ELFI

Release 0.8.7

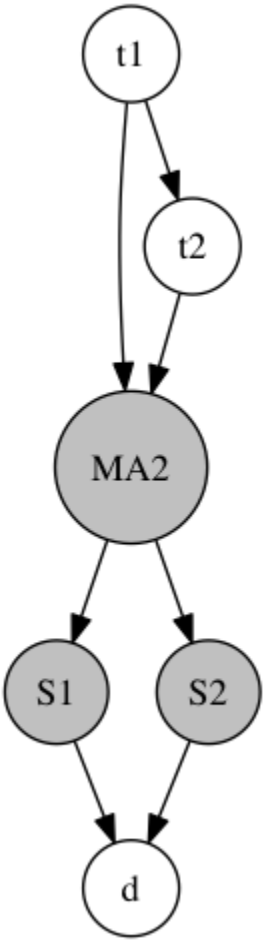
ELFI authors

May 07, 2025

GETTING STARTED

ELFI is a statistical software package for likelihood-free inference (LFI) such as Approximate Bayesian Computation (ABC). The term LFI refers to a family of inference methods that replace the use of the likelihood function with a data generating simulator function. Other names or related approaches to LFI include simulator-based inference, approximate Bayesian inference, indirect inference, etc.

ELFI features an easy to use syntax and supports parallelized inference out of the box.



See *the quickstart* to get started.

ELFI is licensed under BSD3. The source is in GitHub.

CHAPTER

ONE

CURRENTLY IMPLEMENTED LFI METHODS:

- ABC rejection sampler
- Sequential Monte Carlo ABC sampler
- ABC-SMC sampler with adaptive distance
- ABC-SMC sampler with adaptive threshold selection
- Bayesian Optimization for Likelihood-Free Inference (BOLFI) framework
- Robust Optimization Monte Carlo (ROMC) framework
- Bayesian Optimization for Likelihood-Free Inference by Ratio Estimation (BOLFIRE)
- Bayesian Synthetic Likelihood (BSL)

ELFI also has the following non LFI methods:

- Bayesian Optimization
- No-U-Turn-Sampler, a Hamiltonian Monte Carlo MCMC sampler

Additionally, ELFI integrates tools for visualization, model comparison, diagnostics and post-processing.

1.1 Installation

ELFI requires Python 3.9 or greater (see below how to install). To install ELFI, simply type in your terminal:

```
pip install elfi
```

In some OS you may have to first install `numpy` with `pip install numpy`. If you don't have pip installed, this Python installation guide can guide you through the process.

1.1.1 Installing Python 3.9

If you are new to Python, perhaps the simplest way to install it is with Anaconda that manages different Python versions. After installing Anaconda, you can create a Python 3.9. environment with ELFI:

```
conda create -n elfi python=3.9 numpy
source activate elfi
pip install elfi
```

1.1.2 Optional dependencies

We recommend to install:

- `graphviz` for drawing graphical models (`pip install graphviz` requires Graphviz binaries).

1.1.3 Potential problems with installation

ELFI depends on several other Python packages, which have their own dependencies. Resolving these may sometimes go wrong:

- If you receive an error about missing `numpy`, please install it first.
- If you receive an error about *yaml.load*, install `pyyaml`.
- On OS X with Anaconda virtual environment say *conda install python.app* and then use *pythonw* instead of *python*.
- Note that ELFI requires Python 3.9 or greater
- In some environments `pip` refers to Python 2.x, and you have to use `pip3` to use the Python 3.x version
- Make sure your Python installation meets the versions listed in requirements.

1.1.4 Developer installation from sources

The sources for ELFI can be downloaded from the Github repo.

You can either clone the public repository:

```
git clone https://github.com/elfi-dev/elfi.git
```

Or download the development tarball:

```
curl -OL https://github.com/elfi-dev/elfi/tarball/dev
```

Note that for development it is recommended to base your work on the *dev* branch instead of *master*.

Once you have a copy of the source, go to the folder and type:

```
pip install -e .
```

This will install ELFI along with its default requirements. Note that the dot in the end means the current folder.

