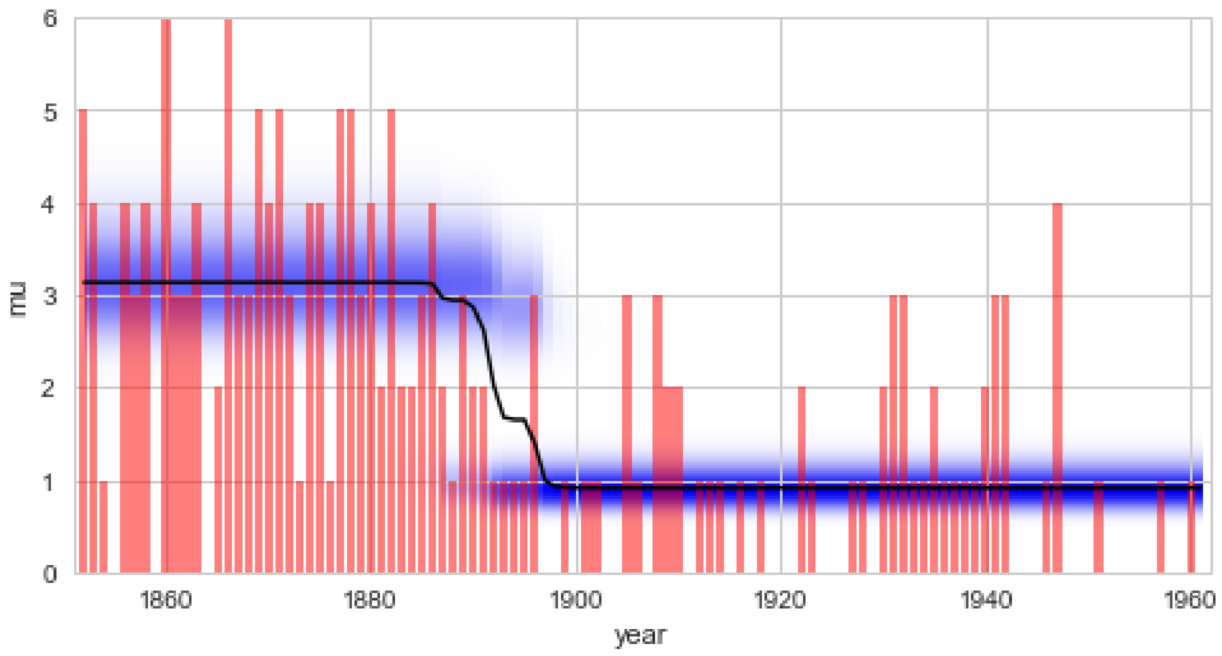
As a first example, we re-define the observation model of the coal mining study `S` defined above, but this time use the `scipy.stats` version of the Poisson distribution. First, we check the parameter names:

```
In [3]: import scipy.stats

        scipy.stats.poisson.shapes

Out[3]: 'mu'
```

In `scipy.stats`, the rate of events in one time interval of the Poisson distribution is called *mu*. Additionally, as a discrete distribution, `stats.poisson` has an additional parameter `loc` (which is **not** shown by `.shapes` attribute!). As we do not want to shift the distribution, we have to set this parameter to zero in *bayesloop* by passing a dictionary for fixed parameters when initializing the class instance. As for the SymPy model, we have to pass the names and values of all free parameters of the model (here only mu):

```
In [4]: L = bl.om.SciPy(scipy.stats.poisson, 'mu', bl.oint(0, 6, 1000), fixedParameters={'loc': 0})
        S.set(L)
        S.fit()

        plt.figure(figsize=(8, 4))
        plt.bar(S.rawTimestamps, S.rawData, align='center', facecolor='r', alpha=.5)
        S.plot('mu')
        plt.xlim([1851, 1962])
        plt.xlabel('year');
```

```
+ Observation model: poisson. Parameter(s): ('mu',)
+ Started new fit:
    + Formatted data.
    + Set uniform prior with parameter boundaries.


    + Finished forward pass.
    + Log10-evidence: -80.49098


    + Finished backward pass.
    + Computed mean parameter values.
```



Comparing this result with the regime-switching example, we find that the model evidence value obtained using the scipy.stats implementation of the Poisson distribution is *different* from the value obtained using the built-in implementation or the sympy.stats version. The deviation is explained by a different prior distribution for the parameter λ. While both the built-in version and the sympy.stats version use the Jeffreys prior of the Poisson model, the scipy.stats implementation uses a flat prior instead. Since the scipy.stats module does not provide symbolic representations of probability distributions, *bayesloop* cannot determine the correct Jeffreys prior in this case. Custom priors are still possible, using the keyword argument prior.

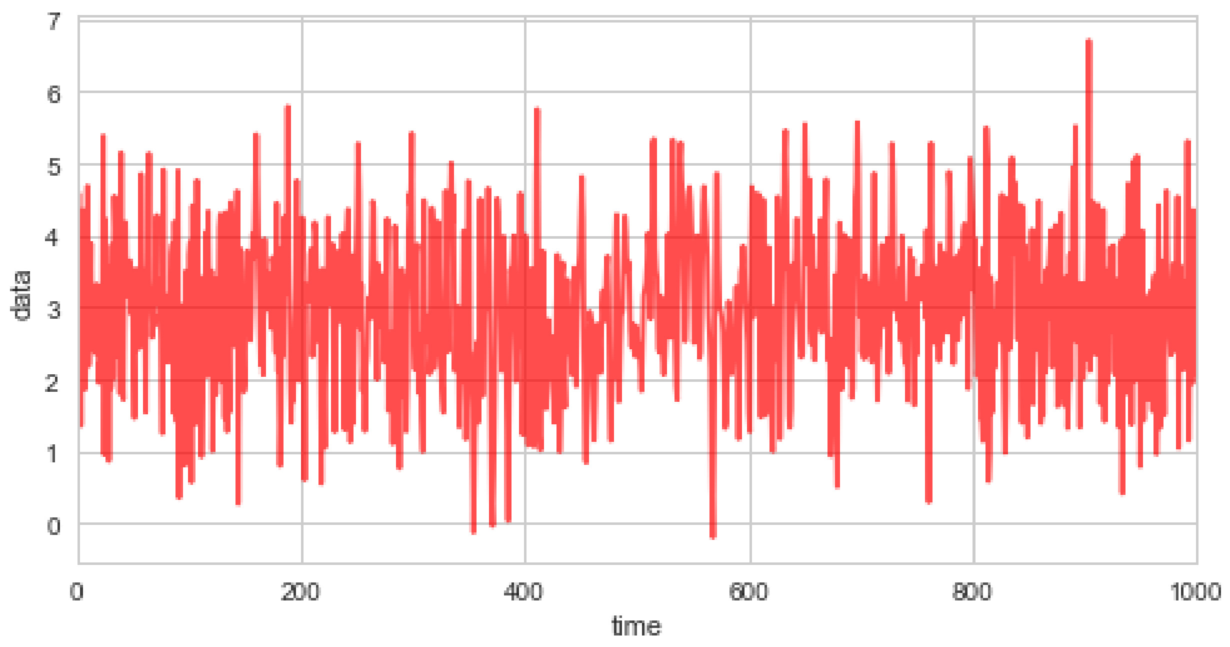
NumPy likelihood functions

In some cases, the data at hand cannot be described by a common statistical distribution contained in either `scipy.stats` or `sympy.stats`. In the following example, we assume normally distributed data points with known standard deviation σ, but unknown mean μ. Additionally, we suspect that the data points may be serially correlated and that the correlation coefficient ρ possibly changes over time. For this multivariate problem with the known standard deviation as "extra" data points, we need more flexibility than either the `SymPy` or the `SciPy` class of `bayesloop` can offer. Instead, we may define the likelihood function of the observation model directly, with the help of NumPy functions.

First, we simulate 1000 random variates with $\mu = 3$, $\sigma = 1$, and a linearly varying correlation coefficient ρ:

```
In [5]: n = 1000

        # parameters
        mean = 3
        sigma = 1
        rho = np.concatenate([np.linspace(-0.5, 0.9, 500), np.linspace(0.9, -0.5, 499)])

        # covariance matrix
        cov = np.diag(np.ones(n)*sigma**2.) + np.diag(np.ones(n-1)*rho*sigma**2., 1) + np.diag(np.one

        # random variates
        np.random.seed(123456)
        obs_data = np.random.multivariate_normal([mean]*n, cov)

        plt.figure(figsize=(8, 4))
        plt.plot(obs_data, c='r', alpha=0.7, lw=2)
        plt.xlim([0, 1000])
        plt.xlabel('time')
        plt.ylabel('data');
```

```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:13: RuntimeWarning: covariance is not positive-s
  del sys.path[0]
```



Before we create an observation model to be used by `bayesloop`, we define a pure Python function that takes a segment of data as the first argument, and NumPy arrays with parameter grids as further arguments. Here, one data segment includes two subsequent data points `x1` and `x2`, and their known standard deviations `s1` and `s2`. The likelihood function we evaluate states the probability of observing the current data point `x2`, given the previous data point `x1`, the known standard deviations `s2`, `s1` and the parameters μ and ρ:

$$P(x_2 \mid x_1, s_2, s_1, \mu, \rho) = \frac{P(x_2, x_1 \mid s_2, s_1, \mu, \rho)}{P(x_1 \mid s_1, \mu)} ,$$

where $P(x_2, x_1 \mid s_2, s_1, \mu, \rho)$ denotes the bivariate normal distribution, and $P(x_1 \mid s_1, \mu)$ is the marginal, univariate normal distribution of x_1. The resulting distribution is expressed as a Python function below. Note that all mathematical functions use NumPy functions, as the function needs to work with arrays as input arguments for the parameters:

```
In [6]: def likelihood(data, mu, rho):
            x2, x1, s2, s1 = data

            exponent = -(((x1-mu)*rho/s1)**2. - (2*rho*(x1-mu)*(x2-mu))/(s1*s2) + ((x2-mu)/s2)**2.) /
            norm = np.sqrt(2*np.pi)*s2*np.sqrt(1-rho**2.)

            like = np.exp(exponent)/norm
            return like
```

As `bayesloop` still needs to know about the parameter boundaries and discrete values of the parameters μ and ρ, we need to create an observation model from the custom likelihood function defined above. This can be done with the `NumPy` class:

```
In [7]: L = bl.om.NumPy(likelihood, 'mu', bl.cint(0, 6, 100), 'rho', bl.oint(-1, 1, 100))
```

Before we can load the data into a `Study` instance, we have to format data segments in the order defined by the likelihood function:

```
[[x1, x0, s1, s0],
 [x2, x1, s2, s1],
 [x3, x2, s3, s2],
 ...]
```

Note that in this case, the standard deviation $\sigma = 1$ for all time steps.

```
In [8]: data_segments = input_data = np.array([obs_data[1:], obs_data[:-1], [sigma]*(n-1), [sigma]*(r
```

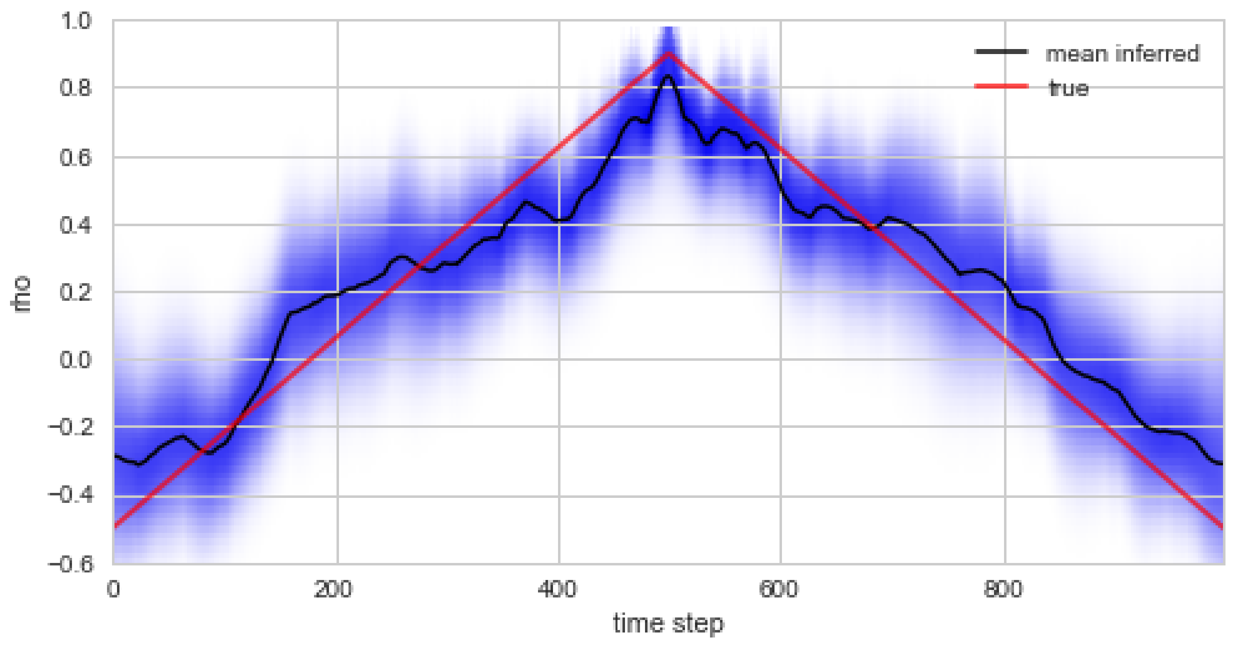
Finally, we create a new `Study` instance, load the formatted data, set the custom observation model, set a suitable transition model, and fit the model parameters:

```
In [9]: S = bl.Study()
        S.loadData(data_segments)
        S.set(L)

        T = bl.tm.GaussianRandomWalk('d_rho', 0.03, target='rho')
        S.set(T)

        S.fit()
```

```
+ Created new study.
+ Successfully imported array.
+ Observation model: likelihood. Parameter(s): ('mu', 'rho')
+ Transition model: Gaussian random walk. Hyper-Parameter(s): ['d_rho']
+ Started new fit:
    + Formatted data.
    + Set uniform prior with parameter boundaries.


    + Finished forward pass.
    + Log10-evidence: -605.35934
    + Finished backward pass.
    + Computed mean parameter values.
```

Plotting the true values of ρ used in the simulation of the data together with the inferred distribution (and posterior mean values) below, we see that the custom model accurately infers the time-varying serial correlation in the data.

```
In [10]: plt.figure(figsize=(8, 4))
         S.plot('rho', label='mean inferred')
         plt.plot(rho, c='r', alpha=0.7, lw=2, label='true')
         plt.legend()
         plt.ylim([-.6, 1]);
```



Finally, we note that the NumPy observation model allows to access multiple data points at once, as we can pass arbitrary data segments to it (in the example above, each data segment contained the current and the previous data point). This also means that there is no check against looking at the data points twice, and the user has to make sure that the likelihood function at time t always states the probability of **only the current** data point:

$$P(\text{data}_t \mid \{\text{data}_{t'}\}_{t'<t}, \text{parameters})$$

If the left side of this conditional probability contains data points from more than one time step, the algorithm will look at each data point more than once, and this generally results in an underestimation of the uncertainty teid to the model parameters!

1.2.9 Probability parser

bayesloop as a probabilistic programming framework is focused on two-level hierarchical models for time series analysis, and is therefore not as flexible as other frameworks. PyMC3, for example, allows the user to create hierarchical models of any structure and to apply arbitrary arithmetic operations on the random variables contained in the model. While bayesloop allows the user to build custom low-level (observation) models and choose from a variety of high-level (transition) models, one has no direct influence on the choice of parameters. In other words, the Gaussian observation model includes the parameters *mean* and *standard deviation*, but there is no direct way of evaluating the *variance* or the *ratio of mean and standard deviation* instead. In many applications, however, these combinations or transformed variables are the key to informed desicions! In finance, for example, the performance of a financial asset is often evaluated by its Sharpe ratio S:

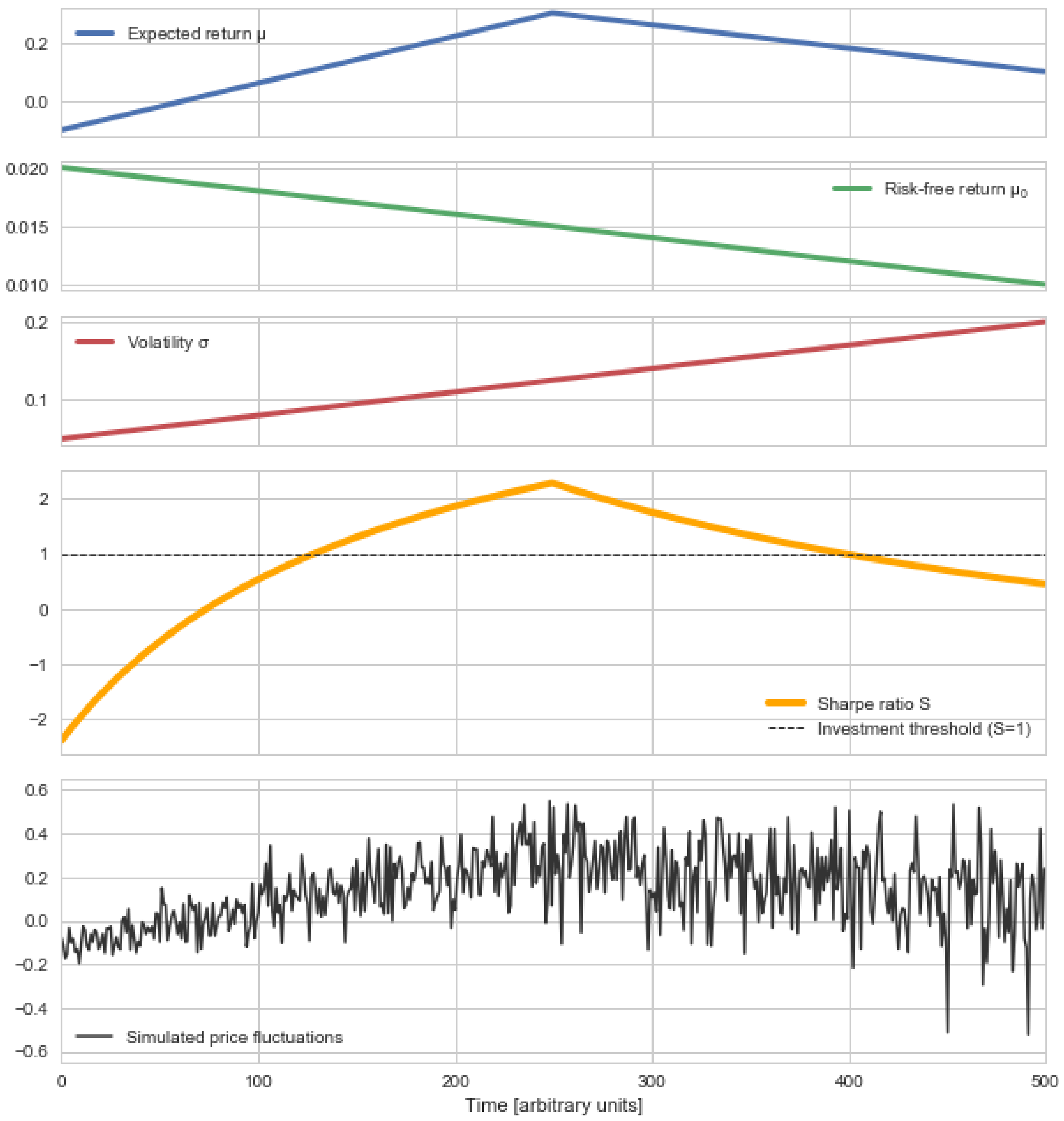
$$S = \frac{\mu - \mu_0}{\sigma} ,$$

where μ is the expected return of the financial asset, μ_0 is the risk-free return, and σ is the volatility. Assuming Gaussian returns, we can obtain time-varying estimates of μ and σ with a simple `Gaussian` observation model with `bayesloop`, while μ_0 is just a series of given numbers we may obtain from the central bank.

This tutorial shows how to evaluate arithmetic combinations of inferred parameter distributions after the model has been fitted, using the `eval` method of the `Study` class and the `Parser` class. With these tools, we compute the probability that the time-varying Sharpe ratio of a simulated series of returns is greater than one (a common threshold for investing in financial assets).

