pyABC Documentation

Release 0.9.12

The pyABC developers

May 05, 2019
# USER’S GUIDE

## 1 What is pyABC about?
1.1 What you need ................................................................. 3
1.2 What you don’t need .......................................................... 3
1.3 When better not to use pyABC ............................................. 3
1.4 Why to use pyABC? ............................................................ 3

## 2 Installation and Upgrading
2.1 Preparation ................................................................. 5
2.2 PIP Installation .............................................................. 5
2.3 GIT Installation ............................................................ 6
2.4 Upgrading ................................................................. 6
2.5 Installing Anaconda on a Cluster environment ...................... 6
2.6 Optional dependencies ....................................................... 6

## 3 Examples
3.1 Quickstart .............................................................. 9
3.2 Parameter inference ......................................................... 12
3.3 Early stopping of model simulations .................................... 16
3.4 Resuming stored ABC runs ............................................... 20
3.5 Using R with pyABC ....................................................... 22
3.6 Ordinary Differential Equations: Conversion Reaction .......... 27
3.7 Markov Jump Process: Reaction Network .......................... 32
3.8 Multi-scale model: Tumor spheroid growth ......................... 38
3.9 Stochastic Differential Equation: Ion channel noise in Hodgkin-Huxley neurons ....................................................... 45
3.10 Adaptive Distances ........................................................ 53
3.11 Download the examples as notebooks ............................... 57

## 4 Parallel and Distributed Sampling
4.1 Strategies .............................................................. 59
4.2 How to setup a Redis based distributed cluster ..................... 60

## 5 Parallel job execution on an SGE cluster environment
5.1 Quick start ............................................................ 63
5.2 Information about running jobs ......................................... 63
5.3 Usage notes ............................................................ 64

## 6 Exporting pyABC’s database to other file formats

## 7 Web based visualizations

## 8 Release Notes
pyABC is a framework for distributed, likelihood-free inference. That means, if you have a model and some data and want to know the posterior distribution over the model parameters, i.e. you want to know with which probability which parameters explain the observed data, then pyABC might be for you.

All you need is some way to numerically draw samples from the model, given the model parameters. pyABC “inverts” the model for you and tells you which parameters were well matching and which ones not. You do **not** need to analytically calculate the likelihood function.

pyABC runs efficiently on multi-core machines and distributed cluster setups. It is easy to use and flexibly extensible. If you use it in your work, you can cite the paper:

Emmanuel Klinger, Dennis Rickert, Jan Hasenauer; pyABC: distributed, likelihood-free inference; Bioinformatics 2018; https://doi.org/10.1093/bioinformatics/bty361
WHAT IS PYABC ABOUT?

pyABC helps to solve the problem of parameter inference, in settings where all you can do is simulate from the (black-box) model but no further analysis is possible. Putting it differently, if you have a forward simulator, then pyABC does the backwards parameter inference step for you.

1.1 What you need

- Some kind of experimentally observed or synthetically generated data
- A parametrized stochastic simulator which supposedly explains the data (e.g. a function which possibly uses a random number generator)

1.2 What you don’t need

- The likelihood function: $p(\text{data} | \text{parameter})$ is not required.

1.3 When better not to use pyABC

Fig. 1: Not everything is a nail.

ABC-SMC likelihood-free inference is not a hammer for every nail. If you’re actually able to write down the likelihood then using it and applying a different inference technique which exploits it might be the better approach. This package helps to solve the much harder problem of likelihood-free inference.

1.4 Why to use pyABC?

This is a package for Approximate Bayesian Computation, using a Sequential Monte Carlo scheme. This provides a particularly efficient technique for Bayesian posterior estimation in cases where it is very hard to calculate the likelihood function efficiently.

pyABC was designed to perform well on

- multicore and
• distributed environments.

pyABC integrates well with SGE like environments, as commonly found in scientific settings, but can also be deployed to the cloud. Amongst other backend modes, Dask.distributed can be used under the hood. A Redis based broker or IPython parallel is also supported.

It sounds like a contradiction, but pyABC is on the one hand easy to use for standard applications, on the other hand it allows for flexible experimentation, exploring all aspects of new ABC-SMC schemes. Apart of a rich set of default choices, it is easy to parametrize aspects of your algorithm through the implementation of custom classes.
2.1 Preparation

This package requires Python 3.6 or later. The package is tested on Linux (using Travis continuous integration).

Not all of the package’s functionality is available for Microsoft Windows. As some of the multi-core parallelizations rely on forking, these won’t work on Windows. However, most other parts of the package should work on Windows as well.

2.1.1 My system’s Python distribution is outdated, what now?

Several Python distributions can coexist on a single system. If you don’t have access to a recent Python version via your system’s package manager (this might be the case for old Debian or Ubuntu operating systems), it is recommended to install the latest version of the Anaconda Python 3 distribution. See also: *Installing Anaconda on a Cluster environment*.

2.2 PIP Installation

2.2.1 Install with root rights into you system’s Python distribution

The package can be installed via pip:

```
pip install pyabc
```

into your system’s Python distribution. This requires usually root access.

2.2.2 Install as user into your home directory (recommended)

Installing pyABC into your system’s Python distribution can be problematic as you might not want to change your system’s Python installation or you don’t have root rights. The recommended alternative is to install pyABC into your home directory with:

```
pip install --user pyabc
```
2.3 GIT Installation

If you want the bleeding edge version, install directly from github:

```
pip install git+https://github.com/icb-dcm/pyabc.git
```

2.4 Upgrading

If you want to upgrade from a previous pyABC version, use:

```
pip install --upgrade pyabc
```

instead of `pip install`. You can also consult the `pip documentation` on how to manage packages. If you installed pyABC into your home directory with `pip install --user pyabc`, then upgrade also with the `--user` flag:

```
pip install --upgrade --user pyabc
```

2.5 Installing Anaconda on a Cluster environment

We’re assuming you’re on a Linux environment. Use the most recent Anaconda Python 3.x distribution. As of writing this documentation, this is the Anaconda Python 3.6 distribution. To install it, run:

```
wget https://repo.continuum.io/archive/Anaconda3-4.4.0-Linux-x86_64.sh
```

To execute the installer run:

```
bash Anaconda3-4.4.0-Linux-x86_64.sh
```

and follow the guided installation process (i.e. approve the license and tell the installer where to install it to). You might want to replace the “4.4.0” by the most recent version of Anaconda. Find out on the Anaconda Download page which one it is.

**Note:** The Anaconda installer asks you at the end of the installation whether you want to use Anaconda Python as your default Python:: bash

```
Do you wish the installer to prepend the Anaconda3 install location to PATH in your /home/username/.bashrc? [yes|no] [no] >>>
```

If you answer yes, the path to the Anaconda installation is prepended to your PATH environment variable and subsequent calls to `pip` (see below) use the Anaconda Python pip (check with the command `which pip`). If you answer no, you need to ensure manually, that the correct Python installation is used. Just saying “yes” here might safe you from some difficulties later on.

2.6 Optional dependencies

pyABC has an optional interface to the R language. To enable it install pyabc via `pip install pyabc[R]`. All Python based features will work just fine if R is not installed. See also *pyABC’s external API*. 

6 Chapter 2. Installation und Upgrading
pyABC optionally uses git to store commit hashed in its database leveraging the gitpython package. This feature can be installed via `pip install pyabc[git]`. 
The following examples should help to get a better idea of how to use pyABC.

### 3.1 Quickstart

Here is a small example on how to do Bayesian model selection.

There are more examples in the *examples section* of the documentation, such as a *parameter inference example* with a single model only.

The notebook can be downloaded here: [Quickstart](#).

The following classes from the pyABC package are used for this example:

- `ABCSMC`
- `RV`
- `Distribution`
- `PercentileDistanceFunction`

#### 3.1.1 Step by step explanation

**Defining a model**

To do model selection, we first need some models. A model, in the simplest case, is just a callable which takes a single `dict` as input and returns a single `dict` as output. The keys of the input dictionary are the parameters of the model, the output keys denote the summary statistics. Here, the `dict` is passed as `parameters` and has the entry `x`, which denotes the mean of a Gaussian. It returns the observed summary statistics `y`, which is just the sampled value.

```python
[1]: %matplotlib inline
import os
import tempfile
import scipy.stats as st
import pyabc

# Define a gaussian model
sigma = .5
```
def model(parameters):
    # sample from a gaussian
    y = st.norm(parameters.x, sigma).rvs()
    # return the sample as dictionary
    return {"y": y}

For model selection we usually have more than one model. These are assembled in a list. We require a Bayesian prior
over the models. The default is to have a uniform prior over the model classes. This concludes the model definition.

[2]: # We define two models, but they are identical so far
models = [model, model]

# However, our models’ priors are not the same.
# Their mean differs.
mu_x_1, mu_x_2 = 0, 1
parameter_priors = [
    pyabc.Distribution(x=pyabc.RV("norm", mu_x_1, sigma)),
    pyabc.Distribution(x=pyabc.RV("norm", mu_x_2, sigma))
]

Configuring the ABCSMC run

Having the models defined, we can plug together the ABCSMC class. We need a distance function, to measure the
distance of obtained samples.

[3]: # We plug all the ABC options together
abc = pyabc.ABCSMC(
    models, parameter_priors,
    pyabc.PercentileDistance(measures_to_use=["y"]))

Setting the observed data

Actually measured data can now be passed to the ABCSMC. This is set via the new method, indicating that we start
a new run as opposed to resuming a stored run (see the “resume stored run” example). Moreover, we have to set the
output database where the ABC-SMC run is logged.

[4]: # y_observed is the important piece here: our actual observation.
y_observed = 1
# and we define where to store the results
db_path = ("sqlite:///" +
    os.path.join(tempfile.gettempdir(), "test.db"))
abc_id = abc.new(db_path, {"y": y_observed})

INFO:History:Start <ABCSMC(id=1, start_time=2019-04-01 23:16:46.185404, _
˓
→
end_time=None)>
INFO:Epsilon:initial epsilon is 0.4773527724895183

The new method returns an id, which is the id of the ABC-SMC run in the database. We’re not usint this id for now.
But it might be important when you load the stored data or want to continue an ABC-SMC run in the case of having
more than one ABC-SMC run stored in a single database.
Running the ABC

We run the ABC using the ABC-SMC specifying the epsilon value at which to terminate. The default epsilon strategy is the `pyabc.epsilon.MedianEpsilon`. Whatever is reached first, the epsilon or the maximum number allowed populations, terminates the ABC run. The method returns a `pyabc.storage.History` object, which can, for example, be queried for the posterior probabilities.

```python
# We run the ABC until either criterion is met
history = abc.run(minimum_epsilon=0.2, max_nr_populations=5)
```

Note that the history object is also always accessible from the abcsmc object:

```python
history is abc.history
```

The `pyabc.storage.History` object can, for example, be queried for the posterior probabilities in the populations:

```python
# Evaluate the model probabilities
model_probabilities = history.get_model_probabilities()
model_probabilities
```

And now, let’s visualize the results:

```python
pyabc.visualization.plot_model_probabilities(history)
```

```
```
So model 1 is the more probable one. Which is expected as it was centered at 1 and the observed data was also 1, whereas model 0 was centered at 0, which is farther away from the observed data.

### 3.2 Parameter inference

This example illustrates parameter inference for a single model. (Check also the model selection example if you're interested in comparing multiple models.)

This notebook can be downloaded here: Parameter Inference.

We're going to use the following classes from the pyABC package:

- **ABCSMC**, our entry point to parameter inference,
- **RV**, to define the prior over a single parameter,
- **Distribution**, to define the prior over a possibly higher dimensional parameter space,
- **MultivariateNormalTransition**, to do a kernel density estimate (KDE) for visualization purposes.

Let's start to import the necessary classes. We also set up matplotlib and we’re going to use pandas as well.

```python
[1]: import pyabc
import scipy as sp
import scipy.stats as st
import tempfile
import os
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

Our model is about as simple as it gets. We assume a Gaussian model \( \mathcal{N}(\text{mean}, 0.5^2) \) with the single parameter mean. The variance is 0.5². In this case, the parameter dictionary that is passed to the model has only the single key `mean`. We name the sampled data just `data`. It might look like overcomplicating things to return a whole dictionary, but as soon as we return heterogeneous data this starts to make a lot of sense.
We then define the prior for the mean to be uniform over the interval [0, 5):

(Actually, this has to be read as [0, 0+5]. For example, RV("uniform", 1, 5) is uniform over the interval [1, 6]. Check the scipy.stats package for details of the definition.)

We also need to specify when we consider data to be close in form of a distance function. We just take the absolute value of the difference here.

Now we create the ABCSMC object, passing the model, the prior and the distance to it.

To get going, we have to specify where to log the ABC-SMC runs.

We can later query the database with the help of the History class.

Usually you would now have some measure data which you want to know the posterior of. Here, we just assume, that the measured data was 2.5.

The **new** method returned an integer. This is the id of the ABC-SMC run. This id is only important if more than one ABC-SMC run is stored in the same database.

Let’s start the sampling now. We’ll sample until the acceptance threshold epsilon drops below 0.2. We also specify that we want a maximum number of 10 populations. So whatever is reached first, minimum_epsilon or max_nr_populations, will stop further sampling.

For the simple model we defined above, this should only take a couple of seconds:
The `History` object returned by ABCSMC.run can be used to query the database. This object is also available via `abc.history`.

```python
[8]: history is abc.history
[8]: True
```

Now we visualize the probability density functions. The vertical line indicates the location of the observation. Given our model, we expect the mean to be close to the observed data.

```python
[9]: fig, ax = plt.subplots()
for t in range(history.max_t+1):
    df, w = history.get_distribution(m=0, t=t)
    pyabc.visualization.plot_kde_1d(
        df, w,
        xmin=0, xmax=5,
        x="mean", ax=ax,
        label="PDF t={}".format(t))
ax.axvline(observation, color="k", linestyle="dashed");
ax.legend();
```

![Probability density functions](image)

pyABC also offers various other visualization routines in order to analyze the parameter estimation run:

```python
[10]: pyabc.visualization.plot_sample_numbers(history)
pyabc.visualization.plot_epsilons(history)
pyabc.visualization.plot_confidence_intervals(history)
pyabc.visualization.plot_effective_sample_sizes(history)
```

```python
[10]: <matplotlib.axes._subplots.AxesSubplot at 0x7f70b4b09ba8>
```
3.2. Parameter inference
That's it. Now you can go ahead and try more sophisticated models.

### 3.3 Early stopping of model simulations

This notebook can be downloaded here: Early Stopping.

For certain distance functions and certain models it is possible to calculate the distance on-the-fly while the model is running. This is e.g. possible if the distance is calculated as a cumulative sum and the model is a stochastic process. For example, Markov Jump Processes belong to this class. However, we want to keep things simple here and only demonstrate how to use the pyABC interface in such cases. So don’t expect a sophisticated (or even useful) model implementation here.

In this example we’ll use in particular the following classes for integrated simulation and accepting/rejecting a parameter:

- `IntegratedModel`
Let's start with the necessary imports:

```python
%matplotlib inline
from pyabc import ABCSMC, RV, Distribution, IntegratedModel, ModelResult, MedianEpsilon, LocalTransition
from pyabc.sampler import SingleCoreSampler
import matplotlib.pyplot as plt
import os
import tempfile
import pandas as pd
import scipy as sp

db_path = ("sqlite:///" +
           os.path.join(tempfile.gettempdir(), "test.db"))
```

We define here a (very) simple stochastic process, purely for demonstrative reasons. First, we fix the number of steps `n_steps` to 30.

```python
n_steps = 30
```

We then define our process as follows:

\[ x(t + 1) = x(t) + s\xi, \]

in which \( \xi \sim U(0, 1) \) denotes a uniformly in \([0, 1]\) distributed random variable, and \( s \) is the step size, \( s = \) step_size.

The function `simulate` implements this stochastic process:

```python
def simulate(step_size):
    trajectory = sp.zeros(n_steps)
    for t in range(1, n_steps):
        xi = sp.rand()
        trajectory[t] = trajectory[t-1] + xi * step_size
    return trajectory
```

We take as distance function between two such generated trajectories the sum of the absolute values of the pointwise differences.

```python
def distance(trajectory_1, trajectory_2):
    return sp.absolute(trajectory_1 - trajectory_2).sum()
```

Let's run the simulation and plot the trajectories to get a better idea of the so generated data. We set the ground truth step size `gt_step_size` to

```python
gt_step_size = 5
```

This will be used to generate the data which will be subject to inference later on.
dist_1_2 = distance(gt_trajectory, trajectory_2)

plt.plot(gt_trajectory,
         label="Step size = {} (Ground Truth)".format(gt_step_size))
plt.plot(trajectory_2,
         label="Step size = 2")
plt.legend();
plt.title("Distance={:.2f}".format(dist_1_2));

As you might have noted already we could calculate the distance on the fly. After each step in the stochastic process, we could increment the cumulative sum. This will supposedly save time in the ABC-SMC run later on.

To implement this in pyABC we use the \texttt{IntegratedModel}.

Let's start with the code first and explain it afterwards.

```python
[7]:
class MyStochasticProcess(IntegratedModel):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.n_early_stopped = 0

    def integrated_simulate(self, pars, eps):
        cumsum = 0
        trajectory = sp.zeros(n_steps)
        for t in range(1, n_steps):
            xi = sp.rand()
            next_val = trajectory[t-1] + xi * pars["step_size"]
            cumsum += abs(next_val - gt_trajectory[t])
            trajectory[t] = next_val
            if cumsum > eps:
                self.n_early_stopped += 1
                return ModelResult(accepted=False)

        return ModelResult(accepted=True,
                            distance=cumsum,
                            sum_stats={"trajectory": trajectory})
```

Our \texttt{MyStochasticProcess} class is a subclass of \texttt{IntegratedModel <pyabc.model.IntegratedModel>}. 
The `__init__` method is not really necessary. Here, we just want to keep track of how often early stopping has actually happened.

More interesting is the `integrated_simulate` method. This is where the real thing happens. As already said, we calculate the cumulative sum on the fly. In each simulation step, we update the cumulative sum. Note that `gt_trajectory` is actually a global variable here. If `cumsum > eps` at some step of the simulation, we return immediately, indicating that the parameter was not accepted by returning `ModelResult(accepted=False)`. If the `cumsum` never passed `eps`, the parameter got accepted. In this case we return an accepted result together with the calculated distance and the trajectory. Note that, while it is mandatory to return the distance, returning the trajectory is optional. If it is returned, it is stored in the database.

We define a uniform prior over the interval $[0, 10]$ over the step size

```python
[8]: prior = Distribution(step_size=RV("uniform", 0, 10))
```

and create an instance of our integrated model `MyStochasticProcess`

```python
[9]: model = MyStochasticProcess()
```

We then configure the ABC-SMC run. As the distance function is calculated within `MyStochasticProcess`, we just pass `None` to the `distance_function` parameter. As sampler, we use the `SingleCoreSampler` here. We do so to correctly keep track of `MyStochasticProcess.n_early_stopped`. Otherwise, the counter gets incremented in subprocesses and we don’t see anything here. Of course, you could also use the `MyStochasticProcess` model in a multi-core or distributed setting.

Importantly, we pre-specify the initial acceptance threshold to a given value, here to 300. Otherwise, pyABC will try to automatically determine it by drawing samples from the prior and evaluating the distance function. However, we do not have a distance function here, so this approach would break down.

```python
[10]: abc = ABCSMC(models=model,  
    parameter_priors=prior,  
    distance_function=None,  
    sampler=SingleCoreSampler(),  
    population_size=30,  
    transitions=LocalTransition(k_fraction=.2),  
    eps=MedianEpsilon(300, median_multiplier=0.7))
```

We then indicate that we want to start a new ABC-SMC run:

```python
[11]: abc.new(db_path)
```

We do not need to pass any data here. However, we could still pass additionally a dictionary `{"trajectory": gt_trajectory}` only for storage purposes to the `new` method. The data will however be ignored during the ABC-SMC run.

Then, let’s start the sampling

```python
[12]: h = abc.run(minimum_epsilon=40, max_nr_populations=3)
```

3.3. Early stopping of model simulations
and check how often the early stopping was used:

```python
[13]: model.n_early_stopped
[13]: 1069

Quite a lot actually.

Lastly we estimate KDEs of the different populations to inspect our results and plot everything (the vertical dashed line is the ground truth step size).

```python
[14]: from pyabc.visualization import plot_kde_1d
    fig, ax = plt.subplots()
    for t in range(h.max_t+1):
        particles = h.get_distribution(m=0, t=t)
        plot_kde_1d(*particles, "step_size",
                     label="t={}".format(t), ax=ax,
                     xmin=0, xmax=10, numx=300)
    ax.axvline(gt_step_size, color="k", linestyle="dashed");
```

That’s it. You should be able to see how the distribution contracts around the true parameter.

### 3.4 Resuming stored ABC runs

In this example, it is illustrated how stored ABC runs can be loaded and continued later on. This might make sense if you decide later on to run a couple more populations for increased accuracy.

The models used in this example are similar to the ones from the parameter inference tutorial. This notebook can be downloaded here: Resuming stored ABC runs.

In this example, we’re going to use the following classes:

- `ABCSMC`, our entry point to parameter inference,
- `RV`, to define the prior over a single parameter,
- `Distribution`, to define the prior over a possibly higher dimensional parameter space,

Let’s start with the imports.
As usually, we start with the definition of the model, the prior and the distance function.

```python
[2]: def model(parameter):
    return {"data": parameter["mean"] + sp.randn()}

prior = Distribution(mean=RV("uniform", 0, 5))

[3]: def distance(x, y):
    return abs(x["data"] - y["data"])
```

We next make a new ABC-SMC run and also print the id of this run. We’ll use the id later on to resume the run.

```python
[4]: abc = ABCSMC(model, prior, distance)
run_id = abc.new(db, {"data": 2.5})
print("Run ID: ", run_id)
```

Let’s verify that we have 3 populations.

```python
[5]: history.n_populations
```

We now create a completely new ABCSMC object. We pass the same model, prior and distance from before.

```python
[6]: abc_continued = ABCSMC(model, prior, distance)
```

Note: You could actually pass different models, priors and distance functions here. This might make sense if, for example, in the meantime you came up with a more efficient model implementation or distance function.

For the experts: under certain circumstances it can even be mathematically correct to change the prior after a couple of populations.

To resume a run, we use the `load` method. This loads the necessary data. We pass to this method the id of the run we want to continue.

---

### 3.4. Resuming stored ABC runs

---
Let's check the number of populations of the resumed run. It should be 4, as we did 3 populations before and added another one.

That's it. This was a basic tutorial on how to continue stored ABC-SMC runs.

---

**Note:** For advanced users:

In situations where the distance function or epsilon require initialization, it is possible that resuming a run via load(), we lose information because not everything can be stored in the database. This concerns hyper-parameters in individual objects specified by the user.

If that is the case, however the user can somehow store e.g. the distance function used in the first run, and pass this very object to abc_continued. Then it is ideally fully initialized, so that setting distance_function.require_initialize = False, it is just as if the first run had not been interrupted.

However, even if information was lost, after load() the process usually quickly re-adjusts itself in 1 or 2 iterations, so that this is not much of a problem.

---

### 3.5 Using R with pyABC

This example illustrates how to use models, summary statistics and distance functions defined in R. We're assuming you're already familiar with the basic workings of pyABC. If not, consult the other tutorial examples.

Download this notebook Using R with pyABC.

In this example, we're introducing the new class `R` which is our interface with R. We use this class to to load an external R script.

```python
%matplotlib inline
from pyabc.external import R
r = R("myRModel.R")
/home/yannik/pyabc/pyabc/external.py:58: UserWarning: The support of the R language for ABC-SMC is considered experimental. The API might change in the future.
warnings.warn("The support of the R language for ABC-SMC is "
```
Note:  `R("myRModel.R")` does, amongst other things, the equivalent to R’s `source("myRModel.R")`. That is, the entire script is executed with all the possible side effects this might have.

You can download the file here: `myRModel.R`. But now, let’s have a look at it.

```python
r.display_source_ipython()
```

We see that four relevant objects are defined in the file.

- `myModel`
- `mySummaryStatistics` (optional)
- `myDistance`
- `mySumStatData`

The names of these do not matter. The `mySummaryStatistics` is actually optional and can be omitted in case the model calculates the summary statistics directly. We load the defined functions using the `r` object:

```python
model = r.model("myModel")
distance = r.distance("myDistance")
sum_stat = r.summary_statistics("mySummaryStatistics")
```

From there on, we can use them (almost) as if they were ordinary Python functions.

```python
from pyabc import Distribution, RV, ABCSMC
prior = Distribution(meanX=RV("uniform", 0, 10),
                      meanY=RV("uniform", 0, 10))
abc = ABCSMC(model, prior, distance,
             summary_statistics=sum_stat)
```

We also load the observation with `r.observation` and pass it to a new ABC run.

```python
import os
from tempfile import gettempdir
db = "sqlite:///" + os.path.join(gettempdir(), "test.db")
abc.new(db, r.observation("mySumStatData"))
```

We start a run which terminates as soon as an acceptance threshold of 0.9 or less is reached or the maximum number of 4 populations is sampled.

```python
history = abc.run(minimum_epsilon=0.9, max_nr_populations=4)
```
Lastly, we plot the results and observe how the generations contract slowly around the observed value. (Note, that the contraction around the observed value is a particular property of the chosen example and not always the case.)

```python
from pyabc.visualization import plot_kde_2d

for t in range(history.n_populations):
    df, w = abc.history.get_distribution(0, t)
    ax = plot_kde_2d(df, w, "meanX", "meanY",
                     xmin=0, xmax=10,
                     ymin=0, ymax=10,
                     numx=100, numy=100)
    ax.scatter([4], [8],
               edgecolor="black",
               facecolor="white",
               label="Observation");
    ax.legend();
    ax.set_title("PDF t={}".format(t))
```

![PDF t=0](image)
And we can also retrieve summary statistics such as a stored DataFrame, although the DataFrame was actually defined in R.

```python
history.get_weighted_sum_stats_for_model(m=0, t=1)[1][0]["cars"].head()
```

<table>
<thead>
<tr>
<th>speed</th>
<th>dist</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>4.0</td>
<td>10.0</td>
</tr>
<tr>
<td>7.0</td>
<td>4.0</td>
</tr>
<tr>
<td>7.0</td>
<td>22.0</td>
</tr>
<tr>
<td>8.0</td>
<td>16.0</td>
</tr>
</tbody>
</table>

### 3.5.1 Dumping the results to a file format R can read

Although you could query pyABC’s database directly from R since the database is just a SQL database (e.g. SQLite), pyABC ships with a utility for facilitate export of the database. Use the `abc-dump` utility provided by pyABC to dump results to file formats such as csv, feather, html, json and others. These can be easily read in by R. See Exporting pyABC’s database for how to use this utility.

Assume you dumped to the feather format:

```bash
abc-export --db results.db --out exported.feather --format feather
```

You could read the results in with the following R snippet

```r
install.packages("feather")
install.packages("jsonlite")

library("feather")
library("jsonlite")

loadedDf <- data.frame(feather("exported.feather"))

jsonStr <- loadedDf$sumstat_ss_df[1]

sumStatDf <- fromJSON(jsonStr)
```
If you prefer CSV over the feather format you can also do that.