1.1.5 Docker container

A simple Dockerfile with Jupyter support is also provided. This is especially suitable for running tests. Please see Docker documentation for details.

```
git clone --depth 1 https://github.com/elfi-dev/elfi.git
cd elfi
make docker-build  # builds the image with requirements for dev
make docker  # runs a container with live elfi directory
```

To open a Jupyter notebook, run

```
jupyter notebook --ip 0.0.0.0 --no-browser --allow-root
```

within the container and then on host open the page localhost:8888.

1.2 Good to know

Here we describe some important concepts related to ELFI. These will help in understanding how to implement custom operations (such as simulators or summaries) and can potentially save the user from some pitfalls.

1.2.1 ELFI model

In ELFI, the priors, simulators, summaries, distances, etc. are called operations. ELFI provides a convenient syntax of combining these operations into a network that is called an ElfiModel, where each node represents an operation. Basically, the ElfiModel is a description of how different quantities needed in the inference are to be generated. The structure of the network is a directed acyclic graph (DAG).

1.2.2 Operations

Operations are functions (or more generally Python callables) in the nodes of the ELFI model. Those nodes that deal directly with data, e.g. priors, simulators, summaries and distances should return a numpy array of length `batch_size` that contains their output.

If your operation does not produce data wrapped to numpy arrays, you can use the elfi.tools.vectorize tool to achieve that. Note that sometimes it is required to specify which arguments to the vectorized function will be constants and at other times also specify the datatype (when automatic numpy array conversion does not produce desired result). It is always good to check that the output is sane using the `node.generate` method.

1.2.3 Reusing data

The OutputPool object can be used to store the outputs of any node in the graph. Note however that changing a node in the model will change the outputs of it's child nodes. In Rejection sampling you can alter the child nodes of the nodes in the OutputPool and safely reuse the OutputPool with the modified model. This is especially handy when saving the simulations and trying out different summaries. BOLFI allows you to use the stored data as initialization data.

However passing a modified model with the OutputPool of the original model will produce biased results in other algorithms besides Rejection sampling. This is because more advanced algorithms learn from previous results. If the results change in some way, so will also the following parameter values and thus also their simulations and other nodes that depend on them. The Rejection sampling does not suffer from this because it always samples new parameter values directly from the priors, and therefore modified distance outputs have no effect to the parameter values of any later simulations.

1.3 Quickstart

First ensure you have installed Python 3.9 (or greater) and ELFI. After installation you can start using ELFI:

```
import elfi
```

ELFI includes an easy to use generative modeling syntax, where the generative model is specified as a directed acyclic graph (DAG). Let's create two prior nodes:

```
mu = elfi.Prior('uniform', -2, 4)
sigma = elfi.Prior('uniform', 1, 4)
```

The above would create two prior nodes, a uniform distribution from -2 to 2 for the mean `mu` and another uniform distribution from 1 to 5 for the standard deviation `sigma`. All distributions from `scipy.stats` are available.

For likelihood-free models we typically need to define a simulator and summary statistics for the data. As an example, lets define the simulator as 30 draws from a Gaussian distribution with a given mean and standard deviation. Let's use mean and variance as our summaries:

```
import scipy.stats as ss
import numpy as np

def simulator(mu, sigma, batch_size=1, random_state=None):
    mu, sigma = np.atleast_1d(mu, sigma)
    return ss.norm.rvs(mu[:, None], sigma[:, None], size=(batch_size, 30), random_
↪state=random_state)

def mean(y):
    return np.mean(y, axis=1)

def var(y):
    return np.var(y, axis=1)
```

Let's now assume we have some observed data `y0` (here we just create some with the simulator):

```
# Set the generating parameters that we will try to infer
mean0 = 1
std0 = 3

# Generate some data (using a fixed seed here)
np.random.seed(20170525)
y0 = simulator(mean0, std0)
print(y0)
```

```
[[ 3.7990926   1.49411834  0.90999905  2.46088006 -0.10696721  0.80490023
   0.7413415  -5.07258261  0.89397268  3.55462229  0.45888389 -3.31930036
  -0.55378741  3.00865492  1.59394854 -3.37065996  5.03883749 -2.73279084
   6.10128027  5.09388631  1.90079255 -1.7161259   3.86821266  0.4963219
   1.64594033 -2.51620566 -0.83601666  2.68225112  2.75598375 -6.02538356]]
```