First, we simulate a fictious series of price fluctuations. Here we assume gradual changes in the expected return, the risk-free return and the volatility. Based on the expected return and the volatility, we sample the observed price fluctuations based on a Gaussian distribution:

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt # plotting
        import seaborn as sns           # nicer plots
        sns.set_style('whitegrid')      # plot styling
        sns.set_color_codes()

        import bayesloop as bl
        import numpy as np

        mu_sim = np.concatenate([np.linspace(-0.1, 0.3, 250), np.linspace(0.3, 0.1, 250)])
        sigma_sim = np.linspace(0.05, 0.2, 500)
        mu0 = np.linspace(0.02, 0.01, 500)
        sharpe_sim = (mu_sim-mu0)/sigma_sim

        np.random.seed(123456)
        data = mu_sim + sigma_sim*np.random.randn(500)

        plt.figure(figsize=(10, 11))
        plt.subplot2grid((7, 1), (0, 0))
        plt.plot(mu_sim, c='b', lw=3, label='Expected return $\mathregular{\mu}$')
        plt.xticks([100, 200, 300, 400], ['', '', '', ''])
        plt.xlim([0, 500])
        plt.legend(loc='upper left')

        plt.subplot2grid((7, 1), (1, 0))
        plt.plot(mu0, c='g', lw=3, label='Risk-free return $\mathregular{\mu_0}$')
        plt.xticks([100, 200, 300, 400], ['', '', '', ''])
        plt.xlim([0, 500])
        plt.legend(loc='upper right')

        plt.subplot2grid((7, 1), (2, 0))
        plt.plot(sigma_sim, c='r', lw=3, label='Volatility $\mathregular{\sigma}$')
        plt.xticks([100, 200, 300, 400], ['', '', '', ''])
        plt.xlim([0, 500])
        plt.legend(loc='upper left')

        plt.subplot2grid((7, 1), (3, 0), rowspan=2)
        plt.plot(sharpe_sim, lw=4, c='orange', label='Sharpe ratio S')
        plt.xticks([100, 200, 300, 400], ['', '', '', ''])
        plt.axhline(y=1, c='0.2', ls='dashed', lw=1, label='Investment threshold (S=1)')
        plt.xlim([0, 500])
        plt.legend(loc='lower right')

        plt.subplot2grid((7, 1), (5, 0), rowspan=2)
        plt.plot(data, c='0.2', lw=1.5, label='Simulated price fluctuations')
        plt.xlim([0, 500])
        plt.ylim([-0.65, 0.65])
        plt.legend(loc='lower left')
        plt.xlabel('Time [arbitrary units]');
```



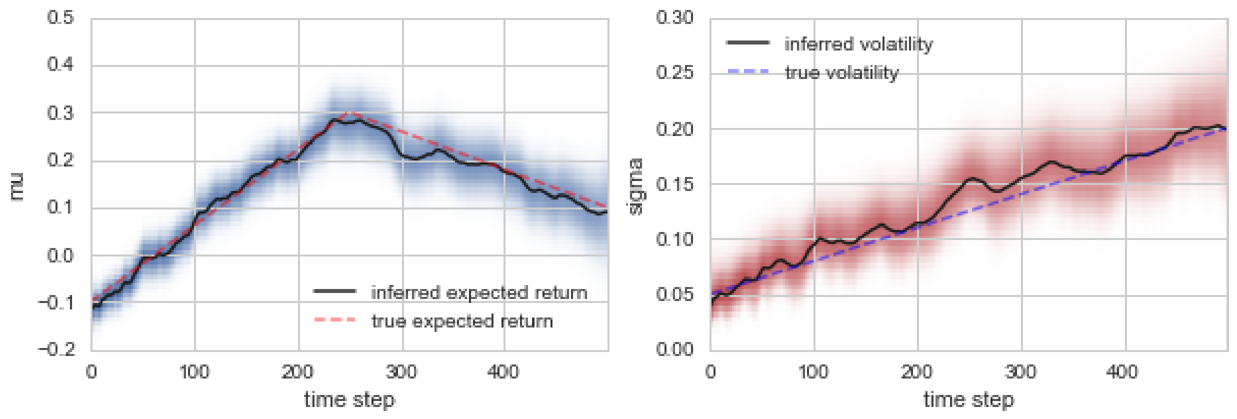
To infer the time-varying expected return μ and volatility σ from the simulated price fluctuations, we assume a `Gaussian` observation model. The two parameters of this observation model are themselves subject to a `GaussianRandomWalk`, the transition model of the study. For simplicity, we set fixed magnitudes for the parameter fluctuations (`sigma_mu`, `sigma_stdev`):

```
In [2]: S = bl.Study()
        S.load(data)

        L = bl.om.Gaussian('mu', bl.cint(-0.5, 0.5, 100), 'sigma', bl.oint(0, 0.5, 100))
        T = bl.tm.CombinedTransitionModel(
                bl.tm.GaussianRandomWalk('sigma_mu', 0.01, target='mu'),
                bl.tm.GaussianRandomWalk('sigma_stdev', 0.005, target='sigma')
            )

        S.set(L, T)
        S.fit()
```

```
+ Created new study.
+ Successfully imported array.
+ Observation model: Gaussian observations. Parameter(s): ['mu', 'sigma']
+ Transition model: Combined transition model. Hyper-Parameter(s): ['sigma_mu', 'sigma_stdev']
+ Started new fit:
    + Formatted data.
    + Set prior (function): jeffreys. Values have been re-normalized.


    + Finished forward pass.
    + Log10-evidence: 130.79643


    + Finished backward pass.
    + Computed mean parameter values.
```

Plotting the inferred expected return and volatility against the true underlying values used in the simulation, we find that the time-varying parameter model we defined above nicely captures the variations in the performance of the fictious financial asset:

```
In [3]: plt.figure(figsize=(10, 3))
        plt.subplot2grid((1, 2), (0, 0))
        S.plot('mu', color='b', label='inferred expected return')
        plt.plot(mu_sim, lw=1.5, c='red', ls='dashed', alpha=0.5, label='true expected return')
        plt.ylim([-0.2, 0.5])
        plt.legend(loc='lower right')

        plt.subplot2grid((1, 2), (0, 1))
        S.plot('sigma', color='r', label='inferred volatility')
        plt.plot(sigma_sim, lw=1.5, c='blue', ls='dashed', alpha=0.5, label='true volatility')
        plt.ylim([0, 0.3])
        plt.legend(loc='upper left');
```



Based on these inferred parameter distributions, we can now evaluate probabilities in the form of inequalities using the `eval` function of the `Study` class. For example, we can evaluate the probability of a positive expected return at time step 50:

```
In [4]: S.eval('mu > 0.', t=50);
```

```
P(mu > 0.) = 0.2486016146556542
```

By default, `eval` prints out the resulting probability. This behavior can be controlled with the keyword-argument `silent`. `eval` further returns the value, for further processing:

```
In [5]: p = S.eval('mu > 0.', t=50, silent=True)
        print(p)
```

```
0.248601614656
```

The keyword-argument `t` is not the only way to define the time step at which to evaluate the probability value. One can also use the `@` operator. In this example, we ask for the probability of a volatility value smaller than 0.2 at `t=400`:

```
In [6]: S.eval('sigma@400 < 0.2');
```

```
P(sigma@400 < 0.2) = 0.934624296075193
```

The evaluation of probability values is not restricted to a single time step, one may also be interested whether the expected return at `t=100` is greater than at `t=400`. Note that `bayesloop` does not infer the joint parameter distribution of all time steps, but the conditional joint parameter distributions for each time step iteratively. This means that correlations between the parameter at different time steps are ignored by the `eval` method. Further note that comparing parameters at different time steps may result in a fairly long computation time and a lot of RAM usage, as `bayesloop` has to take into account all possible combinations of parameter values.

```
In [7]: S.eval('mu@100 > mu@400');
```

```
P(mu@100 > mu@400) = 0.0007986199467668083
```

The `eval` method further computes arithmetic combinations of different parameters. In our example, `bayesloop` computes the joint parameter distribution of `mu` and `sigma` for each time step. When we ask for the probability that the ratio of `mu` and `sigma` is greater than one, `eval` computes the ratio of the parameters from their joint distribution:

```
In [8]: S.eval('mu/sigma > 1', t=100);
```

```
P(mu/sigma > 1) = 0.12878704018028345
```

The parameter ratio above does not yet resemble the Sharpe ratio S, as the risk-free return is not inserted yet. The `eval` method does not only support the parsing of parameter names, but also the parsing of plain numbers. Below, we first check the risk-free return at `t=100`, and then insert this number into the query of the `eval` method. We discover that at `t=100`, the probability that the Sharpe ratio S exceeds the value of one is only about 2.3%:

```
In [9]: mu0[100]
```

```
Out[9]: 0.017995991983967938
```

```
In [10]: S.eval('(mu - 0.018)/sigma > 1', t=100);
```

```
P((mu - 0.018)/sigma > 1) = 0.023246959985844733
```

Of course, we can also insert the risk-free return value directly, via string manipulation:

```
In [11]: S.eval('(mu - ' + str(mu0[100]) + ')/sigma > 1', t=100);
```

```
P((mu - 0.017995991984)/sigma > 1) = 0.023246959985844733
```

The `eval` method contains a bit of computational overhead, as it has to pre-process all parameter values before every single query. As we want to compute $p(S > 1)$ for *all* 500 time steps, we are better off by using a `Parser` instance. The `Parser` is initialized only once, and then queried repeatedly without computational overhead. Below we show one example with a single query to the `Parser` instance, and subsequently repeat the query for all time steps:

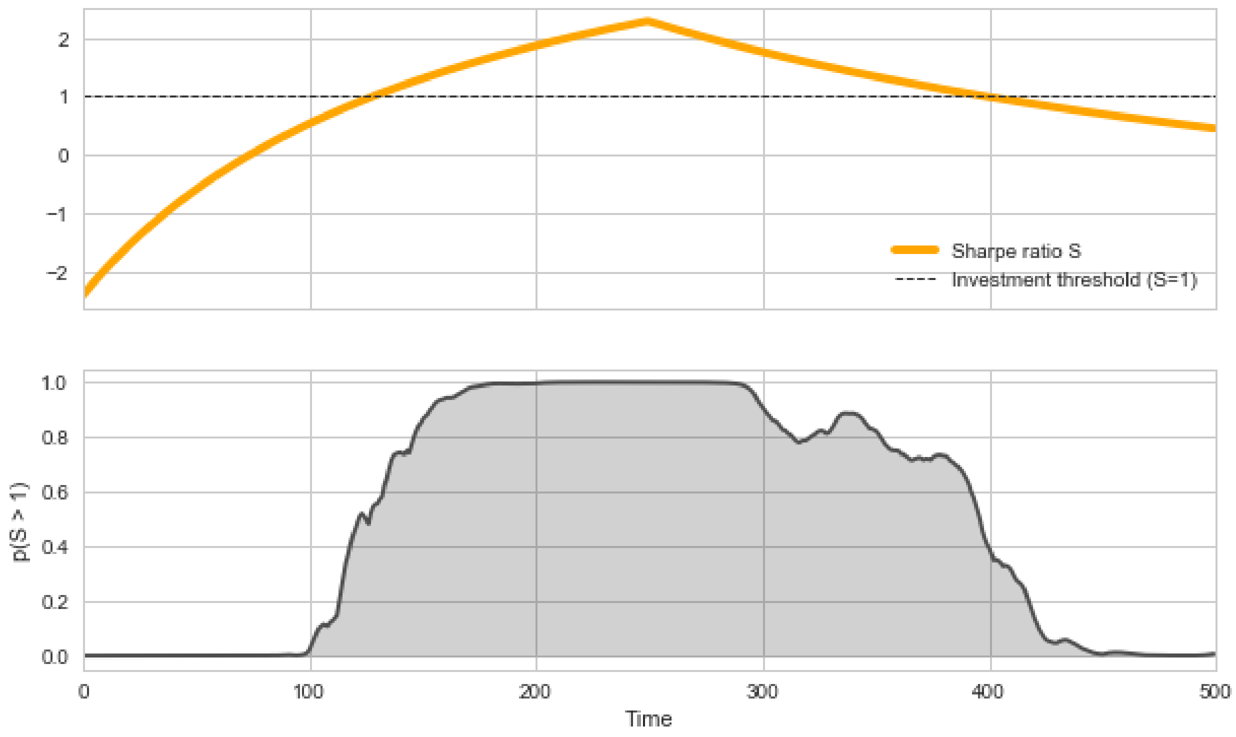
```
In [12]: P = bl.Parser(S)
         P('(mu - ' + str(mu0[100]) + ')/sigma > 1', t=100);
```

```
P((mu - 0.017995991984)/sigma > 1) = 0.023246959985844733
```

```
In [13]: p_invest = []

         for t in range(500):
             p_invest.append(P('(mu - ' + str(mu0[t]) + ')/sigma > 1', t=t, silent=True))
```

Finally, we may plot the probability that our probabilistic, time-varying Sharpe ratio S is greater than one for each time step, together with the underlying true values of S in the simulation. We find that $p(S > 1)$ attains a value of ≈ 0.5 at the times when $S = 1$ in the simulation, and increases further as $S > 1$. Consequently, we find that $p(S > 1) < 0.5$ for $S < 1$ in the simulation. This example thereby illustrates how the combination of inferred time-varying parameters can help to create robust indicators for desicion making in the presence of uncertainty.

```
In [14]: plt.figure(figsize=(10, 6))

         plt.subplot2grid((2, 1), (0, 0))
         plt.plot(sharpe_sim, lw=4, c='orange', label='Sharpe ratio S')
         plt.xticks([100, 200, 300, 400], ['', '', '', ''])
         plt.axhline(y=1, c='0.2', ls='dashed', lw=1, label='Investment threshold (S=1)')
         plt.xlim([0, 500])
         plt.legend(loc='lower right')

         plt.subplot2grid((2, 1), (1, 0))
         plt.plot(p_invest, c='0.3', lw=2)
         plt.fill_between(range(500), p_invest, 0, color='k', alpha=0.2)
         plt.xlim([0, 500])
         plt.ylim([-0.05, 1.05])
         plt.xlabel('Time')
         plt.ylabel('p(S > 1)');
```



So far, we have parsed either parameter names or plain numbers in the queries to the `eval` method or the `Parser` instance. In many applications, however, one also needs to compute the `sqrt`, or the `cos` of a parameter. `bayesloop`'s probability parser therefore allows to parse any function contained in the `numpy` module. Note, however, that only functions which preserve the shape of an input array will return valid results (e.g. using `sum` will not work).

```
In [15]: P('sin(mu) + cos(sigma) > 1', t=50);
```

```
P(sin(mu) + cos(sigma) > 1) = 0.2479001934193268
```

On a final note, the `Parser` class also handles (hyper-)parameters of different study instances. For example, one could imagine to run one study `S1` (which may be a `Study`, `HyperStudy`, `ChangepointStudy` or an `OnlineStudy`) to analyze price fluctuations of a financial asset, and another study `S2` to analyze the frequency of market-related Twitter messages. After fitting data points, one can initialize a `Parser` instance that takes into account both studies:

```
P = bl.Parser(S1, S2)
```

Note that the two studies `S1` and `S2` must not contain duplicate (hyper-)parameter names! This allows to create performance indicators that take into account different kinds of information (e.g. pricing data and social media activity).

1.2.10 Multiprocessing

Conducting extensive data studies based on the `HyperStudy` or `ChangepointStudy` classes may involve several 10.000 or 100.000 individual fits (see e.g. here). Since these individual fits with different hyper-parameter values are independent of each other, the computational workload may be distributed among the individual cores of a multi-core processor. To keep things simple, *bayesloop* uses object serialization to create duplicates of the current `HyperStudy` or `ChangepointStudy` instance and distributes them across the predefined number of cores. In general, this procedure may be handled by the built-in Python module multiprocessing. However, *multiprocessing* relies on the built-in module pickle for object serialization, which fails to serialize the classes defined in *bayesloop*. We therefore use a different version of the *multiprocessing* module that is part of the pathos module.

The latest version of *pathos* can be installed directly via pip, but requires git:

```
pip install git+https://github.com/uqfoundation/pathos
```

Note: Windows users need to install a C compiler *before* installing pathos. One possible solution for 64-bit systems is to install Microsoft Visual C++ 2008 SP1 Redistributable Package (x64) and Microsoft Visual C++ Compiler for Python 2.7.

Once installed correctly, the number of cores to use in a hyper-study or change-point study can be specified by using the keyword argument `nJobs` within the `fit` method. Example:

```
S.fit(silent=True, nJobs=4)
```

1.3 Examples

1.3.1 Anomalous diffusion

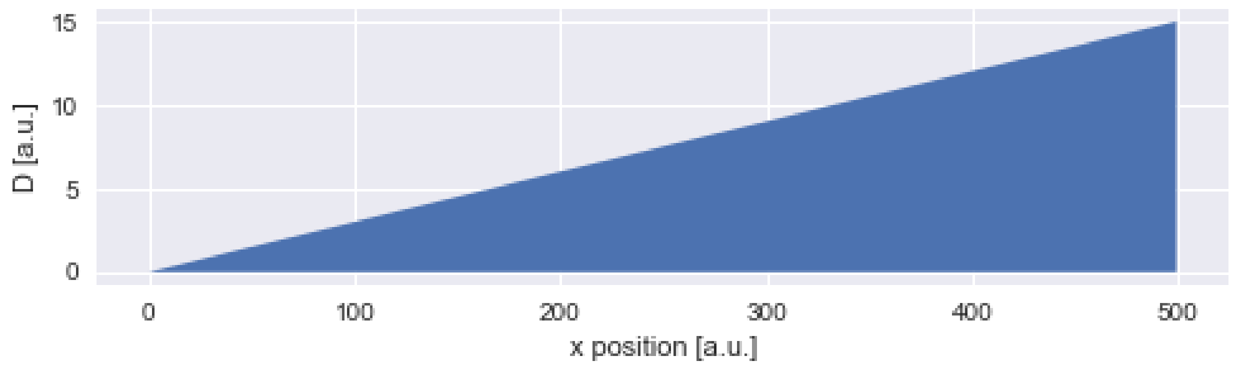
Diffusion processes are mostly used to describe the random motion of particles in fluids, but also apply to biological systems, like animals that search for food or cancer cells migrating through tissue. In an experimental setup, we may simply track the random motions of a particle/animal/cell over time and thereby use it as a probe to measure the diffusion coefficient of the fluid/habitat/tissue. However, by averaging over all measured movements, one may loose important information, as the diffusion coefficient may change over time or depending on the current position of the probe. Changes in the diffusion coefficient may be externally triggered, e.g. by a change in temperature for diffusing particles or a change in terrain within the foraging habitat of an animal. Furthermore, changes in diffusivity may be internally triggered, e.g. by the cell cycle of an invading cancer cell. Here, we present two examples of anomalous diffusion, one with a gradually changing diffusivity, and one with discrete regions of varying diffusivity.

Diffusion gradient

Suppose we have a diffusion gradient, for example due to a temperature gradient across our sample, and we want to determine the diffusion coefficient for different positions along the axis of the gradient. To infer the spatially changing diffusion coefficient, we assume that the amplitude of the random walk of our particle changes gradually over time, as the particle moves through regions of varying diffusivity.

For this example, we assume a positive, constant diffusion gradient, resulting a linear increase of the diffusion coefficient D:

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns

        D = np.linspace(0.0, 15., 500)
        x = np.arange(500)

        plt.figure(figsize=(8,2))
        plt.fill_between(x, D, 0)
        plt.xlabel('x position [a.u.]')
        plt.ylabel('D [a.u.]');
```



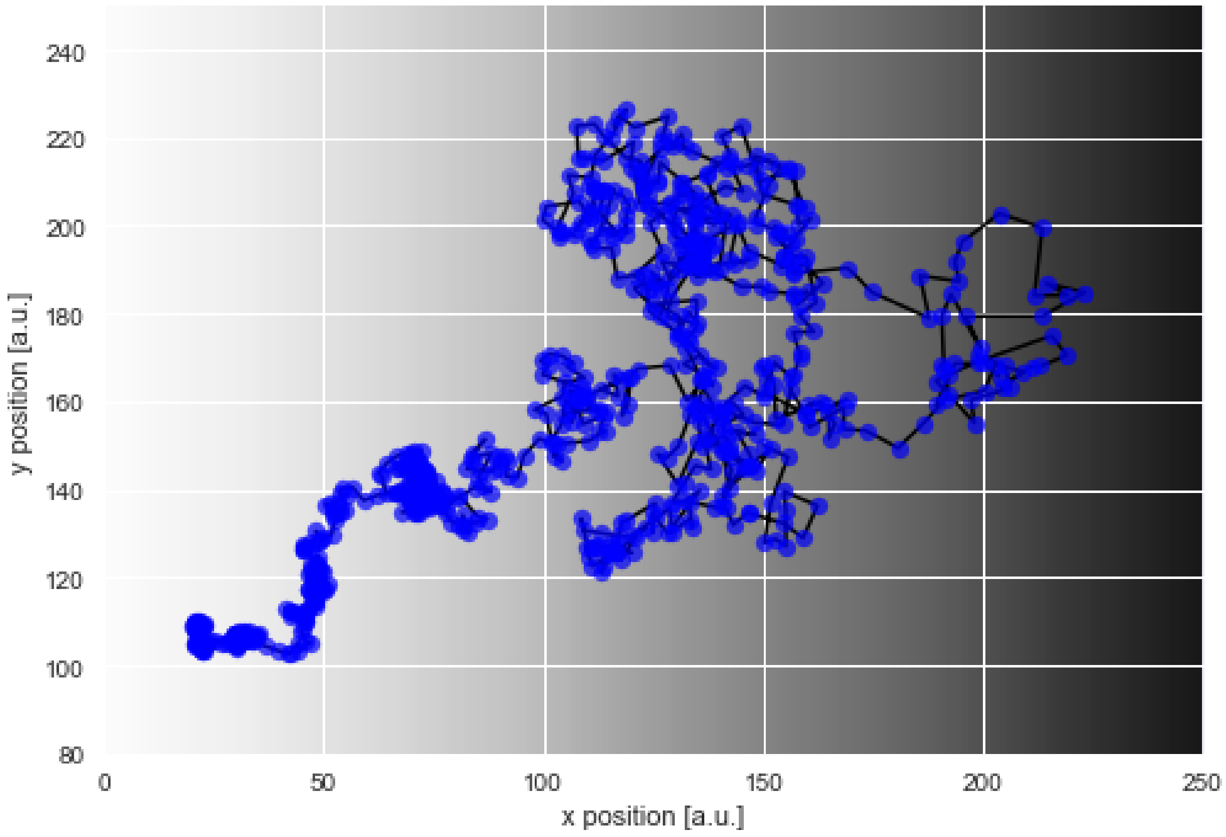
Trajectory simulation

For this example, we simply simulate a random walk of a particle assuming a Gaussian random walk with the standard deviation equal to the diffusion coefficient at the current position:

```
In [2]: np.random.seed(12337) # reproducable results
        trajectory = [[150., 150.]] # starting point

        for t in range(750):
            try:
                dc = D[int(trajectory[-1][0])] # look up diffusion coefficient
            except:
                if int(trajectory[-1][0]) >= 500:
                    dc = 15. # constant diffusion coefficient on the far right
                else:
                    dc = 0. # constant diffusion coefficient on the far left
            trajectory.append([p + np.random.normal(0, dc) for p in trajectory[-1]])

        trajectory = np.array(trajectory)
```

If we plot this simulated trajectory, we see how the distance between two subsequent positions of the particle is significantly smaller on the left as compared to the right of our diffusion gradient (visible as gray shading):

```
In [3]: plt.figure(figsize=(8, 10))

        D_img = np.array([list(D)]*len(D))[:, :250]
        plt.imshow(D_img, cmap='Greys', alpha=0.9, zorder=0, interpolation='nearest')

        plt.scatter(*trajectory.T, lw=0.2, c='b', alpha=0.7, zorder=2, s=50)
        plt.plot(*trajectory.T, lw=1.5, c='k', zorder=1)
        plt.xlim([0, 250])
        plt.ylim([80, 250])
        plt.xlabel('x position [a.u.]')
        plt.ylabel('y position [a.u.]');
```