3.6 Ordinary Differential Equations: Conversion Reaction

This example was kindly contributed by Lukas Sandmeir and Elba Raimundez. It can be downloaded here: Ordinary Differential Equations.

This example provides a model for the interconversion of two species ($X_1$ and $X_2$) following first-order mass action kinetics with the parameters $\Theta_1$ and $\Theta_2$ respectively:

\[
X_1 \rightarrow X_2, \quad \text{rate} = \Theta_1 \cdot [X_1]
\]

\[
X_2 \rightarrow X_1, \quad \text{rate} = \Theta_2 \cdot [X_2]
\]

Measurement of $[X_2]$ is provided as $Y = [X_2]$.

We will show how to estimate $\Theta_1$ and $\Theta_2$ using pyABC.

```
[1]: %matplotlib inline
from pyabc import (ABCSMC,
                   RV, Distribution,
                   MedianEpsilon,
                   LocalTransition)
from pyabc.visualization import plot_kde_2d
import matplotlib.pyplot as plt
import os
import tempfile
import scipy as sp

db_path = ("sqlite:///" +
            os.path.join(tempfile.gettempdir(), "test.db"))
```

3.6.1 Data

We use an artificial data set which consists of a vector of time points $t$ and a measurement vector $Y$. This data was created using the parameter values which are assigned to $\Theta_{\text{true}}$ and by adding normally distributed measurement noise with variance $\sigma^2 = 0.015^2$.

3.6.2 ODE model

\[
\frac{dX_1}{dt} = -\Theta_1 \cdot X_1 + \Theta_2 \cdot X_2
\]

\[
\frac{dX_2}{dt} = \Theta_1 \cdot X_1 - \Theta_2 \cdot X_2
\]

Define the true parameters

```
[2]: theta1_true, theta2_true = sp.exp([-2.5, -2])
```

and the measurement data
as well as the time points at which to evaluate

and the initial conditions for $X_1$ and $X_2$

Define the ODE model

Integration of the ODE model for the true parameter values

Let’s visualize the results
Define the prior for $\Theta_1$ and $\Theta_2$

```python
[10]:
    parameter_prior = Distribution(theta1=RV("uniform", 0, 1),
                               theta2=RV("uniform", 0, 1))
    parameter_prior.get_parameter_names()
[10]: ['theta1', 'theta2']
```

```python
[11]:
    abc = ABCSMC(models=model,
                 parameter_priors=parameter_prior,
                 distance_function=distance,
                 population_size=50,
                 transitions=LocalTransition(k_fraction=.3),
                 eps=MedianEpsilon(500, median_multiplier=0.7))
```

```python
[12]:
    abc.new(db_path, {"X_2": measurement_data});
```

Visualization of the probability density functions for $\Theta_1$ and $\Theta_2$
for t in range(h.max_t+1):
    ax = plot_kde_2d(*h.get_distribution(m=0, t=t),
        "theta1", "theta2",
        xmin=0, xmax=1, numx=300,
        ymin=0, ymax=1, numy=300)
    ax.scatter([theta1_true], [theta2_true],
        color="C1",
        label='$\Theta$ true = {:.3f}, {:.3f}'.format(
            theta1_true, theta2_true))
    ax.set_title("Posterior t={}").format(t)
    ax.legend()
3.7 Markov Jump Process: Reaction Network

In the following, we fit stochastic chemical reaction kinetics with pyABC and show how to perform model selection between two competing models.

This notebook can be downloaded here: Markov Jump Process: Reaction Network.

We consider the two Markov jump process models \( m_1 \) and \( m_2 \) for conversion of (chemical) species \( X \) to species \( Y \):

\[
m_1 : X + Y \xrightarrow{k_1} 2Y \\
m_2 : X \xrightarrow{k_2} Y.
\]

Each model is equipped with a single rate parameter \( k \). To simulate these models, we define a simple Gillespie simulator:

```
[1]: import scipy as sp

def h(x, pre, c):
    return (x**pre).prod(1) * c

def gillespie(x, c, pre, post, max_t):
    """
    Gillespie simulation

    Parameters
    --------

    x: 1D array of size n_species
    The initial numbers.

    c: 1D array of size n_reactions
    The reaction rates.

    pre: array of size n_reactions x n_species
    What is to be consumed.
    """
```

(continues on next page)
Next, we define the models in terms of their initial molecule numbers $x_0$, an array `pre` which determines what is to be consumed (the left hand side of the reaction equations) and an array `post` which determines what is to be produced (the right hand side of the reaction equations). Moreover, we define that the simulation time should not exceed `MAX_T` seconds.

Model 1 starts with initial concentrations $X = 40$ and $Y = 3$. The reaction $X + Y \rightarrow 2Y$ is encoded in `pre = [[1, 1]]` and `post = [[0, 2]]`. 

```python
class Model1:
    __name__ = "Model 1"
    x0 = sp.array([40, 3])  # Initial molecule numbers
    pre = sp.array([[1, 1]], dtype=int)
    post = sp.array([[0, 2]])
```

[2]: MAX_T = 0.1
def __call__(self, par):
    t, X = gillespie(self.x0,
                    sp.array([float(par["rate"])]),
                    self.pre, self.post,
                    MAX_T)
    return {"t": t, "X" : X}

Model 2 inherits the initial concentration from model 1. The reaction $X \rightarrow Y$ is encoded in $\text{pre} = [[1, 0]]$ and $\text{post} = [[0, 1]]$.

We draw one stochastic simulation from model 1 (the “Observation”) and one from model 2 (the “Competition”) and visualize both.

We observe that species $X$ is converted into species $Y$ in both cases. The difference of the concentrations over time can be quite subtle.
We define a distance function as \( L_1 \) norm of two trajectories, evaluated at 20 time points:

\[
\text{distance}(X_1, X_2) = \frac{1}{N} \sum_{n=1}^{N} |X_1(t_n) - X_2(t_n)|, \quad t_n = \frac{n}{N} T, \quad N = 20.
\]

Note that we only consider the concentration of species \( Y \) for distance calculation. And in code:

```python
[5]:
N_TEST_TIMES = 20

t_test_times = sp.linspace(0, MAX_T, N_TEST_TIMES)
def distance(x, y):
    xt_ind = sp.searchsorted(x["t"], t_test_times) - 1
    yt_ind = sp.searchsorted(y["t"], t_test_times) - 1
    error = (sp.absolute(x["X"][:,1][xt_ind] - y["X"][:,1][yt_ind]).sum() / t_test_times.size)
    return error
```

For ABC, we choose for both models a uniform prior over the interval \([0, 100]\) for their single rate parameters:

```python
[6]:
from pyabc import Distribution, RV
prior = Distribution(rate=RV("uniform", 0, 100))
```

We initialize the ABCSMC class passing the two models, their priors and the distance function.

```python
[7]:
from pyabc import ABCSMC
from pyabc.populationstrategy import AdaptivePopulationSize
abc = ABCSMC([Model1(), Model2()], [prior, prior], distance, population_size=AdaptivePopulationSize(500, 0.15))
```

We initialize a new ABC run, taking as observed data the one generated by model 1. The ABC run is to be stored in the sqlite database located at `/tmp/mjp.db`.

```python
[8]:
abc_id = abc.new("sqlite:///tmp/mjp.db", observations[0])
INFO:History:Start <ABCSMC(id=4, start_time=2018-05-08 16:16:00.358367, \])( endpoint=None)\>
INFO:Epsilon:initial epsilon is 14.074999999999992
```

We start pyABC which automatically parallelizes across all available cores.

```python
[9]:
history = abc.run(minimum_epsilon=0.7, max_nr_populations=15)
INFO:ABC:t:0 eps:14.074999999999992
INFO:ABC:t:1 eps:10.6
INFO:Adaptation:Change nr particles 500 -> 101
INFO:ABC:t:2 eps:7.457087679174012
INFO:Adaptation:Change nr particles 101 -> 85
INFO:ABC:t:3 eps:6.270538173626088
INFO:Adaptation:Change nr particles 85 -> 82
INFO:ABC:t:4 eps:5.3
INFO:Adaptation:Change nr particles 82 -> 89
INFO:ABC:t:5 eps:4.85
```

(continues on next page)
INFO:Adaptation:Change nr particles 89 -> 97
INFO:ABC:t:6 eps:4.281990310808671
INFO:Adaptation:Change nr particles 97 -> 107
INFO:ABC:t:7 eps:3.336777623025941
INFO:Adaptation:Change nr particles 107 -> 100
INFO:ABC:t:8 eps:2.3319007349977956
INFO:Adaptation:Change nr particles 100 -> 57
INFO:ABC:t:9 eps:1.8334187576797563
INFO:Adaptation:Change nr particles 57 -> 54
INFO:ABC:t:10 eps:1.55
INFO:Adaptation:Change nr particles 54 -> 53
INFO:ABC:t:11 eps:1.35
INFO:Adaptation:Change nr particles 53 -> 56
INFO:ABC:t:12 eps:1.2286362088943308
INFO:Adaptation:Change nr particles 56 -> 60
INFO:ABC:t:13 eps:1.05
INFO:Adaptation:Change nr particles 60 -> 44
INFO:ABC:t:14 eps:0.95
INFO:Adaptation:Change nr particles 44 -> 53
INFO:History:Done <ABCSMC(id=4, start_time=2018-05-08 16:16:00.358367, end_time=2018-05-08 16:30:48.219049)>

We first inspect the model probabilities.

```python
[10]: ax = history.get_model_probabilities().plot.bar();
ax.set_ylabel("Probability");
ax.set_xlabel("Generation");
ax.legend([1, 2], title="Model", ncol=2,
          loc="lower center", bbox_to_anchor=(.5, 1));
```

![Probability vs Generation](image)

The mass at model 2 decreased, the mass at model 1 increased slowly. The correct model 1 is detected towards the later generations. We then inspect the distribution of the rate parameters:

```python
[11]: from pyabc.visualization import plot_kde_1d
fig, axes = plt.subplots(2)
(continues on next page)```
fig.set_size_inches((6, 6))
axes = axes.flatten()
axes[0].axvline(true_rate, color="black", linestyle="dotted")

for m, ax in enumerate(axes):
    for t in range(0, history.n_populations, 2):
        df, w = history.get_distribution(m=m, t=t)
        if len(w) > 0:  # Particles in a model might die out
            plot_kde_1d(df, w, "rate", ax=ax, label=f"t={t}",
                        xmin=0, xmax=20 if m == 0 else 100,
                        numx=200)
        ax.set_title(f"Model {m+1}")

axes[0].legend(title="Generation",
                loc="upper left", bbox_to_anchor=(1, 1))

fig.tight_layout()

The true rate is closely approximated by the posterior over the rate of model 1. It is a little harder to interpret the posterior over model 2. Apparently a rate between 20 and 40 yields data most similar to the observed data.

Lastly, we visualize the evolution of the population sizes. The population sizes were automatically selected by pyABC and varied over the course of the generations. (We do not plot the size of the first generation, which was set to 500)

[12]: populations = history.get_all_populations()
ax = populations[populations.t >= 1].plot("t", "particles",
                                                style = "o-"
ax.set_xlabel("Generation")

3.7. Markov Jump Process: Reaction Network
The initially chosen population size was adapted to the desired target accuracy. A larger population size was automatically selected by pyABC while both models were still alive. The population size decreased during the later populations thereby saving computational time.

### 3.8 Multi-scale model: Tumor spheroid growth

Download this notebook here: Multi-scale model: Tumor spheroid growth.

#### 3.8.1 Installation prerequisites

To run this example, install the tumor2d package available here: Agent-based 2D Tumor growth model. For details on the model, see Jagiella, Rickert, Theis, Hasenauer (2017). (Note that the tumor2d package is not part of the paper by Jagiella, Rickert, Theis, Hasenauer 2017, but was specifically developed and adapted for pyABC, however based on the original source code).

#### 3.8.2 Model description

Briefly, the model is a multi-scale model describing the growth of a 2D tumor spheroid (see animations below). It is a hybrid discrete-continuum model which exploits an agent-based description for individual cells and a PDE-based description for the extracellular matrix. The intra-cellular mechanism of cell-division is described by continuous-time Markov chains and decision rules. In particular the initial phase of the growth is substantially stochastic. During the later phases, with higher cell numbers, averaging effects occur.

The proliferating cells are depicted in orange

Extracellular matrix intensity is coded in the intensity of the green color

To simulate a growing spheroid, we employ the simulate function provided by the tumor2d package.
The simulation took a little time. The simulate function does not return each individual cell. Instead, we “measure” the radial density of proliferating cells, the radial extracellular matrix density and the spheroid radius over time. These statistics summarize the underlying data.
3.8.3 Exemplary pyABC inference run with the model

To perform parameter inference with pyABC, we first define a prior over the seven parameters. We do this on the log-domain.

```python
[3]: from pyabc import Distribution, RV
    limits = dict(log_division_rate=(-3, -1),
                   log_division_depth=(1, 3),
                   log_initial_spheroid_radius=(0, 1.2),
                   log_initial_quiescent_cell_fraction=(-5, 0),
                   log_ecm_production_rate=(-5, 0),
                   log_ecm_degradation_rate=(-5, 0),
                   log_ecm_division_threshold=(-5, 0))
    prior = Distribution(**{key: RV("uniform", a, b - a)
                           for key, (a,b) in limits.items()})
```

Note that we define the `limits` dictionary for later visualization usages and then initialize the prior indirectly to not to repeat ourselves.

Each spheroid simulation runs already in its own subprocess, which is forked off from the main process by the simulate function. We therefore do not need to parallelize using multiprocessing, but will instead use multithreading. pyABC ships with a flexible sampler which accepts any backend implementing the `concurrent.futures.Executor` interface. For instance, we can use the `ThreadPoolExecutor`. For distributed execution the IPython parallel client would be an alternative, but the `ThreadPoolExecutor` is good enough for illustrative purposes.

Since we’ve defined the priors on the log domain, we also use a model accepting log-transformed parameters. This model is defined as `tumor2d.log_model`. It is a thin wrapper around the `tumor2d.simulate` function and transforms the parameters accordingly before executing the simulate function.

The tumor2d package also provides a distance function and default data. These data were obtained from 100 samples from the so called “reference parameters”, which we also used above to simulate one growing spheroid. We take the mean from these 100 simulations as the observed data. The distance function calculates $L_2$ norms for each of the three summary statistics and adds them up. (These $L_2$ norms are normalized by the pointwise variances.)

```python
[4]: from tumor2d import log_model, distance, load_default
from pyabc import ABCSMC
from pyabc.sampler import ConcurrentFutureSampler
from concurrent.futures import ThreadPoolExecutor
```

(continues on next page)
As usually, we initialize a new ABC inference run with the `new` method.

```python
[5]: abc.new("sqlite:///tmp/test.db", data_mean)
INFO:Epsilon:initial epsilon is 22450777.905396923
INFO:History:Start <ABCSMC(id=6, start_time=2018-04-11 08:43:27.549224, end_time=None)>
```

Since the inference would run very long on a single local machine, we sample only a single population for illustration purposes. Note, that we’ve also set the number of particles to a very small number. This is much too low for a real inference run, but serves here for illustration purposes and saves sampling time.

```python
[6]: history = abc.run(max_nr_populations=1, minimum_epsilon=0)
INFO:ABC:t:0 eps:22450777.905396923
INFO:History:Done <ABCSMC(id=6, start_time=2018-04-11 08:43:27.549224, end_time=2018-04-11 08:46:15.690193)>
```

We visualize the first population with pyABC’s `plot_kde_matrix` function.

```python
[7]: from pyabc.visualization import plot_kde_matrix

df, w = history.get_distribution(m=0)
plot_kde_matrix(df, w, limits=limits);
```
We see kernel density estimates on the lower diagonal and scatter plots of the weighted particles on the upper diagonal. Note that the weight of the particles is not encoded in the scatter plots. On the diagonal we see the marginal distributions.

### 3.8.4 Results of a stored, distributedly executed pyABC inference run

We’ve run the inference described above in a distributed fashion on one of our clusters on about 800 cores simultaneously. The stored data is provided as part of the tumor2d package. (Note, that the summary statistics were actually also stored in the database. However, to not to distribute a 200MB database, we deleted them afterwards. So they are not contained in the databases distributed with the tumor2d package. If you’re interested in the full database, don’t hesitate to contact us)

We load the stored database and visualize generation 5.
from pyabc import History
from tumor2d import stored_data_db

h_loaded = History("sqlite:///" + stored_data_db)

df, w = h_loaded.get_distribution(m=0, t=5)
plot_kde_matrix(df, w, limits=limits);

The posterior is still quite broad. At the last generation the posterior is notably sharper:

df, w = h_loaded.get_distribution(m=0)
plot_kde_matrix(df, w, limits=limits);

3.8. Multi-scale model: Tumor spheroid growth
3.8.5 Adaptive population sizes

For this run, we employed one of pyABC’s unique features: Adaptive population sizes. We initialized with population size 500. pyABC tuned the necessary population size automatically.

```python
[10]: populations = h_loaded.get_all_populations()
    populations[populations.t >= 0]
    .plot("t", "particles", marker="o");
```
In this example, the population size was roughly constant, however, this is not always the case. The full execution of this notebook takes a little due to the tumor growth simulations performed:

```python
print(f"Execution time: {((time() - start_time)/60:.1f})m")
```
Execution time: 8.8m

### 3.8.6 The complete stored inference

We also compiled an animated gif from the complete course of the generations. Observe how the posterior slowly contracts:

![Particles over time](image)

### 3.9 Stochastic Differential Equation: Ion channel noise in Hodgkin-Huxley neurons

Download this notebook here: Stochastic Differential Equation: Ion channel noise in Hodgkin-Huxley neurons

In the following, we estimate parameters of the stochastic differential equation model of ion channel noise in Hodgkin-Huxley neurons presented in:


The code was implemented in Fortran 95 and made available in ModelDB: ModelDB. (The code is not included in pyABC and neither developed nor maintained by the pyABC developers.)

#### 3.9.1 Download and compilation of the Fortran model

We start by downloading the code from ModelDB. For this, the `requests` package is needed.
The zip file to which URL points is stored in memory. The code is then extracted into a temporary directory and compiled using make HH_run provided as part of the download from ModelDB. The Fortran compiler gfortran is required for compilation.

The executable location is /tmp/tmpujmwnetn/ModelDBFolder/HH_run

The variable EXEC points to the executable.

A simulate function is defined which uses the subprocess.run function to execute the external binary. The external binary writes to stdout. The output is captured and stored in a pandas dataframe. This dataframe is returned by the simulate function.
3.9.2 Generating the observed data

We run a simulations and plot the fraction of open “K” channels and open “Na” channels:

```python
import matplotlib.pyplot as plt
%matplotlib inline

gt = {"dc": 20, "membrane_dim": 10}
fig, axes = plt.subplots(nrows=2, sharex=True)
fig.set_size_inches((12,8))
for _ in range(10):
    observation = simulate(**gt)
    observation.plot(y="K", color="C1", ax=axes[0]);
    observation.plot(y="Na", color="C0", ax=axes[1]);
for ax in axes:
    ax.legend().set_visible(False)
axes[0].set_title("K")
axes[0].set_ylabel("K")
axes[1].set_title("Na")
axes[1].set_ylabel("Na");
```

```python
str(voltage_clamp), str(data_to_print),
str(rng_seed)],
stdout=subprocess.PIPE)
df = pd.read_table(BytesIO(re.stdout),
delim_whitespace=True,
header=None, index_col=0,
names=["t", "V", "Na", "K"])
return df
```
We observe how the channels open and close and also that the individual trajectories differ from realization to realization, even though we simulate for the exact same parameter set. We take the last simulation as observed data.

### 3.9.3 Defining distance and prior

We’ll now demonstrate how to use pyABC to estimate parameters of the model. Here, we’ll focus on the $dc$ and the $membrane\_dim$ parameters. The $dc$ parameter describes the input current, the $membrane\_dim$ is the square root of the membrane area. We choose uniform priors:

```
[5]: from pyabc import Distribution, RV, ABCSMC

dcmin, dcmax = 2, 30
memmin, memmax = 1, 12
prior = Distribution(
    dc=RV("uniform", dcmin, dcmax - dcmin),
    membrane_dim=RV("uniform", memmin, memmax - memmin))
```

The distance function is defined as $L_2$ norm between the fractions of open “K” channels.

```
[6]: def distance(x, y):
    diff = x[\"data\"]\[\"K\"] - y[\"data\"]\[\"K\"]
    dist = np.sqrt(np.sum(diff**2))
    return dist
```

We also define a small `simulate_pyabc` wrapper function, which wraps the `simulate` function. This is needed to comply with the interface expected by pyABC.
3.9.4 Performing parameter inference with pyABC

We are now ready to start pyABC. As usually, we first initialize the ABCSMC object, then pass the observed data and the database location in which to store the logged parameters and summary statistics, and finally run the inference until the maximum number of allowed populations $\text{max}_\text{nr}_\text{populations}$ or the final acceptance threshold $\text{minimum}_\text{epsilon}$ is reached.

```python
abc = ABCSMC(simulate_pyabc, prior, distance,
    population_size=35)
abc_id = abc.new("sqlite:///"
    + os.path.join(tempdir, "test.db"),
    "data": observation)
history = abc.run(max_nr_populations=10, minimum_epsilon=6)
```

3.9.5 Visualization of the estimated parameters

We plot the posterior distribution after a few generations together with the parameters generating the observed data (the dotted line and the orange dot).

```python
from pyabc.visualization import plot_kde_matrix

dfw = history.get_distribution(m=0)
grid = plot_kde_matrix(dfw,
    limits={"dc": (dcmin, dcmax),
        "membrane_dim": (memmin, memmax)})
grid.map_diag(lambda x, **kwargs: plt.gca().axvline(
    gt[x.name], color="k", linestyle="dotted"))
grid.map_lower(lambda x, y, **kwargs: plt.gca().scatter(
    [gt[x.name]], [gt[y.name]], color="orange"))
plt.gcf().set_size_inches(8, 8)
```
The $dc$ parameter is very well detected. (Don’t get confused by the y-axis. It applies to the scatterplot, not to the marginal distribution.) The distribution of $membrane\_dim$ is broader. (Note that even the exact posterior is not necessarily peaked at the ground truth parameters).

### 3.9.6 Evaluation of the fit

We compare four types of data:

1. samples from the prior distribution,
2. samples from the posterior distribution,
3. the data generated by the accepted parameters, and
4. the observation.

```python
[10]: from pyabc.transition import MultivariateNormalTransition
```

```python
fig, axes = plt.subplots(nrows=3, sharex=True)
fig.set_size_inches(8, 12)
```

(continues on next page)
n = 5  # Number of samples to plot from each category
# Plot samples from the prior
alpha = .5
for _ in range(n):
    prior_sample = simulate(**prior.rvs())
    prior_sample.plot(y="K", ax=axes[0],
                      color="C1", alpha=alpha)

# Fit a posterior KDE and plot samples form it
posterior = MultivariateNormalTransition()
prior_sample = simulate(**prior.rvs())
prior_sample.plot(y="K", ax=axes[0],
                   color="C1", alpha=alpha)

for _ in range(n):
    posterior_sample = simulate(**posterior.rvs())
    posterior_sample.plot(y="K", ax=axes[1],
                           color="C0", alpha=alpha)

# Plot the stored summary statistics
sum_stats = history.get_weighted_sum_stats_for_model(m=0, t=history.max_t)
for stored in sum_stats[1][n):
    stored["data"].plot(y="K", ax=axes[2],
                        color="C2", alpha=alpha)

# Plot the observation
for ax in axes:
    observation.plot(y="K", ax=ax, color="k", linewidth=1.5)
    ax.legend().set_visible(False)
    ax.set_ylabel("K");

# Add a legend with pseudo artists to first plot
axes[0].legend([plt.plot([0], color="C1")[0],
                plt.plot([0], color="C0")[0],
                plt.plot([0], color="C2")[0],
                plt.plot([0], color="k")[0]],
               ["Prior", "Posterior",
                "Stored, accepted", "Observation"],
               bbox_to_anchor=(.5, 1),
               loc="lower center",
               ncol=4);
We observe that the samples from the prior exhibit the largest variation and do not resemble the observation well. The samples from the posterior are much closer to the observed data. Even a little bit closer are the samples generated by the accepted parameters. This has at least two reasons: First, the posterior KDE-fit smoothes the particle populations. Second, the sample generated by a parameter that was accepted is biased towards being more similar to the observed data as compared to a random sample from that parameter.
3.10 Adaptive Distances

In this example, we show how and when to use the adaptive distances feature of pyabc. “Adaptive distances” means that the distance function is not pre-defined (e.g. after pre-processing), but evolves over time during the ABC run, depending on the observed summary statistics. This can be useful if different summary statistics vary on different scales, but it is not immediately clear how to weight them. For this case, in adaptive distances weights are adjusted in each iteration so as to balance the impact of all summary statistics on the computed distance.

Currently, adaptively weighted p-norm distances (e.g. Euclidean) are implemented in pyABC, but it is easily possible to define arbitrary adaptive distances.

The notebook can be downloaded [here](#).

For illustration, we consider a simple Gaussian model:

```python
import scipy
import tempfile
import os
import matplotlib.pyplot as pyplot
import pyabc.visualization
import logging
# for debugging
df_logger = logging.getLogger('Distance')
df_logger.setLevel(logging.DEBUG)

# model definition
def model(p):
    return {'ss1': p['theta'] + 1 + 0.1*scipy.randn(),
            'ss2': 2 + 10*scipy.randn()}

# true model parameter
theta_true = 3

# observed summary statistics
observation = {'ss1': theta_true + 1, 'ss2': 2}

# prior distribution
prior = pyabc.Distribution(theta=pyabc.RV('uniform', 0, 10))

# database
db_path = "sqlite:///" + os.path.join(tempfile.gettempdir(), "tmp.db")
```

Summary statistic ss2 has a high variance compared to summary statistic ss1. In addition, ss1 is informative about the model parameters \( \theta \), ss2 not. We expect that the proposal distribution for \( \theta \) iteratively centers around the true value \( \theta = 3 \). Thus, the variability for the sampled ss1 decreases iteratively, while the variability of the sampled ss2 stays approximately constant. If both summary statistics are weighted similarly in the calculation of the distance between sample and observation, there is hence an undesirable high impact of ss2, so that convergence can be slowed down. In contrast, if we weight ss1 higher, we may hope that our estimation of \( \theta \) is improved.

These informal expectations being stated, let us continue with the implementation. First, we consider a non-adaptive Euclidean distance:

```python
[2]: distance = pyabc.PNormDistance(p=2)

abc = pyabc.ABCSMC(model, prior, distance)
```

(continues on next page)
abc.new(db_path, observation)

history0 = abc.run(minimum_epsilon=.1, max_nr_populations=8)

INFO:History:Start <ABCSMC(id=3, start_time=2019-05-01 16:05:34.676351, end_time=None)>
INFO:Epsilon:initial epsilon is 8.323532755675517
INFO:ABC:t:0 eps:8.323532755675517
INFO:ABC:t:1 eps:4.91440739009507
INFO:ABC:t:2 eps:3.4217303156404038
INFO:ABC:t:3 eps:2.370056613878034
INFO:ABC:t:4 eps:1.6268813652396334
INFO:ABC:t:5 eps:1.186727790009716
INFO:ABC:t:6 eps:0.8548463991669453
INFO:ABC:t:7 eps:0.5934272512057183
INFO:History:Done <ABCSMC(id=3, start_time=2019-05-01 16:05:34.676351, end_time=2019-05-01 16:07:50.875925)>

Let us visualize the results for the non-adaptive distance:

```python
# plotting
fig, ax = pyplot.subplots()
for t in range(history0.max_t + 1):
    df, w = history0.get_distribution(m=0, t=t)
    pyabc.visualization.plot_kde_1d(df, w, xmin=0, xmax=10,
        x='theta', ax=ax,
        label="PDF t={}".format(t))
ax.axvline(theta_true, color='k', linestyle='dashed', label="True value")
ax.legend()
```

Second, we consider an adaptive Euclidean distance:

```python
distance_adaptive = pyabc.AdaptivePNormDistance(p=2)
abc = pyabc.ABCSMC(
    model, prior, distance_adaptive,
    (continues on next page)
acceptor = pyabc.acceptor.accept_use_complete_history

abc.new(db_path, observation)

history1 = abc.run(minimum_epsilon=.1, max_nr_populations=8)

INFO:History:Start <ABCSMC(id=4, start_time=2019-05-01 16:07:55.074538,
---end_time=None)>
DEBUG:Distance:update distance weights = {'ss1': 1.5466414404249624, 'ss2': 0.
---4533585595750377}
INFO:Epsilon:initial epsilon is 6.043428511588377
INFO:ABC:t:0 eps:6.043428511588377
DEBUG:Distance:update distance weights = {'ss1': 1.5570776915018665, 'ss2': 0.
---442923084981334}
INFO:ABC:t:1 eps:4.389515937519
DEBUG:Distance:update distance weights = {'ss1': 1.6713705978285887, 'ss2': 0.
---3262940217141107}
INFO:ABC:t:2 eps:2.804497748453505
DEBUG:Distance:update distance weights = {'ss1': 1.7212145149951485, 'ss2': 0.
---278784850048517}
INFO:ABC:t:3 eps:1.6024394835463
DEBUG:Distance:update distance weights = {'ss1': 1.8360864969496702, 'ss2': 0.
---163931503053006}
INFO:ABC:t:4 eps:1.03169141122295
DEBUG:Distance:update distance weights = {'ss1': 1.8861028922004666, 'ss2': 0.
---113897107799533}
INFO:ABC:t:5 eps:0.5907815126190052
DEBUG:Distance:update distance weights = {'ss1': 1.9359517130547306, 'ss2': 0.
---640482864526930}
INFO:ABC:t:6 eps:0.3291787896369823
DEBUG:Distance:update distance weights = {'ss1': 1.958912672910412, 'ss2': 0.
---41087327099587976}
INFO:ABC:t:7 eps:0.19112613860322253
DEBUG:Distance:update distance weights = {'ss1': 1.963773404833426, 'ss2': 0.
---36226595166573825}
INFO:History:Done <ABCSMC(id=4, start_time=2019-05-01 16:07:55.074538,
---end_time=2019-05-01 16:09:01.403688)>

In the debug output of abc.run above, it can be seen how the weights evolve over time. Note that we set the ac-
ceptor to pyabc.acceptor.accept_use_complete_history instead of the default pyabc.acceptor.
accept_use_current_time in order to get nested acceptance regions. This is optional here but may be benefi-
cial sometimes. Let us visualize the results for the adaptive distance:

[5]: # plotting

    fig, ax = pyplot.subplots()
    for t in range(history1.max_t + 1):
        df, w = history1.get_distribution(m=0, t=t)
        pyabc.visualization.plot_kde_1d(df, w, xmin=0, xmax=10,
            x='theta', ax=ax,
            label="PDF t={}".format(t))
        ax.axvline(theta_true, color='k', linestyle='dashed', label="True value")
        ax.legend()

[5]: <matplotlib.legend.Legend at 0x7f71e20a7080>

3.10. Adaptive Distances 55
We observe differences compared to the non-adaptive setting. In particular, the densities tend to be narrower around the true parameter $\theta = 3$. In addition, despite, the better convergence, the required number of samples in total is lower, as not so much time was wasted trying to match an uninformative summary statistic:

```python
[6]: pyabc.visualization.plot_sample_numbers([history0, history1], ["Fixed distance", "Adaptive distance"])

[6]: <matplotlib.axes._subplots.AxesSubplot at 0x7f71e42210f0>
```

In detail, the adaptive distance feature works as follows: In each iteration of the ABCSMC run, after having obtained the desired number of accepted particles (and once at the beginning using a sample from the prior), the method `DistanceFunction.update()` is called. It is given a set of summary statistics which can be used to e.g. compute weights for the distance measure in the next iteration. In order to avoid bias, via `DistanceFunction.configure_sampler()`, the distance function can tell the sampler to not only record accepted particles, but all that were generated during the sampling process. So, when you want to define your own adaptive distance function, you will typically only need to overwrite these two methods. For implementation details and an example of how this can look in practice, please inspect the code of `AdaptivePNormDistance`. 

In detail, the adaptive distance feature works as follows: In each iteration of the ABCSMC run, after having obtained the desired number of accepted particles (and once at the beginning using a sample from the prior), the method `DistanceFunction.update()` is called. It is given a set of summary statistics which can be used to e.g. compute weights for the distance measure in the next iteration. In order to avoid bias, via `DistanceFunction.configure_sampler()`, the distance function can tell the sampler to not only record accepted particles, but all that were generated during the sampling process. So, when you want to define your own adaptive distance function, you will typically only need to overwrite these two methods. For implementation details and an example of how this can look in practice, please inspect the code of `AdaptivePNormDistance`. 

```python
Total required samples
```

In detail, the adaptive distance feature works as follows: In each iteration of the ABCSMC run, after having obtained the desired number of accepted particles (and once at the beginning using a sample from the prior), the method `DistanceFunction.update()` is called. It is given a set of summary statistics which can be used to e.g. compute weights for the distance measure in the next iteration. In order to avoid bias, via `DistanceFunction.configure_sampler()`, the distance function can tell the sampler to not only record accepted particles, but all that were generated during the sampling process. So, when you want to define your own adaptive distance function, you will typically only need to overwrite these two methods. For implementation details and an example of how this can look in practice, please inspect the code of `AdaptivePNormDistance`. 

```python
Total required samples
```

In detail, the adaptive distance feature works as follows: In each iteration of the ABCSMC run, after having obtained the desired number of accepted particles (and once at the beginning using a sample from the prior), the method `DistanceFunction.update()` is called. It is given a set of summary statistics which can be used to e.g. compute weights for the distance measure in the next iteration. In order to avoid bias, via `DistanceFunction.configure_sampler()`, the distance function can tell the sampler to not only record accepted particles, but all that were generated during the sampling process. So, when you want to define your own adaptive distance function, you will typically only need to overwrite these two methods. For implementation details and an example of how this can look in practice, please inspect the code of `AdaptivePNormDistance`. 

```python
Total required samples
```
3.11 Download the examples as notebooks

- Quickstart
- Parameter inference
- Model selection (Quickstart)
- Early stopping of model simulations
- Resuming stored ABC runs
- Using R with pyABC
- Ordinary Differential Equations: Conversion Reaction
- Markov Jump Process: Reaction Network
- Multi-scale model: Tumor spheroid growth
- Stochastic Differential Equation: Ion channel noise in Hodgkin-Huxley neurons
- Adaptive Distances

**Warning:** Upgrade to the latest pyABC version before running the examples. If you installed pyABC some weeks (or days) ago, some new features might have been added in the meantime. Refer to the *Upgrading* section on how to upgrade pyABC.
4.1 Strategies

The pyABC package offers a variety of different parallel and distributed sampling strategies. Single-core, multi-core and distributed execution is supported in a couple different ways. The ParticleParallel samplers and the MappingSampler implement the “Static Scheduling (STAT)” strategy. The EvalParallel samplers, the DaskDistributedSampler and the ConcurrentFutureSampler implement the “DynamicScheduling (DYN)” strategy. The batchsize argument of the DaskDistributedSampler and the ConcurrentFutureSampler allow to interpolate between dynamic and static scheduling.

4.1.1 Single-core execution

For single-core execution, pyABC offers the pyabc.sampler.SingleCoreSampler. This one just generates sample by sample sequentially. This sampler is intended for debugging purposes as debugging parallel code can be hard sometimes.

4.1.2 Multi-core only samplers

For multi-core execution, pyABC implements two possible parallelization strategies.

- First, the pyabc.sampler.MulticoreParticleParallelSampler implements the STAT sampling strategy.
- Next, the pyabc.sampler.MulticoreEvalParallelSampler implements the DYN strategy. This is currently the default sampler.

Both samplers are highly specialized to the multi-core setting and have very little communication overhead. Even for very small model evaluation times these samplers are about as fast as the single core sampler. This is achieved circumventing object serialization by forking. As Microsoft Windows does not support forking, these samplers might not work as expected on Windows.

4.1.3 Distributed samplers

The distributed samplers can be used in a distributed setting, and of course also locally by setting up a local cluster. However, for local execution, the multi-core samplers are recommended as they are easier to set up.

The pyabc.sampler.RedisEvalParallelSampler has very low communication overhead, and when running workers and redis-server locally is actually competitive with the multi-core only samplers. The pyabc.sampler.RedisEvalParallelSampler performs parameter sampling on a per worker basis, and can handle fast function evaluations efficiently.
The `pyabc.sampler.DaskDistributedSampler` has slightly higher communication overhead, however this can be compensated with the batch submission mode. As the `pyabc.sampler.DaskDistributedSampler` performs parameter sampling locally on the master, it is unsuitable for simulation functions with a runtime below 100ms, as network communication becomes prohibitive at this point.

The Redis based sampler can require slightly more effort in setting up than the Dask based sampler, but has fewer constraints regarding simulation function runtime. The Dask sampler is in turn better suited to handle worker failures and unexpected execution host terminations.

### 4.1.4 General extensible samplers

Moreover, there are two more generic samplers which can be used in a multicore and distributed fashion. These samplers facilitate adaptation of pyABC to new parallel environments.

The `pyabc.sampler.MappingSampler` can be used in a multi-core context if the provided map implementation is a multi-core one, such as, e.g. `multiprocessing.Pool.map`, or distributed if the map is a distributed one, such as `pyabc.sge.SGE.map`.

Similarly, the `pyabc.sampler.ConcurrentFutureSampler` can use any implementation of the python `concurrent.futures.Executor` interface. Again, implementations are available for both multi-core (e.g. `concurrent.futures.ProcessPoolExecutor`) and distributed (e.g. Dask) environments.

Check the API documentation for more details.

### 4.2 How to setup a Redis based distributed cluster

#### 4.2.1 Step 1: Start a Redis server without password authentication

Start one some machine, which is reachable by the machine running the pyABC main application and by the workers, a Redis server, disabling password authentication:

```
redis-server --protected-mode no
```

You should get an output looking similar to the one below:

```
30656:M 23 May 13:19:20.718 # You requested maxclients of 10000 requiring at least
→10032 max file descriptors.
30656:M 23 May 13:19:20.718 # Server can't set maximum open files to 10032 because of
→OS error: Operation not permitted.
30656:M 23 May 13:19:20.718 # Current maximum open files is 4096. maxclients has been
→reduced to 4064 to compensate for low ulimit. If you need higher maxclients,
→increase 'ulimit -n'.
```

Redis 3.2.9 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 30656
http://redis.io

(continues on next page)
If you’re on Linux, you can install redis either via your package manager or if you’re using anaconda via

```bash
conda install redis
```

At this point, Windows is not officially supported by the Redis developers. We assume for now, that the IP address of the machine running the Redis server is 111.111.111.111.

### 4.2.2 Step 2 or 3: Start pyABC

It does not matter what you do first: starting pyABC or starting the workers. Assuming the models, priors and the distance function are defined, configure pyABC to use the Redis sampler

```python
from pyabc.sampler import RedisEvalParallelSampler
redis_sampler = RedisEvalParallelSampler(host="111.111.111.111")
abc = pyabc.ABCSMC(models, priors, distance, sampler=redis_sampler)
```

Note that 111.111.111.111 is the IP address of the machine running the Redis server. Then start the ABC-SMC run as usual with

```bash
abc.run(...)```

passing the stopping conditions.

### 4.2.3 Step 2 or 3: Start the workers

It does not matter what you do first: starting pyABC or starting the workers. You can even dynamically add workers after the sampling has started. Start as many workers as you wish on the machines you wish. Up to 10,000 workers should not pose any problem if the model evaluation times are on the second scale or longer.

```bash
abc-redis-worker --host=111.111.111.111```
Again, 111.111.111.111 is the IP address of the machine running the Redis server. You should get an output similar to

```
INFO:REDIS-WORKER:Start redis worker. Max runtime 7200.0s, PID=2731
```

Note that the `abc-redis-worker` command also has options to set the maximal runtime of a worker, e.g. `--runtime=2h`, `--runtime=3600s`, `--runtime=2d`, to start a worker running for 2 hours, 3600 seconds or 2 days. The default is 2 hours. It is OK if a worker stops during the sampling of a generation. You can add new workers during the sampling process. The `abc-redis-worker` command also has an option `--processes` which allows you to start several worker processes in parallel. This might be handy in situations where you have to use a whole cluster node with several cores.

### 4.2.4 Optional: Monitoring

pyABC ships with a small utility to manage the Redis based sampler setup. To monitor the ongoing sampling, execute

```
abc-redis-manager info --host=111.111.111.111
```

again, assuming 111.111.111.111 is the IP of the Redis server. If no sampling has happened yet, the output should look like

```
Workers=None Evaluations=None Particles=None
```

The keys are to be read as follows:

- **Workers**: Currently sampling workers. This will show None or zero, even if workers are connected but they are not running. This number drops to zero at the end of a generation.

- **Evaluations**: Number of accumulated model evaluations for the current generations. This is a sum across all workers.

- **Particles**: Number of particles which remain to be accepted. This number decreases over the course of a population and reaches 0 (or a negative number due to excess sampling) at the end of a population. At the very start, this is just the population size.

### 4.2.5 Optional: Stopping workers

Use `abc-redis-manager stop` to send a signal to the workers that they should shutdown at the end of the current generation.

You can also stop workers with `Ctrl-C`, or even send a kill signal when pyABC has finished.

### 4.2.6 Optional: Something with the workers went wrong in the middle of a run

It can happen, that workers get unexpectedly killed. If they are not able to communicate to the redis-server that they’ve finished working on the current population before they’re killed, the pyABC master process will wait forever. In such cases, the following can be done

1. Terminate all running workers (but not the pyABC master process and also not the redis-server)

2. Execute `abc-redis-manager reset-workers` to manually reset the number of registered workers to zero.

3. Start worker processes again.
PARALLEL JOB EXECUTION ON AN SGE CLUSTER ENVIRONMENT

5.1 Quick start

The pyabc.sge package provides as most important class the SGE. Its map method automatically parallelizes across an SGE/UGE cluster. The SGE class can be used in standalone mode or in combination with the ABCSMC class (see below Usage notes).

Usage of the parallel package is fairly easy. For example:

```python
from pyabc.sge import SGE
sge = SGE(priority=-200, memory="3G")

def f(x):
    return x * 2

tasks = [1, 2, 3, 4]
result = sge.map(f, tasks)
print(result)

[2, 4, 6, 8]
```

The job scheduling is either done via an SQLite database or a REDIS instance. REDIS is recommended as it works more robustly, in particular in cases where distributed file systems are rather slow.

**Note:** A configuration file in `~/.parallel` is required. See SGE.

The `pyabc.sge.sge_available` can be used to check if an SGE cluster can be used on the machine.

Check the API documentation for more details.

5.2 Information about running jobs

Use the `python -m pyabc.sge.job_info_redis` to get a nicely formatted output of the current execution state, in case the REDIS mode is used. Check `python -m pyabc.sge.job_info_redis --help` for more details.
5.3 Usage notes

The `SGE` class can be used in standalone mode for convenient parallelization of jobs across a cluster, completely independent of the rest of the pyABC package. The `SGE` class can also be combined, for instance, with the `pyabc.sampler.MappingSampler` class for simple parallelization of ABC-SCM runs across an SGE cluster.
pyABC ships with a small convenience utility, which allows to export pyABC’s SQL database to other formats. Supported formats are:

- CSV
- feather
- json
- msgpack
- html
- hdf
- stata

An example call might look as follows:

```
abc-export --db results.db --out exported.feather --format feather
```

This exports the database “results.db” generated by pyABC to “exported.feather” which is written in in the feather format as indicated by the option `--format feather` (the file extension of the exported file is not parsed).

Check:

```
abc-export --help
```

for further options to customize the export.
WEB BASED VISUALIZATIONS

The pyABC package comes with a web server, which displays lots of useful information on the currently running and already completed ABC tasks. You can launch it from the command line with

```
abc-server <databasename>
```

It opens per default a web server on port 5000.

You should see something similar to the following:

```
pyABC
```

Via “Go to ABC Run List”, you can see all running and finished ABC runs, which you can then inspect in more detail.

You can get overviews over the models:
Information about individual model parameters for each model and time point is also displayed:
Type in the command line

```
abc-server --help
```

To get more information on available options, such as selecting another port:

```
abc-server --port=8888 <databasename>
```
8.1 0.9 series

8.1.1 0.9.12 (2019-05-02)

- Reorganize distance module (minor API change: distance_functions -> distance, and some classes shortened accordingly)
- Allow to pass parameters to Acceptor and Distance.
- Make time and parameter arguments to distance functions optional.
- Rewrite lazy evaluation for calibration sample in ABCSMC class.
- Give default values for ABCSMC.run arguments, which set no stopping criterion.
- Add function and plot for effective sample size.

8.1.2 0.9.11 (2019-04-01)

- Run some notebooks as part of the tests.
- Automatize pypi upload via travis.

8.1.3 0.9.10 (2019-03-27)

- Save number of samples taken in calibration step in database.
- Fix error with reported number of simulations in EpsMixin based samplers.
- Fix several warnings.

8.1.4 0.9.9 (2019-03-25)

- Monitor code quality using codacy and codecov.
- Extend visualization routines: Add histogram, sample number, epsilon trajectory, model probability, and credible interval plots.
- Test visualization routines on travis.
- Fix problem with the History.get_weighted_distances function after update to sqlalchemy>=1.3.0.
- Add random walk based transition for discrete parameters.
8.1.5 0.9.8 (2019-02-21)

- Tidy up returning of rejected samples in Sample (not only summary statistics).
- Recreate a population from file in History.get_population().
- Speed up loading from database by eager loading.
- Document the change of the contribution scheme to master+develop.

8.1.6 0.9.7 (2019-02-20)

- Allow for the database to save no summary statistics for testing purposes.
- Tidy up some pyabc.History methods.
- pyabc.History.id set by default to the largest index (previously 0), corresponding to the latest inserted analysis.

8.1.7 0.9.6 (2019-02-01)

- Fix several errors with the readthedocs (rtd) documentation.
- Speed-up rtd build by removing unnecessary conda and pip requirements.
- Clean-up requirements for travis and rtd.
- Change rtd design from alabaster to sphinx_rtd_theme since it implements better navigation.

8.1.8 0.9.5 (2019-01-17)

- ABCSMC can pass observed summary statistics to distance functions (required for some scale functions, and to make the methods robust to volatile summary statistics).
- Implementation of more scale functions (distance_functions.scales), in particular some taking into account the bias to the observed data.
- AdaptivePNormDistance accepts a Callable as scaling scheme, allowing for more flexibility.

8.1.9 0.9.4 (2018-12-18)

- Can specify kde and number of bins for all visualization routines.
- Can re-submit observed sum stats to ABCSMC.load() function in case it cannot be read correctly from the db.

8.1.10 0.9.3 (2018-12-01)

- Fix serious memory problem resulting from pickling more than necessary for parallel sampling.
- Update logo, readme.
- Make tidying optional in abc-export (default behavior not changed).
8.1.11 0.9.2 (2018-09-10)

- Minor error and warning fixes due to API changes in pandas, seaborn (not used any more), and change of the R installation on travis.

8.1.12 0.9.1 (2018-06-05)

- Default visualizations like plot_kde_matrix() can plot reference values, useful for testing purposes.

8.1.13 0.9.0

- Acceptance transferred to an Acceptor object to allow for more flexibility (i.e. not only on a single comparison as per default).
- This acceptor is passed to the ABCSMC object.
- Update of distance and epsilon synchronized after each iteration and moved to update() methods.
- initialize() for DistanceFunction and Epsilon also called in load() method, given a time point to initialize for, and made optional via a require_initialize flag. This makes sure these objects are always correctly initialized.
- PNormDistance and AdaptivePNormDistance (prev. WeightedPNormDistance) improved to allow for more customization.
- ABCSMC.set_data() method removed.
- API breaks for DistanceFunction, Epsilon, Model.

8.2 0.8 series

8.2.1 0.8.21

- Implementation of adaptive distances feature. Distance functions can adapt via an update() method.
- In particular add WeightedPNormDistance (special case: WeightedEuclideanDistance). Also add non-weighted versions.
- Simplify Sampler.sample_until_n_accepted interface.
- Extend Sampler class to allow for customization, e.g. by the distance functions.
- Generalize MedianEpsilon to QuantileEpsilon.
- Make Viserver work with latest bokeh version.

8.2.2 0.8.20

- Add batch sampling now also to the REDIS evaluation parallel sampler (dynamic scheduling)

8.2.3 0.8.19

- Bug fix. Fix a race condition in the redis evaluation parallel sampler (dynamic scheduling). An error occured if a worker tried to start to work on a population after the other workers had already terminated the population.
8.2.4 0.8.18

- Minor bug fix. Ensure that the multicore samplers raise an Exception if an Exception occurs in the worker processes.
- Clarify that weighted distances are not normalized in case of having more than a single simulation per proposed parameter. Also add corresponding tests.
- Add n_worker method to the RedisEvalParallelSampler to enable querying of the number of connected workers.
- Add in-memory database support. Useful, e.g., for benchmarking on slow filesystems or with rather slow network connections.

8.2.5 0.8.17

Make git and gitpython an optional dependency.

8.2.6 0.8.16

- Add “abc-redis-manager reset-workers” command in case workers were unexpectedly killed.
- Adapt web server to changed bkcharts API.

8.2.7 0.8.15

- Bug fix. Rand seed initialization in case of starting multiple workers with --processes in redis server was not correct.

8.2.8 0.8.14

- Bug fix in MulticoreEvalParallelSampler. The multiprocessing.Queue could fill up and cause a deadlock on joining the workers. This is now fixed.
- Rename population_specification to population_size.
- Improve plot_kde_matrix plot ranger are now handled in a less confusing way

8.2.9 0.8.13

- Minor doc fixes
- Python 3.5 support dropped. It might still work for a while with Python 3.5 but this is not guaranteed anymore.
- Add kde matrix visualization function
- Add 2d tumor growth example
- Add Gillespie example
- Change license
8.2.10 0.8.12

- Minor bug fix. Visualization server produced error when JSON information was empty.
- Adapt to new bkcharts package.

8.2.11 0.8.11

Ensure R source file is reloaded when unpickling R objects.

8.2.12 0.8.10

Add \texttt{--id} option to abc-export to handle databases with multiple ABC runs.

8.2.13 0.8.9

Ensure that summary statistics have names. Also add kwargs to \texttt{plot_kde_2d} which are passed to \texttt{pcolormesh}.

8.2.14 0.8.8

Add \texttt{--processes} option to abc-redis-worker to start a number of workers in parallel.

8.2.15 0.8.7

Make rpy2 an optional dependency. If rpy2 is installed, then R can be used if not, the rest will still work.

8.2.16 0.8.6

 minor bug fixes

8.2.17 0.8.5

- minor bug fix in \texttt{plot_kde_2d} if the axis is provided

8.2.18 0.8.5

- minor bug fix. The external.R interface did not display the source code correctly.
- minor doc updates

8.2.19 0.8.4

- support serialization of DataFrames used as summary statistics for storage in the database. This feature is still considered experimental.
- Add command line utility to export pyABC’s database to different file formats such as csv, feather, html, json and more.
8.2.20 0.8.3

- Add (experimental) support for models defined in R.
- Add some visualization functions for convenience.

8.2.21 0.8.2

Bug fixes for web server.

8.2.22 0.8.1

Minor internal refactorings and minor documentation updates. Nothing a user should notice.

8.2.23 0.8.0

- Deprecate the “set_data” method of the ABCSMC class. Use the “new” method instead.
- Add a “load” method to the ABCSMC class for easier resuming stored ABCSMC runs.
- Add an example to the documentation how to resume stored ABC-SMC runs.
- Rename the acceptance_rate parameter form ABCSMC.run to min_acceptance_rate for clarity. Usage of acceptance_rate is deprecated.
- Various documentation improvements, correcting typos, clarifications, etc.

8.3 0.7 series

8.3.1 0.7.2

Easier early stopping models via the IntegratedModel class. Also has now examples.

8.3.2 0.7.1

- Minor refactoring for better Windows compatibility. But runs in serial on Windows

8.3.3 0.7.0

- ABCSMC.run gets a new parameter “acceptance_rate” to stop sampling if the acceptance rate drops too low.
- History.get_all_populations returns a DataFrame with columns “t”, “population_end_time”, “samples”, “epsilon”, “particles”. That is “nr_samples” got renamed to “samples” and “particles” is new.
## 8.4 0.6 series

### 8.4.1 0.6.4

Performance improvement. Use MulticoreEvalParallelSampler as default. This should bring better performance for machines with many cores and comparatively small population sizes.

### 8.4.2 0.6.3

Bug fix. Ensure numpy.int64 can also be passed to History methods were an integer argument is expected.

### 8.4.3 0.6.2

Bug fix. Forgot to add the new Multicore base class.

### 8.4.4 0.6.1

MulticoreEvalParallelSampler gets an n_procs parameter.

### 8.4.5 0.6.0

**History API**

Change the signature from `History.get_distribution(t, m)` to `History.get_distribution(m, t)` and make the time argument optional defaulting to the last time point

## 8.5 0.5 series

### 8.5.1 0.5.2

- **Minor History API changes**
  - Remove `History.get_results_distribution`
  - rename `History.get_weighted_particles_dataframe` to `History.get_distribution`

### 8.5.2 0.5.1

- **Minor ABCSMC API changes**
  - Mark the de facto private methods as private by prepending an underscore. This should not cause trouble as usually noone would ever use these methods.
8.5.3 0.5.0

- **Usability improvements and minor API changes**
  - ABCSMC accepts now an integer to be passed for constant population size
  - The maximum number populations specification has moved from the PopulationStrategy classes to the ABCSMC.run method. The ABCSMC.run method will be where it is defined when to stop.

8.6 0.4 series

8.6.1 0.4.4

- **Improvements to adaptive population size strategy**
  - Use same CV estimation algorithm for Transition and PopulationStrategy
  - Bootstrapping on full joint space for model selection

8.6.2 0.4.3

- Fix edge case of models without parameters for population size adaptation

8.6.3 0.4.2

- **Changes to the experimental adaptive population strategy.**
  - Smarter update for model selection
  - Better CV estimation

8.6.4 0.4.1

- Fix minor bug in RVs wrapper. args and keyword args were not passed to the wrapper random variable.