Now we have all the components needed. Let's complete our model by adding the simulator, the observed data, summaries and a distance to our model:

```
# Add the simulator node and observed data to the model
sim = elfi.Simulator(simulator, mu, sigma, observed=y0)

# Add summary statistics to the model
S1 = elfi.Summary(mean, sim)
S2 = elfi.Summary(var, sim)

# Specify distance as euclidean between summary vectors (S1, S2) from simulated and
# observed data
d = elfi.Distance('euclidean', S1, S2)
```

If you have `graphviz` installed to your system, you can also visualize the model:

```
# Plot the complete model (requires graphviz)
elfi.draw(d)
```

Note: The automatic naming of nodes may not work in all environments e.g. in interactive Python shells. You can alternatively provide a name argument for the nodes, e.g. `S1 = elfi.Summary(mean, sim, name='S1')`.

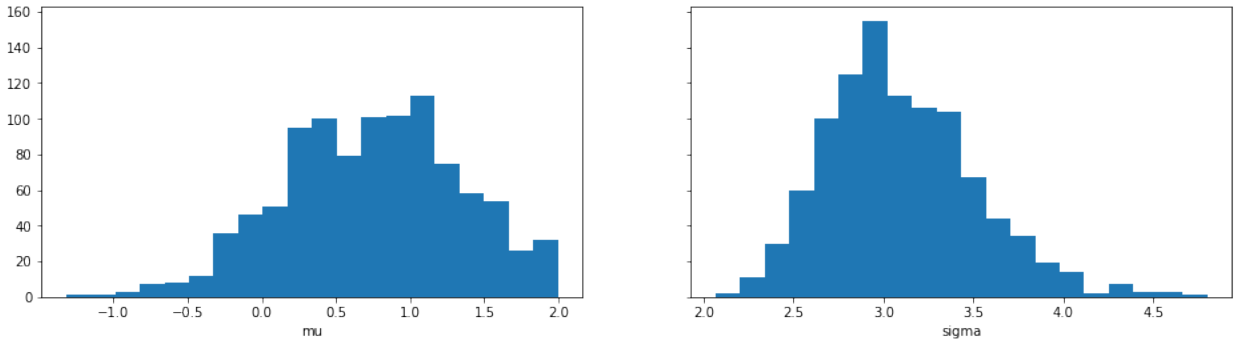
We can try to infer the true generating parameters `mean0` and `std0` above with any of ELFI's inference methods. Let's use ABC Rejection sampling and sample 1000 samples from the approximate posterior using threshold value 0.5:

```
rej = elfi.Rejection(d, batch_size=10000, seed=30052017)
res = rej.sample(1000, threshold=.5)
print(res)
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 120000
Threshold: 0.492
Sample means: mu: 0.748, sigma: 3.1
```

Let's plot also the marginal distributions for the parameters:

```
import matplotlib.pyplot as plt
res.plot_marginals()
plt.show()
```



1.4 API

This file describes the classes and methods available in ELFI.

1.4.1 Modelling API

Below is the API for creating generative models.

elfi.ElfiModel([name, observed, source_net])	A container for the inference model.

General model nodes

elfi.Constant(value, **kwargs)	A node holding a constant value.
elfi.Operation(fn, *parents, **kwargs)	A generic deterministic operation node.
elfi.RandomVariable(distribution, *params[, ...])	A node that draws values from a random distribution.

LFI nodes

elfi.Prior(distribution, *params[, size])	A parameter node of an ELFI graph.
elfi.Simulator(fn, *params, **kwargs)	A simulator node of an ELFI graph.
elfi.Summary(fn, *parents, **kwargs)	A summary node of an ELFI graph.
elfi.Discrepancy(discrepancy, *parents, **kwargs)	A discrepancy node of an ELFI graph.
elfi.Distance(distance, *summaries, **kwargs)	A convenience class for the discrepancy node.
elfi.AdaptiveDistance(*summaries, **kwargs)	Euclidean (2-norm) distance calculation with adaptive scale.