Modeling

To model this example of anomalous diffusion, we first generate a low-level model - a simple Gaussian random walk - using `SymPy` random variables. Here, we simply assume that the velocity components of our particle are normally distributed, with zero mean. As we further assume that the diffusion coefficient changes gradually across our sample, our particle should also gradually change its diffusivity as it moves across the sample. We therefore also use a Gaussian random walk as the high-level model. As we do not know the amplitude of the change of diffusivity, we choose a `HyperStudy` as it enables the inference of this hyper-parameter.

```
In [4]: import bayesloop as bl
        import sympy.stats
        from sympy import Symbol
        S = bl.HyperStudy()

        # load data
        velocity = trajectory[1:] - trajectory[:-1] # compute velocity vectors from positions
        S.load(velocity)

        # create low-level model
        std = Symbol('D', positive=True)
        normal = sympy.stats.Normal('normal', 0, std)

        # we assume the diffusivity to lie within the interval ]0, 15[
        # within this interval, we create 5000 equally spaced parameter values
        L = bl.om.SymPy(normal, 'D', bl.oint(0, 15, 5000))

        # create high-level model
        # we assume a Gaussian random walk of the parameter 'D' with a
        # standard deviation within the interval [0, 0.3] (20 equally
        # spaced hyper-parameter values)
        T = bl.tm.GaussianRandomWalk('sigma', bl.cint(0, 0.3, 20), target='D')

        S.set(L, T)
        S.fit()
```

```
+ Created new study.
  --> Hyper-study
+ Successfully imported array.
    + Trying to determine Jeffreys prior. This might take a moment...
    + Successfully determined Jeffreys prior: sqrt(2)/D. Will use corresponding lambda function.
+ Observation model: normal. Parameter(s): ('D',)
+ Transition model: Gaussian random walk. Hyper-Parameter(s): ['sigma']
+ Set hyper-prior(s): ['uniform']
+ Started new fit.
    + 20 analyses to run.
```

```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

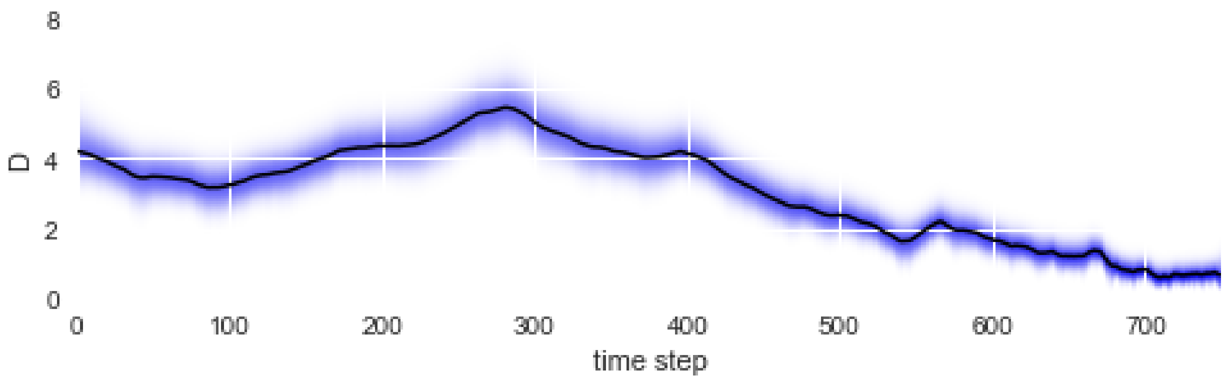
```
C:\Anaconda3\lib\site-packages\scipy\ndimage\filters.py:139: RuntimeWarning: divide by zero encounter
  p = numpy.polynomial.Polynomial([0, 0, -0.5 / (sigma * sigma)])
C:\Anaconda3\lib\site-packages\numpy\polynomial\polynomial.py:780: RuntimeWarning: invalid value enco
  c0 = c[-i] + c0*x
```

```
    ! WARNING: Forward pass distribution contains only zeros, check parameter boundaries!
      Stopping inference process. Setting model evidence to zero.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: -1583.15297
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```

Analysis

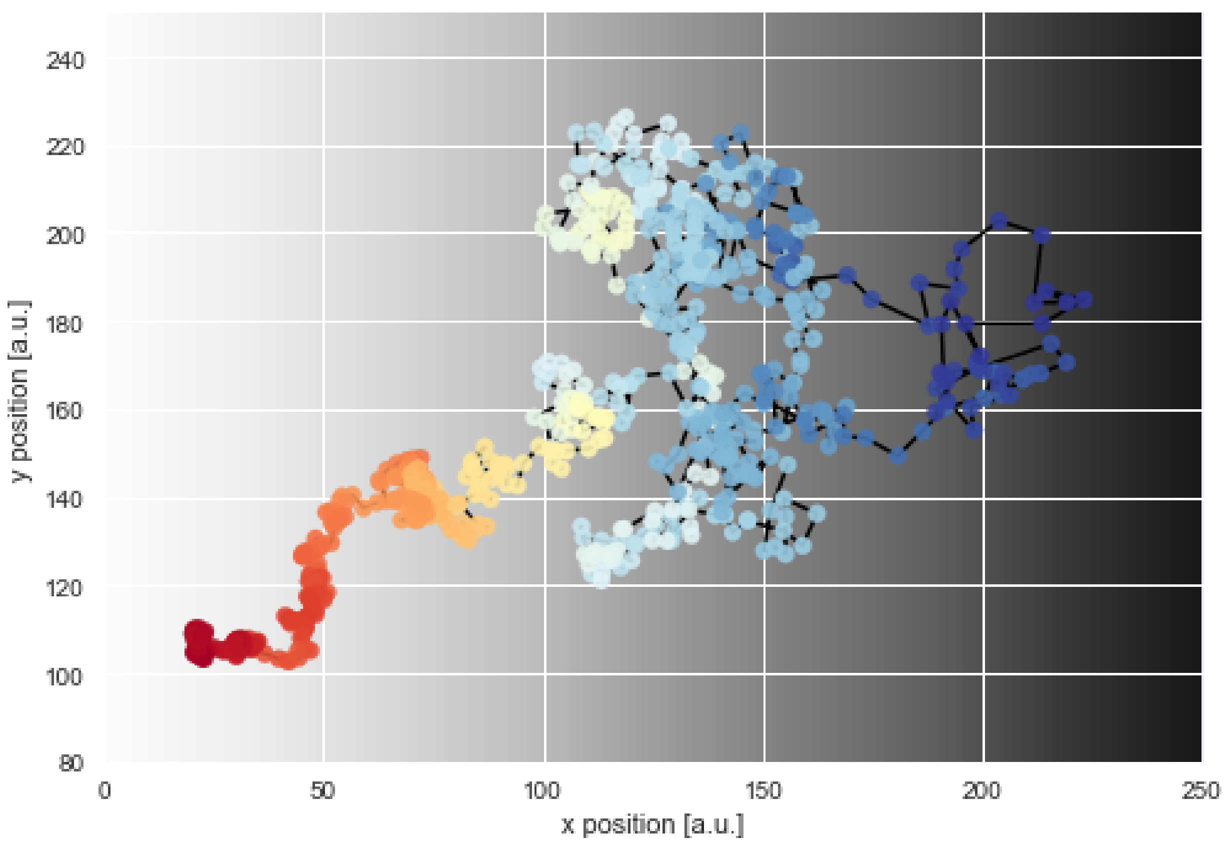
After fitting is done, we may evaluate how the diffusion coefficient of our particle changed over time by plotting the mean values of the posterior distributions (black line) together with the posterior distributions of the amplitude of the particle motions (blue shading):

```
In [5]: plt.figure(figsize=(8,2))
        S.plot('D')
        plt.xlim([0, 750])
        plt.ylim([0, 8]);
```



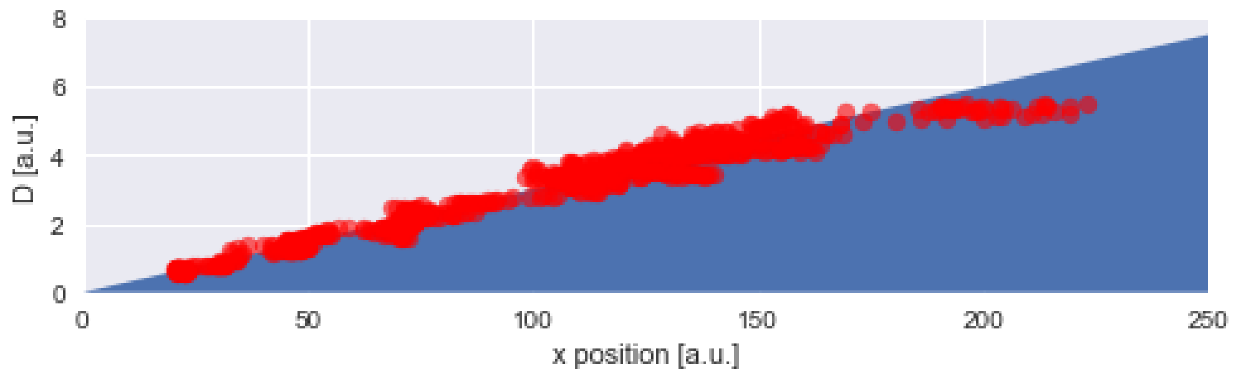
As we can see, the diffusion coefficient changes quite irregularly over time, but this irregularity is just the result of the irregular path of our particle across the sample. However, this plots confirms that the chosen parameter boundaries of $\sigma \in [0, 20]$ suffice. We may re-plot the particle path from above, but this time color-code the particle positions with the corresponding inferred diffusivity values:

```
In [6]: D_mean = S.getParameterMeanValues('D') # extract posterior mean values of first (and only) pa

        # prepare color coding
        m = plt.cm.ScalarMappable(cmap='RdYlBu')
        m.set_array(D_mean)
        m.autoscale()

        plt.figure(figsize=(8, 10))
        plt.imshow(D_img, cmap='Greys', alpha=0.9, zorder=0, interpolation='nearest')
        plt.scatter(*trajectory[1:].T, lw=0.2, c=[m.to_rgba(x) for x in D_mean], alpha=0.9, zorder=2,
        plt.plot(*trajectory[1:].T, lw=1.5, c='k', zorder=1)
        plt.xlim([0, 250])
        plt.ylim([80, 250])
        plt.xlabel('x position [a.u.]')
        plt.ylabel('y position [a.u.]');
```



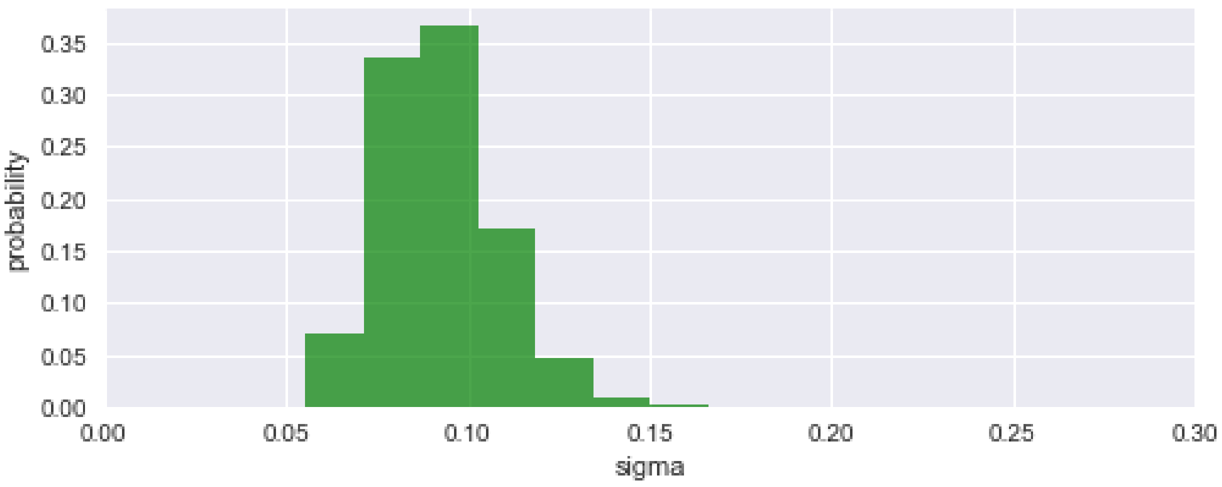
Finally, we can directly compare the inferred mean values to the true values of the diffusion coefficient used in the simulation:

```
In [7]: D = np.linspace(0.0, 15., 500)
        x = np.arange(500)

        plt.figure(figsize=(8,2))
        plt.fill_between(x, D, 0)
        plt.scatter(trajectory[1:, 0], D_mean, c='r', alpha=0.6)

        plt.xlim([0, 250])
        plt.ylim([0, 8])
        plt.xlabel('x position [a.u.]')
        plt.ylabel('D [a.u.]');
```



To assess our choice for the hyper-parameter boundaries, we may further plot the inferred distribution of the magnitude of parameter changes between time steps. In this case, the distribution falls off to (approximately) zero on both sides, therefore the boundaries suffice.

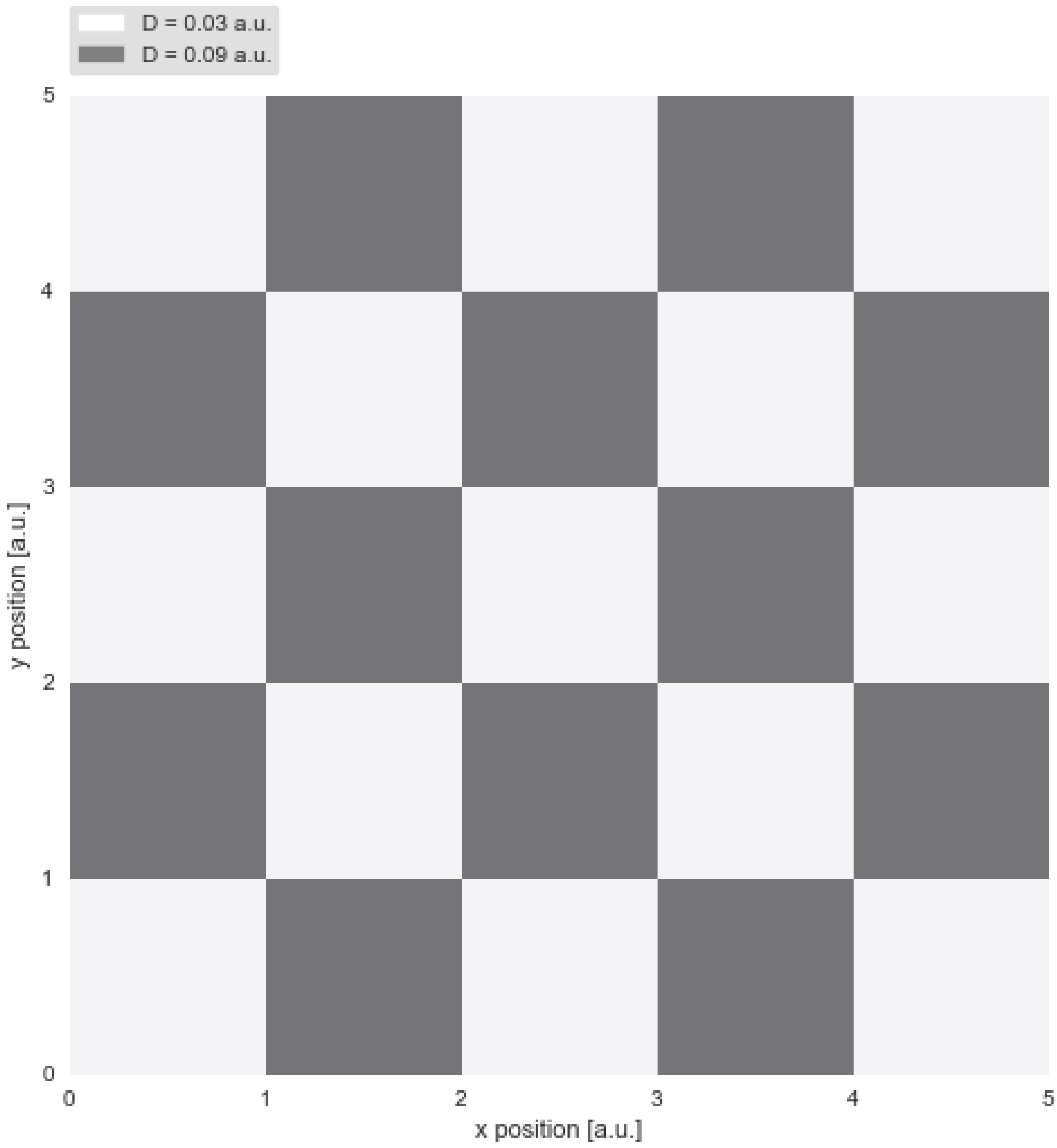
```
In [8]: plt.figure(figsize=(8,3))
        S.plot('sigma', alpha=0.7, facecolor='g')
        plt.xlim([0, 0.3]);
```



Regime-switch diffusion process

While the first example investigated gradual variations, we now turn to abrupt jumps in diffusivity, as they may occur for proteins diffusing on a cell membrane. The binding and unbinding of the protein to other constituents of the membrane may inhibit or favor diffusion. In a very simplistic attempt to model this complex process, we assume only two different regimes of diffusivity, and arrange them spatially like a chessboard pattern. On the dark patches, the diffusivity reaches three times the value as on the light patches:

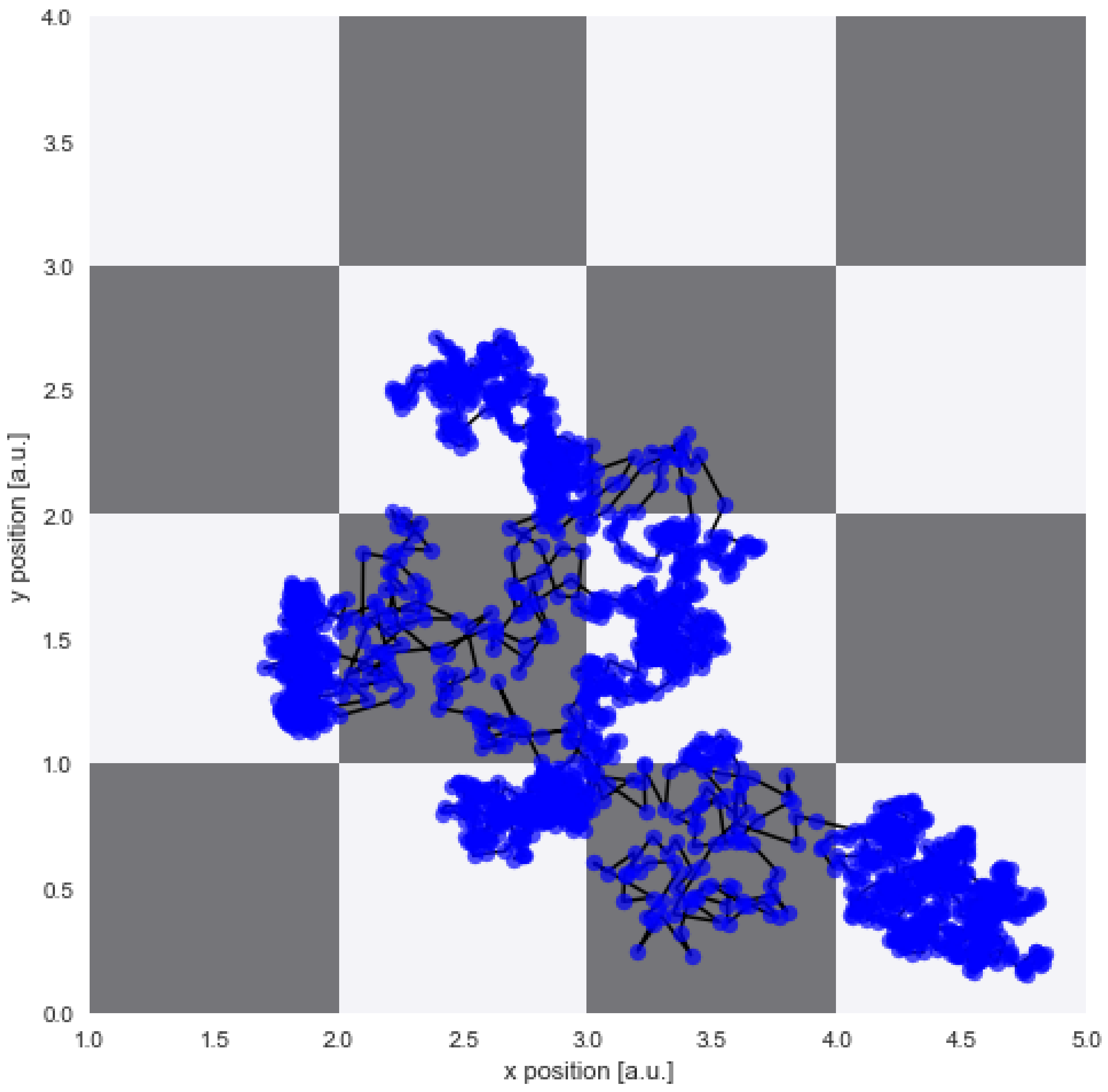
```
In [9]: import matplotlib.patches as mpatches

        # helper function for chessboard pattern
        def check_region(x, y):
            if (int(x) % 2 == 0 and int(y) % 2 == 0) or (int(x) % 2 == 1 and int(y) % 2 == 1):
                return True
            else:
                return False

        # create chessboard pattern
        n = 5
        D_img = np.array([0, 1]*int(np.ceil(0.5*(n**2))))[:-1].reshape([n, n])

        # plot chessboard pattern
        plt.figure(figsize=(8,8))
        plt.imshow(D_img, cmap='Greys', alpha=0.5, extent=[0, n, 0, n], zorder=0, interpolation='near
        plt.grid(b=False, which='major')
        plt.xlabel('x position [a.u.]')
        plt.ylabel('y position [a.u.]')

        # legend
        white_patch = mpatches.Patch(color='white', label='D = 0.03 a.u.')
        gray_patch = mpatches.Patch(color='0.5', label='D = 0.09 a.u.')
        legend = plt.legend(handles=[white_patch, gray_patch],
                            bbox_to_anchor=(0., 1.02, 1., .102),
                            loc=3,
                            borderaxespad=0.,
                            frameon = 1)
        frame = legend.get_frame()
        frame.set_facecolor('0.85')
```