8.6.5 0.4.0

- Add local transition class which makes a local KDE fit.
- Fix corner cases of adaptive population size strategy
- Change the default: Do not stop if only a single model is alive.
- Also include population 0, i.e. a sample from the prior, in the webserver visualization
- **Minor bug fixes**
  - Fix inconsistency in ABC options if db_path given as sole string argument
- **Add four evaluation parallel samplers**
  - Dask based implementation
    - More communication overhead
  - Future executor evaluation parallel sampler
* Very similar to the Dask implementation

  – **Redis based implementation**
    * Less communication overhead
    * Performs also well for short running simulations

  – **Multicore evaluation parallel sampler**
    * In most common cases, where the population size is much bigger than the number of cores, this sampler is not going to be faster than the multicore particle parallel sampler.
    * However, on machines with lots of cores and moderate sized populations this sampler might be faster

### 8.7 0.3 series

#### 8.7.1 0.3.3

  * Fix SGE regression. Forgot to update a module path on refactoring.

#### 8.7.2 0.3.2

**PEP8**

Comply with PEP8 with a few exceptions where it does not make sense. Flake8 runs now with the test. The tests do not pass if flake8 complains.

**Legacy code cleanup**

Remove legacy classes such as the MultivariateMultiTypeNormalDistributions and the legacy covariance calculation. Also remove devideas folder.

#### 8.7.3 0.3.1

**Easier usage**

Refactor the ABCSMC.set_data and provide defaults.

#### 8.7.4 0.3.0

**Easier usage**

Provide more default values for ABCSMC. This improves usability.
8.8 0.2 series

8.8.1 0.2.0

Add an efficient multicore sampler

The new sampler relies on forking instead of pickling for the `sample_one`, `simulate_one` and `accept_one` functions. This brings a huge performance improvement for single machine multicore settings compared to `multiprocessing.Pool.map` like execution which repeatedly pickles.

8.9 0.1 series

8.9.1 0.1.3

Initial release to the public.
CHAPTER NINE

ABOUT

pyABC version: 0.9.12
Source code: https://github.com/icb-dcm/pyabc

9.1 Authors

The package was mainly developed by Emmanuel Klinger. Dennis Rickert contributed through discussions and code. Lukas Sandmeir and Elba Raimundez contributed to the examples. Yannik Schälte contributed several new features.

9.2 Contact

Discovered an error? Need help? Not sure if something works as intended? Please contact us!
- Yannik Schälte: yannik.schaelte@helmholtz-muenchen.de
- Elba Raimundez: elba.raimundez@helmholtz-muenchen.de

If you think that your issue could be of general interest, please consider creating an issue on github, which will then also be helpful for other users: https://github.com/icb-dcm/pyabc/issues

9.3 License

This package is licensed under a permissive 3-clause BSD license.

Copyright 2017 the pyABC developers

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from (continues on next page)
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
CONTRIBUTE DOCUMENTATION

Documentation is an essential part of the software development process. We want to provide a useful piece of software. It is therefore necessary to have a good documentation, such that the user knows how to use our package. Contributions to the documentation are as welcome as contributions to the code.

10.1 Docstrings

We follow the numpy docstring standard. Check here for a detailed explanation.
We’re committed to testing our code. Tests that are required to pass are located in the `test` folder. All files starting with `test_` contain tests and are automatically run on Travis CI. To run them manually, type:

```
python3 -m pytest test
```

You can also run specific tests only.

We encourage to test whatever possible. However, it might not always be easy to test code which is based on random sampling. We still encourage to provide general sanity and integration tests. We highly encourage a test-driven development (TDD) style.

### 11.1 Writing tests

Tests can be written with `pytest` or the `unittest` module.

### 11.2 PEP8 Style Guide

We try to respect the PEP8 standard. We run `flake8` as part of the test suite. The tests won’t pass if `flake8` complains.
CONTRIBUTE CODE

If you start working on a new feature or a fix, if not already done, please create an issue on github, shortly describing your plans, and assign it to yourself. Your starting point should not be the master branch, but the develop branch, which contains the latest updates.

Create an own branch or fork, on which you can implement your changes. To get your work merged, please:

1. create a pull request to the develop branch,
2. check that all tests on travis pass,
3. check that the documentation is up-to-date,
4. request a code review from the main developers.

Document all your changes in the pull request, and make sure to appropriately resolve issues, and delete stale branches after a successful merge.
New features and bug fixes are continually added to the develop branch. On every merge to master, the version number in `pyabc/version.py` should be incremented as described below.

### 13.1 Versioning scheme

For version numbers, we use $A.B.C$, where

- $C$ is increased for bug fixes,
- $B$ is increased for new features and minor API breaking changes,
- $A$ is increased for major API breaking changes.

Thus, we roughly follow the versioning scheme suggested by the Python packaging guide.

### 13.2 Create a new release

After new commits have been added via pull requests to the develop branch, changes can be merged to master and a new version of pyABC can be released. Every merge to master should coincide with an incremented version number and a git tag on the respective merge commit.

#### 13.2.1 Merge into master

1. create a pull request from develop to master,
2. check that all tests on travis pass,
3. check that the documentation is up-to-date,
4. adapt the version number in `pyabc/version.py` (see above),
5. update the release notes in `doc/releasenotes.rst`,
6. request a code review,
7. merge into the origin master branch.

To be able to actually perform the merge, sufficient rights may be required. Also, at least one review is required.
13.2.2 Create a release on github

After merging into master, create a new release on GitHub. This can be done either directly on the project GitHub website, or via the CLI as described in Git Basics - Tagging. In the release form,

- specify a tag with the new version as specified in \texttt{pyabc/version.py},
- include the latest additions to \texttt{doc/releasenotes.rst} in the release description.

13.3 Upload to the Python package index PyPI

The upload to pypi has been automatized via travis since pyabc version 0.9.11, and is triggered whenever a new release tag is created on the github master branch.

To manually upload a new version to pypi, proceed as follows:

First, a so called “wheel” is created via:

```bash
python setup.py bdist_wheel
```

A wheel is essentially a zip archive which contains the source code and the binaries (if any).

This archive is uploaded using twine:

```bash
twine upload dist/pyabc-x.y.z-py3-non-any.wheel
```

replacing \texttt{x.y.z} by the appropriate version number.

For a more in-depth discussion see also the section on distributing packages of the Python packaging guide.
14.1 Parallel Approximate Bayesian Computation - Sequential Monte Carlo

The ABCSMC class is the most central class of the pyABC package. Most of the other classes serve to configure it. (I.e. the other classes implement a Strategy pattern.)


Bases: object

Approximate Bayesian Computation - Sequential Monte Carlo (ABCSMC).

This is an implementation of an ABCSMC algorithm similar to\(^1\).

Parameters

- **models** (list of models, single model, list of functions, or single function) –
  - If models is a function, then the function should have a single parameter, which is of dictionary type, and should return a single dictionary, which contains the simulated data.
  - If models is a list of functions, then the first point applies to each function.
  - Models can also be a list of Model instances or a single Model instance.

This model’s output is passed to the summary statistics calculation. Per default, the model is assumed to already return the calculated summary statistics. Accordingly, the default summary_statistics function is just the identity. Note that the sampling and evaluation of particles happens in the model’s methods, so overriding these offers a great deal of flexibility, in particular the freedom to use or ignore the distance_function, summary_statistics, and eps parameters here.

- **parameter_priors** (List[Distribution]) – A list of prior distributions for the models’ parameters. Each list entry is the prior distribution for the corresponding model.

• **distance_function** *(Distance, optional)* – Measures the distance of the tentatively sampled particle to the measured data.

• **population_size** *(int, PopulationStrategy, optional)* – Specify the size of the population. If `population_specification` is an int, then the size is constant. Adaptive population sizes are also possible by passing a `pyabc.populationstrategy.PopulationStrategy` object. The default is 100 particles per population.

• **summary_statistics** *(Callable[[model_output], dict])* – A function which takes the raw model output as returned by any of the models and calculates the corresponding summary statistics. Note that the default value is just the identity function. I.e. the model is assumed to already calculate the summary statistics. However, in a model selection setting it can make sense to have the model produce some kind or raw output and then take the same summary statistics function for all the models.

• **model_prior** *(RV, optional)* – A random variable giving the prior weights of the model classes. The default is a uniform prior over the model classes, `RV("randint", 0, len(models))`.

• **model_perturbation_kernel** *(ModelPerturbationKernel)* – Kernel which governs with which probability to switch from one model to another model for a given sample while generating proposals for the subsequent population from the current population.

• **transitions** *(List[Transition], Transition, optional)* – A list of `pyabc.transition.Transition` objects or a single `pyabc.transition.Transition` in case of a single model. Defaults to multivariate normal transitions for every model.

• **eps** *(Epsilon, optional)* – Accepts any `pyabc.epsilon.Epsilon` subclass. The default is the `pyabc.epsilon.MedianEpsilon` which adapts automatically. The object passed here determines how the acceptance threshold scheduling is performed.

• **sampler** *(Sampler, optional)* – In some cases, a mapper implementation will require initialization to run properly, e.g. database connection, grid setup, etc.. The sampler is an object that encapsulates this information. The default sampler `pyabc.sampler.MulticoreEvalParallelSampler` will parallelize across the cores of a single machine only.

• **acceptor** *(Acceptor, optional)* – Takes a distance function, summary statistics and an epsilon threshold to decide about acceptance of a particle. Argument accepts any subclass of `pyabc.acceptor.Acceptor` or a function convertible to an acceptor.

**stop_if_only_single_model_alive**

Defaults to False. Set this to true if you want to stop ABCSMC automatically as soon as only a single model has survived.

**Type** `bool`

```
```

Initialize self. See help(type(self)) for accurate signature.
**load** *(db: str, abc_id: int = 1, observed_sum_stat: dict = None) → int*

Load an ABC-SMC run for continuation.

**Parameters**

- **db** *(str)* – A SQLAlchemy database identifier pointing to the database from which to continue a run.
- **abc_id** *(int, optional)* – The id of the ABC-SMC run in the database which is to be continued. The default is 1. If more than one ABC-SMC run is stored, use the abc_id parameter to indicate which one to continue.
- **observed_sum_stat** *(dict, optional)* – The observed summary statistics. This field should be used only if the summary statistics cannot be reproduced exactly from the database (in particular when they are no numpy or pandas objects, e.g. when they were generated in R). If None, then the summary statistics are read from the history.

**new** *(db: str, observed_sum_stat: dict = None, *, gt_model: int = None, gt_par: dict = None, meta_info=None) → int*

Make a new ABCSMC run.

**Parameters**

- **db** *(str)* – Has to be a valid SQLAlchemy database identifier. This indicates the database to be used (and created if necessary and possible) for the ABC-SMC run.

  To use an in-memory database pass “sqlite://”. Note that in-memory databases are only available on the master mode. If workers are started on different nodes they won’t be able to access the database. This should not be a problem in most scenarios. The in-memory option is mainly useful for benchmarking (and maybe) for testing.

- **observed_sum_stat** *(dict, optional)* – This is the really important parameter here. It is of the form `{‘statistic_1’: val_1, ‘statistic_2’: val_2, ... }`. The dictionary provided here represents the measured data. Particle during ABCSMC sampling are compared against the summary statistics provided here.

  The summary statistics’ values can be integers, floats, strings and everything which is a numpy array or can be converted to one (e.g. lists). In addition, pandas.DataFrames can also be used as summary statistics. **Note that storage of pandas DataFrames in pyABC’s database is still considered experimental.**

  This parameter is optional, as the distance function might implement comparison to the observed data on its own. Not giving this parameter is equivalent to passing an empty dictionary `{}`.

- **gt_model** *(int, optional)* – This is only meta data stored to the database, but not actually used for the ABCSMC algorithm. If you want to predict your ABCSMC procedure against synthetic samples, you can use this parameter to indicate the ground truth model number. This helps with further analysis. If you use actually measured data (and don’t know the ground truth) you don’t have to set this.

- **gt_par** *(dict, optional)* – Similar to ground_truth_model, this is only for recording purposes in the database, but not used in the ABCSMC algorithm. This stores the parameters of the ground truth model if it was synthetically obtained. Don’t give this parameter if you don’t know the ground truth.

- **meta_info** *(dict, optional)* – Can contain an arbitrary number of keys, only for recording purposes. Store arbitrary meta information in this dictionary. Can be used for really anything. This dictionary is stored in the database.
Returns `run_id` – The history.id, which is the id under which the generated ABCSMC run entry in the database can be identified.

Return type `int`

```python
def run(minimum_epsilon: float = 0.0, max_nr_populations: int = inf, **kwargs) -> pyabc.storage.history.History
```

Run the ABCSMC model selection until either of the stopping criteria is met.

Parameters

- `minimum_epsilon` (`float`, optional (default = 0.0)) – Stop if epsilon is smaller than minimum epsilon specified here.
- `max_nr_populations` (`int`, optional (default = np.inf)) – The maximum number of populations. Stop if this number is reached.
- `min_acceptance_rate` (`float`, optional (default = 0.0)) – Minimal allowed acceptance rate. Sampling stops if a population has a lower rate.

Population after population is sampled and particles which are close enough to the observed data are accepted and added to the next population. If an adaptive Epsilon is specified (this is the default), then the acceptance threshold decreases from population to population automatically in a data dependent way.

Sampling of further populations is stopped, when either of the three stopping criteria is met:

- the maximum number of populations `max_nr_populations` is reached,
- the acceptance threshold for the last sampled population was smaller than `minimum.epsilon`,
- or the acceptance rate dropped below `acceptance_rate`.

The value of `minimum.epsilon` determines the quality of the ABCSMC approximation. The smaller the better. But sampling time also increases with decreasing `minimum.epsilon`.

This method can be called repeatedly to sample further populations after sampling was stopped once.

### 14.2 Distance functions

Distance functions measure closeness of observed and sampled data. This module implements various commonly used distance functions for ABC, featuring a few advanced concepts.

For custom distance functions, either pass a plain function to ABCSMC or subclass the `pyabc.Distance` class.

```python
class pyabc.distance.Distance
    Bases: abc.ABC

    Abstract base class for distance objects. Any object that computes the similarity between observed and simulated data should inherit from this class.

    __call__(x: dict, x_0: dict, t: int = None, par: dict = None) -> float
    Evaluate at time point t the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

    Abstract method. This method has to be overwritten by all concrete implementations.

    Parameters

    - `x` (`dict`) – Summary statistics of the data simulated for the tentatively sampled parameter.
    - `x_0` (`dict`) – Summary statistics of the observed data.
```

Chapter 14. API reference
• \(t\) (int) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.

• \(\text{par}\) (dict) – The parameters used to create the summary statistics \(x\). These can be required by some distance functions. Usually, the distance will not depend on the parameters.

Returns distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

Return type float

__init__()
Default constructor.

configure_sampler (sampler: pyabc.sampler.base.Sampler)
This is called by the ABCSMC class and gives the distance the opportunity to configure the sampler. For example, the distance might request the sampler to also return rejected particles in order to adapt the distance to the statistics of the sample. The method is called by the ABCSMC framework before the first used of the distance (at the beginning of ABCSMC.run()), after initialize().

The default is to do nothing.

Parameters
sampler (Sampler) – The sampler used in ABCSMC.

get_config () \rightarrow \text{dict}
Return configuration of the distance.

Returns config – Dictionary describing the distance.

Return type dict

initialize (t: int, get_sum_stats: Callable[[()], List[dict]], x_0: dict = None)
This method is called by the ABCSMC framework before the first use of the distance (at the beginning of ABCSMC.run()), and can be used to calibrate it to the statistics of the samples.

The default is to do nothing.

Parameters
• \(t\) (int) – Time point for which to initialize the distance.

• get_sum_stats (Callable[[[], List[dict]]) – Returns on command the initial summary statistics.

• x_0 (dict, optional) – The observed summary statistics.

to_json () \rightarrow \text{str}
Return JSON encoded configuration of the distance.

Returns json_str – JSON encoded string describing the distance. The default implementation is to try to convert the dictionary returned by get_config.

Return type str:

update (t: int, sum_stats: List[dict]) \rightarrow \text{bool}
Update the distance for the upcoming generation \(t\). The default is to do nothing.

Parameters
• \(t\) (int) – Time point for which to update the distance.

• sum_stats (List[dict]) – List of all summary statistics from the finished generation that should be used to update the distance.
Returns is_updated – Whether the distance has changed compared to beforehand. Depending on the result, the population needs to be updated in ABCSMC before preparing the next generation. Defaults to False.

Return type bool

class pyabc.distance.NoDistance
Bases: pyabc.distance.base.Distance

Implements a kind of null object as distance function. This can be used as a dummy distance function if e.g. integrated modeling is used.

Note: This distance function cannot be evaluated, so currently it is in particular not possible to use an epsilon threshold which requires initialization, because during initialization the distance function is invoked directly and not via the acceptor as usual. Conceptually, this would be possible and can be implemented on request.

__call__(x: dict, x_0: dict, t: int = None, par: dict = None) → float
Evaluate at time point t the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

Parameters

• x (dict) – Summary statistics of the data simulated for the tentatively sampled parameter.
• x_0 (dict) – Summary statistics of the observed data.
• t (int) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.
• par (dict) – The parameters used to create the summary statistics x. These can be required by some distance functions. Usually, the distance will not depend on the parameters.

Returns distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

Return type float

__init__()
Default constructor.

class pyabc.distance.IdentityFakeDistance
Bases: pyabc.distance.base.Distance

A fake distance function, which just passes the summary statistics on. This class assumes that the model already returns the distance. This can be useful in cases where simulating can be stopped early, when during the simulation some condition is reached which makes it impossible to accept the particle.

__call__(x: dict, x_0: dict, t: int = None, par: dict = None) → float
Evaluate at time point t the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

Parameters

• x (dict) – Summary statistics of the data simulated for the tentatively sampled parameter.
• x_0 (dict) – Summary statistics of the observed data.
• **t** (*int*) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.

• **par** (*dict*) – The parameters used to create the summary statistics x. These can be required by some distance functions. Usually, the distance will not depend on the parameters.

**Returns** distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

**Return type** float

class pyabc.distance.AcceptAllDistance  
Bases: pyabc.distance.base.Distance

Just a mock distance function which always returns -1. So any sample should be accepted for any sane epsilon object.

Can be used for testing.

```python
__call__(x: dict, x_0: dict, t: int = None, par: dict = None) → float
```

Evaluate at time point t the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

**Parameters**

• **x** (*dict*) – Summary statistics of the data simulated for the tentatively sampled parameter.

• **x_0** (*dict*) – Summary statistics of the observed data.

• **t** (*int*) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.

• **par** (*dict*) – The parameters used to create the summary statistics x. These can be required by some distance functions. Usually, the distance will not depend on the parameters.

**Returns** distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

**Return type** float

class pyabc.distance.SimpleFunctionDistance(*fun*)  
Bases: pyabc.distance.base.Distance

This is a wrapper around a simple function which calculates the distance. If a function/callable is passed to the ABCSMC class, which is not subclassed from pyabc.Distance, then it is converted to an instance of the SimpleFunctionDistance class.

```python
Parameters function (Callable[[dict, dict], float]) – A Callable accepting as parameters (a subset of) the arguments of the pyabc.Distance.__call__ function. Usually at least the summary statistics x and x_0. Returns the distance between both.
```

```python
__call__(x: dict, x_0: dict, t: int = None, par: dict = None) → float
```

Evaluate at time point t the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

**Parameters**

• **x** (*dict*) – Summary statistics of the data simulated for the tentatively sampled parameter.
• \texttt{x\_0 (dict)} – Summary statistics of the observed data.

• \texttt{t (int)} – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.

• \texttt{par (dict)} – The parameters used to create the summary statistics \texttt{x}. These can be required by some distance functions. Usually, the distance will not depend on the parameters.

Returns \texttt{distance} – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

Return type float

\texttt{__init__ (fun)}

Default constructor.

\texttt{get\_config ()}

Return configuration of the distance.

Returns \texttt{config} – Dictionary describing the distance.

Return type dict

\texttt{pyabc.distance.to\_distance (maybe\_distance)}

Parameters

• \texttt{maybe\_distance (either a Callable as in SimpleFunctionDistance, or a)} –

• \texttt{object. (pyabc.Distance)} –

Returns

Return type A Distance instance.

class \texttt{pyabc.distance.PNormDistance (p: float = 2, w: dict = None)}

Bases: \texttt{pyabc.distance.base.Distance}

Use weighted p-norm

\[
d(x, y) = \left[ \sum_i w_i (x_i - y_i)^p \right]^{1/p}
\]

to compute distances between sets of summary statistics. E.g. set \texttt{p=2} to get a Euclidean distance.

Parameters

• \texttt{p (float)} – p for p-norm. Required \texttt{p \geq 1}, \texttt{p = np.inf} allowed (infinity-norm).

• \texttt{w (dict)} – Weights. Dictionary indexed by time points. Each entry contains a dictionary of numeric weights, indexed by summary statistics labels. If None is passed, a weight of 1 is considered for every summary statistic. If no entry is available in \texttt{w} for a given time point, the maximum available time point is selected.

\texttt{__call__ (x: dict, x\_0: dict, t: int, par: dict = None)} \rightarrow float

Evaluate at time point \texttt{t} the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

Parameters

• \texttt{x (dict)} – Summary statistics of the data simulated for the tentatively sampled parameter.
• $x_0$ (dict) – Summary statistics of the observed data.

• $t$ (int) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.

• $\text{par}$ (dict) – The parameters used to create the summary statistics $x$. These can be required by some distance functions. Usually, the distance will not depend on the parameters.

Returns distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

Return type float

__init__ ($p$: float = 2, $w$: dict = None)
Default constructor.

get_config () -> dict
Return configuration of the distance.

Returns config – Dictionary describing the distance.

Return type dict

class pyabc.distance.AdaptivePNormDistance ($p$: float = 2, $\text{adaptive}$: bool = True, $\text{scale}_\text{function}$=None, $\text{normalize}_\text{weights}$: bool = True, $\text{max}_\text{weight}_\text{ratio}$: float = None)
Bases: pyabc.distance.distance.PNormDistance

In the $p$-norm distance, adapt the weights for each generation, based on the previous simulations. This class is motivated by $^1$.

Parameters

• $p$ (float, optional (default = 2)) – $p$ for $p$-norm. Required $p \geq 1$, $p = \infty$ allowed (infinity-norm).

• $\text{adaptive}$ (bool, optional (default = True)) – True: Adapt distance after each iteration. False: Adapt distance only once at the beginning in initialize(). This corresponds to a pre-calibration.

• $\text{scale}_\text{function}$ (Callable, optional (default = standard_deviation)) – (data: list, $x_0$: float) -> scale: float. Computes the scale (i.e. inverse weight $s = 1 / w$) for a given summary statistic. Here, data denotes the list of simulated summary statistics, and $x_0$ the observed summary statistic. Implemented are absolute_median_deviation, standard_deviation (default), centered_absolute_median_deviation, centered_standard_deviation.

• $\text{normalize}_\text{weights}$ (bool, optional (default = True)) – Whether to normalize the weights to have mean 1. This just possibly smoothes the decrease of epsilon and might aid numeric stability, but is not strictly necessary.

• $\text{max}_\text{weight}_\text{ratio}$ (float, optional (default = None)) – If not None, large weights will be bounded by the ratio times the smallest non-zero absolute weight. In practice usually not necessary, it is theoretically required to ensure convergence.

__init__ ($p$: float = 2, $\text{adaptive}$: bool = True, $\text{scale}_\text{function}$=None, $\text{normalize}_\text{weights}$: bool = True, $\text{max}_\text{weight}_\text{ratio}$: float = None)
Default constructor.

configure_sampler (sampler: pyabc.sampler.base.Sampler)

Make the sampler return also rejected particles, because these are needed to get a better estimate of the summary statistic variabilities, avoiding a bias to accepted ones only.

Parameters sampler (Sampler) – The sampler employed.

get_config () → dict

Return configuration of the distance.

Returns config – Dictionary describing the distance.

Return type dict

initialize (t: int, get_sum_stats: Callable[List[dict]], x_0: dict = None)

Initialize weights.

update (t: int, sum_stats: List[dict])

Update weights based on all simulations.

class pyabc.distance.ZScoreDistance (measures_to_use='all')

Bases: pyabc.distance.distance.DistanceWithMeasureList

Calculate distance as sum of ZScore over the selected measures. The measured Data is the reference for the ZScore.

Hence

\[d(x, y) = \sum_{i \in \text{measures}} \left| \frac{x_i - y_i}{y_i} \right|\]

__call__ (x: dict, x_0: dict = None, t: int = None, par: dict = None) → float

Evaluate at time point t the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

Parameters

- **x** (dict) – Summary statistics of the data simulated for the tentatively sampled parameter.
- **x_0** (dict) – Summary statistics of the observed data.
- **t** (int) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.
- **par** (dict) – The parameters used to create the summary statistics x. These can be required by some distance functions. Usually, the distance will not depend on the parameters.

Returns distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

Return type float

class pyabc.distance.PCADistance (measures_to_use='all')

Bases: pyabc.distance.distance.DistanceWithMeasureList

Calculate distance in whitened coordinates.

A whitening transformation \(X\) is calculated from an initial sample. The distance is measured as euclidean distance in the transformed space. I.e

\[d(x, y) = \|Wx - Wy\|\]
__call__ (x: dict, x_0: dict, t: int = None, par: dict = None) \rightarrow float

Evaluate at time point \( t \) the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

**Parameters**

- \( x (\text{dict}) \) – Summary statistics of the data simulated for the tentatively sampled parameter.
- \( x_0 (\text{dict}) \) – Summary statistics of the observed data.
- \( t (\text{int}) \) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.
- \( \text{par} (\text{dict}) \) – The parameters used to create the summary statistics \( x \). These can be required by some distance functions. Usually, the distance will not depend on the parameters.

**Returns**

- \( \text{distance} \) – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

**Return type** float

__init__ (measures_to_use='all')

Default constructor.

initialize (t: int, get_sum_stats: Callable[List[dict]], x_0: dict = None)

This method is called by the ABCSMC framework before the first use of the distance (at the beginning of ABCSMC.run()), and can be used to calibrate it to the statistics of the samples.

The default is to do nothing.

**Parameters**

- \( t (\text{int}) \) – Time point for which to initialize the distance.
- \( \text{get_sum_stats} (\text{Callable}[\text{List}[\text{dict}]], \text{x_0} : \text{dict} = \text{None}) \) – Returns on command the initial summary statistics.
- \( \text{x_0 (dict, optional)} \) – The observed summary statistics.

class pyabc.distance.MinMaxDistance (measures_to_use='all')

Bases: pyabc.distance.distance.RangeEstimatorDistance

Calculate upper and lower margins as max and min of the parameters. This works surprisingly well for normalization in simple cases

**static lower** (parameter_list)

Calculate the lower margin form a list of parameter values.

**Parameters**

- \( \text{parameter_list} (\text{List[\text{float}}]) \) – List of values of a parameter.

**Returns**

- \( \text{lower_margin} \) – The lower margin of the range calculated from these parameters

**Return type** float

**static upper** (parameter_list)

Calculate the upper margin form a list of parameter values.

**Parameters**

- \( \text{parameter_list} (\text{List[\text{float}}]) \) – List of values of a parameter.