Other

elfi.new_model([name, set_default])	Create a new `ElfiModel` instance.
elfi.load_model(name[, prefix, set_default])	Load the pickled ElfiModel.
elfi.get_default_model()	Return the current default `ElfiModel` instance.
elfi.set_default_model([model])	Set the current default `ElfiModel` instance.

elfi.draw(G[, internal, param_names, ...])	Draw the *ElfiModel*.
elfi.plot_params_vs_node(node[, n_samples, ...])	Plot some realizations of parameters vs.

1.4.2 Inference API

Below is a list of inference methods included in ELFI.

elfi.Rejection(model[, discrepancy_name, ...])	Parallel ABC rejection sampler.
elfi.SMC(model[, discrepancy_name, output_names])	Sequential Monte Carlo ABC sampler.
elfi.AdaptiveDistanceSMC(model[, ...])	SMC-ABC sampler with adaptive threshold and distance function.
elfi.AdaptiveThresholdSMC(model[, ...])	ABC-SMC sampler with adaptive threshold selection.
elfi.BayesianOptimization(model[, ...])	Bayesian Optimization of an unknown target function.
elfi.BOLFI(model[, target_name, bounds, ...])	Bayesian Optimization for Likelihood-Free Inference (BOLFI).
elfi.ROMC(model[, bounds, discrepancy_name, ...])	Robust Optimisation Monte Carlo inference method.
elfi.BSL(model, n_sim_round[, ...])	Bayesian Synthetic Likelihood for parameter inference.

Result objects

OptimizationResult(x_min, **kwargs)	Base class for results from optimization.
Sample(method_name, outputs, parameter_names)	Sampling results from inference methods.
SmcSample(method_name, outputs, ...)	Container for results from SMC-ABC.
McmcSample(method_name, chains, ...)	Container for MCMC results.

Post-processing

elfi.adjust_posterior(sample, model, ...[, ...])	Adjust the posterior using local regression.

LinearAdjustment(**kwargs)	Regression adjustment using a local linear model.

Diagnostics

elfi.TwoStageSelection(simulator, fn_distance)	Perform the summary-statistics selection proposed by Nunes and Balding (2010).

Acquisition methods

LCBSC(*args[, delta])	Lower Confidence Bound Selection Criterion.
MaxVar(model, prior[, quantile_eps])	The maximum variance acquisition method.
RandMaxVar(model, prior[, quantile_eps, ...])	The randomised maximum variance acquisition method.
ExpIntVar(model, prior[, quantile_eps, ...])	The Expected Integrated Variance (ExpIntVar) acquisition method.
UniformAcquisition(model[, prior, n_inits, ...])	Acquisition from uniform distribution.

1.4.3 Other

Data pools

elfi.OutputPool([outputs, name, prefix])	Store node outputs to dictionary-like stores.
elfi.ArrayPool([outputs, name, prefix])	OutputPool that uses binary .npy files as default stores.

Module functions

elfi.get_client()	Get the current ELFI client instance.
elfi.set_client([client])	Set the current ELFI client instance.

Tools

elfi.tools.vectorize(operation[, constants, ...])	Vectorize an operation.
elfi.tools.external_operation(command[, ...])	Wrap an external command as a Python callable (function).

1.4.4 Class documentations

Modelling API classes

class elfi.**ElfiModel**(*name=None*, *observed=None*, *source_net=None*)

A container for the inference model.

The ElfiModel is a directed acyclic graph (DAG), whose nodes represent parts of the inference task, for example the parameters to be inferred, the simulator or a summary statistic.

Initialize the inference model.

Parameters

- **name** (*str, optional*) –
- **observed** (*dict, optional*) – Observed data with node names as keys.
- **source_net** (*nx.DiGraph, optional*) –
- **set_current** (*bool, optional*) – Sets this model as the current (default) ELFI model

copy()

Return a copy of the ElfiModel instance.

Return type
ElfiModel

generate(*batch_size=1*, *outputs=None*, *with_values=None*, *seed=None*)

Generate a batch of outputs.

This method is useful for testing that the ELFI graph works.

Parameters

- **batch_size** (*int, optional*) –
- **outputs** (*list, optional*) –

- **with_values** (*dict, optional*) – You can specify values for nodes to use when generating data
- **seed** (*int, optional*) – Defaults to global numpy seed.

get_reference(*name*)

Return a new reference object for a node in the model.

Parameters
name (*str*) –

get_state(*name*)

Return the state of the node.