Trajectory simulation

Again, we simulate a random walk of a particle assuming a Gaussian random walk with the standard deviation equal to the diffusion coefficient at the current position:

```
In [10]: np.random.seed(1274)

         D1 = 0.03
         D2 = 0.09

         trajectory = [[2.5, 2.5]]

         for t in range(2000):
             dc = D1 if check_region(*trajectory[-1]) else D2
             trajectory.append([x + np.random.normal(0, dc) for x in trajectory[-1]])

         trajectory = np.array(trajectory)

         # plotting
         plt.figure(figsize=(8,8))
         plt.imshow(D_img, cmap='Greys', alpha=0.5, extent=[0, n, 0, n], zorder=0, interpolation='nea
         plt.grid(b=False, which='major')
         plt.xlabel('x position [a.u.]')
         plt.ylabel('y position [a.u.]')

         plt.scatter(*trajectory.T, lw=0.2, c='b', alpha=0.7, zorder=2, s=50)
         plt.plot(*trajectory.T, lw=1.5, c='k', zorder=1)

         plt.xlim([1, 5])
         plt.ylim([0, 4]);
```



Modeling

The low-level model stays the same as in the previous example, a Gaussian random walk. As the parameter changes are not gradual in this example, however, we assume a regime-switching process for the high-level model. Here, we assign a small probability to all possible parameter values in each time step to account for the rare, but possible event of a abrupt parameter change. As we do not know this minimal probability value a-priori, we again use a hyper-study to test different values. Note that we may keep our study-instance defined as above and just load new data, adjust the parameter boundaries and alter the high-level model:

```
In [11]: # load new data
         velocity = trajectory[1:] - trajectory[:-1]
         S.load(velocity)

         # change parameter boudaries of low-level model
         L = bl.om.SymPy(normal, 'D', bl.oint(0, 0.2, 2000))

         # create high-level model
         T = bl.tm.RegimeSwitch('log10pMin', bl.cint(-10, 3, 20))

         S.set(L, T)
         S.fit()
```

```
+ Successfully imported array.
    + Trying to determine Jeffreys prior. This might take a moment...
    + Successfully determined Jeffreys prior: sqrt(2)/D. Will use corresponding lambda function.
+ Observation model: normal. Parameter(s): ('D',)
+ Transition model: Regime-switching model. Hyper-Parameter(s): ['log10pMin']
+ Set hyper-prior(s): ['uniform']
+ Started new fit.
    + 20 analyses to run.
```

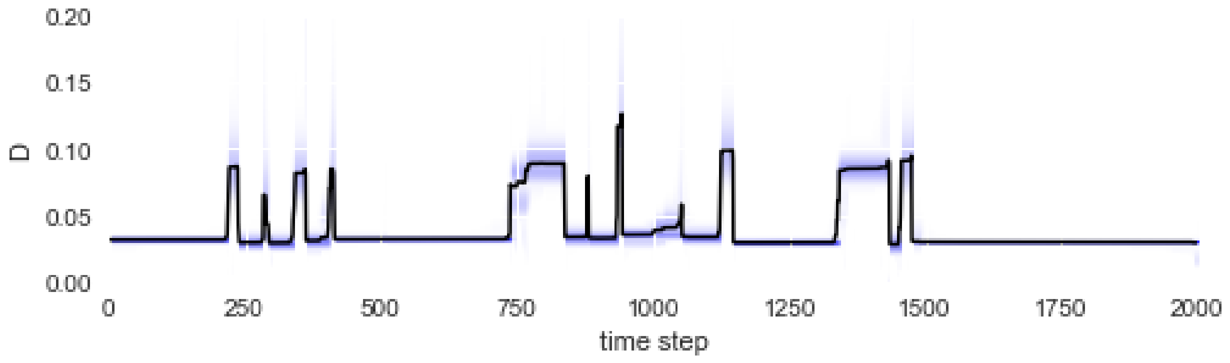
```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

```
    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: 3236.20040
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```

Analysis

After fitting is done, we may check whether we are indeed able to detect the abrupt parameter changes over time:

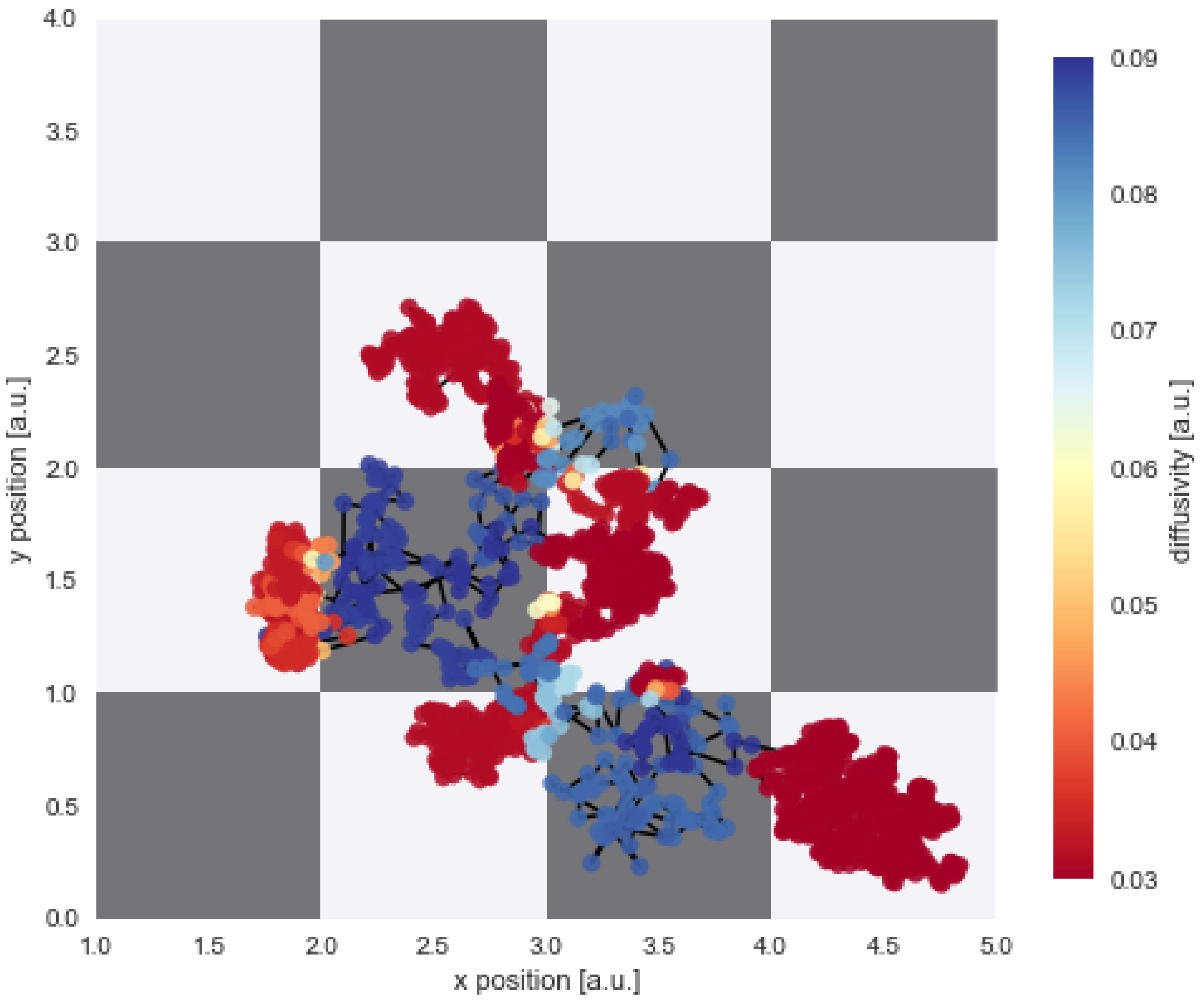
```
In [12]: plt.figure(figsize=(8,2))
         S.plot('D')
         plt.xlim([0, 2000])
         plt.ylim([0, .2]);
```



The irregular spacing of the parameter jumps in time gets more regular once the parameter evolution is plotted on the chessboard pattern:

```
In [13]: D_mean = S.getParameterMeanValues('D')

         # prepare color coding
         m = plt.cm.ScalarMappable(cmap='RdYlBu')
         m.set_array([0.03, 0.09])
         m.autoscale()
```

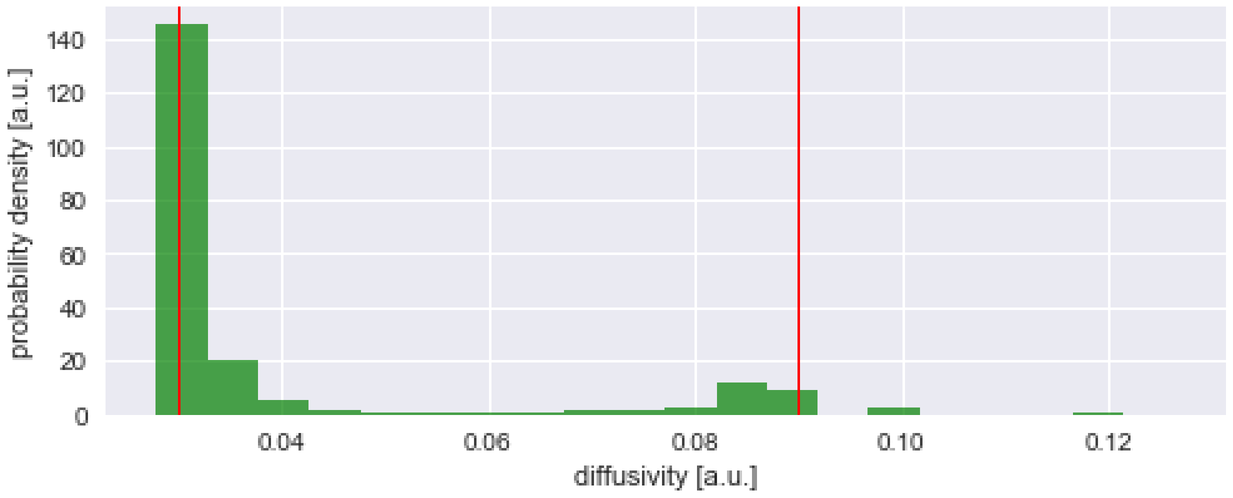
```
# plotting
plt.figure(figsize=(8,8))
plt.imshow(D_img, cmap='Greys', alpha=0.5, extent=[0, n, 0, n], zorder=0, interpolation='nea
plt.grid(b=False, which='major')
plt.xlabel('x position [a.u.]')
plt.ylabel('y position [a.u.]')

plt.scatter(*trajectory[1:].T, lw=0.2, c=[m.to_rgba(x) for x in D_mean], alpha=0.9, zorder=2
plt.plot(*trajectory[1:].T, lw=1.5, c='k', zorder=1)

cb = plt.colorbar(m, shrink=0.75)
cb.set_label('diffusivity [a.u.]')
cb.set_clim(0.03, 0.09)

plt.xlim([1, 5])
plt.ylim([0, 4]);
```



The figure above shows that we can nicely reconstruct the spatial pattern in diffusivity for the region covered by the diffusing particle. Finally, we may plot a histogram to confirm that the inferred values for the diffusion coefficient (green histogram) align with the true values used in the simulation (red lines):

```
In [14]: plt.figure(figsize=(8,3))
         plt.hist(D_mean, 20, alpha=0.7, facecolor='g', normed=True)
```

```
        plt.axvline(0.03, 0, 1600, lw=1, c='r')
        plt.axvline(0.09, 0, 1600, lw=1, c='r')
        plt.xlabel('diffusivity [a.u.]')
        plt.ylabel('probability density [a.u.]');
```



Model selection

In the example above, we a-priori assumed abrupt changes of the diffusion coefficient, because we knew the parameter dynamics from the simulation of our trajectory. In an experimental setup, however, one hardly knows the exact nature of the parameter dynamics before-hand. The big advantage of *bayesloop* is to compute the model evidence, i.e. the probability (density) of observing the data given the model one assumes. For a simple demonstration, we again choose the high-level model of the first example and assume gradual parameter fluctuations (we only adjust the scale for this example) and re-run the fit:

```
In [15]: T = bl.tm.GaussianRandomWalk('sigma', bl.cint(0, 0.01, 20), target='D')
         S.set(T)

         S.fit()
```

```
+ Transition model: Gaussian random walk. Hyper-Parameter(s): ['sigma']
+ Set hyper-prior(s): ['uniform']
+ Started new fit.
    + 20 analyses to run.

Widget Javascript not detected.  It may not be installed or enabled properly.

C:\Anaconda3\lib\site-packages\scipy\ndimage\filters.py:139: RuntimeWarning: divide by zero encounte
  p = numpy.polynomial.Polynomial([0, 0, -0.5 / (sigma * sigma)])
C:\Anaconda3\lib\site-packages\numpy\polynomial\polynomial.py:780: RuntimeWarning: invalid value enc
  c0 = c[-i] + c0*x

    ! WARNING: Forward pass distribution contains only zeros, check parameter boundaries!
      Stopping inference process. Setting model evidence to zero.

    + Computed average posterior sequence
    + Computed hyper-parameter distribution
    + Log10-evidence of average model: 3206.17483
    + Computed local evidence of average model
    + Computed mean parameter values.
+ Finished fit.
```

Here, we obtain a model evidence value of **3206** on a $\log_{10}$ scale (these probability density values can get absurdly small or large if many data points are involved), compared to **3236** for the correct, regime-switching model. This means that it is **10^{30}-times more likely** that the 2000 data points of our trajectory are created by a regime-switching process, compared to gradual variations of the diffusion coefficient!

1.3.2 Stock market fluctuations

The fluctuations of stock prices represent an intriguing example of a complex random walk. Stock prices are influenced by transactions that are carried out over a broad range of time scales, from micro- to milliseconds for high-frequency hedge funds over several hours or days for day-traders to months or years for long-term investors. We therefore expect that the statistical properties of stock price fluctuations, like volatility and auto-correlation of returns, are not constant over time, but show significant fluctuations of their own. Time-varying parameter models can account for such changes in volatility and auto-correlation and update their parameter estimates in real-time.

There are, however, certain events that render previously gathered information about volatility or autocorrelation of returns completely useless. News announcements that are unexpected at least to some extent, for example, can induce increased trading activity, as market participants update their orders according to their interpretation of novel information. The resulting *"non-scheduled"* price corrections can often not be adequately described by the current estimates of volatility. Even a model that accounts for gradual variations in volatility cannot reproduce these large price corrections. Instead, when such an event happens, it becomes favorable to forget about previously gathered parameter estimates and completely start over. In this example, we use the *bayesloop* framework to specifically evaluate the probability of previously acquired information becoming useless. We interpret this probability value as a risk metric and evaluate it for each minute of an individual trading day (Nov 28, 2016) for the exchange-traded fund SPY. The announcement of macroeconomic indicators on this day results in a significant increase of our risk metric in intra-day trading.

DISCLAIMER: This website does not provide tax, legal or accounting advice. This material has been prepared for informational purposes only, and is not intended to provide, and should not be relied on for, tax, legal or accounting advice. You should consult your own tax, legal and accounting advisors before engaging in any transaction.

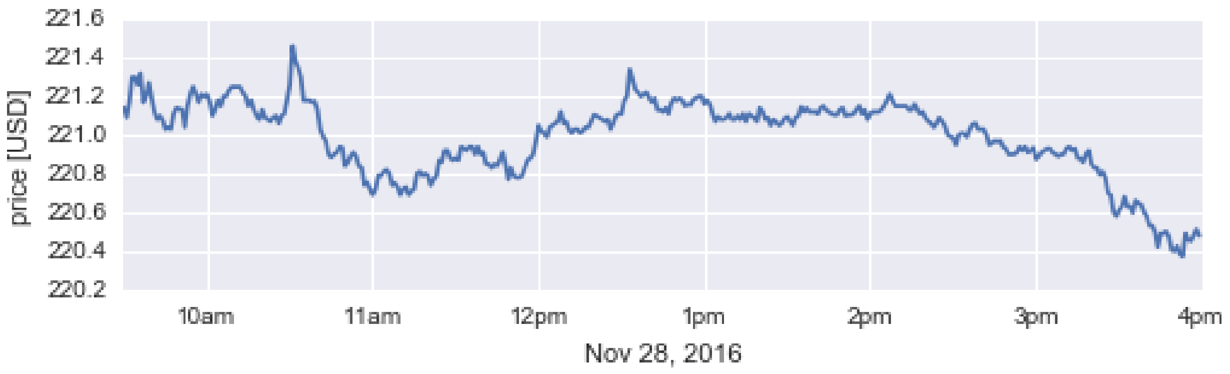
Note: The intraday pricing data used in this example is obtained via Google Finance:

```
https://www.google.com/finance/getprices?q=SPY&i=60&p=1d&f=d,c
```

This request returns a list of minute close prices (and date/time information; `f=d,c`) for SPY for the last trading period. The maximum look-back period is 14 days (`p=14d`) for requests of minute-scale data (`i=60`).

```
In [1]: %matplotlib inline
        import numpy as np
        import bayesloop as bl
        import sympy.stats as stats
        from tqdm import tqdm_notebook
        import matplotlib.pyplot as plt
        import seaborn as sns
        sns.set_color_codes()  # use seaborn colors

        # minute-scale pricing data
        prices = np.array(
             [ 221.14 ,  221.09 ,  221.17 ,  221.3  ,  221.3  ,  221.26 ,
               221.32 ,  221.17 ,  221.2  ,  221.27 ,  221.19 ,  221.12 ,
               221.08 ,  221.1  ,  221.075,  221.03 ,  221.04 ,  221.03 ,
               221.11 ,  221.14 ,  221.135,  221.13 ,  221.04 ,  221.15 ,
               221.21 ,  221.25 ,  221.21 ,  221.17 ,  221.21 ,  221.2  ,
               221.21 ,  221.17 ,  221.1  ,  221.13 ,  221.18 ,  221.15 ,
               221.2  ,  221.2  ,  221.23 ,  221.25 ,  221.25 ,  221.25 ,
               221.25 ,  221.22 ,  221.2  ,  221.15 ,  221.18 ,  221.13 ,
               221.1  ,  221.08 ,  221.13 ,  221.09 ,  221.08 ,  221.07 ,
               221.09 ,  221.1  ,  221.06 ,  221.1  ,  221.11 ,  221.18 ,
               221.26 ,  221.46 ,  221.38 ,  221.35 ,  221.3  ,  221.18 ,
               221.18 ,  221.18 ,  221.17 ,  221.175,  221.13 ,  221.03 ,
               220.99 ,  220.97 ,  220.9  ,  220.885,  220.9  ,  220.91 ,
               220.94 ,  220.935,  220.84 ,  220.86 ,  220.89 ,  220.91 ,
               220.89 ,  220.84 ,  220.83 ,  220.74 ,  220.755,  220.72 ,
               220.69 ,  220.72 ,  220.79 ,  220.79 ,  220.81 ,  220.82 ,
               220.8  ,  220.74 ,  220.75 ,  220.73 ,  220.69 ,  220.72 ,
               220.73 ,  220.69 ,  220.71 ,  220.72 ,  220.8  ,  220.81 ,
               220.79 ,  220.8  ,  220.79 ,  220.74 ,  220.77 ,  220.79 ,
               220.87 ,  220.86 ,  220.92 ,  220.92 ,  220.88 ,  220.87 ,
               220.88 ,  220.87 ,  220.94 ,  220.93 ,  220.92 ,  220.94 ,
               220.94 ,  220.9  ,  220.94 ,  220.9  ,  220.91 ,  220.85 ,
               220.85 ,  220.83 ,  220.85 ,  220.84 ,  220.87 ,  220.91 ,
               220.85 ,  220.77 ,  220.83 ,  220.79 ,  220.78 ,  220.78 ,
               220.79 ,  220.83 ,  220.87 ,  220.88 ,  220.9  ,  220.97 ,
               221.05 ,  221.02 ,  221.01 ,  220.99 ,  221.04 ,  221.05 ,
               221.06 ,  221.07 ,  221.12 ,  221.06 ,  221.07 ,  221.03 ,
               221.01 ,  221.03 ,  221.03 ,  221.01 ,  221.02 ,  221.04 ,
               221.04 ,  221.07 ,  221.105,  221.1  ,  221.09 ,  221.08 ,
               221.07 ,  221.08 ,  221.03 ,  221.06 ,  221.1  ,  221.11 ,
               221.11 ,  221.18 ,  221.2  ,  221.34 ,  221.29 ,  221.235,
               221.22 ,  221.2  ,  221.21 ,  221.22 ,  221.19 ,  221.17 ,
               221.19 ,  221.13 ,  221.13 ,  221.12 ,  221.14 ,  221.11 ,
               221.165,  221.19 ,  221.18 ,  221.19 ,  221.18 ,  221.15 ,
               221.16 ,  221.155,  221.185,  221.19 ,  221.2  ,  221.2  ,
               221.16 ,  221.18 ,  221.16 ,  221.11 ,  221.07 ,  221.095,
               221.08 ,  221.08 ,  221.09 ,  221.11 ,  221.08 ,  221.08 ,
               221.1  ,  221.08 ,  221.11 ,  221.07 ,  221.11 ,  221.1  ,
               221.09 ,  221.07 ,  221.14 ,  221.12 ,  221.08 ,  221.09 ,
               221.05 ,  221.08 ,  221.065,  221.05 ,  221.06 ,  221.085,
               221.095,  221.07 ,  221.05 ,  221.09 ,  221.1  ,  221.145,
               221.12 ,  221.14 ,  221.12 ,  221.12 ,  221.12 ,  221.11 ,
               221.14 ,  221.15 ,  221.13 ,  221.12 ,  221.11 ,  221.105,
               221.105,  221.13 ,  221.14 ,  221.1  ,  221.105,  221.105,
               221.11 ,  221.13 ,  221.15 ,  221.11 ,  221.13 ,  221.08 ,
               221.11 ,  221.12 ,  221.12 ,  221.12 ,  221.13 ,  221.15 ,
               221.18 ,  221.21 ,  221.18 ,  221.15 ,  221.15 ,  221.15 ,
               221.15 ,  221.15 ,  221.13 ,  221.13 ,  221.16 ,  221.13 ,
               221.11 ,  221.12 ,  221.09 ,  221.07 ,  221.06 ,  221.04 ,
               221.06 ,  221.09 ,  221.07 ,  221.045,  221.   ,  220.99 ,
               220.985,  220.95 ,  221.   ,  221.01 ,  221.005,  220.99 ,
               221.03 ,  221.055,  221.06 ,  221.03 ,  221.03 ,  221.03 ,
               221.   ,  220.95 ,  220.96 ,  220.97 ,  220.965,  220.97 ,
               220.94 ,  220.93 ,  220.9  ,  220.9  ,  220.9  ,  220.91 ,
               220.94 ,  220.92 ,  220.94 ,  220.91 ,  220.92 ,  220.935,
               220.875,  220.89 ,  220.91 ,  220.92 ,  220.93 ,  220.93 ,
               220.91 ,  220.9  ,  220.89 ,  220.9  ,  220.9  ,  220.93 ,
               220.94 ,  220.92 ,  220.93 ,  220.88 ,  220.88 ,  220.86 ,
               220.9  ,  220.92 ,  220.85 ,  220.83 ,  220.83 ,  220.795,
               220.81 ,  220.78 ,  220.7  ,  220.69 ,  220.6  ,  220.58 ,
               220.61 ,  220.63 ,  220.68 ,  220.63 ,  220.63 ,  220.595,
               220.66 ,  220.645,  220.64 ,  220.6  ,  220.579,  220.53 ,
               220.53 ,  220.5  ,  220.42 ,  220.49 ,  220.49 ,  220.5  ,
               220.475,  220.405,  220.4  ,  220.425,  220.385,  220.37 ,
               220.49 ,  220.46 ,  220.45 ,  220.48 ,  220.51 ,  220.48 ]
        )

        plt.figure(figsize=(8,2))
```

```
        plt.plot(prices)
        plt.ylabel('price [USD]')
        plt.xlabel('Nov 28, 2016')
        plt.xticks([30, 90, 150, 210, 270, 330, 390],
                   ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
        plt.xlim([0, 390]);
```