**Returns**

- \( \text{upper_margin} \) – The upper margin of the range calculated from these parameters

**Return type** float
class pyabc.distance.PercentileDistance(measures_to_use='all')
Bases: pyabc.distance.distance.RangeEstimatorDistance

Calculate normalization 20% and 80% from percentiles as lower and upper margins

PERCENTILE = 20
The percentiles

get_config()
Return configuration of the distance.

Returns config – Dictionary describing the distance.

Return type dict

static lower(parameter_list)
Calculate the lower margin form a list of parameter values.

Parameters parameter_list (List[float]) – List of values of a parameter.

Returns lower_margin – The lower margin of the range calculated from these parameters

Return type float

static upper(parameter_list)
Calculate the upper margin form a list of parameter values.

Parameters parameter_list (List[float]) – List of values of a parameter.

Returns upper_margin – The upper margin of the range calculated from these parameters

Return type float

class pyabc.distance.RangeEstimatorDistance(measures_to_use='all')
Bases: pyabc.distance.distance.DistanceWithMeasureList

Abstract base class for distance functions which estimate is based on a range.

It defines the two template methods lower and upper.

Hence

\[ d(x, y) = \sum_{i \in \text{measures}} \frac{|x_i - y_i|}{u_i - l_i} \]

where \( l_i \) and \( u_i \) are the lower and upper margin for measure \( i \).

__call__(x: dict, x_0: dict, t: int = None, par: dict = None) → float
Evaluate at time point \( t \) the distance of the summary statistics of the data simulated for the tentatively sampled particle to those of the observed data.

Abstract method. This method has to be overwritten by all concrete implementations.

Parameters

- \( x (\text{dict}) \) – Summary statistics of the data simulated for the tentatively sampled parameter.
- \( x_0 (\text{dict}) \) – Summary statistics of the observed data.
- \( t (\text{int}) \) – Time point at which to evaluate the distance. Usually, the distance will not depend on the time.
- \( \text{par} (\text{dict}) \) – The parameters used to create the summary statistics \( x \). These can be required by some distance functions. Usually, the distance will not depend on the parameters.
Returns distance – Quantifies the distance between the summary statistics of the data simulated for the tentatively sampled particle and of the observed data.

Return type float

__init__ (measures_to_use='all')
Default constructor.

get_config ()
Return configuration of the distance.

Returns config – Dictionary describing the distance.

Return type dict

initialize (t: int, get_sum_stats: Callable[List[dict]], x_0: dict = None)
This method is called by the ABCSMC framework before the first use of the distance (at the beginning of ABCSMC.run()), and can be used to calibrate it to the statistics of the samples.

The default is to do nothing.

Parameters

• t (int) – Time point for which to initialize the distance.

• get_sum_stats (Callable[], List[dict]) – Returns on command the initial summary statistics.

• x_0 (dict, optional) – The observed summary statistics.

static lower (parameter_list: List[float])
Calculate the lower margin form a list of parameter values.

Parameters parameter_list (List[float]) – List of values of a parameter.

Returns lower_margin – The lower margin of the range calculated from these parameters

Return type float

static upper (parameter_list: List[float])
Calculate the upper margin form a list of parameter values.

Parameters parameter_list (List[float]) – List of values of a parameter.

Returns upper_margin – The upper margin of the range calculated from these parameters

Return type float

class pyabc.distance.DistanceWithMeasureList (measures_to_use='all')
Bases: pyabc.distance.base.Distance

Base class for distance functions with measure list. This class is not functional on its own.

Parameters measures_to_use (Union[str, List[str]]) –

• If set to “all”, all measures are used. This is the default.

• If a list is provided, the measures in the list are used.

• measures refers to the summary statistics.

__init__ (measures_to_use='all')
Default constructor.

get_config ()
Return configuration of the distance.

Returns config – Dictionary describing the distance.
Return type  dict

initialize(t: int, get_sum_stats: Callable[List[dict]], x_0: dict = None)
This method is called by the ABCSMC framework before the first use of the distance (at the beginning of
ABCSMC.run()), and can be used to calibrate it to the statistics of the samples.
The default is to do nothing.

Parameters

• t (int) – Time point for which to initialize the distance.

• get_sum_stats (Callable[[], List[dict]]) – Returns on command the ini-
tial summary statistics.

• x_0 (dict, optional) – The observed summary statistics.

pyabc.distance.median_absolute_deviation(**kwargs)
Calculate the sample median absolute deviation (MAD) from the median, defined as median(abs(data - me-
dian(data)).

pyabc.distance.mean_absolute_deviation(**kwargs)
Calculate the mean absolute deviation from the mean.

pyabc.distance.standard_deviation(**kwargs)
Calculate the sample standard deviation (SD).

pyabc.distance.bias(**kwargs)
Bias of sample to observed value.

pyabc.distance.root_mean_square_deviation(**kwargs)
Square root of the mean squared error, i.e. of the bias squared plus the variance.

pyabc.distance.median_absolute_deviation_to_observation(**kwargs)
Median absolute deviation of data w.r.t. the observation x_0.

pyabc.distance.mean_absolute_deviation_to_observation(**kwargs)
Mean absolute deviation of data w.r.t. the observation x_0.

pyabc.distance.combined_median_absolute_deviation(**kwargs)
Compute the sum of the median absolute deviations to the median of the samples and to the observed value.

pyabc.distance.combined_mean_absolute_deviation(**kwargs)
Compute the sum of the mean absolute deviations to the mean of the samples and to the observed value.

pyabc.distance.standard_deviation_to_observation(**kwargs)
Standard deviation of absolute deviations of the data w.r.t. the observation x_0.

14.3 Acceptor

After summary statistics for samples for given parameters have been generated, it must be checked whether these are
to be accepted or not. This happens in the Acceptor class.
The most typical and simple way is to compute the distance between simulated and observed summary statistics, and
accept if this distance is below some epsilon threshold. However, also more complex acceptance criteria are possible,
in particular when the distance measure and epsilon criteria develop over time.

class pyabc.acceptor.Acceptor
    Bases: object

    This class encodes the acceptance step.
__call__ \((t, distance\_function, eps, x, x\_0, par)\)
Compute distance between summary statistics and evaluate whether to accept or reject.

This class is abstract and cannot be used on its own. The simplest usable class is the derived SimpleAcceptor.

**Parameters**

- **t** \((\text{int})\) – Time point for which to check.
- **distance\_function** \((\text{pyabc.Distance})\) – The distance function. The user is free to use or ignore this function.
- **eps** \((\text{pyabc.Epsilon})\) – The acceptance thresholds. The user is free to use or ignore this object.
- **x** \((\text{dict})\) – Current summary statistics to evaluate.
- **x\_0** \((\text{dict})\) – The observed summary statistics.
- **par** \((\text{pyabc.Parameter})\) – The model parameters used to simulate \(x\).

**Returns**

- **(distance, accept)** \(((\text{float, bool})\) – Distance value obtained and a flag indicating the recommendation to accept or reject. More specifically: True: The distance is below the epsilon threshold. False: The distance is above the epsilon threshold.
- **.. note:** – Currently, only one value encoding the distance is returned (and stored in the database), namely that at time \(t\), even if also other distances affect the acceptance decision, e.g. distances from previous iterations, or in general if the distance is not scalar. This is because the last distance is likely to be most informative for the further process, and because in some parts of ABC a scalar distance value is required.

__init__()
Default constructor.

class pyabc.acceptor.SimpleFunctionAcceptor \((fun=None)\)
Bases: pyabc.acceptor.Acceptor

Initialize from function.

**Parameters**

- **fun** \((\text{Callable, optional})\) – Callable with the same signature as the __call__ method. Per default, accept\_use\_current\_time is used.

__call__ \((t, distance\_function, eps, x, x\_0, par)\)
Compute distance between summary statistics and evaluate whether to accept or reject.

This class is abstract and cannot be used on its own. The simplest usable class is the derived SimpleAcceptor.

**Parameters**

- **t** \((\text{int})\) – Time point for which to check.
- **distance\_function** \((\text{pyabc.Distance})\) – The distance function. The user is free to use or ignore this function.
- **eps** \((\text{pyabc.Epsilon})\) – The acceptance thresholds. The user is free to use or ignore this object.
- **x** \((\text{dict})\) – Current summary statistics to evaluate.
- **x\_0** \((\text{dict})\) – The observed summary statistics.
- **par** \((\text{pyabc.Parameter})\) – The model parameters used to simulate \(x\).
Returns

• **(distance, accept)** ((float, bool)) – Distance value obtained and a flag indicating the recommendation to accept or reject. More specifically: True: The distance is below the epsilon threshold. False: The distance is above the epsilon threshold.

• .. **note::** – Currently, only one value encoding the distance is returned (and stored in the database), namely that at time \( t \), even if also other distances affect the acceptance decision, e.g. distances from previous iterations, or in general if the distance is not scalar. This is because the last distance is likely to be most informative for the further process, and because in some parts of ABC a scalar distance value is required.

```
__init__(fun=None)
    Default constructor.
```

```
static assert_acceptor(maybe_acceptor)
    Create an acceptor object from input.

    Parameters
    maybe_acceptor (Acceptor or Callable) – Either pass a full acceptor,
    or a callable which is then filled into a SimpleAcceptor.

    Returns
    acceptor – An Acceptor object in either case.
```

```
pyabc.acceptor.accept_use_complete_history(t, distance_function, eps, x, x_0, par)
    Use the acceptance criteria from the complete history to evaluate whether to accept or reject.

    This includes time points \( 0, \ldots, t \), as far as these are available. If either the distance function or the epsilon criterion cannot handle any time point in this interval, the resulting error is simply intercepted and the respective time not used for evaluation. This situation can frequently occur when continuing a stopped run. A different behavior is easy to implement.
```

```
pyabc.acceptor.accept_use_current_time(t, distance_function, eps, x, x_0, par)
    Use only the distance function and epsilon criterion at the current time point to evaluate whether to accept or reject.
```

14.4 Models

Models for ABCSMC.

```
class pyabc.model.IntegratedModel(name: str = 'model')
    Bases: pyabc.model.Model

    A model class which integrates simulation, distance calculation and rejection/acceptance.

    This can bring performance improvements if the user can calculate the distance function on the fly during model simulation and interrupt the simulation if the current acceptance threshold cannot be satisfied anymore.

    Subclass this model and implement integrated_simulate to define your own integrated model.
```

```
    Sample, calculate summary statistics, calculate distance, and then accept or not accept a parameter.

    Called from within ABCSMC in each iteration to evaluate a parameter.

    Parameters

    • t (int) – Current time point.
```
- **pars** (*Parameter*) – The model parameters.
- **sum_stats_calculator** (*Callable*) – A function which calculates summary statistics. The user is free to use or ignore this function.
- **distance_calculator** (*pyabc.Distance*) – The distance function. The user is free to use or ignore this function.
- **acceptor** (*pyabc.Acceptor*) – The acceptor judging whether to accept, based on distance and epsilon.
- **x_0** (*dict*) – The observed summary statistics.

**Returns** **model_result** – The result with filled accepted field.

**Return type** **ModelResult**

### integrated_simulate

```
integrated_simulate(pars: pyabc.parameters.Parameter, eps: float) → pyabc.model.ModelResult
```

Method which integrates simulation and acceptance/rejection in a single method.

**Parameters**

- **pars** (*Parameter*) – Parameters at which to evaluate the model
- **eps** (*float*) – Current acceptance threshold. If required, it is effortlessly possible to instead use the entire epsilon_calculator object passed to accept().

**Returns** **model_result** – In case the parameter evaluation is rejected, this method should simply return **ModelResult**(accepted=False). If the parameter was accepted, this method should return either **ModelResult**(accepted=True, distance=distance) or **ModelResult**(accepted=True, distance=distance, sum_stats=sum_stats) in which distance denotes the achieved distance and sum_stats the summary statistics (e.g. simulated data) of the run. Note that providing the summary statistics is optional. If they are provided, then they are also logged in the database.

**Return type** **ModelResult**

### class pyabc.model.Model

#### class pyabc.model.Model

```
class pyabc.model.Model(name: str = 'model')
```

General model. This is the most flexible model class, but also the most complicated one to use. This is an abstract class and not functional on its own. Derive concrete subclasses for actual usage.

The individual steps

- sample
- summary_statistics
- distance
- accept

can be overwritten.

To use this class, at least the sample method has to be overridden.

**Note:** Most likely you do not want to use this class directly, but the **SimpleModel** instead, or even just pass a plain function as model.
Parameters `name (str, optional (default = "model"))`—A descriptive name of the model. This name can simplify further analysis for the user as it is stored in the database.

```python
__init__(name: str = 'model')
```

Initialize self. See help(type(self)) for accurate signature.

```python
```

Sample, calculate summary statistics, calculate distance, and then accept or not accept a parameter.

Called from within ABCSMC in each iteration to evaluate a parameter.

Parameters

- **t (int)** — Current time point.
- **pars (Parameter)** — The model parameters.
- **sum_stats_calculator (Callable)** — A function which calculates summary statistics. The user is free to use or ignore this function.
- **distance_calculator (pyabc.Distance)** — The distance function. The user is free to use or ignore this function.
- **eps_calculator (pyabc.Epsilon)** — The acceptance thresholds.
- **acceptor (pyabc.Acceptor)** — The acceptor judging whether to accept, based on distance and epsilon.
- **x_0 (dict)** — The observed summary statistics.

Returns `model_result` — The result with filled accepted field.

Return type `ModelResult`

```python
```

Sample, calculate summary statistics, and then calculate the distance.

Not required in the current implementation.

Parameters

- **t (int)** — Current time point.
- **pars (Parameter)** — Model parameters.
- **sum_stats_calculator (Callable)** — A function which calculates summary statistics, as passed to `pyabc.smc.ABCSMC`. The user is free to use or ignore this function.
- **distance_calculator (Callable)** — A function which calculates the distance, as passed to `pyabc.smc.ABCSMC`. The user is free to use or ignore this function.
- **x_0 (dict)** — Observed summary statistics.

Returns `model_result` — The result with filled distance.

Return type `ModelResult`

```python
sample(pars: pyabc.parameters.Parameter)
```

Return a sample from the model evaluated at parameters `pars`. This can be raw data, or already summarized statistics thereof.

This method has to be implemented by any subclass.
Parameters `pars (Parameter)` – Dictionary of parameters.

Returns `sample` – The sampled data.

Return type `any`

`summary_statistics (t: int, pars: pyabc.parameters.Parameter, sum_stats_calculator: Callable) \rightarrow pyabc.model.ModelResult`

Sample, and then calculate the summary statistics.

Called from within ABCSMC during the initialization process.

Parameters

- `t (int)` – Current time point.
- `pars (Parameter)` – Model parameters.
- `sum_stats_calculator (Callable)` – A function which calculates summary statistics, as passed to `pyabc.smc.ABCSMC`. The user is free to use or ignore this function.

Returns `model_result` – The result with filled summary statistics.

Return type `ModelResult`

```python
class pyabc.model.ModelResult (sum_stats=None, distance=None, accepted=None)
Bases: object

Result of a model evaluation. Allows to flexibly return summary statistics, distances and accepted/rejected.

__init__ (sum_stats=None, distance=None, accepted=None)
    Initialize self. See help(type(self)) for accurate signature.
```

```python
class pyabc.model.SimpleModel (sample_function: Callable[pyabc.parameters.Parameter, Any], name: str = None)
Bases: pyabc.model.Model

A model which is initialized with a function which generates the samples. For most cases this class will be adequate. Note that you can also pass a plain function to the ABCSMC class, which then gets automatically converted to a SimpleModel.

Parameters

- `sample_function (Callable[[Parameter], Any])` – Returns the sample to be passed to the summary statistics method. This function as a single argument which is a Parameter.
- `name (str, optional)` – The name of the model. If not provided, the names if inferred from the function name of `sample_function`.

__init__ (sample_function: Callable[pyabc.parameters.Parameter, Any], name: str = None)
    Initialize self. See help(type(self)) for accurate signature.
```

```python
static assert_model (model_or_function)
    Alternative constructor. Accepts either a Model instance or a function and returns always a Model instance.

Parameters `model_or_function (Model, function)` – Constructs a SimpleModel instance if a function is passed. If a Model instance is passed, the Model instance itself is returned.

Returns `model`

Return type `SimpleModel` or `Model`
```
sample \( (\text{pars: pyabc.parameters.Parameter}) \)
Return a sample from the model evaluated at parameters \( \text{pars} \). This can be raw data, or already summarized statistics thereof.

This method has to be implemented by any subclass.

**Parameters** \( \text{pars} \) (Parameter) – Dictionary of parameters.

**Returns** sample – The sampled data.

**Return type** any

## 14.5 Acceptance threshold scheduling strategies

Acceptance thresholds (= epsilon) can be calculated based on the distances from the observed data, can follow a pre-defined list, can be constant, or can have a user-defined implementation.

```python
class pyabc.epsilon.ConstantEpsilon(constant_epsilon_value: float)
    Bases: pyabc.epsilon.Epsilon

    Keep epsilon constant over all populations. This acceptance threshold scheduling strategy is most likely only interesting for debugging purposes.

    **Parameters** constant_epsilon_value (float) – The epsilon value for all populations

    **__call__**(t: int)
    Get epsilon value for generation t.

    **Parameters** t (int) – The time point to get the epsilon threshold for.

    **Returns** eps – The epsilon for population t.

    **Return type** float

    **__init__**(constant_epsilon_value: float)
    Constructor.

    **get_config**()
    Return configuration of the distance function.

    **Returns** config – Dictionary describing the distance function.

    **Return type** dict

class pyabc.epsilon.Epsilon
    Bases: abc.ABC

    Abstract epsilon base class.

    This class encapsulates a strategy for setting a new epsilon for each new population.

    **__call__**(t: int) \( \rightarrow \) float
    Get epsilon value for generation t.

    **Parameters** t (int) – The time point to get the epsilon threshold for.

    **Returns** eps – The epsilon for population t.

    **Return type** float

    **__init__**()
    Constructor.
```
get_config()
Return configuration of the distance function.

Returns config – Dictionary describing the distance function.

Return type dict

initialize(t: int, get_weighted_distances: Callable[[], pd.DataFrame])
This method is called by the ABCSMC framework before the first usage of the epsilon and can be used to
 calibrate it to the statistics of the samples.

Parameters

• t (int) – The time point to initialize the epsilon for.

• get_weighted_distances (Callable[[], pd.DataFrame]) – Returns on
demand the distances for initializing the epsilon.

to_json()
Return JSON encoded configuration of the distance function.

Returns json_str – JSON encoded string describing the distance function. The default imple-
mentation is to try to convert the dictionary returned my get_config.

Return type str

update(t: int, weighted_distances: pandas.core.frame.DataFrame)
Update epsilon value to be used as acceptance criterion for generation t.

Default: Do nothing.

Parameters

• t (int) – The generation index to update / set epsilon for. Counting is zero-based. So the
first population has t=0.

• weighted_distances (pd.DataFrame) – The distances that should be used to up-
date epsilon, as returned by Population.get_weighted_distances(). These are usually the
distances of samples accepted in population t-1. The distances may differ from those used
for acceptance in population t-1, if the distance function for population t has been updated.

class pyabc.epsilon.ListEpsilon(values: List[float])
Bases: pyabc.epsilon.Epsilon

Return epsilon values from a predefined list. For every time point enquired later, an epsilon value must exist in
the list.

Parameters values (List[float]) – List of epsilon values. values[t] is the value for
population t.

__call__(t: int)
Get epsilon value for generation t.

Parameters t (int) – The time point to get the epsilon threshold for.

Returns eps – The epsilon for population t.

Return type float

__init__(values: List[float])
Constructor.

get_config()
Return configuration of the distance function.

Returns config – Dictionary describing the distance function.
Return type: dict
class pyabc.epsilon.MedianEpsilon(initial_epsilon: Union[str, int, float] = 'from_sample', median_multiplier: float = 1, weighted: bool = True)
Bases: pyabc.epsilon.QuantileEpsilon
Calculate epsilon as median of the distances from the last population.
__init__(initial_epsilon: Union[str, int, float] = 'from_sample', median_multiplier: float = 1, weighted: bool = True)
Constructor.
class pyabc.epsilon.QuantileEpsilon(initial_epsilon: Union[str, int, float] = 'from_sample', alpha: float = 0.5, quantile_multiplier: float = 1, weighted: bool = True)
Bases: pyabc.epsilon.Epsilon
Calculate epsilon as alpha-quantile of the distances from the last population.
This strategy works even if the posterior is multi-modal. Note that the acceptance threshold calculation is based on the distance to the observation, not on the parameters which generated data with that distance.
If completely different parameter sets produce equally good samples, the distances of their samples to the ground truth data should be comparable.
The idea behind weighting is that the probability $p_k$ of obtaining a distance $\text{eps}_k$ in the next generation should be proportional to the weight $w_k$ of respective particle $k$ in the current generation. Both weighted and non-weighted median should lead to correct results.

Parameters

- **initial_epsilon** (Union[str, int]) –
  - If ‘from_sample’, then the initial quantile is calculated from a sample of the current population size from the prior distribution.
  - If a number is given, this number is used.
- **alpha** (float) – The alpha-quantile to be used, e.g. alpha=0.5 means median.
- **quantile_multiplier** (float) – Multiplies the quantile by that number. Also applies it to the initial quantile if it is calculated from samples. However, it does not apply to the initial quantile if it is given as a number.
- **weighted** (bool) – Flag indicating whether the new epsilon should be computed using weighted (True, default) or non-weighted (False) distances.

__call__(t: int) → float
Epsilon value for time $t$, set before via update() method.

Returns **eps** – The epsilon value for time $t$ (throws error if not existent).

Return type **float**

__init__(initial_epsilon: Union[str, int, float] = 'from_sample', alpha: float = 0.5, quantile_multiplier: float = 1, weighted: bool = True)
Constructor.

get_config()
Return configuration of the distance function.

Returns **config** – Dictionary describing the distance function.

Return type **dict**
**initialize** (*t: int, get_weighted_distances: Callable[[], pandas.core.frame.DataFrame]*)

This method is called by the ABCSMC framework before the first usage of the epsilon and can be used to calibrate it to the statistics of the samples.

**Parameters**

- **t** (*int*) – The time point to initialize the epsilon for.
- **get_weighted_distances** (*Callable[[], pd.DataFrame]*) – Returns on demand the distances for initializing the epsilon.

**update** (*t: int, weighted_distances: pandas.core.frame.DataFrame*)

Compute quantile of the (weighted) distances given in population, and use this to update epsilon.

---

### 14.6 Data store

#### 14.6.1 Purpose of the data store

The most important class here is the History class. The History class is the interface to the database in which pyABC stores and logs information during the ABC-SMC run, but also the interface which allows you to query that information later on.

#### 14.6.2 Initializing the database interface from a file

For querying, you initialize a History object with a valid SQLAlchemy database identifier. For example, if your ABC-SMC data is stored in a file “data.db”, you initialize the History with:

```python
history = History("sqlite:///data.db")
```

Don’t mind the three slashes. This is SQLAlchemy syntax.

If more than one ABC-SMC run is stored in your database file, these runs will have ids. The first run has id=1, the second run id=2, and so on. Per default, the first run found in the database is automatically selected. To select a specific run n (e.g. n=3), do

```python
history.id = n
```

#### 14.6.3 Querying the database

The History class has a number of methods which are relevant for querying the stored data. The most important ones are:

- **History.get_distribution** to retrieve information on the parameter posteriors,
- **History.get_model_probabilities** to retrieve information on the model probabilities in case you’re doing model selection,
- **History.get_all_populations**, to retrieve information on the evolution of the acceptance threshold and the number of sample attempts per population,
- **History.get_nr_particles_per_population**, to retrieve the number of particles per population (this number os not necessarily constant),
- **History.get_weighted_distances**, to retrieve the distances the parameter samples achieved,
- **History.n_populations** to get the total number of populations, and
• `History.total_nr_simulations` to get the total number of simulations, i.e. sample attempts.

Use `get_distribution` to retrieve your posterior particle population. For example,

```python
df, w = history.get_distribution(m)
```

will return a DataFrame `df` of parameters and an array `w` of weights of the particles of model `m` in the last available population. If you’re interested in intermediate populations, add the optional `t` parameter, which indicates the population number (the first population is `t=0`)

```python
df, w = history.get_distribution(m, t)
```

### 14.6.4 What can be stored as summary statistics

Currently, integers, floats, strings, and in general everything that can be converted to a numpy array, can be stored. In addition, it is also possible to store pandas DataFrames.

**Warning:** Storage of pandas DataFrames is considered experimental at this point.

```python
class pyabc.storage.History(db: str, stores_sum_stats: bool = True)
    Bases: object
    History for ABCSMC.
    This class records the evolution of the populations and stores the ABCSMC results.

    db_identifier
        SQLalchemy database identifier. For a relative path use the template “sqlite:///file.db”, for an absolute path “sqlite:///path/to/file.db”, and for an in-memory database “sqlite://”.
        Type str

    stores_sum_stats
        Whether to store summary statistics to the database. Note: this is True by default, and should be set to False only for testing purposes (i.e. to speed up the writing to the file system), as it can not be guaranteed that all methods of pyabc work correctly if the summary statistics are not stored.
        Type bool, optional (default = True)

    id
        The id of the ABCSMC analysis that is currently in use. If there are analyses in the database already, this defaults to the latest id. Manually set if another run is wanted.
        Type int

    __init__(self, db: str, stores_sum_stats: bool = True)
        Initialize history object.

    alive_models(t: int = None) → List
        Get the models which are still alive at time `t`.
        Parameters t (int, optional (default = self.max_t)) – Population index.
        Returns alive – A list which contains the indices of those models which are still alive.
        Return type List

    all_runs()
        Get all ABCSMC runs which are stored in the database.
```
append_population\( (t: \text{int}, \text{current\_epsilon}: \text{float}, \text{population}: \text{pyabc\_storage\_db\_model\_Population}, \text{nr\_simulations}: \text{int}, \text{model\_names}) \)
Append population to database.

**Parameters**

- \( t (\text{int}) \) – Population number.
- \( \text{current\_epsilon} (\text{float}) \) – Current epsilon value.
- \( \text{population} (\text{Population}) \) – List of sampled particles.
- \( \text{nr\_simulations} (\text{int}) \) – The number of model evaluations for this population.
- \( \text{model\_names} (\text{list}) \) – The model names.

**Note.** This function is called by the \text{pyabc\_ABC\_SMC} class internally. You should most likely not find it necessary to call this method under normal circumstances.

db_size
Size of the database.

**Returns**

\( \text{db\_size} \) – Size of the SQLite database in MB. Currently this only works for SQLite databases.

Returns an error string if the DB size cannot be calculated.

**Return type** \( \text{int, str} \)

done()
Close database sessions and store end time of population.

**Note.** This function is called by the \text{pyabc\_ABC\_SMC} class internally. You should most likely not find it necessary to call this method under normal circumstances.

get_all_populations()
Returns a pandas DataFrame with columns

- \( t \) : Population number
- \( \text{population\_end\_time} \) : The end time of the population
- \( \text{samples} \) : The number of sample attempts performed for a population
- \( \text{epsilon} \) : The acceptance threshold for the population

**Returns** \( \text{all\_populations} \) – DataFrame with population info

**Return type** \( \text{pd.DataFrame} \)

get_distribution\( (m: \text{int} = 0, t: \text{int} = \text{None}) \rightarrow (\text{<class 'pandas.core.frame.DataFrame'>}, \text{<class 'numpy.ndarray'>}) \)
Returns the weighted population sample as pandas DataFrame.

**Parameters**

- \( m (\text{int, optional (default = 0)}) \) – Model index.
- \( t (\text{int, optional (default = self\_max\_t}) \) – Population index. If \( t \) is not specified, then the last population is returned.

**Returns**

\( \text{df, w} \) –
• df: a DataFrame of parameters
• w: are the weights associated with each parameter

Return type pandas.DataFrame, np.ndarray

get_model_probabilities (t: Optional[int] = None) → pandas.core.frame.DataFrame
Model probabilities.

Parameters t (int or None (default = None)) – Population index. If None, all pop-
ulations of indices \geq 0 are considered.

Returns probabilities – Model probabilities.

Return type np.ndarray

get_nr_particles_per_population () → pandas.core.series.Series

Returns nr_particles_per_population – A pandas DataFrame containing the number of parti-
cles for each population

Return type pd.DataFrame

get_population (t: int = None)
Create a pyabc.Population object containing all particles, as far as those can be recreated from the database.
In particular, rejected particles are currently not stored.

Parameters t (int, optional (default = self.max_t)) – The population index.

get_population_extended (*, m: Optional[int] = None, t: Union[int, str] = 'last', tidy: bool = True) → pandas.core.frame.DataFrame
Get extended population information, including parameters, distances, summary statistics, weights and more.

Parameters
• m (int or None, optional (default = None)) – The model to query. If
omitted, all models are returned.

• t (int or str, optional (default = "last")) – Can be “last” or “all”, or
a population index (i.e. an int). In case of “all”, all populations are returned. If “last”, only
the last population is returned, for an int value only the corresponding population at that
time index.

• tidy (bool, optional) – If True, try to return a tidy DataFrame, where the individ-
ual parameters and summary statistics are pivoted. Setting tidy to true will only work for
a single model and a single population.

Returns full_population

Return type DataFrame

get_population_strategy ()

Returns The population strategy.

Return type population_strategy

get_weighted_distances (t: int = None) → pandas.core.frame.DataFrame
Population’s weighted distances to the measured sample. These weights do not necessarily sum up to 1.
In case more than one simulation per parameter is performed and accepted the sum might be larger.

Parameters t (int, optional (default = self.max_t)) – Population index. If t
is None, the last population is selected.
Returns `df_weighted` – Weighted distances. The dataframe has column “w” for the weights and column “distance” for the distances.

Return type `pd.DataFrame`

`get_weighted_sum_stats(t: int = None) -> (typing.List[float], typing.List[dict])`

Population’s weighted summary statistics. These weights do not necessarily sum up to 1. In case more than one simulation per parameter is performed and accepted, the sum might be larger.

Parameters `t (int, optional (default = self.max_t))` – Population index. If `t` is None, the latest population is selected.

Returns `(weights, sum_stats)` – In the same order in the first array the weights (multiplied by the model probabilities), and in the second array the summary statistics.

Return type `(List[float], List[dict])`

`get_weighted_sum_stats_for_model(m: int = 0, t: int = None) -> (np.ndarray, typing.List)`

Summary statistics for model `m`. The weights sum to 1, unless there were multiple acceptances per particle.

Parameters
- `m (int, optional (default = 0))` – Model index.
- `t (int, optional (default = self.max_t))` – Population index.

Returns `w, sum_stats` –
- `w`: the weights associated with the summary statistics
- `sum_stats`: list of summary statistics

Return type `np.ndarray, list`

`max_t`

The population number of the last populations. This is equivalent to `n_populations - 1`.

`model_names(t: int = -1)`

Get the names of alive models for population `t`.

Parameters `t (int, optional (default = -1))` – Population index.

`n_populations`

Number of populations stored in the database. This is equivalent to `max_t + 1`.

`nr_of_models_alive(t: int = None) → int`

Number of models still alive.

Parameters `t (int, optional (default = self.max_t))` – Population index.

Returns `nr_alive` – Number of models still alive. None is for the last population

Return type `int >= 0 or None`


Store the initial configuration data.

Parameters
- `ground_truth_model (int)` – Index of the ground truth model.
• \textbf{options}(\textit{dict}) – Of ABC metadata.

• \textbf{observed\_summary\_statistics}(\textit{dict}) – The measured summary statistics.

• \textbf{ground\_truth\_parameter}(\textit{dict}) – The ground truth parameters.

• \textbf{model\_names}(\textit{List}) – A list of model names.

• \textbf{distance\_function\_json\_str}(\textit{str}) – The distance function represented as json string.

• \textbf{eps\_function\_json\_str}(\textit{str}) – The epsilon represented as json string.

• \textbf{population\_strategy\_json\_str}(\textit{str}) – The population strategy represented as json string.

\textbf{Note.} This function is called by the \texttt{pyabc.ABCSMC} class internally. You should most likely not find it necessary to call this method under normal circumstances.

\begin{verbatim}
store_pre_population(ground_truth_model: int, observed_summary_statistics: dict, ground_truth_parameter: dict, model_names: List[str])
\end{verbatim}

Store a dummy pre-population containing some configuration data and in particular some ground truth values.

For the parameters, see \texttt{store\_initial\_data}.

\textbf{Note.} This function is called by the \texttt{pyabc.ABCSMC} class internally. You should most likely not find it necessary to call this method under normal circumstances.

\begin{verbatim}
total_nr_simulations
\end{verbatim}

Number of sample attempts for the ABC run.

\textbf{Returns} \texttt{nr\_sim} – Total nr of sample attempts for the ABC run.

\textbf{Return type} \texttt{int}

\begin{verbatim}
update_nr_samples(t: int = -1, nr_samples: int = 0)
\end{verbatim}

Update the number of samples used in iteration \(t\). The default time of \texttt{PRE\_TIME} implies an update of the number of samples used in calibration.

\textbf{Parameters}

• \texttt{t}(int, optional (default = -1)) – Time to update for.

• \texttt{nr\_samples}(int, optional (default = 0)) – Number of samples reported.

\textbf{Note.} This function is called by the \texttt{pyabc.ABCSMC} class internally. You should most likely not find it necessary to call this method under normal circumstances.

\section{14.7 Transitions (Perturbation Kernels)}

Perturbation strategies. The classes defined here transition the current population to the next one. pyABC implements global and local transitions. Proposals for the subsequent generation are generated from the current generation density estimates of the current generations. This is equivalent to perturbing randomly chosen particles.

These can be passed to \texttt{pyabc.smc.ABCSMC} via the \texttt{transitions} keyword argument.

\textbf{class} \texttt{pyabc.transition.Transition}

\textbf{Bases:} \texttt{sklearn.base.BaseEstimator}

Abstract Transition base class. Derive all Transitions from this class.
Note: This class does a little bit of meta-programming. The fit, pdf and rvs methods are automatically wrapped to handle the special case of no parameters. Hence, you can safely assume that you encounter at least one parameter. All the defined transitions will then automatically generalize to the case of no parameter.

**fit** *(X: pandas.core.frame.DataFrame, w: numpy.ndarray)*
Fit the density estimator (perturber) to the sampled data. Concrete implementations might do something like fitting a KDE.

The parameters given as X and w are automatically stored in self.X and self.w.

**Parameters**
- **X** *(pd.DataFrame)* – The parameters.
- **w** *(array)* – The corresponding weights

**mean_cv** *(n_samples: Union[None, int] = None) → float*
Estimate the uncertainty on the KDE.

**Parameters**
- **n_samples** *(int, optional)* – Estimate the CV for n_samples samples. If this parameter is not given, the sample size of the last fit is used.

**Returns**
- **mean_cv** – The estimated average coefficient of variation.

**Return type** float

Note: A call to this method, as a side effect, also sets the attributes test_points_, test_weights_ and variation_at_test_points_. These are the individual points, weights and variations used to calculate the mean.

**pdf** *(x: Union[pandas.core.series.Series, pandas.core.frame.DataFrame]) → Union[float, numpy.ndarray]*
Evaluate the probability density function (PDF) at x.

**Parameters**
- **x** *(pd.Series, pd.DataFrame)* – Parameter. If x is a series, then x should have the the columns from X passed to the fit method as indices. If x is a DataFrame, then x should have the same columns as X passed before to the fit method. The order of the columns is not important

**Returns**
- **density** – Probability density at x.

**Return type** float

**rvs** *(size=None)*
Sample from the density.

**Parameters**
- **size** *(int, optional)* – Number of independent samples to draw. Defaults to 1 and is in this case equivalent to calling “rvs_single”.

**Returns**
- **samples**

**Return type** The samples as pandas DataFrame

Note: This method can be overridden for efficient implementations. The default is to call rvs_single repeatedly (which might not be the most efficient way).
rvs_single() → pandas.core.series.Series
Random variable sample (rvs).

Returns sample – A sample from the fitted model.

Return type pd.Series

class pyabc.transition.DiscreteTransition
Bases: pyabc.transition.base.Transition
This is a base class for discrete transition kernels.

fit(X: pandas.core.frame.DataFrame, w: numpy.ndarray)
Fit the density estimator (perturber) to the sampled data. Concrete implementations might do something like fitting a KDE.

The parameters given as X and w are automatically stored in self.X and self.w.

Parameters

• X (pd.DataFrame) – The parameters.
• w (array) – The corresponding weights

pdf(x: Union[pandas.core.series.Series, pandas.core.frame.DataFrame]) → Union[float, numpy.ndarray]
Evaluate the probability density function (PDF) at x.

Parameters x (pd.Series, pd.DataFrame) – Parameter. If x is a series, then x should have the columns from X passed to the fit method as indices. If x is a DataFrame, then x should have the same columns as X passed before to the fit method. The order of the columns is not important

Returns density – Probability density at x.

Return type float

rvs_single() → pandas.core.series.Series
Random variable sample (rvs).

Sample from the fitted distribution.

Returns sample – A sample from the fitted model.

Return type pd.Series

class pyabc.transition.MultivariateNormalTransition(scaling=1,
bandwidth_selector=<function silverman_rule_of_thumb>)
Bases: pyabc.transition.base.Transition
Transition via a multivariate Gaussian KDE estimate.

Parameters

• scaling (float) – Scaling is a factor which additionally multiplies the covariance with. Since Silverman and Scott usually have too large bandwidths, it should make most sense to have 0 < scaling <= 1
• bandwidth_selector (optional) – Defaults to silverman_rule_of_thumb. The bandwidth selector is a function of the form f(n_samples: float, dimension: int), where n_samples denotes the (effective) samples size (and is therefore) a float and dimension is the parameter dimension.
__init__ (scaling=1, bandwidth_selector=<function silverman_rule_of_thumb>)
Initialize self. See help(type(self)) for accurate signature.

fit (X: pandas.core.frame.DataFrame, w: numpy.ndarray)
Fit the density estimator (perturber) to the sampled data. Concrete implementations might do something like fitting a KDE.

The parameters given as X and w are automatically stored in self.X and self.w.

Parameters
• X (pd.DataFrame) – The parameters.
• w (array) – The corresponding weights

density (x: Union[pandas.core.series.Series, pandas.core.frame.DataFrame])
Evaluate the probability density function (PDF) at x.

Parameters
x (pd.Series, pd.DataFrame) – Parameter. If x is a series, then x should have the the columns from X passed to the fit method as indices. If x is a DataFrame, then x should have the same columns as X passed before to the fit method. The order of the columns is not important

Returns density – Probability density at x.

Return type float

rvs_single ()
Random variable sample (rvs).

Sample from the fitted distribution.

Returns sample – A sample from the fitted model.

Return type pd.Series

class pyabc.transition.GridSearchCV (estimator=None, param_grid=None, scoring=None, fit_params=None, n_jobs=1, iid=True, refit=True, cv=5, verbose=0, pre_dispatch='2*n_jobs', error_score='raise', return_train_score=True)

Bases: sklearn.model_selection._search.GridSearchCV

Do a grid search to automatically select the best parameters for transition classes such as the pyabc.transition.MultivariateNormalTransition.

This is essentially a thin wrapper around ‘sklearn.model_selection.GridSearchCV’. It translates the scikit-learn interface to the interface used in pyABC. It implements hence a thin adapter pattern.

The parameters are just as for sklearn.model_selection.GridSearchCV. Major default values:
• estimator = MultivariateNormalTransition()
• param_grid = {'scaling': np.linspace(0.05, 1.0, 5)}
• cv = 5

__init__ (estimator=None, param_grid=None, scoring=None, fit_params=None, n_jobs=1, iid=True, refit=True, cv=5, verbose=0, pre_dispatch='2*n_jobs', error_score='raise', return_train_score=True)
Initialize self. See help(type(self)) for accurate signature.

fit (X, y=None, groups=None)
Run fit with all sets of parameters.

Parameters
- **X** (*array-like, shape = [n_samples, n_features]*) – Training vector, where n_samples is the number of samples and n_features is the number of features.

- **y** (*array-like, shape = [n_samples] or [n_samples, n_output], optional*) – Target relative to X for classification or regression; None for unsupervised learning.

- **groups** (*array-like, with shape (n_samples,), optional*) – Group labels for the samples used while splitting the dataset into train/test set.

- **fit_params** (*dict of string -> object*) – Parameters passed to the fit method of the estimator

---

**exception** pyabc.transition.NotEnoughParticles

Bases: Exception

**class** pyabc.transition.LocalTransition(*k=None, k_fraction=0.25, scaling=1*)

Bases: pyabc.transition.base.Transition

Local KDE fit. Takes into account only the k nearest neighbors, similar to [Filippi].

**Parameters**

- **k** (*int*) – Number of nearest neighbors for local covariance calculation.

- **scaling** (*float*) – Scaling factor for the local covariance matrices.

- **k_fraction** (*float, optional*) – Calculate number of nearest neighbors to use according to $k = k_{fraction} \times population\_size$ (and rounds it).

**EPS**

Scaling of the identity matrix to be added to the covariance in case the covariances are not invertible.

**Type** float

**__init__**(k=None, k_fraction=0.25, scaling=1)

Initialize self. See help(type(self)) for accurate signature.

**fit**(*X, w*)

Fit the density estimator (perturber) to the sampled data. Concrete implementations might do something like fitting a KDE.

The parameters given as X and w are automatically stored in self.X and self.w.

**Parameters**

- **X** (*pd.DataFrame*) – The parameters.

- **w** (*array*) – The corresponding weights

**pdf**(*x*)

Evaluate the probability density function (PDF) at x.

**Parameters**

- **x** (*pd.Series, pd.DataFrame*) – Parameter. If x is a series, then x should have the the columns from X passed to the fit method as indices. If x is a DataFrame, then x should have the same columns as X passed before to the fit method. The order of the columns is not important

**Returns**

- **density** – Probability density at x.

**Return type** float

**rvs_single**()

Random variable sample (rvs).

Sample from the fitted distribution.
Returns sample – A sample from the fitted model.

Return type pd.Series

pyabc.transition.scott_rule_of_thumb (n_samples, dimension)
Scott’s rule of thumb.

\[
\left( \frac{1}{n} \right)^{\frac{1}{d+4}}
\]

(see also scipy.stats.kde.gaussian_kde.scotts_factor)

pyabc.transition.silverman_rule_of_thumb (n_samples, dimension)
Silverman’s rule of thumb.

\[
\left( \frac{4}{n(d+2)} \right)^{\frac{1}{d+4}}
\]

(see also scipy.stats.kde.gaussian_kde.silverman_factor)

class pyabc.transition.DiscreteRandomWalkTransition (n_steps: int = 1, p_l: float = 0.3333333333333333, p_r: float = 0.3333333333333333, p_c: float = 0.3333333333333333)
Bases: pyabc.transition.base.DiscreteTransition

This transition is based on a discrete random walk. This may be useful for discrete ordinal parameter distributions that can be described as lying on the grid of integers.

Note: This transition does not adapt to the problem structure and thus has potentially slow convergence. Further, the transition does not satisfy proposal >> prior, so that it is indeed not valid as an importance sampling distribution. This can be overcome by selecting the number of steps as a random variable.

Parameters

**n_steps** (int, optional (default = 1)) – Number of random walk steps to take.

__init__ (n_steps: int = 1, p_l: float = 0.3333333333333333, p_r: float = 0.3333333333333333, p_c: float = 0.3333333333333333)
Initialize self. See help(type(self)) for accurate signature.

fit (X: pandas.core.frame.DataFrame, w: numpy.ndarray)
Fit the density estimator (perturber) to the sampled data. Concrete implementations might do something like fitting a KDE.

The parameters given as X and w are automatically stored in self.X and self.w.

Parameters

- **X** (pd.DataFrame) – The parameters.
- **w** (array) – The corresponding weights

pdf (x: Union[pandas.core.series.Series, pandas.core.frame.DataFrame]) → Union[float, numpy.ndarray]
Evaluate the probability mass function (PMF) at x.

rvs_single () → pandas.core.series.Series
Random variable sample (rvs).
Sample from the fitted distribution.
Returns sample – A sample from the fitted model.
Return type pd.Series

14.8 Population strategy

Strategies to choose the population size.
The population size can be constant or can change over the course of the generations.

```python
class pyabc.populationstrategy.AdaptivePopulationSize(start_nr_particles,
mean_cv=0.05,
*,
max_population_size=inf,
min_population_size=10,
nr_samples_per_parameter=1,
n_bootstrap=10)
```

Bases: pyabc.populationstrategy.PopulationStrategy

Adapt the population size according to the mean coefficient of variation error criterion, as detailed in\(^1\). This strategy tries to respond to the shape of the current posterior approximation by selecting the population size such that the variation of the density estimates matches the target variation given via the mean_cv argument.

Parameters

- **start_nr_particles** (int) – Number of particles in the first populations
- **mean_cv** (float, optional) – The error criterion. Defaults to 0.05. A smaller value leads generally to larger populations. The error criterion is the mean coefficient of variation of the estimated KDE.
- **max_population_size** (int, optional) – Max nr of allowed particles in a population. Defaults to infinity.
- **min_population_size** (int, optional) – Min number of particles allowed in a population. Defaults to 10
- **nr_samples_per_parameter** (int, optional) – Defaults to 1.
- **n_bootstrap** (int, optional) – Number of bootstrapped populations to use to estimate the CV. Defaults to 10.

__init__(start_nr_particles, mean_cv=0.05, *, max_population_size=inf, min_population_size=10, nr_samples_per_parameter=1, n_bootstrap=10)

Initialize self. See help(type(self)) for accurate signature.

adapt_population_size(transitions: List[pyabc.transition.base.Transition], model_weights: numpy.ndarray)

Select the population size for the next population.

Parameters

- **transitions** (List of Transitions) –
- **model_weights** (array of model weights) –

Returns n – The new population size
Return type int

get_config()
Get the configuration of this object.

Returns Configuration of the class as dictionary
Return type dict
class pyabc.populationstrategy.ConstantPopulationSize(nr_particles: int, *, nr_samples_per_parameter: int = 1)
Bases: pyabc.populationstrategy.PopulationStrategy
Constant size of the different populations

Parameters

• nr_particles (int) – Number of particles per populations
• nr_samples_per_parameter (int) – Number of samples to draw for a proposed parameter

adapt_population_size(transitions, model_weights)
Select the population size for the next population.

Parameters

• transitions (List of Transitions) –
• model_weights (array of model weights) –

Returns n – The new population size
Return type int
class pyabc.populationstrategy.PopulationStrategy(nr_particles: int, *, nr_samples_per_parameter: int = 1)
Bases: object
Strategy to select the sizes of the populations.

This is a non-functional abstract base implementation. Do not use this class directly. Subclasses must override the adapt_population_size method.

Parameters

• nr_particles (int) – Number of particles per populations
• nr_samples_per_parameter (int, optional) – Number of samples to draw for a proposed parameter. Default is 1.

__init__(nr_particles: int, *, nr_samples_per_parameter: int = 1)
Initialize self. See help(type(self)) for accurate signature.

adapt_population_size(transitions: List[pyabc.transition.base.Transition], model_weights: numpy.ndarray)
Select the population size for the next population.

Parameters

• transitions (List of Transitions) –
• model_weights (array of model weights) –

Returns n – The new population size
Return type int
get_config()
Get the configuration of this object.

Returns  Configuration of the class as dictionary
Return type  dict
to_json()
Return the configuration as json string. Per default, this converts the dictionary returned by get_config to json.

Returns  Configuration of the class as json string.
Return type  str

14.9 Multi-core and Distributed Sampling

The choice of the sampler determines in which way parallelization is performed. See also the explanation of the samplers.

class pyabc.sampler.Sample (record_rejected: bool = False)
Bases: object

A Sample is created and filled during the sampling process by the Sampler.

Parameters record_rejected (bool) – Whether to record rejected particles as well, along with accepted ones.

__init__ (record_rejected: bool = False)
Initialize self. See help(type(self)) for accurate signature.

all_sum_stats
Get all summary statistics.

Returns  all_sum_stats – Concatenation of all the all_sum_stats lists of all particles added and accepted to this sample via append().

Return type  List
append (particle: pyabc.population.Particle)
Add new particle to the sample.

Parameters particle (Particle) – Sampled particle containing all information needed later.

get_accepted_population () → pyabc.population.Population

Returns  population – A population of only the accepted particles.

Return type  Population

n_accepted
returns: n_accepted – Number of accepted particles. :rtype: int

class pyabc.sampler.Sampler
Bases: abc.ABC

Abstract Sampler base class.

Produce valid particles: pyabc.parameters.ValidParticle.

Parameters
• **nr_evaluations** *(int)* – This is set after a population and counts the total number of model evaluations. This can be used to calculate the acceptance rate.

• **sample_factory** *(SampleFactory)* – A factory to create empty samples.

```python
def __init__(self)
    Initialize self. See help(type(self)) for accurate signature.
```

```python
class pyabc.sampler.SingleCoreSampler
    Bases: pyabc.sampler.base.Sampler

    Sample on a single core. No parallelization.
```

```python
class pyabc.sampler.MulticoreParticleParallelSampler(n_procs=None, daemon=True)
    Bases: pyabc.sampler.multicorebase.MultiCoreSampler

    Samples on multiple cores using the multiprocessing module. This sampler is optimized for low latencies and is efficient, even if the individual model evaluations are fast.

    Requires no pickling of the `sample_one`, `simulate_one` and `accept_one` function. This is achieved using fork on linux (see `Sampler`).

    The simulation results are still pickled as they are transmitted from the worker processes back to the parent process. Depending on the kind of summary statistics this can be fast or slow. If your summary statistics are only a dict with a couple of numbers, the overhead should not be substantial. However, if your summary statistics are large numpay arrays or similar, this could cause overhead.

    **Parameters**

    - **n_procs** *(int, optional)* – If set to None, the Number of cores is determined according to `pyabc.sge.nr_cores_available()`.

        **Warning:** Windows support is not tested. As there is no fork on Windows. This sampler might not work.

```python
class pyabc.sampler.MappingSampler(map_=<class 'map'>, mapper_pickles=False)
    Bases: pyabc.sampler.base.Sampler

    Parallelize via a map operation. This sampler can be applied in a multi-core or in a distributed setting.

    **Parameters**

    - **map** *(map like function)* – A function which works like the built in `map`. The map can be really any generic map operations. Possible candidates include:

        - multiprocessing.pool.map (see `https://docs.python.org/3/library/multiprocessing.html#multiprocessing.pool.Pool`)

        - `pyabc.sge.SGE`’s map method. This mapper is useful in SGE-like environments where you don’t want to start workers which run forever.

        - Dask’s distributed `distributed.Client`’s map (see `https://distributed.readthedocs.io/en/latest/api.html#client`)

        - Ipython parallel’ map (see `http://ipyparallel.readthedocs.io/en/latest/task.html#quick-and-easy-parallelism`)

        and many other implementations.

        Each of the mapped function calls samples until it gets one accepted particle. This could have a performance impact if one of the sample tasks runs very long and all the other tasks are already finished. The sampler then has to wait until the last sample task is finished.

14.9. Multi-core and Distributed Sampling 127
• **mapper_pickles** *(bool, optional)* – Whether the mapper handles the pickling itself or the MappingSampler class should handle serialization.

The default is `False`. While this setting is compatible with a larger range of map functions, its performance can be suboptimal. As possibly too much serialization and deserialization is done, which could limit overall performance if the model evaluations are comparatively fast. The passed map function might implement more efficient serialization. For example, for the `pyabc.sge.SGE` mapper, this option should be set to `True` for better performance.

```
__init__(map_=<class 'map'>, mapper_pickles=False)
```

Initialize self. See `help(type(self))` for accurate signature.

```py
class pyabc.sampler.DaskDistributedSampler(dask_client=None, client_max_jobs=inf, default_pickle=False, batch_size=1)
Bases: pyabc.sampler.eps_mixin.EPSMixin, pyabc.sampler.base.Sampler
```

Parallelize with Dask. This sampler is intended to be used with a pre-configured Dask client, but is able to initialize client, scheduler and workers on its own on the local machine for testing/debugging purposes.

**Parameters**

• **dask_client** *(dask.Client, optional)* – The configured dask Client. If none is provided, then a local dask distributed cluster is created.

• **client_max_jobs** – Maximum number of jobs that can submitted to the client at a time. If this value is smaller than the maximum number of cores provided by the distributed infrastructure, the infrastructure will not be utilized fully.

• **default_pickle** – Specify if the sampler uses python's default pickle function to communicate the submit function to python; if this is the case, a cloud-pickle based workaround is used to pickle the simulate and evaluate functions. This allows utilization of locally defined functions, which can not be pickled using default pickle, at the cost of an additional pickling overhead. For Dask, this workaround should not be necessary and it should be save to use `default_pickle=False`.

• **batch_size** *(int, optional)* – Number of parameter samples that are evaluated in one remote execution call. Batch submission can be used to reduce the communication overhead for fast (ms-s) model evaluations. Large batch sizes can result in unnecessary model evaluations. By default, `batch_size=1`, i.e. no batching is done.

```
__init__(dask_client=None, client_max_jobs=inf, default_pickle=False, batch_size=1)
```

Initialize self. See `help(type(self))` for accurate signature.

```py
class pyabc.sampler.RedisEvalParallelSampler(host='localhost', port=6379, batch_size=1)
Bases: pyabc.sampler.base.Sampler
```

Redis based low latency sampler. This sampler is well performing in distributed environments. It is usually faster than the `pyabc.sampler.DaskDistributedSampler` for short model evaluation runtimes. The longer the model evaluation times, the less the advantage becomes. It requires a running Redis server as broker.

This sampler requires workers to be started via the command `abc-redis-worker`. An example call might look like `abc-redis-worker --host=123.456.789.123 --runtime=2h` to connect to a Redis server on IP `123.456.789.123` and to terminate the worker after finishing the first population which ends after 2 hours since worker start. So the actual runtime might be longer than 2h. See `abc-redis-worker --help` for its options.

Use the command `abc-redis-manager` to retrieve info and stop the running workers.

Start as many workers as you wish. Workers can be dynamically added during the ABC run.

**Parameters**
• **host** *(str, optional)* – IP address or name of the Redis server. Default is “localhost”.

• **port** *(int, optional)* – Port of the Redis server. Default is 6379.