Persistent random walk model

In this example, we choose to model the price evolution of SPY as a simple, well-known random walk model: the auto-regressive process of first order. We assume that subsequent log-return values r_t of SPY obey the following recursive instruction:

$$r_t = \rho_t \cdot r_{t-1} + \sqrt{1 - \rho^2} \cdot \sigma_t \cdot \epsilon_t$$

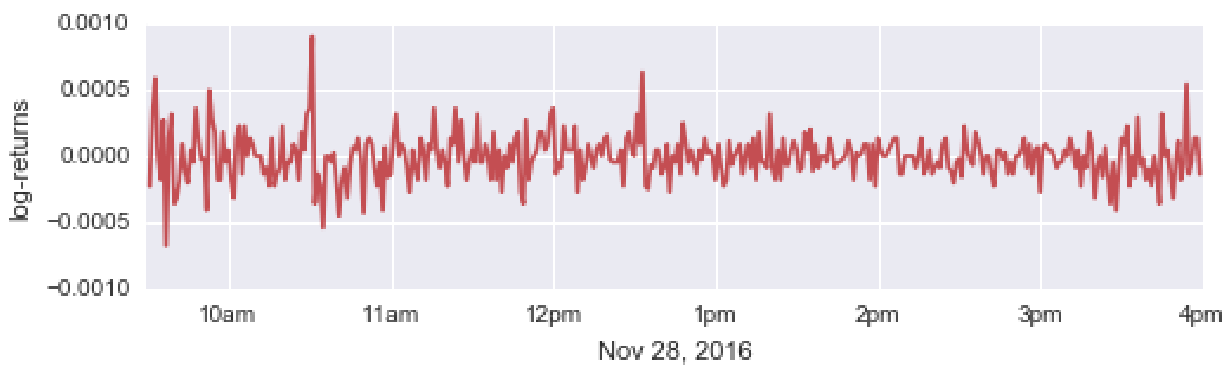
with the time-varying correlation coefficient ρ_t and the time-varying volatility parameter σ_t. Here, ϵ_t is drawn from a standard normal distribution and represents the driving noise of the process and the scaling factor $\sqrt{1 - \rho_t^2}$ makes sure that σ_t is the standard deviation of the process. In *bayesloop*, we define this observation model as follows:

```
bl.om.ScaledAR1('rho', bl.oint(-1, 1, 100), 'sigma', bl.oint(0, 0.006, 400))
```

This implementation of the correlated random walk model will be discussed in detail in the next section.

Looking at the log-returns of our example, we find that the magnitude of the fluctuations (i.e. the volatility) is higher after market open and before market close. While these variations happen quite gradually, a large peak around 10:30am (and possibly another one around 12:30pm) represents an abrupt price correction.

```
In [2]: logPrices = np.log(prices)
        logReturns = np.diff(logPrices)

        plt.figure(figsize=(8,2))
        plt.plot(np.arange(1, 390), logReturns, c='r')
        plt.ylabel('log-returns')
        plt.xlabel('Nov 28, 2016')
        plt.xticks([30, 90, 150, 210, 270, 330, 390],
                   ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
        plt.yticks([-0.001, -0.0005, 0, 0.0005, 0.001])
        plt.xlim([0, 390]);
```



Online study

bayesloop provides the class `OnlineStudy` to analyze on-going data streams and perform model selection for each new data point. In contrast to other Study types, more than just one transition model can be assigned to an `OnlineStudy` instance, using the method `addTransitionModel`. Here, we choose to add two distinct scenarios:

- **normal:** Both volatility and correlation of subsequent returns are allowed to vary gradually over time, to account for periods with above average trading activity after market open and before market close. This scenario therefore represents a smoothly running market.
- **chaotic:** This scenario assumes that we know nothing about the value of volatility or correlation. The probability that this scenario gets assigned in each minute therefore represents the probability that previously gathered knowledge about market dynamics cannot explain the last price movement.

By evaluating how likely the *chaotic* scenario explains each new minute close price of SPY compared to the *normal* scenario, we can identify specific events that lead to extreme fluctuations in intra-day trading.

First, we create a new instance of the `OnlineStudy` class and set the observation model introduced above. The keyword argument `storeHistory` is set to `True`, because we want to access the parameter estimates of all time steps afterwards, not only the estimates of the last time step.

```
In [3]: S = bl.OnlineStudy(storeHistory=True)

        L = bl.om.ScaledAR1('rho', bl.oint(-1, 1, 100),
                            'sigma', bl.oint(0, 0.006, 400))

        S.set(L)
```

```
+ Created new study.
  --> Hyper-study
  --> Online study
+ Observation model: Scaled autoregressive process of first order (AR1). Parameter(s): ['rho', 'sigma
```

Note: While the parameter `rho` is naturally constrained to the interval]-1, 1[, the parameter boundaries of `sigma` have to be specified by the user. Typically, one can review past log-return data and estimate the upper boundary as a multiple of the standard deviation of past data points.

Both scenarios for the dynamic market behavior are implemented via the method `add`. The *normal* case consists of a combined transition model that allows both volatility and correlation to perform a Gaussian random walk. As the standard deviation (magnitude) of the parameter fluctuations is a-priori unknown, we supply a wide range of possible values (`bl.cint(0, 1.5e-01, 15)` for `rho`, which corresponds to 15 equally spaced values within the closed interval [0, 0.15], and 50 equally spaced values within the interval [0, 1.5$ ·10^{-4}$] for `sigma`).

Since we have no prior assumptions about the standard deviations of the Gaussian random walks, we let *bayesloop* assign equal probability to all values. If one wants to analyze more than just one trading day, the (hyper-)parameter distributions from the end of one day can be used as the prior distribution for the next day! One might also want to suppress large variations of `rho` or `sigma` with an exponential prior, e.g.:

```
bl.tm.GaussianRandomWalk('s1', bl.cint(0, 1.5e-01, 15), target='rho',
                         prior=stats.Exponential('expon', 1./3.0e-02))
```

The *chaotic* case is implemented by the transition model `Independent`. This model sets a flat prior distribution for the parameters volatility and correlation in each time step. This way, previous knowledge about the parameters is not used when analyzing a new data point.

```
In [4]: T1 = bl.tm.CombinedTransitionModel(
                 bl.tm.GaussianRandomWalk('s1', bl.cint(0, 1.5e-01, 15), target='rho'),
                 bl.tm.GaussianRandomWalk('s2', bl.cint(0, 1.5e-04, 50), target='sigma')
             )

        T2 = bl.tm.Independent()

        S.add('normal', T1)
        S.add('chaotic', T2)
```

```
+ Added transition model: normal (750 combination(s) of the following hyper-parameters: ['s1', 's2'])
+ Added transition model: chaotic (no hyper-parameters)
```

Before any data points are passed to the study instance, we further provide prior probabilities for the two scenarios. We expect about one news announcement containing unexpected information per day and set a prior probability of $1/390$ for the *chaotic* scenario (one normal trading day consists of 390 trading minutes).

```
In [5]: S.setTransitionModelPrior([389/390., 1/390.])
```

```
+ Set custom transition model prior.
```

Finally, we can supply log-return values to the study instance, data point by data point. We use the `step` method to infer new parameter estimates and the updated probabilities of the two scenarios. Note that in this example, we use a `for` loop to feed all data points to the algorithm because all data points are already available. In a real application of the `OnlineStudy` class, one can supply each new data point as it becomes available and analyze it in real-time.

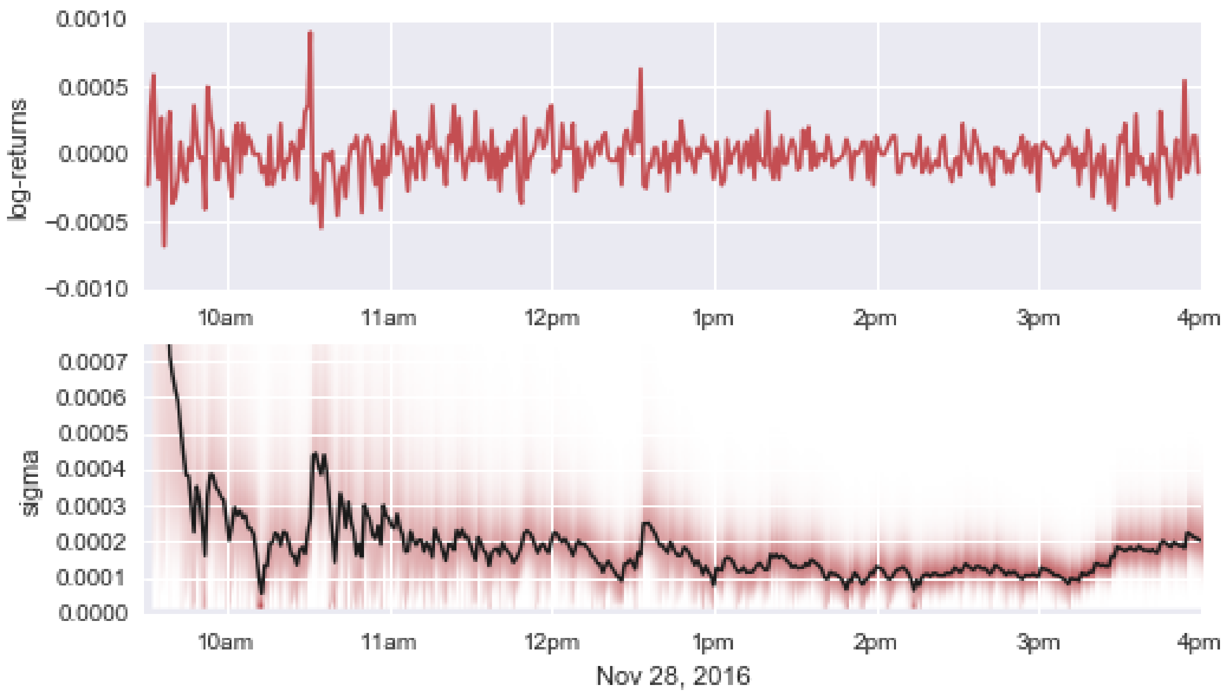
```
In [6]: for r in tqdm_notebook(logReturns):
            S.step(r)
```

```
+ Start model fit
+ Not enough data points to start analysis. Will wait for more data.
    + Set uniform prior with parameter boundaries.
```

Volatility spikes

Before we analyze how the probability values of our two market scenarios change over time, we check whether the inferred temporal evolution of the time-varying parameters is realistic. Below, the log-returns are displayed together with the inferred marginal distribution (shaded red) and mean value (black line) of the volatility parameter, using the method `plotParameterEvolution`.

```
In [7]: plt.figure(figsize=(8, 4.5))

        # data plot
        plt.subplot(211)
        plt.plot(np.arange(1, 390), logReturns, c='r')

        plt.ylabel('log-returns')
        plt.xticks([30, 90, 150, 210, 270, 330, 390],
                   ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
        plt.yticks([-0.001, -0.0005, 0, 0.0005, 0.001])
        plt.xlim([0, 390])

        # parameter plot
        plt.subplot(212)
        S.plot('sigma', color='r')

        plt.xticks([28, 88, 148, 208, 268, 328, 388],
                   ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
        plt.xlabel('Nov 28, 2016')
        plt.ylim([0, 0.00075])
        plt.xlim([-2, 388]);
```



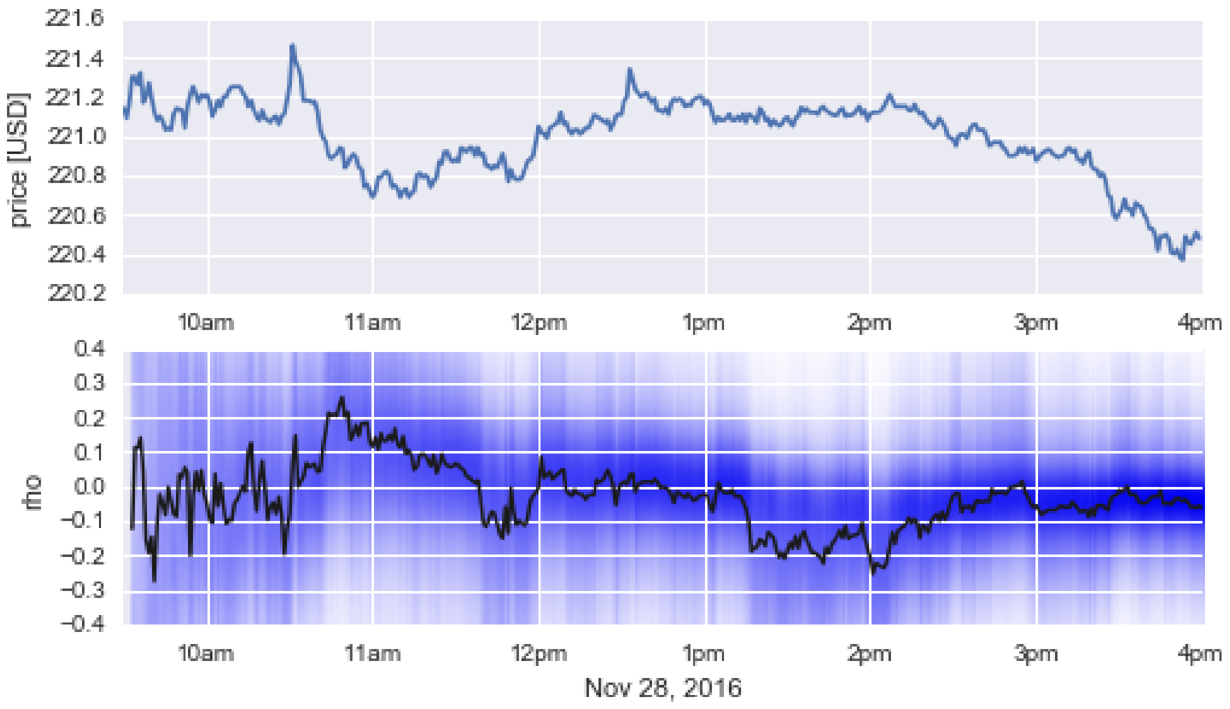
Note that the volatility estimates of the first few trading minutes are not as accurate as later ones, as we initialize the algorithm with a non-informative prior distribution. One could of course provide a custom prior distribution as a more realistic starting point. Despite this *fade-in* period, the period of increased volatility after market open is captured nicely, as well as the (more subtle) increase in volatility during the last 45 minutes of the trading day. Large individual log-return values also result in an *volatility spikes* (around 10:30am and more subtle around 12:30pm).

Islands of stability

The persistent random walk model does not only provide information about the magnitude of price fluctuations, but further quantifies whether subsequent log-return values are correlated. A positive correlation coefficient indicates diverging price movements, as a price increase is more likely followed by another increase, compared to a decrease. In contrast, a negative correlation coefficient indicates *islands of stability*, i.e. trading periods during which prices do not diffuse randomly (as with a corr. coeff. of zero). Below, we plot the price evolution of SPY on November 28, together with the inferred marginal distribution (shaded blue) and the corresponding mean value (black line) of the time-varying correlation coefficient.

```
In [8]: plt.figure(figsize=(8, 4.5))
```

```
        # data plot
        plt.subplot(211)
        plt.plot(prices)

        plt.ylabel('price [USD]')
        plt.xticks([30, 90, 150, 210, 270, 330, 390],
                   ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
        plt.xlim([0, 390])

        # parameter plot
        plt.subplot(212)
        S.plot('rho', color='#0000FF')

        plt.xticks([28, 88, 148, 208, 268, 328, 388],
                   ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
        plt.xlabel('Nov 28, 2016')
        plt.ylim([-0.4, 0.4])
        plt.xlim([-2, 388]);
```



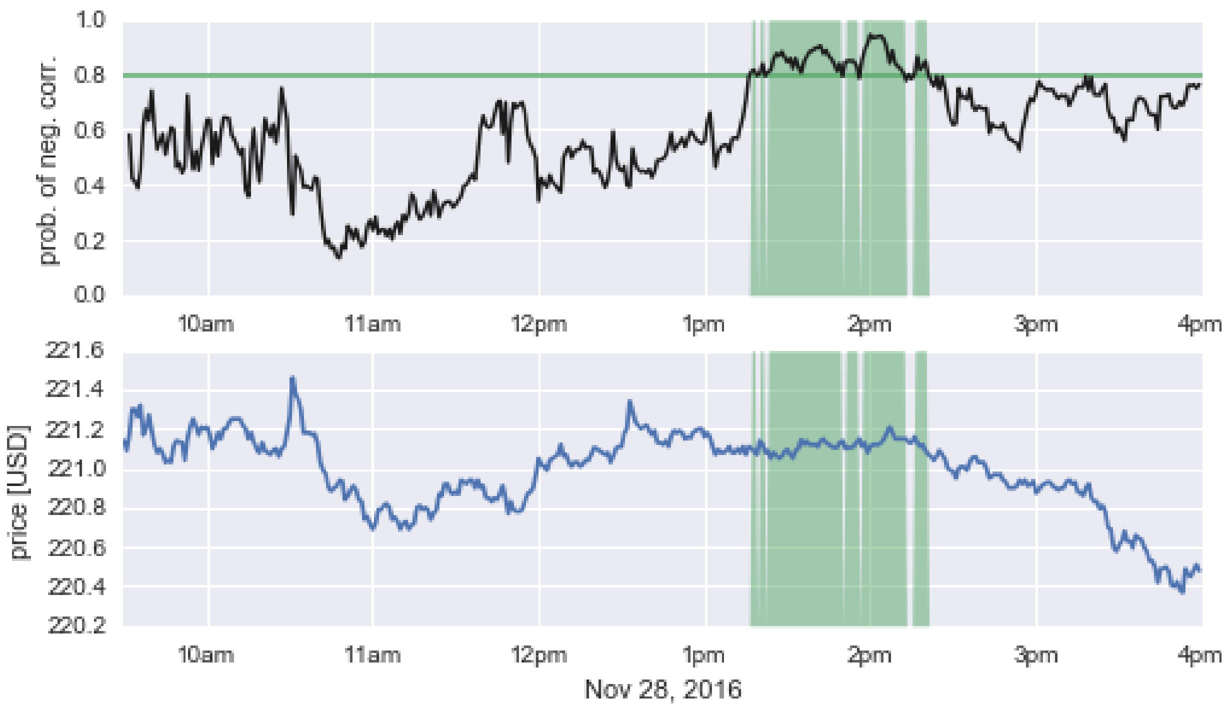
As a correlation coefficient that deviates significantly from zero would be immediately exploitable to predict future price movements, we mostly find correlation values near zero (in accordance with the efficient market hypothesis). However, between 1:15pm and 2:15pm, we find a short period of negative correlation with a value around -0.2. During this period, subsequent price movements tend to cancel each other out, resulting in an unusually strong price stability.

Using the `Parser` sub-module of *bayesloop*, we can evaluate the probability that subsequent return values are negatively correlated. In the figure below, we tag all time steps with a probability for `rho < 0` of 80% or higher and find that this indicator nicely identifies the period of increased market stability!