• **batch_size** *(int, optional)* – Number of model evaluations the workers perform before contacting the REDIS server. Defaults to 1. Increase this value if model evaluation times are short or the number of workers is large to reduce communication overhead.

```python
__init__(host='localhost', port=6379, batch_size=1)
```
Initialize self. See help(type(self)) for accurate signature.

```python
n_worker()
```
Get the number of connected workers.

**Returns**

**Return type** Number of workers connected.

class **pyabc.sampler.MulticoreEvalParallelSampler**(n_procs=None, daemon=True)
Bases: pyabc.sampler.multicorebase.MultiCoreSampler

Multicore Evaluation parallel sampler.

Implements the same strategy as `pyabc.sampler.RedisEvalParallelSampler` or `pyabc.sampler.DaskDistributedSampler`.

However, parallelization is restricted to a single machine with multiple processes. This sampler has very low communication overhead and is thus suitable for short running model evaluations.

Requires no pickling of the `sample_one`, `simulate_one` and `accept_one` function. This is achieved using fork on linux (see `Sampler`).

The simulation results are still pickled as they are transmitted from the worker processes back to the parent process. Depending on the kind of summary statistics this can be fast or slow. If your summary statistics are only a dict with a couple of numbers, the overhead should not be substantial. However, if your summary statistics are large numpy arrays or similar, this could cause overhead.

**Parameters**

**n_procs** *(int, optional)* – If set to None, the Number of cores is determined according to `pyabc.sge.nr_cores_available()`.

```python
class **pyabc.sampler.RedisEvalParallelSamplerServerStarter**(host='localhost', port=6379, batch_size=1, workers=2, processes_per_worker=1)
```
Bases: pyabc.sampler.redis_eps.sampler.RedisEvalParallelSampler

Simple routine to start a redis-server with 2 workers for test purposes.

```python
__init__(host='localhost', port=6379, batch_size=1, workers=2, processes_per_worker=1)
```
Initialize self. See help(type(self)) for accurate signature.

```python
cleanup()
```
Cleanup workers and server.

class **pyabc.sampler.ConcurrentFutureSampler**(cfuture_executor=None, client_max_jobs=200, default_pickle=True, batch_size=1)
Bases: pyabc.sampler.eps_mixin.EPSMixin, pyabc.sampler.base.Sampler

Parallelize with an arbitrary sampler that implements the python concurrent futures executor interface. Specifically, it needs to implement a “submit” function that is able to evaluate arbitrary function handles and return a concurrent future result object.
Parameters

- **cfuture_executor** (*concurrent.futures.Executor, required*) – Configured object that implements the concurrent.futures.Executor interface

- **client_max_jobs** – Maximum number of jobs that can submitted to the client at a time. If this value is smaller than the maximum number of cores provided by the distributed infrastructure, the infrastructure will not be utilized fully.

- **default_pickle** – Specify if the sampler uses python's default pickle function to communicate the submit function to python; if this is the case, a cloud-pickle based workaround is used to pickle the simulate and evaluate functions. This allows utilization of locally defined functions, which can not be pickled using default pickle, at the cost of an additional pickling overhead.

- **batch_size** (*int, optional*) – Number of parameter samples that are evaluated in one remote execution call. Batch submission can be used to reduce the communication overhead for fast (ms-s) model evaluations. Large batch sizes can result in unnecessary model evaluations. By default, batch_size=1, i.e. no batching is done.

```
__init__(cfuture_executor=None, client_max_jobs=200, default_pickle=True, batch_size=1)
```

Initialize self. See help(type(self)) for accurate signature.

## 14.10 Parameters

```python
class pyabc.parameters.Parameter(*args, **kwargs)
Bases: pyabc.parameters.ParameterStructure

A single model parameter.

Parameters are essentially a dictionary with the additional functionality to add and subtract parameters. I.e. `par_1 + par_2` adds key wise.

Contents can be accessed with square brackets or in dot notation.

For example

```python
>>> p = Parameter(a=1, b=2)
>>> assert p.a == p["a"]
```

or

```python
>>> p = Parameter({"a": 1, "b": 2})
>>> assert p.a == p["a"]
```
```
copy() \rightarrow pyabc.parameters.Parameter
Copy the parameter.
```
14.11 Particles and Populations

A particle contains the sampled parameters and simulated data. A population gathers all particles collected in one SMC iteration.

class pyabc.population.Particle(m: int, parameter: pyabc.parameters.Parameter, weight: float, accepted_sum_stats: List[dict], accepted_distances: List[float], rejected_sum_stats: List[dict] = None, rejected_distances: List[float] = None, accepted: bool = True)

Bases: object

An (accepted or rejected) particle, containing the information that will also be stored in the database. Stores all summary statistics that were generated during the creation of this particle, and a flag indicating whether this particle was accepted or not.

Parameters
• m (int) – The model index.
• parameter (Parameter) – The model specific parameter.
• weight (float, 0 <= weight <= 1) – The weight of the particle.
• accepted_sum_stats (List[dict]) – List of accepted summary statistics. This list is usually of length 1. This list is longer only if more than one sample is taken for a particle. This list has length 0 if the particle is rejected.
• accepted_distances (List[float]) – A particle can contain more than one sample. If so, the distances of the individual samples are stored in this list. In the most common case of a single sample, this list has length 1.
• rejected_sum_stats (List[dict]) – List of rejected summary statistics.
• rejected_distances (List[float]) – List of rejected distances.
• accepted (bool) – True if particle was accepted, False if not.

Note: There are two different ways of weighting particles: First, the weights can be calculated as emerges from the importance sampling. Second, the weights of particles belonging to one model can be summed to, after normalization, find model probabilities. Then, the weights of all particles belonging to one model can be summed to one. Weighting is transferred to the second way in _normalize_weights() in order to also have access to model probabilities. This mode is also stored in the database. If one needs access to the first weighting scheme later on again, one has to perform backwards transformation, multiplying the weights with the model probabilities.

__init__(m: int, parameter: pyabc.parameters.Parameter, weight: float, accepted_sum_stats: List[dict], accepted_distances: List[float], rejected_sum_stats: List[dict] = None, rejected_distances: List[float] = None, accepted: bool = True)

Initialize self. See help(type(self)) for accurate signature.

class pyabc.population.Population(particles: List[pyabc.population.Particle])

Bases: object

A population contains a list of particles and offers standardized access to them. Upon initialization, the particle weights are normalized and model probabilities computed as described in _normalize_weights.

__init__(particles: List[pyabc.population.Particle])

Initialize self. See help(type(self)) for accurate signature.
get_accepted_sum_stats() → List[dict]
Return a list of all accepted summary statistics.

get_for_keys(keys)
Get dataframe of population values. Possible entries of keys: weight, distance, sum_stat, parameter.

Returns
• Dictionary where the keys are associated to same-ordered lists
• of the corresponding values.

get_list() → List[pyabc.population.Particle]
Returns
Return type A copy of the underlying particle list.

get_model_probabilities() → dict
Get probabilities of the individual models.

Returns model_probabilities – The model probabilities.
Return type List

get_weighted_distances() → pandas.core.frame.DataFrame
Create DataFrame of (distance, weight)'s. The particle weights are multiplied by the model probabilities.
If one simulation per particle was performed, the weights thus sum to 1. If more than one simulation per
particle was performed, this does not have to be the case, and post-normalizing may be necessary.

Returns weighted_distances – A pd.DataFrame containing in column ‘distance’ the distances
and in column ‘w’ the scaled weights.

Return type pd.DataFrame:

to_dict() → dict
Create a dictionary representation, creating a list of particles for each model.

Returns store – A dictionary with the models as keys and a list of particles for each model as
values.

Return type dict
update_distances(distance_to_ground_truth: Callable[dict, float])
Update the distances of all summary statistics of all particles according to the passed distance function
(which is typically different from the distance function with which the original distances were computed).

Parameters distance_to_ground_truth – Distance function to the observed summary
statistics.

14.12 Random variables

class pyabc.random_variables.Distribution(*args, **kwargs)
Bases: pyabc.parameters.ParameterStructure

Distribution of parameters for a model.

A distribution is a collection of RVs and/or distributions. Essentially something like a dictionary of random
variables or distributions. The variables from which the distribution is initialized are independent.

This should be used to define a prior.
copy() \rightarrow \text{pyabc.random_variables.Distribution}

Copy the distribution

Returns \text{copied\_distribution} – A copy of the distribution.

Return type \text{Distribution}

classmethod \text{from\_dictionary\_of\_dictionaries}(dict\_of\_dicts: \text{dict}) \rightarrow \text{pyabc.random_variables.Distribution}

Create distribution from dictionary of dictionaries

Parameters \text{dict\_of\_dicts}(\text{dict}) – The keys of the dict indicate the parameters names. The values are itself dictionaries representing scipy.stats distribution. I.e. the have the key “name” and at least one of the keys “args” or “kwargs”.

Returns \text{distribution} – Created distribution.

Return type \text{Distribution}

get\_parameter\_names() \rightarrow \text{list}

Sorted list of parameter names.

Returns \text{sorted\_names} – Sorted list of parameter names.

Return type \text{list}

pdf(x: \text{Union[pyabc.parameters.Parameter, dict]})

Get combination of probability density function (for continuous variables) and probability mass function (for discrete variables) at point x

Parameters \text{x}(\text{Union[Parameter, dict]}) – Evaluate at the given Parameter \text{x}.

rvs() \rightarrow \text{pyabc.parameters.Parameter}

Sample from joint distribution

Returns \text{parameter} – A parameter which was sampled.

Return type \text{Parameter}

update\_random\_variables(**random\_variables)

Update random variables within the distribution

Parameters **\text{random\_variables} – keywords are the parameters’ names, the values are random variable.

class \text{pyabc.random_variables.LowerBoundDecorator}(\text{component: pyabc.random_variables.RV, lower\_bound: float})

\text{Bases: pyabc.random_variables.RVDecorator}

Impose a strict lower bound on a random variable. Condition RV X to X > lower bound. In particular P(X = lower\_bound) = 0.

Note: Sampling is done via rejection. Up to 10000 samples are taken from the decorated RV. The first sample within the permitted range is then taken. Otherwise None is returned.

Parameters

• \text{component}(\text{RV}) – The decorated random variable.

• \text{lower\_bound}(\text{float}) – The lower bound.
class pyabc.random_variables.ModelPerturbationKernel(nr_of_models: int, probability_to_stay: Optional[float] = None)

Bases: object

Model perturbation kernel.

Parameters

- **nr_of_models** (int) – Number of models
• **probability_to_stay** *(Union[float, None])* – If None, probability to stay is set to 1/nr_of_models. Otherwise, the supplied value is used.

```python
__init__(nr_of_models: int, probability_to_stay: Optional[float] = None)
```
Initialize self. See help(type(self)) for accurate signature.

```python
pmf(n: int, m: int) \rightarrow float
```
Parameters

- **n** *(int)* – Model target nr.
- **m** *(int)* – Model source nr.

Returns **probability** – Probability with which to jump from m to n.

Return type **float**

```python
rvs(m: int) \rightarrow int
```
Sample a Kernel jump from model m to another model.

Parameters **m** *(int)* – Model source nr.

Returns **target** – Target model nr.

Return type **int**

```python
class pyabc.random_variables.RV(name: str, *args, **kwargs)
```
Concrete random variable.

Parameters

- **name** *(str)* – Name of the distribution as in scipy.stats
- **args** – Arguments as in scipy.stats matching the distribution with name “name”.
- **kwargs** – Keyword arguments as in scipy.stats matching the distribution with name “name”.

```python
__init__(name: str, *args, **kwargs)
```
Initialize self. See help(type(self)) for accurate signature.

```python
cdf(x, *args, **kwargs)
```
Cumulative distribution function.

Parameters **x** *(float)* – Cumulative distribution function at x.

Returns **density** – Cumulative distribution function at x.

Return type **float**

```python
copy()
```
Copy the random variable.

Returns **copied_rv** – A copy of the random variable.

Return type **RVBase**

```python
distribution = None
```
the scipy.stats. . . . distribution object

```python
classmethod from_dictionary(dictionary: dict) \rightarrow pyabc.random_variables.RV
```
Construct random variable from dictionary.

Parameters **dictionary** *(dict)* – A dictionary with the keys
• "name" (mandatory)
• "args" (optional)
• "kwargs" (optional)
as in scipy.stats.

**Note:** Either the “args” or the “kwargs” key has to be present.

```python
pdf(x, *args, **kwargs)
```
Probability density function

**Parameters**
- `x (float)` – Probability density at x.

**Returns**
- `density` – Probability density at x.

**Return type**
- `float`

```python
pmf(x, *args, **kwargs)
```
Probability mass function

**Parameters**
- `x (int)` – Probability mass at x.

**Returns**
- `mass` – The mass at x.

**Return type**
- `float`

```python
rvs(*args, **kwargs)
```
Sample from the RV.

**Returns**
- `sample` – A sample from the random variable.

**Return type**
- `float`

**class** `pyabc.random_variables.RVBase`

Bases: `abc.ABC`

Random variable abstract base class.

**Note:** Why introduce another random variable class and not just use the one’s provided in scipy.stats? This funny construction is done because scipy.stats distributions are not pickleable. This class is really a very thin wrapper around scipy.stats distributions to make them pickleable. It is important to be able to pickle them to execute the ACBSMC algorithm in a distributed cluster environment.

```python
cdf(x: float, *args, **kwargs) → float
```
Cumulative distribution function.

**Parameters**
- `x (float)` – Cumulative distribution function at x.

**Returns**
- `density` – Cumulative distribution function at x.

**Return type**
- `float`

```python
copy() → pyabc.random_variables.RVBase
```
Copy the random variable.

**Returns**
- `copied_rv` – A copy of the random variable.

**Return type**
- `RVBase`
**pdf**(*x*, *float, *args, **kwargs*) → float

Probability density function

**Parameters**

- **x** (*float*) – Probability density at x.

**Returns**

- **density** – Probability density at x.

**Return type**

- float

**pmf**(*x*, *args, **kwargs*) → float

Probability mass function

**Parameters**

- **x** (*int*) – Probability mass at x.

**Returns**

- **mass** – The mass at x.

**Return type**

- float

**rvs**(*args, **kwargs*) → float

Sample from the RV.

**Returns**

- **sample** – A sample from the random variable.

**Return type**

- float

---

**class** **pyabc.random_variables.RVDecorator**

**Bases:** pyabc.random_variables.RVBase

Random variable decorator base class.

Implement a decorator pattern.

Further decorators should derive from this class.

It stores the decorated random variable in self.component

Overwrite the method decorator_repr the represent the decorator type. The decorated variable will then be automatically included in the call to **__repr__**.

**Parameters**

- **component** (*RVBase*) – The random variable to be decorated.

**__init__**(*component* : pyabc.random_variables.RVBase)

Initialize self. See help(type(self)) for accurate signature.

**cdf**(*x*, *args, **kwargs*)

Cumulative distribution function.

**Parameters**

- **x** (*float*) – Cumulative distribution function at x.

**Returns**

- **density** – Cumulative distribution function at x.

**Return type**

- float

**component** = None

The decorated random variable

**copy**()

Copy the random variable.

**Returns**

- **copied_rv** – A copy of the random variable.

**Return type**

- RVBase

**decorator_repr**() → str

Represent the decorator itself.

Template method.
The __repr__ method used decorator_repr and the __repr__ of the decorated RV to build a combined representation.

**Returns** decorator_repr – A string representing the decorator only.

**Return type** str

**pdf**(*x, *args, **kwargs*)

Probability density function

**Parameters**

- **x** (*float*) – Probability density at x.

**Returns** density – Probability density at x.

**Return type** float

**pmf**(*x, *args, **kwargs*)

Probability mass function

**Parameters**

- **x** (*int*) – Probability mass at x.

**Returns** mass – The mass at x.

**Return type** float

**rvs**(*args, **kwargs*)

Sample from the RV.

**Returns** sample – A sample from the random variable.

**Return type** float

### 14.13 Parallel job execution on SGE like environments

The functions and classes in the pyabc.sge package can be used for at least two purposes:

1. The SGE.map method can be used together with the MappingSampler to parallelize ABC-SMC in a SGE/UGE infrastructure.

2. SGE.map can be used in a standalone mode to execute jobs on a SGE/UGE cluster. This is completely independent of ABC-SMC inference.

**class** pyabc.sge.SGE(*tmp_directory: str = None, memory: str = '3G', time_h: int = 100, python_executable_path: str = None, sge_error_file: str = None, sge_output_file: str = None, parallel_environment=None, name='map', queue=None, priority=None, num_threads: int = 1, execution_context=<class 'pyabc.sge.execution_contexts.DefaultContext'>, chunk_size=1)

**Bases:** object

Map a function to be executed on an SGE cluster environment. Reads a config file (if it exists) in your home directory which should look as the default in sge.config. The mapper reads commonly used parameters from a configuration file stored in ~/.parallel An example configuration file could look as follows:

```plaintext
#!/.parallel
[DIRECTORIES]
TMP=/tmp

[ BROKER]
# The value of TYPE can be SQLITE or REDIS
```

(continues on next page)
Parameters

- **`tmp_directory (str or None)`** – Directory where temporary job pickle files are stored. If set to None a tmp directory is read from the ‘~/.parallel’ configuration file. It this file does not exist a tmp directory within the user home directory is created.

- **`memory (str, optional (default = '3G'))`** – Ram requested by each job, e.g. ‘10G’.

- **`time_h (int (default = 100))`** – Job run time in hours.

- **`python_executable_path (str or None)`** – The python interpreter which executes the jobs. If set to None, the currently executing interpreter is used as returned by `sys.executable`.

- **`sge_error_file (str or None)`** – File to which stderr messages from workers are stored. If set to None, a file within the `tmp_directory` is used.

- **`sge_output_file (str or None)`** – File to which stdout messages from workers are stored. If set to None, a file within the `tmp_directory` is used.

- **`parallel_environment (str, optional (default = 'map'))`** – The SGE environment. (This is what is passed to the -pe option in the qsub script).

- **`name (str)`** – A name for the job.

- **`queue (str)`** – The SGE queue.

- **`priority (int, optional.)`** – SGE job priority. A value between -1000 and 0. Note that a priority of 0 automatically enables the reservation flag.

- **`num_threads (int, optional (default = 1))`** – Number of threads for each worker. This also sets the environment variable MKL_NUM_THREADS, OMP_NUM_THREADS to the specified number to handle jobs which use OpenMP etc. correctly.

- **`execution_context`** (DefaultContext, ProfilingContext, NamedPrinter) – Any context manager can be passed here. The `__enter__` method is called before evaluating the function on the cluster. The `__exit__` method directly after the function run finished.

- **`chunk_size (int, optional (default = 1))`** – nr of tasks executed within one job.

**Warning:** If `chunk_size` is larger than 1, this can have side effects as all the jobs within one chunk are executed within the python process.

Returns `sge` – The configured sge mapper.
**Return type**  
*SGE*

```python
def __init__(tmp_directory: str = None, memory: str = '3G', time_h: int = 100, python_executable_path: str = None, sge_error_file: str = None, sge_output_file: str = None, parallel_environment=None, name='map', queue=None, priority=None, num_threads: int = 1):
    Initialize self. See help(type(self)) for accurate signature.
```

**map** *(function, array)*  

Da what `map(function, array)` would do, but do it via a array job on the SGE by pickling objects, storing them in a temporary folder, submitting them to SGE and then reading and returning the results.

**Parameters**

- `function` *(Callable)* —
- `array` *(iterable)* —

**Returns** **result_list** – List of results of function application. This list can also contain `Exception` objects.

**Return type**  
`list`

```python
def sge_available() -> bool:
    Makes a simple heuristic test to check if the SGE is available on the machine. It tries to execute the `qstat` command. In case it is found, it is assumed that the SGE is available.

    **Returns** available – Whether SGE is available or not.

    **Return type**  
    bool
```

```python
def nr_cores_available() -> int:
    Determine the number of available cores in a manner which is safer for cluster environments than counting the number of CPUs the machine has. The CPU count might not be adequate as a job on a cluster might not have access to all the cores present on the cluster node on which it executes due to resource restrictions, such as for example done by SGE, SLURM etc.

    The following heuristic scheme is used to get the available number of cores:
    1. Tries to determine cores from the SGE environment variable `NSLOTS`
    2. From the environment variable `OMP_NUM_THREADS`
    3. From the environment variable `MKL_NUM_THREADS`
    4. from Python’s `os.cpu_count`

    **Returns** nr_cores – The number of cores available.

    **Return type**  
    int
```

```python
class DefaultContext(tmp_path, job_nr):
    Bases: pyabc.sge.execution_contexts.ExecutionContextMixin

    Does nothing special.
```

```python
class ProfilingContext(tmp_path, job_nr):
    Bases: pyabc.sge.execution_contexts.ExecutionContextMixin

    Profiles the running jobs and stores the profiles in the temporary job folder in the subdirectory “profiling”.

    Useful for debugging. Do not use in production.

    **__init__** *(tmp_path, job_nr)*
    Initialize self. See help(type(self)) for accurate signature.
```
class pyabc.sge.NamedPrinter(tmp_path, job_nr)
    Bases: pyabc.sge.execution_contexts.ExecutionContextMixin

    Context with appends the job name and number to anything printed by that job.

    __init__(tmp_path, job_nr)
        Initialize self. See help(type(self)) for accurate signature.

class pyabc.sge.SGE(tmp_directory: str = None, memory: str = '3G', time_h: int = 100, python_executable_path: str = None, sge_error_file: str = None, sge_output_file: str = None, parallel_environment=None, name='map', queue=None, priority=None, num_threads: int = 1, execution_context=<class 'pyabc.sge.execution_contexts.DefaultContext'>, chunk_size=1)

    Map a function to be executed on an SGE cluster environment. Reads a config file (if it exists) in you home directory which should look as the default in sge.config. An example configuration file could look as follows:

    ```
    #~/.parallel
    [DIRECTORIES]
    TMP=/tmp
    [BROKER]
    # The value of TYPE can be SQLITE or REDIS
    TYPE=REDIS
    [SGE]
    QUEUE=p.openmp
    PARALLEL_ENVIRONMENT=openmp
    PRIORITY=-500
    [REDIS]
    HOST=127.0.0.1
    ```

    Parameters

    • **tmp_directory** *(str or None)* – Directory where temporary job pickle files are stored. If set to None a tmp directory is read from the “~/.parallel” configuration file. It this file does not exist a tmp directory within the user home directory is created.

    • **memory** *(str, optional (default = '3G'))* – Ram requested by each job, e.g. ‘10G’.

    • **time_h** *(int (default = 100))* – Job run time in hours.

    • **python_executable_path** *(str or None)* – The python interpreter which executes the jobs. If set to None, the currently executing interpreter is used as returned by sys.executable.

    • **sge_error_file** *(str or None)* – File to which stderr messages from workers are stored. If set to None, a file within the tmp_directory is used.

    • **sge_output_file** *(str or None)* – File to which stdout messages from workers are stored. If set to None, a file within the tmp_directory is used.

    • **parallel_environment** *(str, optional (default = 'map'))* – The SGE environment. (This is what is passed to the -pe option in the qsub script).

    • **name** *(str)* – A name for the job.

    • **queue** *(str)* – The SGE queue.
• **priority** (*int, optional*) – SGE job priority. A value between -1000 and 0. Note that a priority of 0 automatically enables the reservation flag.

• **num_threads** (*int, optional (default = 1)*) – Number of threads for each worker. This also sets the environment variable MKL_NUM_THREADS, OMP_NUM_THREADS to the specified number to handle jobs which use OpenMP etc. correctly.

• **execution_context** (DefaultValue, ProfilingContext, NamedPrinter) – Any context manager can be passed here. The __enter__ method is called before evaluating the function on the cluster. The __exit__ method directly after the function run finished.

• **chunk_size** (*int, optional (default = 1)*) – Number of tasks executed within one job.

**Warning:** If chunk_size is larger than 1, this can have side effects as all the jobs within one chunk are executed within the python process.

Returns sge – The configured sge mapper.

Return type SGE

map (*function, array*)

Do what map(function, array) would do, but do it via a array job on the SGE by pickling objects, storing them in a temporary folder, submitting them to SGE and then reading and returning the results.

Parameters

• **function** (*Callable*) –

• **array** (*iterable*) –

Returns result_list – List of results of function application. This list can also contain Exception objects.

Return type list

pyabc.sge.sge_available()

Makes a simple heuristic test to check if the SGE is available on the machine. It tries to execute the qstat command. In case it is found, it is assumed that the SGE is available.

Returns available – Whether SGE is available or not.

Return type bool

pyabc.sge.nr_cores_available() \rightarrow \text{int}

Determine the number of available cores in a manner which is safer for cluster environments than counting the number of CPUs the machine has. The CPU count might not be adequate as a job on a cluster might not have access to all the cores present on the cluster node on which it executes due to resource restrictions, such as for example done by SGE, SLURM etc.

The following heuristic scheme is used to get the available number of cores:

1. Tries to determine cores form the SGE environment variable NSLOTS
2. From the environment variable OMP_NUM_THREADS
3. From the environment variable MKL_NUM_THREADS
4. from Python’s os.cpu_count

Chapter 14. API reference
Returns **nr_cores** – The number of cores available.

**Return type** int

class pyabc.sge.NamedPrinter(tmp_path, job_nr)

Context with appends the job name and number to anything printed by that job.

class pyabc.sge.DefaultContext(tmp_path, job_nr)

Does nothing special.

class pyabc.sge.ProfilingContext(tmp_path, job_nr)

Profiles the running jobs and stores the profiles in the temporary job folder in the subdirectory “profiling”.

Useful for debugging. Do not use in production.

### 14.14 Visualizations

Helper functions to visualize results of ABCSMC runs.

pyabc.visualization.plot_kde_1d(df, w, x, xmin=None, xmax=None, numx=50, ax=None, title: str = None, refval=None, kde=None, **kwargs)

Plots a 1d histogram.

**Parameters**

- **df (Pandas Dataframe)** – The rows are the observations, the columns the variables
- **w (The corresponding weights)** –
- **x (str)** – The variable for the x-axis
- **xmin (float, optional)** – The lower limit in x for the histogram. If left empty, it is set to the minimum of the observations of the variable to be plotted as x.
- **xmax (float, optional)** – The upper limit in x for the histogram. If left empty, it is set to the maximum of the observations of the variable to be plotted as x.
- **numx (int, optional)** – The number of bins in x direction. Defaults to 50.
- **title (str, optional)** – Title for the plot. Defaults to None.
- **refval (dict, optional)** – A reference value for x (as refval[x]: float). If not None, the value will be highlighted in the plot. Default: None.
- **kde (pyabc.Transition, optional)** – The kernel density estimator to use for creating a smooth density from the sample. If None, a multivariate normal kde with cross-validated scaling is used.

**Returns** **ax** – axis of the plot

**Return type** matplotlib axis

pyabc.visualization.plot_kde_2d(df, w, x, y, xmin=None, xmax=None, numx=50, numy=50, ax=None, colorbar=True, title: str = None, refval=None, kde=None, **kwargs)

Plots a 2d histogram.

**Parameters**

- **df (Pandas Dataframe)** – The rows are the observations, the columns the variables
- **w (The corresponding weights.)** –
- **x (str)** – The variable for the x-axis.
• **y** (*str*) – The variable for the y-axis.
• **xmin** (*float, optional*) – The lower limit in x for the histogram. If left empty, it is set to the minimum of the observations of the variable to be plotted as x.
• **xmax** (*float, optional*) – The upper limit in x for the histogram. If left empty, it is set to the maximum of the observations of the variable to be plotted as x.
• **ymin** (*float, optional*) – The lower limit in y for the histogram. If left empty, it is set to the minimum of the observations of the variable to be plotted as y.
• **ymax** (*float, optional*) – The upper limit in y for the histogram. If left empty, it is set to the maximum of the observations of the variable to be plotted as y.
• **numx** (*int, optional*) – The number of bins in x direction. Defaults to 50.
• **numy** (*int, optional*) – The number of bins in y direction. Defaults to 50.
• **colorbar** (*bool, optional*) – Whether to plot a colorbar. Defaults to True.
• **title** (*str, optional*) – Title for the plot. Defaults to None.
• **refval** (*dict, optional*) – A reference parameter to be shown in the plots. Default: None.
• **kde** (*pyabc.Transition, optional*) – The kernel density estimator to use for creating a smooth density from the sample. If None, a multivariate normal kde with cross-validated scaling is used.

**Returns ax** – axis of the plot

**Return type** matplotlib axis