Note: The arbitrary threshold of 80% for our market indicator is of course chosen with hindsight in this example. In a real application, more than one trading day of data needs to be analyzed to create robust indicators!

```
In [9]: # extract parameter grid values (rho) and corresponding prob. values (p)
        rho, p = S.getParameterDistributions('rho')
```

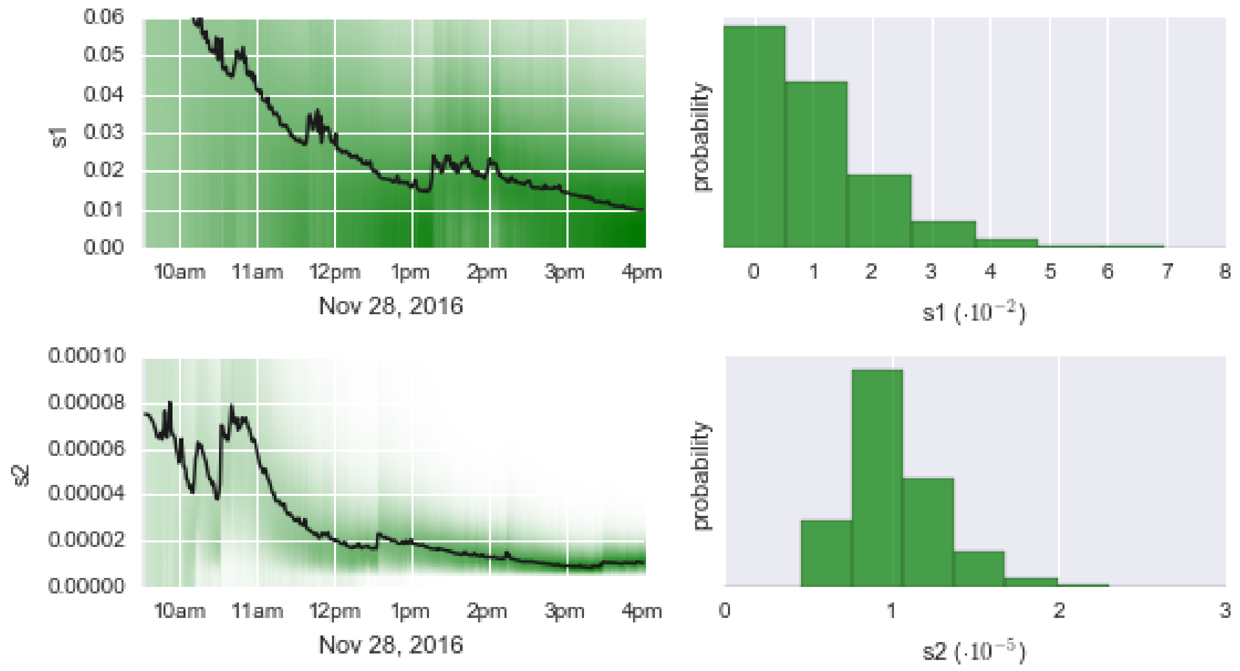
```
# evaluate Prob.(rho < 0) for all time steps
P = bl.Parser(S)
p_neg_rho = np.array([P('rho < 0.', t=t, silent=True) for t in range(1, 389)])

# plotting
plt.figure(figsize=(8, 4.5))
plt.subplot(211)
plt.axhline(y=0.8, lw=1.5, c='g')
plt.plot(p_neg_rho, lw=1.5, c='k')
plt.fill_between(np.arange(len(p_neg_rho)), 0, p_neg_rho > 0.8, lw=0, facecolor='g', alpha=0

plt.xticks([28, 88, 148, 208, 268, 328, 388],
           ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
plt.ylabel('prob. of neg. corr.')
plt.xlim([-2, 388])

plt.subplot(212)
plt.plot(prices)
plt.fill_between(np.arange(2, len(p_neg_rho)+2), 220.2, 220.2 + (p_neg_rho > 0.8)*1.4, lw=0,
plt.ylabel('price [USD]')
plt.xticks([30, 90, 150, 210, 270, 330, 390],
           ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
plt.xlim([0, 390])
plt.ylim([220.2, 221.6])
plt.xlabel('Nov 28, 2016');
```



Automatic tuning

One major advantage of the `OnlineStudy` class is that it not only infers the time-varying parameters of the *low-level* correlated random walk (the observation model `ScaledAR1`), but further infers the magnitude (the standard deviation of the transition model `GaussianRandomWalk`) of the parameter fluctuations and thereby tunes the parameter dynamics as new data arrives. As we can see below (left sub-figures), both magnitudes - one for `rho` and one for sigma - start off at a high level. This is due to our choice of a uniform prior, assigning equal probability to all hyper-parameter values before seeing any data. Over time, the algorithm *learns* that the true parameter fluctuations are less severe than previously assumed and adjusts the hyper-parameters accordingly. This newly gained information, summarized in the hyper-parameter distributions of the last time step (right sub-figures), could then represent the prior distribution for the next trading day.

```
In [10]: plt.figure(figsize=(8, 4.5))
         plt.subplot(221)
         S.plot('s1', color='green')
         plt.xticks([28, 88, 148, 208, 268, 328, 388],
                    ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
         plt.xlabel('Nov 28, 2016')
         plt.xlim([-2, 388])
         plt.ylim([0, 0.06])

         plt.subplot(222)
         S.plot('s1', t=388, facecolor='green', alpha=0.7)
         plt.yticks([])
         plt.xticks([0, 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08], ['0', '1', '2', '3', '4', '
         plt.xlabel('s1 ($\cdot 10^{-2}$)')
         plt.xlim([-0.005, 0.08])

         plt.subplot(223)
         S.plot('s2', color='green')
         plt.xticks([28, 88, 148, 208, 268, 328, 388],
                    ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
         plt.xlabel('Nov 28, 2016')
         plt.xlim([-2, 388])
         plt.ylim([0, 0.0001])

         plt.subplot(224)
         S.plot('s2', t=388, facecolor='green', alpha=0.7)
         plt.yticks([])
         plt.xticks([0, 0.00001, 0.00002, 0.00003], ['0', '1', '2', '3'])
         plt.xlabel('s2 ($\cdot 10^{-5}$)')
         plt.xlim([0, 0.00003])

         plt.tight_layout()
```

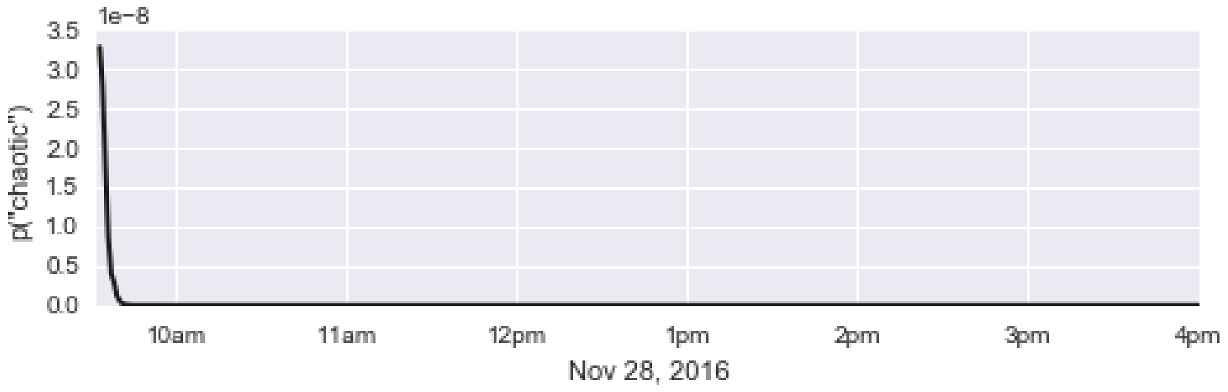


Real-time model selection

Finally, we investigate which of our two market scenarios - *normal* vs. *chaotic* - can explain the price movements best. Using the method `plot('chaotic')`, we obtain the probability values for the *chaotic* scenario compared to the *normal* scenario, **with respect to all past data points**:

```
In [11]: plt.figure(figsize=(8, 2))
         S.plot('chaotic', lw=2, c='k')
         plt.xticks([28, 88, 148, 208, 268, 328, 388],
                    ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
         plt.xlabel('Nov 28, 2016')
         plt.xlim([0, 388])
         plt.ylabel('p("chaotic")')
```

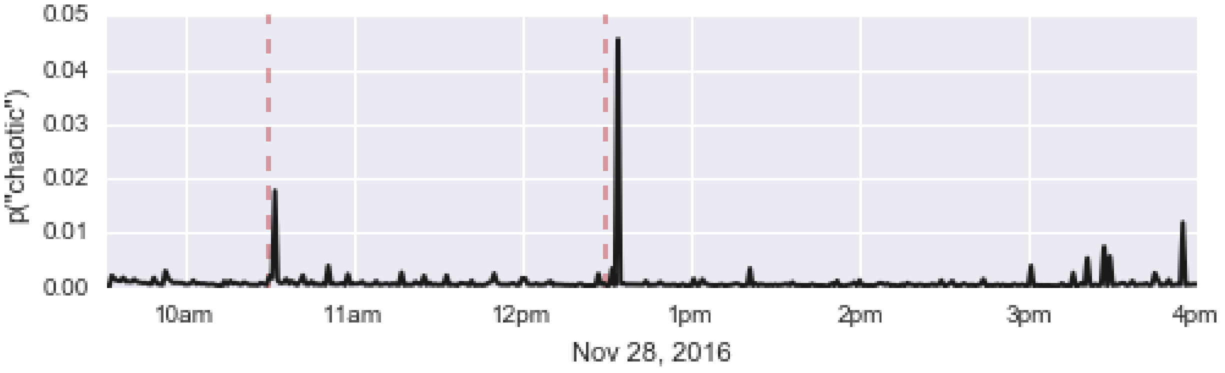
```
Out[11]: <matplotlib.text.Text at 0xefc86a0>
```



As expected, the probability that the *chaotic* scenario can explain all past log-return values at a given point in time quickly falls off to practically zero. Indeed, a correlated random walk with slowly changing volatility and correlation of subsequent returns is better suited to describe the price fluctuations of SPY **in the majority of time steps**.

However, we may also ask for the probability that each **individual** log-return value is produced by either of the two market scenarios by using the keyword argument `local=True`:

```
In [12]: plt.figure(figsize=(8, 2))
         S.plot('chaotic', local=True, c='k', lw=2)
         plt.xticks([28, 88, 148, 208, 268, 328, 388],
                    ['10am', '11am', '12pm', '1pm', '2pm', '3pm', '4pm'])
         plt.xlabel('Nov 28, 2016')
         plt.xlim([0, 388])
         plt.ylabel('p("chaotic")')
         plt.axvline(58, 0, 1, zorder=1, c='r', lw=1.5, ls='dashed', alpha=0.7)
         plt.axvline(178, 0, 1, zorder=1, c='r', lw=1.5, ls='dashed', alpha=0.7);
```



Here, we find clear peaks indicating an increased probability for the *chaotic* scenario, i.e. that previously gained information about the market dynamics has become useless. Lets assume that we are concerned about market behavior as soon as there is at least a 1% risk that *normal* market dynamics can not describe the current price movement. This leaves us with three distinct events in the following time steps:

```
In [13]: p = S.getTransitionModelProbabilities('chaotic', local=True)
         np.argwhere(p > 0.01)
```

```
Out[13]: array([[ 59],
                [181],
                [382]], dtype=int64)
```

These time steps translate to the following trading minutes: **10:31am, 12:33pm and 3:54pm**.

The first peak at 10:31am directly follows the release of the Texas Manufacturing Outlook Survey of the Dallas Fed. The publication of this set of economic indicators has already been announced by financial news pages before the market opened. While the title of the survey (*"Texas Manufacturing Activity Continues to Expand"*) indicated good news, investors reacted sceptically, possibly due to negative readings in both the *new orders index* and *growth rate of orders index* (c.f. this article).

Note: This first peak would be more pronounced if we had supplied a prior distribution that constrains strong parameter fluctuations!

The underlying trigger for the second peak, shortly after 12:30pm, remains unknown. No major macroeconomic indicators were published at that time, at least according to some economic news sites (see e.g. nytimes.com or liveindex.org).

The last peak at 3:54pm is likely connected to the imminent market close at 4pm. To protect themselves from unexpected news after trading hours, market participants often close their open positions before market close, generally leading to an increased volatility. If large market participants follow this behavior, price corrections may no longer be explained by *normal* market dynamics.

This example has shown that *bayesloop*'s `OnlineStudy` class can identify periods of increased volatility as well as periods of increased price stability (accompanied by a negative correlation of subsequent returns), and further automatically tunes its current assumptions about market dynamics. Finally, we have demonstrated that *bayesloop* serves as a viable tool to detect *"anomalous"* price corrections, triggered by large market participants or news announcements, in real-time.

1.4 API Reference

1.4.1 Study types

`Study`([silent])	Fits with fixed hyper-parameters and hyper-parameter optimization.
`HyperStudy`([silent])	Infers hyper-parameter distributions.
`ChangepointStudy`([silent])	Infers change-points and structural breaks.
`OnlineStudy`([storeHistory, silent])	Enables model selection for online data streams.

Note: These Study classes are imported directly into the module namespace for convenient access.

```
import bayesloop as bl
S = bl.Study()
```

In bayesloop, each new data study is handled by an instance of a `Study`-class. In this way, all data, the inference results and the appropriate post-processing routines are stored in one object that can be accessed conveniently or stored in a file. Apart from the basic `Study` class, there exist a number of specialized classes that extend the basic fit method, for example to infer the full distribution of hyper-parameters or to apply model selection to on-line data streams.

class `bayesloop.core.`**`ChangepointStudy`**(*silent=False*)

Infers change-points and structural breaks. This class builds on the HyperStudy-class and the change-point transition model to perform a series of analyses with varying change point times. It subsequently computes the average model from all possible change points and creates a probability distribution of change point times. It supports any number of change-points and arbitarily combined models.

`fit`(*forwardOnly=False*, *evidenceOnly=False*, *silent=False*, *nJobs=1*)

This method over-rides the corresponding method of the HyperStudy-class. It runs the algorithm for all possible combinations of change-points (and possible scans a range of values for other hyper-parameters). The posterior sequence represents the average model of all analyses. Posterior mean values are computed from this average model.

Parameters

- **`forwardOnly`** (*`bool`*) – If set to True, the fitting process is terminated after the forward pass. The resulting posterior distributions are so-called "filtering distributions" which - at each time step - only incorporate the information of past data points. This option thus emulates an online analysis.
- **`evidenceOnly`** (*`bool`*) – If set to True, only forward pass is run and evidence is calculated. In contrast to the forwardOnly option, no posterior mean values are computed and no posterior distributions are stored.
- **`silent`** (*`bool`*) – If set to True, reduced output is generated by the fitting method.
- **`nJobs`** (*`int`*) – Number of processes to employ. Multiprocessing is based on the 'pathos' module.

getDD (*names*, *plot=False*, ***kwargs*)
See `ChangepointStudy.getDurationDistribution()`.

getDurationDistribution (*names*, *plot=False*, ***kwargs*)
Computes the distribution of the number of time steps between two change/break-points. This distribution of duration is created from the joint distribution of the two specified change/break-points.

Parameters

- **names** (`list`) – List of two parameter names of change/break-points to display (first and second model parameter)
- **plot** (`bool`) – If True, a bar chart of the distribution is created
- ****kwargs** – All further keyword-arguments are passed to the bar-plot (see matplotlib documentation)

Returns

The first array contains the number of time steps, the second one the corresponding probability values.

Return type ndarray, ndarray

class bayesloop.core.**HyperStudy** (*silent=False*)
Infers hyper-parameter distributions. This class serves as an extension to the basic Study class and allows to compute the distribution of hyper-parameters of a given transition model. For further information, see the documentation of the fit-method of this class.

fit (*forwardOnly=False*, *evidenceOnly=False*, *silent=False*, *nJobs=1*, *customHyperGrid=False*)
This method over-rides the according method of the Study-class. It runs the algorithm for equally spaced hyper- parameter values as defined by the variable 'hyperGrid'. The posterior sequence represents the average model of all analyses. Posterior mean values are computed from this average model.

Parameters

- **forwardOnly** (`bool`) – If set to True, the fitting process is terminated after the forward pass. The resulting posterior distributions are so-called "filtering distributions" which - at each time step - only incorporate the information of past data points. This option thus emulates an online analysis.
- **evidenceOnly** (`bool`) – If set to True, only forward pass is run and evidence is calculated. In contrast to the forwardOnly option, no posterior mean values are computed and no posterior distributions are stored.
- **silent** (`bool`) – If set to true, reduced output is created by this method.
- **nJobs** (`int`) – Number of processes to employ. Multiprocessing is based on the 'pathos' module.
- **customHyperGrid** (`bool`) – If set to true, the method "_createHyperGrid" is not called before starting the fit. This is used by the class "ChangepointStudy", which employs a custom version of "_createHyperGrid".

getHPD (*name*, *plot=False*, ***kwargs*)
See `HyperStudy.getHyperParameterDistribution()`.

getHyperParameterDistribution (*name*, *plot=False*, ***kwargs*)
Computes marginal hyper-parameter distribution of a single hyper-parameter in a HyperStudy fit.

Parameters

- **name** (`str`) – Name of the hyper-parameter to display (first model hyper-parameter)

- **plot** (*bool*) – If True, a bar chart of the distribution is created
- ****kwargs** – All further keyword-arguments are passed to the bar-plot (see matplotlib documentation)

Returns

The first array contains the hyper-parameter values, the second one the corresponding probability values

Return type ndarray, ndarray

getJHPD(*names*, *plot=False*, *figure=None*, *subplot=111*, ***kwargs*)

See `HyperStudy.getJointHyperParameterDistribution()`.

getJointHyperParameterDistribution(*names*, *plot=False*, *figure=None*, *subplot=111*, ***kwargs*)

Computes the joint distribution of two hyper-parameters of a HyperStudy and optionally creates a 3D bar chart. Note that the 3D plot can only be included in an existing plot by passing a figure object and subplot specification.

Parameters

- **names** (*list*) – List of two hyper-parameter names to display
- **plot** (*bool*) – If True, a 3D-bar chart of the distribution is created
- **figure** – In case the plot is supposed to be part of an existing figure, it can be passed to the method. By default, a new figure is created.
- **subplot** – Characterization of subplot alignment, as in matplotlib. Default: 111
- ****kwargs** – all further keyword-arguments are passed to the bar3d-plot (see matplotlib documentation)

Returns

The first and second array contains the hyper-parameter values, the third one the corresponding probability values

Return type ndarray, ndarray, ndarray

optimize(**args*, ***kwargs*)

Uses the COBYLA minimization algorithm from SciPy to perform a maximization of the log-evidence with respect to all hyper-parameters (the parameters of the transition model) of a time seris model. The starting values are the values set by the user when defining the transition model.

For the optimization, only the log-evidence is computed and no parameter distributions are stored. When a local maximum is found, the parameter distribution is computed based on the optimal values for the hyper-parameters.

Parameters

- **parameterList** (*list*) – List of hyper-parameter names to optimize. For nested transition models with multiple, identical hyper-parameter names, the sub-model index can be provided. By default, all hyper-parameters are optimized.
- **forwardOnly** (*bool*) – If set to True, the fitting process is terminated after the forward pass. The resulting posterior distributions are so-called "filtering distributions" which - at each time step - only incorporate the information of past data points.
- **- All other keyword parameters are passed to the 'minimize' routine of scipy.optimize.** (***kwargs*) –

plot(*name*, ***kwargs*)

Convenience method to plot the temporal evolution of observation model parameters, the distribution of a parameter at a specific time step, or the distribution of a hyper-parameter.

Parameters

- **name** (*str*) – name of the (hyper-)parameter to display
- **color** – color from which a light colormap is created (for parameter evolution only)
- **gamma** (*float*) – exponent for gamma correction of the displayed marginal distribution; default: 0.5 (for parameter evolution only)
- **t** – Time step/stamp for which the parameter distribution is evaluated
- **density** (*bool*) – If true, probability density is plotted; if false, probability values. Note: Only availble for parameters, not hyper-parameters.
- **kwargs** – all further keyword-arguments are passed to the axes object of the plot

class bayesloop.core.**OnlineStudy**(*storeHistory=False*, *silent=False*)

Enables model selection for online data streams. This class builds on the Study-class and features a step-method to include new data points in the study as they arrive from a data stream. This online-analysis is performed in an forward-only way, resulting in filtering-distributions only. In contrast to a normal study, however, one can add multiple transition models to account for different types of parameter dynamics (similar to a Hyper study). The Online study then computes the probability distribution over all transition models for each new data point (or all past data points), enabling real-time model selection.

Parameters **storeHistory** (*bool*) – If true, posterior distributions and their mean values, as well as hyper-posterior distributions are stored for all time steps.

add(*name*, *transitionModel*)

See *OnlineStudy.addTransitionModel()*.

addTM(*name*, *transitionModel*)

See *OnlineStudy.addTransitionModel()*.

addTransitionModel(*name*, *transitionModel*)

Adds a transition model to the list of transition models that are fitted in each time step. Note that a list of hyper-parameter values can be supplied.

Parameters

- **name** (*str*) – a custom name for this transition model to identify it in post-processing methods
- **transitionModel** – instance of a transition model class.

Example

Here, 'S' denotes the OnlineStudy instance. In the first example, we assume a Poisson observation model and add a Gaussian random walk with varying standard deviation to the rate parameter 'lambda':

S.setObservationModel(bl.om.Poisson('lambda', bl.oint(0, 6, 1000))) S.addTransitionModel(bl.tm.GaussianRandomWalk('sigma', [0, 0.1, 0.2, 0.3], target='lambda'))

fit(**args*, ***kwargs*)

This method over-rides the according method of the Study-class. It runs the algorithm for equally spaced hyper- parameter values as defined by the variable 'hyperGrid'. The posterior sequence represents the average model of all analyses. Posterior mean values are computed from this average model.

Parameters

- **forwardOnly** (*bool*) – If set to True, the fitting process is terminated after the forward pass. The resulting posterior distributions are so-called "filtering distributions" which - at each time step - only incorporate the information of past data points. This option thus emulates an online analysis.
- **evidenceOnly** (*bool*) – If set to True, only forward pass is run and evidence is calculated. In contrast to the forwardOnly option, no posterior mean values are computed and no posterior distributions are stored.
- **silent** (*bool*) – If set to true, reduced output is created by this method.
- **nJobs** (*int*) – Number of processes to employ. Multiprocessing is based on the 'pathos' module.
- **customHyperGrid** (*bool*) – If set to true, the method "_createHyperGrid" is not called before starting the fit. This is used by the class "ChangepointStudy", which employs a custom version of "_createHyperGrid".

getCHPD (*name*, *plot=False*, ***kwargs*)
See :meth:'.OnlineStudy.getCurrentHyperParameterDistribution.

getCPD (*name*, *plot=False*, *density=True*, ***kwargs*)
See `OnlineStudy.getCurrentParameterDistribution()`.

getCTMD (*local=False*)
See `OnlineStudy.getCurrentTransitionModelDistribution()`.

getCTMP (*transitionModel*, *local=False*)
See :meth:'.OnlineStudy.getCurrentTransitionModelProbability.

getCurrentHyperParameterDistribution (*name*, *plot=False*, ***kwargs*)
Computes marginal hyper-parameter distribution of a single hyper-parameter at the current time step in an OnlineStudy fit.

Parameters

- **name** (*str*) – hyper-parameter name
- **plot** (*bool*) – If True, a bar chart of the distribution is created
- ****kwargs** – All further keyword-arguments are passed to the bar-plot (see matplotlib documentation)

Returns

The first array contains the hyper-parameter values, the second one the corresponding probability values

Return type ndarray, ndarray

getCurrentParameterDistribution (*name*, *plot=False*, *density=True*, ***kwargs*)
Compute the current marginal parameter distribution.

Parameters

- **name** (*str*) – Name of the parameter to display
- **plot** (*bool*) – If True, a plot of the distribution is created
- **density** (*bool*) – If true, probability density is plotted; if false, probability values.
- ****kwargs** – All further keyword-arguments are passed to the plot (see matplotlib documentation)

Returns

The first array contains the parameter values, the second one the corresponding probability values

Return type ndarray, ndarray

`getCurrentParameterMeanValue`(*name*)
Returns the posterior mean value for a given parameter of the observation model.

Parameters **`name`** (*`str`*) – Name of the parameter

Returns posterior mean value

Return type float

`getCurrentTransitionModelDistribution`(*local=False*)
Returns the current probabilities for each transition model defined in the Online Study.

Parameters **`local`** (*`bool`*) – If true, transition model distribution taking into account only the last data point is returned.

Returns Arrays of transition model names and normalized probabilities.

Return type ndarray, ndarray

`getCurrentTransitionModelProbability`(*transitionModel*, *local=False*)
Returns the current posterior probability for a specified transition model.

Parameters

- **`transitionModel`** (*`str`*) – Name of the transition model
- **`local`** (*`bool`*) – If true, transition model probability taking into account only the last data point is returned.

Returns Posterior probability value for the specified transition model

Return type float

`getHPD`(*t*, *name*, *plot=False*, ***kwargs*)
See :meth:‘.OnlineStudy.getHyperParameterDistribution.

`getHPDs`(*name*)
See :meth:‘.OnlineStudy.getHyperParameterDistributions.

`getHyperParameterDistribution`(*t*, *name*, *plot=False*, ***kwargs*)
Computes marginal hyper-parameter distribution of a single hyper-parameter at a specific time step in an OnlineStudy fit. Only available if Online Study is created with flag ‘storeHistory=True’.

Parameters

- **`t`** (*`int`*) – Time step at which to compute distribution, or ‘avg’ for time-averaged distribution
- **`name`** (*`str`*) – hyper-parameter name
- **`plot`** (*`bool`*) – If True, a bar chart of the distribution is created
- **`**kwargs`** – All further keyword-arguments are passed to the bar-plot (see matplotlib documentation)

Returns

The first array contains the hyper-parameter values, the second one the corresponding probability values

Return type ndarray, ndarray

`getHyperParameterDistributions`(*name*)

Computes marginal hyper-parameter distributions of a single hyper-parameter for all time steps in an OnlineStudy fit. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters **`name`** (*`str`*) – hyper-parameter name

Returns

The first array contains the hyper-parameter values, the second one the corresponding probability values (first axis is time).

Return type ndarray, ndarray

`getHyperParameterMeanValue`(*t*, *name*)

Computes the mean value of the joint hyper-parameter distribution for a given hyper-parameter at a given time step. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **`t`** (*`int`*) – Time step at which to compute distribution
- **`name`** (*`str`*) – name of hyper-parameter

Returns Array containing the mean values of all hyper-parameters of the given transition model

Return type ndarray

`getHyperParameterMeanValues`(*name*)

Computes the sequence of mean value of the joint hyper-parameter distribution for a given hyper-parameter for all time steps. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters **`name`** (*`str`*) – name of hyper-parameter

Returns Array containing the sequences of mean values of the given hyper-parameter

Return type ndarray

`getJointHyperParameterDistribution`(*names*, *plot=False*, *figure=None*, *subplot=111*, ***kwargs*)

Computes the joint distribution of two hyper-parameters of a HyperStudy and optionally creates a 3D bar chart. Note that the 3D plot can only be included in an existing plot by passing a figure object and subplot specification.

Parameters

- **`names`** (*`list`*) – List of two hyper-parameter names to display
- **`plot`** (*`bool`*) – If True, a 3D-bar chart of the distribution is created
- **`figure`** – In case the plot is supposed to be part of an existing figure, it can be passed to the method. By default, a new figure is created.
- **`subplot`** – Characterization of subplot alignment, as in matplotlib. Default: 111
- **`**kwargs`** – all further keyword-arguments are passed to the bar3d-plot (see matplotlib documentation)

Returns

The first and second array contains the hyper-parameter values, the third one the corresponding probability values

Return type ndarray, ndarray, ndarray

`getPD`(*t*, *name*, *plot=False*, *density=True*, ***kwargs*)

See `OnlineStudy.getParameterDistribution()`.

`getPDs`(*name*, *plot=False*, *density=True*, ***kwargs*)
See `OnlineStudy.getParameterDistributions()`.

`getParameterDistribution`(*t*, *name*, *plot=False*, *density=True*, ***kwargs*)
Compute the marginal parameter distribution at a given time step. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **`t`** (`int, float`) – Time step/stamp for which the parameter distribution is evaluated
- **`name`** (`str`) – Name of the parameter to display
- **`plot`** (`bool`) – If True, a plot of the distribution is created
- **`density`** (`bool`) – If true, probability density is plotted; if false, probability values.
- **`**kwargs`** – All further keyword-arguments are passed to the plot (see matplotlib documentation)

Returns

The first array contains the parameter values, the second one the corresponding probability values

Return type ndarray, ndarray

`getParameterDistributions`(*name*, *plot=False*, *density=True*, ***kwargs*)
Computes the time series of marginal posterior distributions with respect to a given model parameter. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **`name`** (`str`) – Name of the parameter to display
- **`plot`** (`bool`) – If True, a plot of the series of distributions is created (density map)
- **`density`** (`bool`) – If true, probability density is plotted; if false, probability values.
- **`**kwargs`** – All further keyword-arguments are passed to the plot (see matplotlib documentation)

Returns

The first array contains the parameter values, the second one the sequence of corresponding posterior distributions.

Return type ndarray, ndarray

`getParameterMeanValue`(*t*, *name*)
Returns the posterior mean value for a given parameter of the observation model at a specified time step. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **`t`** (`int`) – Time step at which to compute parameter mean value
- **`name`** (`str`) – Name of the parameter

Returns posterior mean value

Return type float

`getParameterMeanValues`(*name*)
Returns the posterior mean value for a given parameter of the observation model for all time steps. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters **name** (*str*) – Name of the parameter

Returns posterior mean values

Return type ndarray

getTMPs(*transitionModel*, *local=False*)
See :meth:'.OnlineStudy.getTransitionModelProbabilities.

getTransitionModelDistributions(*local=False*)
The transition model distribution contains posterior probability values for all transition models included in the online study. This distribution is available for all time steps analyzed. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters **local** (*bool*) – If true, transition model distributions taking into account only the data point at the corresponding time step is returned.

Returns

Arrays containing the names and posterior probability values for all transition models included in the online study for all time steps analyzed

Return type ndarray, ndarray

getTransitionModelProbabilities(*transitionModel*, *local=False*)
Returns posterior probability values for a specified transition model. This distribution is available for all time steps analyzed. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **transitionModel** (*str*) – Name of the transition model
- **local** (*bool*) – If true, transition model probabilities taking into account only the data point at the corresponding time step is returned.

Returns

Array containing the posterior probability values for the specified transition model for all time steps analyzed

transitionModel(str): Name of the transition model

Return type ndarray

plot(*name*, ***kwargs*)
Convenience method to plot the temporal evolution of (hyper-)parameters, the distribution of a (hyper-)parameter at a specific time step, or the temporal evolution of the probability of a transition model.

Parameters

- **name** (*str*) – name of the (hyper-)parameter or transition model to display
- **color** – color from which a light colormap is created (for (hyper-)parameter evolution only)
- **gamma** (*float*) – exponent for gamma correction of the displayed marginal distribution; default: 0.5 (for (hyper-)parameter evolution only)
- **t** – Time step/stamp for which the parameter distribution is evaluated
- **density** (*bool*) – If true, probability density is plotted; if false, probability values. Note: Only availble for parameters, not hyper-parameters.
- **local** (*bool*) – If true, transition model probabilities taking into account only the data point at the corresponding time step is returned

- **kwargs** – all further keyword-arguments are passed to the axes object of the plot

plotHyperParameterEvolution(*name*, *color='b'*, *gamma=0.5*, ***kwargs*)
Plot method to display a series of marginal posterior distributions corresponding to a single model parameter. This method includes the removal of plotting artefacts, gamma correction as well as an overlay of the posterior mean values. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **name** (*str*) – hyper-parameter name
- **color** – color from which a light colormap is created
- **gamma** (*float*) – exponent for gamma correction of the displayed marginal distribution; default: 0.5
- **kwargs** – all further keyword-arguments are passed to the plot of the posterior mean values

plotParameterEvolution(*name*, *color='b'*, *gamma=0.5*, ***kwargs*)
Plots a series of marginal posterior distributions corresponding to a single model parameter, together with the posterior mean values. Only available if Online Study is created with flag 'storeHistory=True'.

Parameters

- **name** (*str*) – Name of the parameter to display
- **color** – color from which a light colormap is created
- **gamma** (*float*) – exponent for gamma correction of the displayed marginal distribution; default: 0.5
- **kwargs** – all further keyword-arguments are passed to the plot of the posterior mean values

setTransitionModelPrior(*transitionModelPrior*, *silent=False*)
Sets prior probabilities for transition models added to the online study instance.

Parameters

- **transitionModelPrior** – List/Array of probabilities, one for each transition model. If the list does not sum to one, it will be re-normalised.
- **silent** – If true, no output is generated by this method.

step(*dataPoint*)
Update the current parameter distribution by adding a new data point to the data set.

Parameters **dataPoint** (*float, int, ndarray*) – Float, int, or 1D-array of those (for multidimensional data).

class bayesloop.core.**Study**(*silent=False*)
Fits with fixed hyper-parameters and hyper-parameter optimization. This class implements a forward-backward-algorithm for analyzing time series data using hierarchical models. For efficient computation, all parameter distributions are discretized on a parameter grid.

eval(*query*, *t=None*, *silent=False*)
Convenience method to evaluate arithmetic operations on (hyper-)parameters. See `parser.Parser()` for more information. Note: This method is slow for evaluating a lot of queries subsequently, as it initializes a new Parser instance for each query. Use a dedicated Parser instance in that case (i.e.: P = Parser(S); P('query')).

Parameters

- **query** (*str*) – (In-)equality to compute probability for, or just an arithmetic operation on parameters (the distribution will be returned in the latter case).
- **t** – Time step/stamp to evaluate all parameters at
- **silent** (*bool*) – If true, no output is generated by this method.

Returns Probability of queried (in-)equality, or values and corresponding probability values of derived distribution.

fit (*forwardOnly=False*, *evidenceOnly=False*, *silent=False*)
Computes the sequence of posterior distributions and evidence for each time step. Evidence is also computed for the complete data set.

Parameters

- **forwardOnly** (*bool*) – If set to True, the fitting process is terminated after the forward pass. The resulting posterior distributions are so-called "filtering distributions" which - at each time step - only incorporate the information of past data points. This option thus emulates an online analysis.
- **evidenceOnly** (*bool*) – If set to True, only forward pass is run and evidence is calculated. In contrast to the forwardOnly option, no posterior mean values are computed and no posterior distributions are stored.
- **silent** (*bool*) – If set to True, no output is generated by the fitting method.

getHyperParameterValue (*name*)
Returns the currently set value of a hyper-parameter. Note: The returned value is NOT an inferred value, but simply the last value used by the fitting method.

Parameters **name** (*str*) – Hyper-parameter name.

Returns current value of the specified hyper-parameter.

Return type float

getPD (*t*, *name*, *plot=False*, *density=True*, ***kwargs*)
See `Study.getParameterDistribution()`.

getPDs (*name*, *plot=False*, *density=True*, ***kwargs*)
See `Study.getParameterDistributions()`.

getParameterDistribution (*t*, *name*, *plot=False*, *density=True*, ***kwargs*)
Compute the marginal parameter distribution at a given time step.

Parameters

- **t** – Time step/stamp for which the parameter distribution is evaluated (or 'avg' for time-averaged parameter distribution)
- **name** (*str*) – Name of the parameter to display
- **plot** (*bool*) – If True, a plot of the distribution is created
- **density** (*bool*) – If true, probability density is returned; if false, probability values
- ****kwargs** – All further keyword-arguments are passed to the plot (see matplotlib documentation)

Returns

The first array contains the parameter values, the second one the corresponding probability (density) values

Return type ndarray, ndarray

`getParameterDistributions`(*name*, *plot=False*, *density=True*, ***kwargs*)
Computes the time series of marginal posterior distributions with respect to a given model parameter.

Parameters

- **`name`** (*`str`*) – Name of the parameter to display
- **`plot`** (*`bool`*) – If True, a plot of the series of distributions is created (density map)
- **`density`** (*`bool`*) – If true, probability density is returned; if false, probability values
- **`**kwargs`** – All further keyword-arguments are passed to the plot (see matplotlib documentation)

Returns The first array contains the parameter values, the second one the sequence of corresponding posterior distributions.

Return type ndarray, ndarray

`getParameterMeanValues`(*name*)
Returns posterior mean values for a parameter of the observation model.

Parameters **`name`** (*`str`*) – Name of the parameter to display

Returns array of posterior mean values for the selected parameter

Return type ndarray

`load`(*array*, *timestamps=None*, *silent=False*)
See *`Study.loadData()`*.

`loadData`(*array*, *timestamps=None*, *silent=False*)
Loads Numpy array as data.

Parameters

- **`array`** (*`ndarray`*) – Numpy array containing time series data
- **`timestamps`** (*`ndarray`*) – Array of timestamps (same length as data array)
- **`silent`** (*`bool`*) – If set to True, no output is generated by this method.

`loadExampleData`(*silent=False*)
Loads UK coal mining disaster data.

Parameters **`silent`** (*`bool`*) – If set to True, no output is generated by this method.

`optimize`(*parameterList=[]*, *forwardOnly=False*, ***kwargs*)
Uses the COBYLA minimization algorithm from SciPy to perform a maximization of the log-evidence with respect to all hyper-parameters (the parameters of the transition model) of a time seris model. The starting values are the values set by the user when defining the transition model.

For the optimization, only the log-evidence is computed and no parameter distributions are stored. When a local maximum is found, the parameter distribution is computed based on the optimal values for the hyper-parameters.

Parameters

- **`parameterList`** (*`list`*) – List of hyper-parameter names to optimize. For nested transition models with multiple, identical hyper-parameter names, the sub-model index can be provided. By default, all hyper-parameters are optimized.
- **`forwardOnly`** (*`bool`*) – If set to True, the fitting process is terminated after the forward pass. The resulting posterior distributions are so-called "filtering distributions" which - at each time step - only incorporate the information of past data points.

- **– All other keyword parameters are passed to the 'minimize' routine of scipy.optimize.** (***kwargs*) –

plot(*name*, ***kwargs*)
Convenience method to plot the temporal evolution of observation model parameters, or the parameter distribution at a specific time step. Extended functionality for other study classes.

Parameters

- **name** (*str*) – name of the parameter to display
- **color** – color from which a light colormap is created (for parameter evolution only)
- **gamma** (*float*) – exponent for gamma correction of the displayed marginal distribution; default: 0.5 (for parameter evolution only)
- **t** – Time step/stamp for which the parameter distribution is evaluated
- **density** (*bool*) – If true, probability density is plotted; if false, probability values
- **kwargs** – all further keyword-arguments are passed to the axes object of the plot

plotParameterEvolution(*name*, *color='b'*, *gamma=0.5*, ***kwargs*)
Extended plot method to display a series of marginal posterior distributions corresponding to a single model parameter. In contrast to getMarginalParameterDistributions(), this method includes the removal of plotting artefacts, gamma correction as well as an overlay of the posterior mean values.

Parameters

- **name** (*str*) – name of the parameter to display
- **color** – color from which a light colormap is created
- **gamma** (*float*) – exponent for gamma correction of the displayed marginal distribution; default: 0.5
- **kwargs** – all further keyword-arguments are passed to the plot of the posterior mean values

set(**args*, ***kwargs*)
Set observation model or transition model, or both. See `Study.setTransitionModel()` and `Study.setObservationModel()`.

Parameters

- **args** – Sequence of Observation model instance and Transition model instance, or just one of those two types
- **silent** (*bool*) – If true, no output is printed by this method

setOM(*L*, *silent=False*)
See `Study.setObservationModel()`.

setObservationModel(*L*, *silent=False*)
Sets observation model (likelihood function) for analysis and creates parameter grid for inference routine.

Parameters

- **L** – Observation model class (see observationModels.py)
- **silent** (*bool*) – If set to True, no output is generated by this method.

setTM(*T*, *silent=False*)
See `Study.setTransitionModel()`.

setTransitionModel (*T*, *silent=False*)
Set transition model which describes the parameter dynamics.

Parameters

- **T** – Transition model class (see transitionModels.py)
- **silent** (*bool*) – If true, no output is printed by this method

simulate (*x*, *t=None*, *density=False*)
Computes the probability (density) for a set of observations, based on the inferred parameter distributions of a given time step, or based on the time-averaged parameter distributions. It can be used to compute the expected distribution of the observed data, taking into account the uncertainty in the parameters (as well as hyper-parameters for Hyper-Studies).

Parameters

- **x** – array of observation values
- **t** – Time step/stamp for which the parameter distribution is evaluated
- **density** – If true, probability density is computed; if false, probability value is computed

Returns probability (density) values corresponding to observation values

Return type ndarray

1.4.2 Observation models

SymPy(rv, *args, **kwargs)	Model based on sympy.stats random variable.
SciPy(rv, *args, **kwargs)	Model based on scipy.stats distribution.
NumPy(function, *args, **kwargs)	Model based on NumPy functions.
Bernoulli([name, value, prior])	Bernoulli trial.
Poisson([name, value, prior])	Poisson observation model.
Gaussian([name1, value1, name2, value2, prior])	Gaussian observations.
GaussianMean([name, value, prior])	Observations with given error interval.
WhiteNoise([name, value, prior])	White noise process.
AR1([name1, value1, name2, value2, prior])	Auto-regressive process of first order.
ScaledAR1([name1, value1, name2, value2, prior])	Scaled auto-regressive process of first order. Recusively defined as

Note: You can use the short-form *om* to access all observation models:

```
import bayesloop as bl
L = bl.om.SymPy(...)
```

Observation models refer to likelihood functions, describing the probability (density) of a measurement at a certain time step, given the time-varying parameter values and past measurements. Observation models thus represent the low- level model in a bayesloop study, as compared to transition models that represent the high-level models and specify how the time-varying parameter change over time.

class bayesloop.observationModels.**AR1** (*name1='correlation coefficient'*, *value1=None*, *name2='noise amplitude'*, *value2=None*, *prior=None*)
Auto-regressive process of first order. This model describes a simple stochastic time series model with an exponential autocorrelation-function. It can be recursively defined as: d_t = r_t * d_(t-1) + s_t * e_t, with d_t

being the data point at time t, r_t the correlation coefficient of subsequent data points and s_t being the noise amplitude of the process. e_t is distributed according to a standard normal distribution.

Parameters

- **`name1`** (*`str`*) – custom name for correlation coefficient
- **`value1`** (*`list, tuple, ndarray`*) – Regularly spaced parameter values for the correlation coefficient
- **`name2`** (*`str`*) – custom name for noise amplitude
- **`value2`** (*`list, tuple, ndarray`*) – Regularly spaced parameter values for the noise amplitude
- **`prior`** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

`estimateParameterValues`(*name*, *rawData*)

Returns estimated boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **`name`** (*`str`*) – name of a parameter of the observation model
- **`rawData`** (*`ndarray`*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

`pdf`(*grid*, *dataSegment*)

Probability density function of the Auto-regressive process of first order

Parameters

- **`grid`** (*`list`*) – Parameter grid for discrete values of the correlation coefficient and noise amplitude
- **`dataSegment`** (*`ndarray`*) – Data segment from formatted data (in this case a pair of measurements)

Returns Discretized pdf (for data point d_t, given d_(t-1) and parameters).

Return type ndarray

class `bayesloop.observationModels.`**`Bernoulli`**(*name='p'*, *value=None*, *prior='Jeffreys'*)

Bernoulli trial. This distribution models a random variable that takes the value 1 with a probability of p, and a value of 0 with the probability of (1-p). Subsequent data points are considered independent. The model has one parameter, p, which describes the probability of "success", i.e. to take the value 1.

Parameters

- **`name`** (*`str`*) – custom name for model parameter p
- **`value`** (*`list, tuple, ndarray`*) – Regularly spaced parameter values for the model parameter p
- **`prior`** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

`estimateParameterValues`(*name*, *rawData*)

Returns appropriate boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns Regularly spaced parameter values for the specified parameter.

Return type list

jeffreys(*x*)
Jeffreys prior for Bernoulli model.

pdf(*grid*, *dataSegment*)
Probability density function of the Bernoulli model

Parameters

- **grid** (*list*) – Parameter grid for discrete values of the parameter p
- **dataSegment** (*ndarray*) – Data segment from formatted data (in this case a single number of events)

Returns Discretized Bernoulli pdf (with same shape as grid)

Return type ndarray

class bayesloop.observationModels.**Gaussian**(*name1='mean'*, *value1=None*, *name2='std'*, *value2=None*, *prior='Jeffreys'*)
Gaussian observations. All observations are independently drawn from a Gaussian distribution. The model has two parameters, mean and standard deviation.

Parameters

- **name1** (*str*) – custom name for mean
- **value1** (*list*, *tuple*, *ndarray*) – Regularly spaced parameter values for the model parameter mean
- **name2** (*str*) – custom name for standard deviation
- **value2** (*list*, *tuple*, *ndarray*) – Regularly spaced parameter values for the model parameter standard deviation
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

estimateParameterValues(*name*, *rawData*)
Returns appropriate boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

jeffreys(*mu*, *sigma*)
Jeffreys prior for Gaussian model.

pdf (*grid*, *dataSegment*)

Probability density function of the Gaussian model.

Parameters

- **grid** (*list*) – Parameter grid for discrete values of mean and standard deviation
- **dataSegment** (*ndarray*) – Data segment from formatted data (containing a single measurement)

Returns Discretized Normal pdf (with same shape as grid).

Return type ndarray

class bayesloop.observationModels.**GaussianMean** (*name='mean'*, *value=None*, *prior=None*)

Observations with given error interval. This observation model represents a Gaussian distribution with given standard deviation, only the mean of the distribution is a free parameter. It can be used if the data at hand contains for example mean values and corresponding error intervals. The data is supplied as an array of tuples, where each tuple contains the observed mean value and the corresponding standard deviation for an individual time step:

```
[["mean (t=1)", "std (t=1)"], ["mean (t=2)", "std (t=2)"], ...]
```

Parameters

- **name** (*str*) – custom name for the mean parameter
- **value** (*list, tuple, ndarray*) – Regularly spaced parameter values for the mean parameter
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

estimateParameterValues (*name*, *rawData*)

Returns appropriate boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

pdf (*grid*, *dataSegment*)

Probability density function of the Gaussian mean model.