```python
pyabc.visualization.plot_kde_matrix(df, w, limits=None, colorbar=True, height=2.5, numx=50, numy=50, refval=None, kde=None)
```

Plot a KDE matrix.

**Parameters**

• **df** (*Pandas Dataframe*) – The rows are the observations, the columns the variables.
• **w** (*np.ndarray*) – The corresponding weights.
• **colorbar** (*bool*) – Whether to plot the colorbars or not.
• **limits** (*dictionary, optional*) – Dictionary of the form (**"name":** (lower_limit, upper_limit)).
• **height** (*float, optional*) – Height of each subplot in inches. Default: 2.5.
• **numx** (*int, optional*) – The number of bins in x direction. Defaults to 50.
• **numy** (*int, optional*) – The number of bins in y direction. Defaults to 50.
• **refval** (*dict, optional*) – A reference parameter to be shown in the plots (e.g. the underlying ground truth parameter used to simulate the data for testing purposes). Default: None.
• **kde** (*pyabc.Transition, optional*) – The kernel density estimator to use for creating a smooth density from the sample. If None, a multivariate normal kde with cross-validated scaling is used.
pyabc.visualization.plot_sample_numbers

Plot required numbers of samples over all iterations.

Parameters

- **histories** (Union[List, History]) – The histories to plot from. History ids must be set correctly.
- **labels** (Union[List, str], optional) – Labels corresponding to the histories. If None are provided, indices are used as labels.
- **rotation** (int, optional (default = 0)) – Rotation to apply to the plot’s x tick labels. For longer labels, a tilting of 45 or even 90 can be preferable.
- **title** (str, optional (default = "Total required samples")) – Title for the plot.
- **size** (tuple of float, optional) – The size of the plot in inches.

Returns ax

Return type Axis of the generated plot.

pyabc.visualization.plot_epsilons

Plot epsilon trajectory.

Parameters

- **histories** (Union[List, History]) – The histories to plot from. History ids must be set correctly.
- **labels** (Union[List, str], optional) – Labels corresponding to the histories. If None are provided, indices are used as labels.
- **scale** (str, optional (default='lin')) – Scaling to apply to the y axis. Must be one of ‘lin’, ‘log’, ‘log10’.
- **title** (str, optional (default = "Epsilon values")) – Title for the plot.
- **size** (tuple of float, optional) – The size of the plot in inches.

Returns ax

Return type Axis of the generated plot.

pyabc.visualization.plot_histogram_1d

Plot 1d histogram of parameter samples.

Parameters

- **history** (History) – History to extract data from.
- **x** (str) – Id of the parameter to plot for.
- **m** (int, optional (default = 0)) – Id of the model to plot for.
- **t** (int, optional (default = None, i.e. the last time)) – Time point to plot for.
- **xmax** \((x_{\text{min}},)\) – Bounds for x. Both must be specified for bounds to be applied.
- **ax** \((\text{matplotlib.axis.Axis})\) – Axis object for the plot. If None is passed, a new figure is created.

Returns **ax**

Return type Axis of the generated plot.

`pyabc.visualization.plot_histogram_1d_lowlevel` \((df: \text{pandas.core.frame.DataFrame}, w: \text{pandas.core.frame.DataFrame}, x: \text{str}, x_{\text{min}}=\text{None}, x_{\text{max}}=\text{None}, ax=\text{None}, **\text{kwargs})\)

Lowlevel interface for plot_histogram_1d (see there for the remaining parameters).

Parameters

- **df** \((\text{pd.DataFrame})\) – Contains the parameters. Must have a column ‘x’.
- **w** \((\text{pd.DataFrame})\) – Parameter weights.

`pyabc.visualization.plot_histogram_2d` \((\text{history: pyabc.storage.history.History}, x: \text{str}, y: \text{str}, m: \text{int} = 0, t: \text{int} = \text{None}, x_{\text{min}}=\text{None}, x_{\text{max}}=\text{None}, y_{\text{min}}=\text{None}, y_{\text{max}}=\text{None}, ax=\text{None}, **\text{kwargs})\)

Plot 2d histogram of parameter pair samples.

Parameters

- **history** \((\text{History})\) – History to extract data from.
- **y** \((x, )\) – Ids of the parameters to plot for.
- **m** \((\text{int, optional (default = 0)})\) – Id of the model to plot for.
- **t** \((\text{int, optional (default = None, i.e. the last time)})\) – Time point to plot for.
- **xmax**, **ymin**, **ymax** \((x_{\text{min}},)\) – Bounds for x and y. All must be specified for bounds to be applied.
- **ax** \((\text{matplotlib.axis.Axis})\) – Axis object for the plot. If None is passed, a new figure is created.

Returns **ax**

Return type Axis of the generated plot.

`pyabc.visualization.plot_histogram_2d_lowlevel` \((df: \text{pandas.core.frame.DataFrame}, w: \text{pandas.core.frame.DataFrame}, x, y, x_{\text{min}}=\text{None}, x_{\text{max}}=\text{None}, y_{\text{min}}=\text{None}, y_{\text{max}}=\text{None}, ax=\text{None}, **\text{kwargs})\)

Lowlevel interface for plot_histogram_2d (see there for the remaining parameters).

Parameters

- **df** \((\text{pd.DataFrame})\) – Contains the parameters. Must have a column ‘x’.
- **w** \((\text{pd.DataFrame})\) – Parameter weights.

`pyabc.visualization.plot_histogram_matrix` \((\text{history: pyabc.storage.history.History}, m: \text{int} = 0, t: \text{int} = \text{None}, **\text{kwargs})\)

Plot matrix of 1d and 2d histograms over all parameters.

Parameters

- **history** \((\text{History})\) – History to extract data from.
• m (int, optional (default = 0)) – Id of the model to plot for.
• t (int, optional (default = None, i.e. the last time)) – Time point to plot for.

Returns arr_ax – Axis objects of the generated plots.

Return type list of matplotlib.axis.Axis

pyabc.visualization.plot_histogram_matrix_lowlevel(df: pandas.core.frame.DataFrame, w: pandas.core.frame.DataFrame, **kwargs)

Lowlevel interface for plot_histogram_matrix (see there for the remaining parameters).

Parameters
• df (pd.DataFrame) – Contains the parameters. Must have a column ‘x’.
• w (pd.DataFrame) – Parameter weights.

pyabc.visualization.plot_confidence_intervals(history: pyabc.storage.history.History, m: int = 0, par_names: List = None, confidences: List = None, show_mean: bool = False, size: tuple = None)

Plot confidence intervals over time.

Parameters
• history (History) – The history to extract data from.
• m (int, optional (default = 0)) – The id of the model to plot for.
• par_names (List of str, optional) – The parameter to plot for. If None, then all parameters are used.
• show_mean (bool, optional (default = False)) – Whether to show the mean apart from the median as well.
• size (tuple of float) – Size of the plot.

pyabc.visualization.plot_model_probabilities(history: pyabc.storage.history.History, rotation: int = 0, title: str = 'Model probabilities')

Plot the probabilities of models over time.

Parameters
• history (History) – The history to extract data from.
• rotation (int, optional (default = 0)) – Rotation of x axis labels.
• title (str, optional) – Title of the plot.

pyabc.visualization.plot_effective_sample_sizes(histories: Union[List, pyabc.storage.history.History], labels: Union[List, str] = None, rotation: int = 0, title: str = 'Effective sample size', size: tuple = None)

Plot effective sample sizes over all iterations.

Parameters
• histories (Union[List, History]) – The histories to plot from. History ids must be set correctly.
• **labels** (*Union[List ,str], optional*) – Labels corresponding to the histories. If None are provided, indices are used as labels.

• **rotation** (*int, optional (default = 0)*) – Rotation to apply to the plot’s x tick labels. For longer labels, a tilting of 45 or even 90 can be preferable.

• **title** (*str, optional (default = "Total required samples")*) – Title for the plot.

• **size** (*tuple of float, optional*) – The size of the plot in inches.

Returns **ax**

Return type Axis of the generated plot.

### 14.15 Weighted Statistics

Functions performing statistical operations on weighted points generated via importance sampling.

pyabc.weighted_statistics.**effective_sample_size**(weights)

Compute the effective sample size of weighted points sampled via importance sampling according to the formula

\[ n_{eff} = \frac{\left(\sum_{i=1}^{n} w_i\right)^2}{\sum_{i=1}^{n} w_i^2} \]

pyabc.weighted_statistics.**weight_checked**(function)

Function decorator to check normalization of weights.

pyabc.weighted_statistics.**weighted_mean**(points, weights)

Compute the weighted mean.

pyabc.weighted_statistics.**weighted_median**(points, weights)

Compute the weighted median (i.e. 0.5 quantile).

pyabc.weighted_statistics.**weighted_quantile**(points, weights=None, alpha=0.5)

Compute the weighted alpha-quantile. E.g. alpha = 0.5 -> median.

pyabc.weighted_statistics.**weighted_std**(points, weights)

Compute the weighted standard deviation from the weighted mean.
CHAPTER
FIFTEEN

INDICES AND TABLES

• genindex
• modindex
• search
p
pyabc.acceptor, 104
pyabc.distance, 94
pyabc.epsilon, 110
pyabc.model, 106
pyabc.parameters, 130
pyabc.population, 130
pyabc.populationstrategy, 124
pyabc.random_variables, 132
pyabc.sampler, 126
pyabc.sge, 138
pyabc.smc, 91
pyabc.storage, 113
pyabc.transition, 118
pyabc.visualization, 143
pyabc.weighted_statistics, 148
Symbols

__call__() (pyabc.acceptor.Acceptor method), 104
__call__() (pyabc.acceptor.SimpleFunctionAcceptor method), 105
__call__() (pyabc.distance.AcceptAllDistance method), 97
__call__() (pyabc.distance.Distance method), 94
__call__() (pyabc.distance.IdentityFakeDistance method), 96
__call__() (pyabc.distance.NoDistance method), 96
__call__() (pyabc.distance.PCADistance method), 100
__call__() (pyabc.distance.PNormDistance method), 98
__call__() (pyabc.distance.RangeEstimatorDistance method), 102
__call__() (pyabc.distance.SimpleFunctionDistance method), 97
__call__() (pyabc.distance.ZScoreDistance method), 100
__call__() (pyabc.epsilon.ConstantEpsilon method), 110
__call__() (pyabc.epsilon.Epsilon method), 110
__call__() (pyabc.epsilon.ListEpsilon method), 111
__call__() (pyabc.epsilon.QuantileEpsilon method), 112
__call__() (pyabc.epsilon.MedianEpsilon method), 112
__call__() (pyabc.model.Model method), 108
__call__() (pyabc.model.ModelResult method), 109
__call__() (pyabc.model.SimpleModel method), 109
__call__() (pyabc.parameters.ParameterStructure method), 130
__call__() (pyabc.pipeline.Particle method), 131
__call__() (pyabc.pipeline.Population method), 131
__call__() (pyabc.pipeline.strategy.AdaptivePopulationSize method), 124
__call__() (pyabc.pipeline.strategy.PopulationStrategy method), 125
__call__() (pyabc.random_variables.LowerBoundDecorator method), 133
__call__() (pyabc.random_variables.ModelPerturbationKernel method), 135
__call__() (pyabc.random_variables.RV method), 135
__call__() (pyabc.random_variables.RVDecorator method), 137
__call__() (pyabc.sampler.ConcurrentFutureSampler method), 130
__call__() (pyabc.sampler.DaskDistributedSampler method), 128
__call__() (pyabc.sampler.MappingSampler method), 128
__call__() (pyabc.sampler.RedisEvalParallelSampler method), 129
__call__() (pyabc.sampler.RedisEvalParallelSamplerServerStarter method), 129
__call__() (pyabc.sampler.Sample method), 126
__call__() (pyabc.sampler.Sampler method), 127
__call__() (pyabc.sge.NamedPrinter method), 141
A

cdf() (pyabc.random_variables.RV method), 135
cdf() (pyabc.random_variables.RVBase method), 136
cdf() (pyabc.random_variables.RVDecorator method), 137
accept() (pyabc.model.IntegratedModel method), 106
accept() (pyabc.model.Model method), 108
accept_use_complete_history() (in module pyabc.acceptor), 106
accept_use_current_time() (in module pyabc.acceptor), 106
AdaptivePopulationSize (class in pyabc.populationstrategy), 124
AdaptivePNormDistance (class in pyabc.distance), 99
AdaptivePopulationSize (class in pyabc.populationstrategy), 124
alive_models() (pyabc.storage.History method), 114
all_runs() (pyabc.storage.History method), 114
calls() (pyabc.storage.History method), 114
append() (pyabc.sampler.Sample method), 126
append_population() (pyabc.transition.LocalTransition method), 122
call() (pyabc.transition.LocalTransition method), 122
assert_acceptor() (pyabc.acceptor.SimpleFunctionAcceptor static method), 106
assert_model() (pyabc.model.SimpleModel static method), 109

B

bias() (in module pyabc.distance), 104
Distribution (class in pyabc.random_variables), 132

distribution (pyabc.random_variables.RV attribute), 135
done() (pyabc.storage.History method), 115

effective_sample_size() (in module pyabc.weighted_statistics), 148
EPS (pyabc.transition.LocalTransition attribute), 122
Epsilon (class in pyabc.epsilon), 110

fit() (pyabc.transition.DiscreteRandomWalkTransition method), 123
fit() (pyabc.transition.DiscreteTransition method), 120
fit() (pyabc.transition.GridSearchCV method), 121
fit() (pyabc.transition.LocalTransition method), 122
fit() (pyabc.transition.MultivariateNormalTransition method), 121
fit() (pyabc.transition.Transition method), 119
from_dictionary() (pyabc.random_variables.RV class method), 135
from_dictionary_of_dictionaries() (pyabc.random_variables.Distribution class method), 133

get_accepted_population() (pyabc.sampler.Sample method), 126
get_accepted_sum_stats() (pyabc.population.Population method), 131
get_all_populations() (pyabc.storage.History method), 115
get_config() (pyabc.distance.AdaptivePNormDistance method), 100
get_config() (pyabc.distance.Distance method), 95
get_config() (pyabc.distance.DistanceWithMeasureList method), 103
get_config() (pyabc.distance.PercentileDistance method), 102
get_config() (pyabc.distance.PNormDistance method), 99
get_config() (pyabc.distance.RangeEstimatorDistance method), 103
get_config() (pyabc.distance.SimpleFunctionDistance method), 98
get_config() (pyabc.epsilon.ConstantEpsilon method), 110
get_config() (pyabc.epsilon.Epsilon method), 111
get_config() (pyabc.epsilon.ListEpsilon method), 111
get_config() (pyabc.epsilon.QuantileEpsilon method), 112
get_config() (pyabc.populationstrategy.AdaptivePopulationSize method), 124
get_config() (pyabc.populationstrategy.PopulationStrategy method), 125
get_distribution() (pyabc.storage.History method), 115
get_for_keys() (pyabc.population.Population method), 132
get_list() (pyabc.population.Population method), 132
get_model_probabilities() (pyabc.population.Population method), 132
get_model_probabilities() (pyabc.storage.History method), 116
get_nr_particles_per_population() (pyabc.storage.History method), 116
get_parameter_names() (pyabc.random_variables.Distribution class method), 133
get_population() (pyabc.storage.History method), 116
get_population_extended() (pyabc.storage.History method), 116
get_population_strategy() (pyabc.storage.History method), 116
get_weighted_distances() (pyabc.population.Population method), 132
get_weighted_distances() (pyabc.storage.History method), 116
get_weighted_sum_stats() (pyabc.storage.History method), 117
get_weighted_sum_stats_for_model() (pyabc.storage.History method), 117
GridSearchCV (class in pyabc.transition), 121

History (class in pyabc.storage), 114

id (pyabc.storage.History attribute), 114
IdentityFakeDistance (class in pyabc.distance), 96
initialize() (pyabc.distance.AdaptivePNormDistance method), 100
initialize() (pyabc.distance.Distance method), 95
initialize() (pyabc.distance.DistanceWithMeasureList method), 104
initialize() (pyabc.distance.PCADistance method), 101
initialize() (pyabc.distance.RangeEstimatorDistance method), 103
initialize() (pyabc.epsilon.Epsilon method), 111

Index 157
initialize() (pyabc.epsilon.QuantileEpsilon method), 112
integrated_simulate() (pyabc.model.IntegratedModel method), 107
IntegratedModel (class in pyabc.model), 106
L
ListEpsilon (class in pyabc.epsilon), 111
load() (pyabc.smc.ABCSMC method), 92
LocalTransition (class in pyabc.transition), 122
lower() (pyabc.distance.MinMaxDistance static method), 101
lower() (pyabc.distance.PercentileDistance static method), 102
lower() (pyabc.distance.RangeEstimatorDistance static method), 103
LowerBoundDecorator (class in pyabc.random_variables), 133
M
map() (pyabc.sge.SGE method), 140, 142
MappingSampler (class in pyabc.sampler), 127
max_t (pyabc.storage.History attribute), 117
mean_absolute_deviation() (in module pyabc.distance), 104
mean_absolute_deviation_to_observation() (in module pyabc.distance), 104
mean_cv() (pyabc.transition.Transition method), 119
median_absolute_deviation() (in module pyabc.distance), 104
median_absolute_deviation_to_observation() (in module pyabc.distance), 104
MedianEpsilon (class in pyabc.epsilon), 112
MinMaxDistance (class in pyabc.distance), 101
Model (class in pyabc.model), 107
model_names() (pyabc.storage.History method), 117
ModelPerturbationKernel (class in pyabc.random_variables), 134
ModelResult (class in pyabc.model), 109
MulticoreEvalParallelSampler (class in pyabc.sampler), 129
MulticoreParticleParallelSampler (class in pyabc.sampler), 127
MultivariateNormalTransition (class in pyabc.transition), 120
N
n_accepted (pyabc.sampler.Sample attribute), 126
n_populations (pyabc.storage.History attribute), 117
n_worker() (pyabc.sampler.RedisEvalParallelSampler method), 129
NamedPrinter (class in pyabc.sge), 141, 143
new() (pyabc.smc.ABCSMC method), 93
NoDistance (class in pyabc.distance), 96
NotEnoughParticles, 122
nr_cores_available() (in module pyabc.sge), 140, 142
nr_of_models_alive() (pyabc.storage.History method), 117
P
Parameter (class in pyabc.parameters), 130
ParameterStructure (class in pyabc.parameters), 130
Particle (class in pyabc.population), 131
PCADistance (class in pyabc.distance), 100
pdf() (pyabc.random_variables.Distribution method), 133
df() (pyabc.random_variables.LowerBoundDecorator method), 134
df() (pyabc.random_variables.RV method), 136
df() (pyabc.random_variables.RVBase method), 136
df() (pyabc.random_variables.RVDecorator method), 138
pdf() (pyabc.transition.DiscreteRandomWalkTransition method), 123
pdf() (pyabc.transition.DiscreteTransition method), 120
df() (pyabc.transition.LocalTransition method), 122
df() (pyabc.transition.MultivariateNormalTransition method), 121
df() (pyabc.transition.Transition method), 119
PERCENTILE (pyabc.distance.PercentileDistance attribute), 102
PercentileDistance (class in pyabc.distance), 101
plot_confidence_intervals() (in module pyabc.visualization), 147
plot_effective_sample_sizes() (in module pyabc.visualization), 147
plot_epsilons() (in module pyabc.visualization), 145
plot_histogram_1d() (in module pyabc.visualization), 145
plot_histogram_1d_lowlevel() (in module pyabc.visualization), 146
plot_histogram_2d() (in module pyabc.visualization), 146
plot_histogram_2d_lowlevel() (in module pyabc.visualization), 146
plot_histogram_matrix() (in module pyabc.visualization), 146
plot_histogram_matrix_lowlevel() (in module pyabc.visualization), 146
plot_kde_1d() (in module pyabc.visualization), 143
plot_kde_2d() (in module pyabc.visualization), 143
Index 159

plot_kde_matrix() (in module pyabc.visualization), 144
plot_model_probabilities() (in module pyabc.visualization), 147
plot_sample_numbers() (in module pyabc.visualization), 144
pmf() (pyabc.random_variables.LowerBoundDecorator method), 134
pmf() (pyabc.random_variables.ModelPerturbationKernel method), 135
pmf() (pyabc.random_variables.RV method), 136
pmf() (pyabc.random_variables.RVBase method), 137
pmf() (pyabc.random_variables.RVDecorator method), 138
PNormDistance (class in pyabc.distance), 98
Population (class in pyabc.population), 131
PopulationStrategy (class in pyabc.populationstrategy), 125
ProfilingContext (class in pyabc.sge), 140, 143
pyabc.acceptor (module), 104
pyabc.distance (module), 94
pyabc.epsilon (module), 110
pyabc.model (module), 106
pyabc.parameters (module), 130
pyabc.population (module), 130
pyabc.populationstrategy (module), 124
pyabc.random_variables (module), 132
pyabc.sampler (module), 126
pyabc.sge (module), 138
pyabc.smc (module), 91
pyabc.storage (module), 113
pyabc.transition (module), 118
pyabc.visualization (module), 143
pyabc.weighted_statistics (module), 148
Q
QuantileEpsilon (class in pyabc.epsilon), 112
R
RangeEstimatorDistance (class in pyabc.distance), 102
RedisEvalParallelSampler (class in pyabc.sampler), 128
RedisEvalParallelSamplerServerStarter (class in pyabc.sampler), 129
root_mean_square_deviation() (in module pyabc.distance), 104
run() (pyabc.smc.ABCSMC method), 94
RV (class in pyabc.random_variables), 135
RVBase (class in pyabc.random_variables), 136
RVDecorator (class in pyabc.random_variables), 136
rvs() (pyabc.random_variables.Distribution method), 133
rs() (pyabc.random_variables.LowerBoundDecorator method), 134
rs() (pyabc.random_variables.ModelPerturbationKernel method), 135
rs() (pyabc.random_variables.RV method), 136
rs() (pyabc.random_variables.RVBase method), 137
rs() (pyabc.random_variables.RVDecorator method), 138
RangeEstimatorDistance (class in pyabc.distance), 98
Population (class in pyabc.population), 131
PopulationStrategy (class in pyabc.populationstrategy), 125
ProfilingContext (class in pyabc.sge), 140, 143
pyabc.acceptor (module), 104
pyabc.distance (module), 94
pyabc.epsilon (module), 110
pyabc.model (module), 106
pyabc.parameters (module), 130
pyabc.population (module), 130
pyabc.populationstrategy (module), 124
pyabc.random_variables (module), 132
pyabc.sampler (module), 126
pyabc.sge (module), 138
pyabc.smc (module), 91
pyabc.storage (module), 113
pyabc.transition (module), 118
pyabc.visualization (module), 143
pyabc.weighted_statistics (module), 148
Q
QuantileEpsilon (class in pyabc.epsilon), 112
R
RangeEstimatorDistance (class in pyabc.distance), 102
RedisEvalParallelSampler (class in pyabc.sampler), 128
RedisEvalParallelSamplerServerStarter (class in pyabc.sampler), 129
root_mean_square_deviation() (in module pyabc.distance), 104
run() (pyabc.smc.ABCSMC method), 94
RV (class in pyabc.random_variables), 135
RVBase (class in pyabc.random_variables), 136
RVDecorator (class in pyabc.random_variables), 136
rvs() (pyabc.random_variables.Distribution method), 133
rs() (pyabc.random_variables.LowerBoundDecorator method), 134
rs() (pyabc.random_variables.ModelPerturbationKernel method), 135
rs() (pyabc.random_variables.RV method), 136
rs() (pyabc.random_variables.RVBase method), 137
rs() (pyabc.random_variables.RVDecorator method), 138
RangeEstimatorDistance (class in pyabc.distance), 98
Population (class in pyabc.population), 131
PopulationStrategy (class in pyabc.populationstrategy), 125
ProfilingContext (class in pyabc.sge), 140, 143
pyabc.acceptor (module), 104
pyabc.distance (module), 94
pyabc.epsilon (module), 110
pyabc.model (module), 106
pyabc.parameters (module), 130
pyabc.population (module), 130
pyabc.populationstrategy (module), 124
pyabc.random_variables (module), 132
pyabc.sampler (module), 126
pyabc.sge (module), 138
pyabc.smc (module), 91
pyabc.storage (module), 113
pyabc.transition (module), 118
pyabc.visualization (module), 143
pyabc.weighted_statistics (module), 148
Q
QuantileEpsilon (class in pyabc.epsilon), 112
R
RangeEstimatorDistance (class in pyabc.distance), 102
RedisEvalParallelSampler (class in pyabc.sampler), 128
RedisEvalParallelSamplerServerStarter (class in pyabc.sampler), 129
root_mean_square_deviation() (in module pyabc.distance), 104
run() (pyabc.smc.ABCSMC method), 94
RV (class in pyabc.random_variables), 135
RVBase (class in pyabc.random_variables), 136
RVDecorator (class in pyabc.random_variables), 136
rvs() (pyabc.random_variables.Distribution method), 133
to_distance() (in module pyabc.distance), 98

to_json() (pyabc.distance.Distance method), 95

to_json() (pyabc.epsilon.Epsilon method), 111

to_json() (pyabc.populationstrategy.PopulationStrategy method), 126

total_nr_simulations (pyabc.storage.History attribute), 118

Transition (class in pyabc.transition), 118

U

update() (pyabc.distance.AdaptivePNormDistance method), 100

update() (pyabc.distance.Distance method), 95

update() (pyabc.epsilon.Epsilon method), 111

update() (pyabc.epsilon.QuantileEpsilon method), 113

update_distances() (pyabc.population.Population method), 132

update_nr_samples() (pyabc.storage.History method), 118

update_random_variables()
(pyabc.random_variables.Distribution method), 133

upper() (pyabc.distance.MinMaxDistance static method), 101

upper() (pyabc.distance.PercentileDistance static method), 102

upper() (pyabc.distance.RangeEstimatorDistance static method), 103

W

weight_checked() (in module pyabc.weighted_statistics), 148

weighted_mean() (in module pyabc.weighted_statistics), 148

weighted_median() (in module pyabc.weighted_statistics), 148

weighted_quantile() (in module pyabc.weighted_statistics), 148

weighted_std() (in module pyabc.weighted_statistics), 148

Z

ZScoreDistance (class in pyabc.distance), 100