Parameters

- **grid** (*list*) – Parameter grid for discrete values of the mean
- **dataSegment** (*ndarray*) – Data segment from formatted data (containing a tuple of observed mean value and the given standard deviation)

Returns Discretized Normal pdf (with same shape as grid).

Return type ndarray

class bayesloop.observationModels.**Laplace**(*name1='mean'*, *value1=None*, *name2='scale'*, *value2=None*, *prior='Jeffreys'*)

Laplace model. All observations are independently drawn from a Laplace (double-sided exponential) distribution. The model has two parameters, mean and scale.

Parameters

- **name1** (*str*) – custom name for mean
- **value1** (*list*, *tuple*, *ndarray*) – Regularly spaced parameter values for the model parameter mean
- **name2** (*str*) – custom name for the scale parameter
- **value2** (*list*, *tuple*, *ndarray*) – Regularly spaced parameter values for the scale parameter
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

estimateParameterValues(*name*, *rawData*)

Returns appropriate boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

jeffreys(*mu*, *scale*)

Jeffreys prior for the Laplace model.

pdf(*grid*, *dataSegment*)

Probability density function of the Laplace model.

Parameters

- **grid** (*list*) – Parameter grid for discrete values of mean and scale
- **dataSegment** (*ndarray*) – Data segment from formatted data (containing a single measurement)

Returns Discretized Normal pdf (with same shape as grid).

Return type ndarray

class bayesloop.observationModels.**NumPy**(*function*, **args*, ***kwargs*)

Model based on NumPy functions. This observation model class allows the user to create new observation models by expressing the likelihood function as a Python function that takes a data point (or vector) and arrays of parameter values as input, and outputs the probability density of those parameter values. Note that the Python function must be able to broadcast the arrays of parameter values, so that the output array has the same shape as the input arrays.

Parameters

- **function** – Likelihood function that takes a data point as the first argument and one NumPy array per model parameter (see example below).
- **args** – succession of names and corresponding parameter values (using bayesloop.cint() or bayesloop.oint()) Example: 'mu', bl.cint(-1, 1, 100), 'sigma', bl.oint(0, 3, 100)

- **prior** – custom prior distribution that may be passed as a NumPy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

Example:

```
# Assume that we have a data set of Gaussian random variates. We know the␣
↪standard deviation for each random
# variate, but not the mean value. The data has the form [[variate_1, std_1],␣
↪[variate_2, std_2], ...]. We can
# design an observation model to infer the mean value of the data taking into␣
↪account the known standard
# deviation as follows:

import bayesloop as bl
S = bl.Study()

data = np.array([[0.12, 0.2], [-0.23, 0.2], [-0.03, 0.1], [0.12, 0.1]])
S.loadData(data)

def likelihood(data, mu):
    # read in one data point of the form [variate_n, std_n]
    x, std = data

    # define Gaussian likelihood function (pdf) with known standard deviation
    pdf = np.exp((x - mu)**2./(2*std**2.))/np.sqrt(2*np.pi*std**2.)

    return pdf

L = bl.om.NumPy(likelihood, 'mu', bl.cint(-3, 3, 1000))
S.setOM(L)
```

pdf (*grid*, *dataSegment*)
Probability density function of custom models

Parameters

- **grid** (*list*) – Parameter grid for discrete parameter values
- **dataSegment** (*ndarray*) – Data segment from formatted data

Returns Discretized pdf (with same shape as grid)

Return type ndarray

class bayesloop.observationModels.**ObservationModel**
Observation model class that handles missing data points and multi-dimensional data. All observation models included in bayesloop inherit from this class.

processedPdf (*grid*, *dataSegment*)
This method is called by the fit-method of the Study class (and the step method of the OnlineStudy class) and processes multidimensional data and missing data and passes it to the pdf-method of the child class.

Parameters

- **grid** (*list*) – Discrete parameter grid
- **dataSegment** (*ndarray*) – Data segment from formatted data

Returns Discretized pdf (with same shape as grid)

Return type ndarray

class bayesloop.observationModels.**Poisson**(*name='lambda'*, *value=None*, *prior='Jeffreys'*)

Poisson observation model. Subsequent data points are considered independent and distributed according to the Poisson distribution. Input data consists of integer values, typically the number of events in a fixed time interval. The model has one parameter, often denoted by lambda, which describes the rate of the modeled events.

Parameters

- **name** (*str*) – custom name for rate parameter lambda
- **value** (*list, tuple, ndarray*) – Regularly spaced parameter values for the model parameter lambda
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

estimateParameterValues(*name*, *rawData*)

Returns appropriate boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

jeffreys(*x*)

Jeffreys prior for Poisson model.

pdf(*grid*, *dataSegment*)

Probability density function of the Poisson model

Parameters

- **grid** (*list*) – Parameter grid for discrete rate (lambda) values
- **dataSegment** (*ndarray*) – Data segment from formatted data (in this case a single number of events)

Returns Discretized Poisson pdf (with same shape as grid)

Return type ndarray

class bayesloop.observationModels.**ScaledAR1**(*name1='correlation coefficient'*, *value1=None*, *name2='standard deviation'*, *value2=None*, *prior=None*)

Scaled auto-regressive process of first order. Recusively defined as d_t = r_t * d_(t-1) + s_t*sqrt(1 - (r_t)^2) * e_t,

with r_t the correlation coefficient of subsequent data points and s_t being the standard deviation of the observations d_t. For the standard AR1 process, s_t defines the noise amplitude of the process. For uncorrelated data, the two observation models are equal.

Parameters

- **name1** (*str*) – custom name for correlation coefficient
- **value1** (*list, tuple, ndarray*) – Regularly spaced parameter values for the correlation coefficient

- **name2** (*str*) – custom name for standard deviation
- **value2** (*list, tuple, ndarray*) – Regularly spaced parameter values for the standard deviation
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

estimateParameterValues(*name*, *rawData*)

Returns estimated boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

pdf(*grid*, *dataSegment*)

Probability density function of the Auto-regressive process of first order

Parameters

- **grid** (*list*) – Parameter grid for discerete values of the correlation coefficient and standard deviation
- **dataSegment** (*ndarray*) – Data segment from formatted data (in this case a pair of measurements)

Returns Discretized pdf (for data point d_t, given d_(t-1) and parameters).

Return type ndarray

class bayesloop.observationModels.**SciPy**(*rv*, **args*, ***kwargs*)

Model based on scipy.stats distribution. This observation model class allows to create new observation models on-the-fly from scipy.stats probability distributions.

Parameters

- **rv** – SciPy random distribution
- **args** – succession of names and corresponding parameter values (using bayesloop.cint() or bayesloop.oint()) Example: ‘mu’, bl.cint(-1, 1, 100), ‘sigma’, bl.oint(0, 3, 100)
- **fixedParameters** (*dict*) – Dictionary defining the names and values of fixed parameters
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

Note that scipy.stats does not use the canonical way of naming the parameters of the probability distributions, but instead includes the parameter ‘loc’ (for discrete & continuous distributions) and ‘scale’ (for continuous only).

See http://docs.scipy.org/doc/scipy/reference/stats.html for further information on the available distributions and the parameter notation.

Example:

```
import bayesloop as bl
import scipy.stats
L = bl.om.SciPy(scipy.stats.poisson, 'mu', bl.oint(0, 6, 1000), fixedParameters={
↪'loc': 0})
```

This will result in a model for poisson-distributed observations with a rate parameter 'mu' between 0 and 6. The distribution is not shifted (loc = 0).

Note that while the parameters 'loc' and 'scale' have default values in scipy.stats and do not necessarily need to be set, they have to be added to the fixedParameters dictionary in bayesloop to be treated as a constant. Using SciPy.stats distributions, bayesloop uses a flat prior by default.

pdf (*grid*, *dataSegment*)
Probability density function of custom scipy.stats models

Parameters

- **grid** (*list*) – Parameter grid for discrete rate values
- **dataSegment** (*ndarray*) – Data segment from formatted data

Returns Discretized pdf (with same shape as grid)

Return type ndarray

class bayesloop.observationModels.**SymPy** (*rv*, **args*, ***kwargs*)
Model based on sympy.stats random variable. This observation model class allows to create new observation models on-the-fly from sympy.stats random variables.

Parameters

- **rv** – SymPy random symbol
- **args** – succession of names and corresponding parameter values (using bayesloop.cint() or bayesloop.oint()) Example: 'mu', bl.cint(-1, 1, 100), 'sigma', bl.oint(0, 3, 100)
- **determineJeffreysPrior** (*bool*) – If set to true, Jeffreys prior is analytically derived
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

Observation models can be defined symbolically using the SymPy module in a convenient way. In contrast to the SciPy probability distributions, fixed parameters are directly set and do not have to be passed as a dictionary.

See http://docs.sympy.org/dev/modules/stats.html for further information on the available distributions and the parameter notation.

Example:

```
import bayesloop as bl
from sympy import Symbol
from sympy.stats import Normal

mu = 4
sigma = Symbol('sigma', positive=True)
rv = Normal('normal', mu, sigma)

L = bl.om.SymPy(rv, {'sigma': bl.oint(0, 3, 1000)})
```

This will result in a model for normally distributed observations with a fixed 'mu' (mean) of 4, leaving 'sigma' (the standard deviation) as the only free parameter to be inferred. Using SymPy random variables to create an observation model, bayesloop tries to determine the corresponding Jeffreys prior. This behavior can be turned off by setting the keyword-argument 'determineJeffreysPrior=False'.

pdf (*grid*, *dataSegment*)
Probability density function of custom sympy.stats models

Parameters

- **grid** (*list*) – Parameter grid for discrete rate values
- **dataSegment** (*ndarray*) – Data segment from formatted data

Returns Discretized pdf (with same shape as grid)

Return type ndarray

class bayesloop.observationModels.**WhiteNoise** (*name='std'*, *value=None*, *prior='Jeffreys'*)
White noise process. All observations are independently drawn from a Gaussian distribution with zero mean and a finite standard deviation, the noise amplitude. This process is basically an auto-regressive process with zero correlation.

Parameters

- **name** (*str*) – custom name for standard deviation
- **value** (*list, tuple, ndarray*) – Regularly spaced parameter values for the model parameter standard deviation
- **prior** – custom prior distribution that may be passed as a Numpy array that has tha same shape as the parameter grid, as a(lambda) function or as a (list of) SymPy random variable(s)

estimateParameterValues (*name*, *rawData*)
Returns appropriate boundaries based on the imported data. Is called in case fit method is called and no boundaries are defined.

Parameters

- **name** (*str*) – name of a parameter of the observation model
- **rawData** (*ndarray*) – observed data points that may be used to determine appropriate parameter boundaries

Returns parameter boundaries.

Return type list

jeffreys (*sigma*)
Jeffreys prior for Gaussian model.

pdf (*grid*, *dataSegment*)
Probability density function of the white noise process.

Parameters

- **grid** (*list*) – Parameter grid for discrete values of noise amplitude
- **dataSegment** (*ndarray*) – Data segment from formatted data (containing a single measurement)

Returns Discretized pdf (with same shape as grid).

Return type ndarray

1.4.3 Transition models

Static()	Constant parameters over time.
Deterministic([function, target, prior])	Deterministic parameter variations.
GaussianRandomWalk([name, value, target, prior])	Gaussian parameter fluctuations.
AlphaStableRandomWalk([name1, value1, . . .])	Parameter changes follow alpha-stable distribution.
ChangePoint([name, value, prior])	Abrupt parameter change at a specified time step.
RegimeSwitch([name, value, prior])	Small probability for a parameter jump in each time step.
Independent()	Observations are treated as independent.
NotEqual([name, value, prior])	Unlikely parameter values are preferred in the next time step.
CombinedTransitionModel(*args)	Different models act at the same time.
SerialTransitionModel(*args)	Different models act at different time steps.

Note: You can use the short-form *tm* to access all transition models:

```
import bayesloop as bl
T = bl.tm.ChangePoint(...)
```

Transition models refer to stochastic or deterministic models that describe how the time-varying parameter values of a given time series model change from one time step to another. The transition model can thus be compared to the state transition matrix of Hidden Markov models. However, instead of explicitly stating transition probabilities for all possible states, a transformation is defined that alters the distribution of the model parameters in one time step according to the transition model. This altered distribution is subsequently used as a prior distribution in the next time step.

class bayesloop.transitionModels.**AlphaStableRandomWalk**(*name1='c', value1=None, name2='alpha', value2=None, target=None, prior=(None, None)*)

Parameter changes follow alpha-stable distribution. This model assumes that parameter changes are distributed according to the symmetric alpha-stable distribution. For each parameter, two hyper-parameters can be set: the width of the distribution (c) and the shape (alpha).

Parameters

- **name1** (*str*) – custom name of the hyper-parameter c
- **value1** (*float, list, tuple, ndarray*) – width(s) of the distribution (c >= 0).
- **name2** (*str*) – custom name of the hyper-parameter alpha
- **value2** (*float, list, tuple, ndarray*) – shape(s) of the distribution (0 < alpha <= 2).
- **target** (*str*) – parameter name of the observation model to apply transition model to
- **prior** – list of two hyper-prior distributions, where each may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

computeForwardPrior(*posterior, t*)

Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

convolve(*distribution*)
Convolves distribution with alpha-stable kernel.

Parameters **distribution** (*ndarray*) – Discrete probability distribution to convolve.

Returns convolution

Return type ndarray

createKernel(*c*, *alpha*, *axis*)
Create alpha-stable distribution on a grid as a kernel for convolution.

Parameters

- **c** (*float*) – Scale parameter.
- **alpha** (*float*) – Tail parameter (alpha = 1: Cauchy, alpha = 2: Gauss)
- **axis** (*int*) – Axis along which the distribution is defined, for 2D-Kernels

Returns kernel

Return type ndarray

class bayesloop.transitionModels.**BreakPoint**(*name='tBreak'*, *value=None*, *prior=None*)
Break-point. This class can only be used to specify break-point within a SerialTransitionModel instance.

Parameters

- **name** (*str*) – custom name of the hyper-parameter tBreak
- **value** (*int, list, tuple, ndarray*) – Value(s) of the time step(s) of the break point
- **prior** – hyper-prior distribution that may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

class bayesloop.transitionModels.**ChangePoint**(*name='tChange'*, *value=None*, *prior=None*)
Abrupt parameter change at a specified time step. Parameter values are allowed to change only at a single point in time, right after a specified time step (Hyper-parameter tChange). Note that a uniform parameter distribution is used at this time step to achieve this "reset" of parameter values.

Parameters

- **name** (*str*) – custom name of the hyper-parameter tChange
- **value** (*int, list, tuple, ndarray*) – Integer value(s) of the time step of the change point
- **prior** – hyper-prior distribution that may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

computeForwardPrior(*posterior*, *t*)
Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**CombinedTransitionModel**(**args*)

Different models act at the same time. This class allows to combine different transition models to be able to explore more complex parameter dynamics. All sub-models are passed to this class as arguments on initialization. Note that a different order of the sub-models can result in different parameter dynamics.

Parameters ***args** – Sequence of transition models

computeForwardPrior(*posterior*, *t*)

Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**Deterministic**(*function=None*, *target=None*, *prior=None*)

Deterministic parameter variations. Given a function with time as the first argument and further keyword-arguments as hyper-parameters, plus the name of a parameter of the observation model that is supposed to follow this function over time, this transition model shifts the parameter distribution accordingly. Note that these models are entirely deterministic, as the hyper-parameter values are entered by the user. However, the hyper-parameter distributions can be inferred using a Hyper-study or can be optimized using the 'optimize' method of the Study class.

Parameters

- **function** (*function*) – A function that takes the time as its first argument and further takes keyword-arguments that correspond to the hyper-parameters of the transition model which the function defines.
- **target** (*str*) – The observation model parameter that is manipulated according to the function defined above.
- **prior** – List of hyper-prior distributions (one for each hyper-parameter), where each may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

Example:

```
def quadratic(t, a=0, b=0):
    return a*(t**2) + b*t

S = bl.Study()
...
S.setObservationModel(bl.om.WhiteNoise('std', bl.oint(0, 3, 1000)))
S.setTransitionModel(bl.tm.Deterministic(quadratic, target='signal'))
```

computeForwardPrior(*posterior*, *t*)

Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int, float*) – time stamp (integer time index by default)

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**GaussianRandomWalk**(*name='sigma'*, *value=None*, *target=None*, *prior=None*)

Gaussian parameter fluctuations. This model assumes that parameter changes are Gaussian-distributed. The standard deviation can be set individually for each model parameter.

Parameters

- **name** (*str*) – custom name of the hyper-parameter sigma
- **value** (*float, list, tuple, ndarray*) – standard deviation(s) of the Gaussian random walk for target parameter
- **target** (*str*) – parameter name of the observation model to apply transition model to
- **prior** – hyper-prior distribution that may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

computeForwardPrior(*posterior*, *t*)

Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**Independent**

Observations are treated as independent. This transition model restores the prior distribution for the parameters at each time step, effectively assuming independent observations.

Note: Mostly used with an instance of OnlineStudy.

computeForwardPrior(*posterior*, *t*)

Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**NotEqual**(*name='log10pMin'*, *value=None*, *prior=None*)

Unlikely parameter values are preferred in the next time step. Assumes an “inverse” parameter distribution at each new time step. The new prior is derived by substracting the posterior probability values from their maximal value and subsequently re-normalizing. To assure that no parameter value is set to zero probability, one may specify a minimal probability for all parameter values. This transition model is mostly used in instances of OnlineStudy to detect time step when parameter distributions change significantly.

Parameters

- **name** (*str*) – custom name of the hyper-parameter log10pMin
- **value** (*float, list, tuple, ndarray*) – Log10-value of the minimal probability that is set to all possible parameter values of the inverted parameter distribution
- **prior** – hyper-prior distribution that may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

Note: Mostly used with an instance of OnlineStudy.

computeForwardPrior(*posterior*, *t*)
Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**RegimeSwitch**(*name='log10pMin'*, *value=None*, *prior=None*)
Small probability for a parameter jump in each time step. In case the number of change-points in a given data set is unknown, the regime-switching model may help to identify potential abrupt changes in parameter values. At each time step, all parameter values within the set boundaries are assigned a minimal probability density of being realized in the next time step, effectively allowing abrupt parameter changes at every time step.

Parameters

- **name** (*str*) – custom name of the hyper-parameter log10pMin
- **value** (*float, list, tuple, ndarray*) – Minimal probability density (log10 value) that is assigned to every parameter value
- **prior** – hyper-prior distribution that may be passed as a(lambda) function, as a SymPy random variable, or directly as a Numpy array with probability values for each hyper-parameter value

computeForwardPrior(*posterior*, *t*)
Compute new prior from old posterior (moving forwards in time).

Parameters

- **posterior** (*ndarray*) – Parameter distribution from current time step
- **t** (*int*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class bayesloop.transitionModels.**SerialTransitionModel**(**args*)
Different models act at different time steps. To model fundamental changes in parameter dynamics, different transition models can be serially coupled. Depending on the time step, a corresponding sub-model is chosen to compute the new prior distribution from the posterior distribution. If a break-point lies in between two transition models, the parameter values do not change abruptly at the time step of the break-point, whereas a change-point

not only changes the transition model, but also allows the parameters to change (the parameter distribution is re-set to the prior distribution).

Parameters **`*args`** – Sequence of transition models and break-points/change-points (for n models, n-1 break-points/change-points have to be provided)

Example:

```
T = bl.tm.SerialTransitionModel(bl.tm.Static(),
                                bl.tm.BreakPoint('t_1', 50),
                                bl.tm.RegimeSwitch('log10pMin', -7),
                                bl.tm.BreakPoint('t_2', 100),
                                bl.tm.GaussianRandomWalk('sigma', 0.2, target='x
↪'))
```

In this example, parameters are assumed to be constant until 't_1' (time step 50), followed by a regime-switching- process until 't_2' (time step 100). Finally, we assume Gaussian parameter fluctuations for parameter 'x' until the last time step. Note that models and time steps do not necessarily have to be passed in an alternating way.

`computeForwardPrior`(*posterior*, *t*)
Compute new prior from old posterior (moving forwards in time).

Parameters

- **`posterior`** (*`ndarray`*) – Parameter distribution from current time step
- **`t`** (*`int`*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class `bayesloop.transitionModels.`**`Static`**
Constant parameters over time. This trivial model assumes no change of parameter values over time.

`computeForwardPrior`(*posterior*, *t*)
Compute new prior from old posterior (moving forwards in time).

Parameters

- **`posterior`** (*`ndarray`*) – Parameter distribution from current time step
- **`t`** (*`int`*) – integer time step

Returns Prior parameter distribution for subsequent time step

Return type ndarray

class `bayesloop.transitionModels.`**`TransitionModel`**
Parent class for transition models. All transition models inherit from this class. It is currently only used to identify transition models as such.

1.4.4 File I/O

The following functions save or load instances of all *Study* types using the Python package *dill*.

`bayesloop.fileIO.`**`load`**(*filename*)
Load an instance of a bayesloop study class that was saved using the bayesloop.save() function.

Parameters **`filename`** (*`str`*) – Path + filename to stored bayesloop study

Returns Study instance

`bayesloop.fileIO.save`(*filename*, *study*)

Save an instance of a bayesloop study class to file.

Parameters

- **`filename`** (*str*) – Path + filename to store bayesloop study
- **`study`** – Instance of study class (Study, HyperStudy, etc.)

Note: Both file I/O functions are imported directly into the module namespace for convenient access.

```
import bayesloop as bl
S = bl.Study()
...
bl.save('test.bl', S)
...
S = bl.load('test.bl')
```

1.4.5 Probability Parser

class `bayesloop.Parser`(**studies*)

Computes derived probability values and distributions based on arithmetic operations of (hyper-)parameters.

Parameters **`studies`** – One or more bayesloop study instances. All (hyper-)parameters in the specified study object(s) will be available to the parser.

Example:

```
S = bl.Study()
...
P = bl.Parser(S)
P('sqrt(rate@1910) > 1.')
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index