Parameters
name (*str*) –

classmethod load(*name*, *prefix*)

Load the pickled ElfiModel.

Assumes there exists a file "name.pkl" in the current directory.

Parameters

- **name** (*str*) – Name of the model file to load (without the .pkl extension).
- **prefix** (*str*) – Path to directory where the model file is located, optional.

Return type
ElfiModel

property name

Return name of the model.

property observed

Return the observed data for the nodes in a dictionary.

property parameter_names

Return a list of model parameter names in an alphabetical order.

remove_node(*name*)

Remove a node from the graph.

Parameters
name (*str*) –

save(*prefix=None*)

Save the current model to pickled file.

Parameters
prefix (*str, optional*) – Path to the directory under which to save the model. Default is the current working directory.

update_node(*name*, *updating_name*)

Update *node* with *updating_node* in the model.

The node with name *name* gets the state (operation), parents and observed data (if applicable) of the updating_node. The updating node is then removed from the graph.

Parameters

- **name** (*str*) –

- **updating_name** (*str*) –

class elfi.**Constant**(*value*, ***kwargs*)

A node holding a constant value.

Initialize a node holding a constant value.

Parameters
value – The constant value of the node.

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

class elfi.**Operation**(*fn*, **parents*, ***kwargs*)

A generic deterministic operation node.

Initialize a node that performs an operation.

Parameters
fn (*callable*) – The operation of the node.

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

class elfi.**RandomVariable**(*distribution*, **params*, *size=None*, ***kwargs*)

A node that draws values from a random distribution.

Initialize a node that represents a random variable.

Parameters

- **distribution** (*str or scipy-like distribution object*) –
- **params** (*params of the distribution*) –
- **size** (*int, tuple or None, optional*) – Output size of a single random draw.

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

static compile_operation(*state*)

Compile a callable operation that samples the associated distribution.

Parameters
state (*dict*) –

property distribution

Return the distribution object.

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property size

Return the size of the output from the distribution.

property state

Return the state dictionary of the node.

class elfi.**Prior**(*distribution*, **params*, *size=None*, ***kwargs*)

A parameter node of an ELFI graph.

Initialize a Prior.

Parameters

- **distribution** (*str, object*) – Any distribution from *scipy.stats*, either as a string or an object. Objects must implement at least an *rvs* method with signature *rvs(*parameters, size, random_state)*. Can also be a custom distribution object that implements at least an *rvs* method. Many of the algorithms also require the *pdf* and *logpdf* methods to be available.
- **size** (*int, tuple or None, optional*) – Output size of a single random draw.
- **params** – Parameters of the prior distribution
- **kwargs** –

Notes

The parameters of the *scipy* distributions (typically *loc* and *scale*) must be given as positional arguments.

Many algorithms (e.g. SMC) also require a *pdf* method for the distribution. In general the definition of the distribution is a subset of *scipy.stats.rv_continuous*.

Scipy distributions: https://docs.scipy.org/doc/scipy-0.19.0/reference/stats.html

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

static compile_operation(*state*)

Compile a callable operation that samples the associated distribution.

Parameters
state (*dict*) –

property distribution

Return the distribution object.

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property size

Return the size of the output from the distribution.

property state

Return the state dictionary of the node.

class elfi.Simulator(*fn*, **params*, ***kwargs*)

A simulator node of an ELFI graph.

Simulator nodes are stochastic and may have observed data in the model.

Initialize a Simulator.

Parameters

- **fn** (*callable*) – Simulator function with a signature *sim(*params, batch_size, random_state)*
- **params** – Input parameters for the simulator.
- **kwargs** –

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

class elfi.Summary(*fn*, **parents*, ***kwargs*)

A summary node of an ELFI graph.

Summary nodes are deterministic operations associated with the observed data. if their parents hold observed data it will be automatically transformed.

Initialize a Summary.

Parameters

- **fn** (*callable*) – Summary function with a signature *summary(*parents)*
- **parents** – Input data for the summary function.
- **kwargs** –

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

class elfi.**Discrepancy**(*discrepancy*, **parents*, ***kwargs*)

A discrepancy node of an ELFI graph.

This class provides a convenience node for custom distance operations.

Initialize a Discrepancy.

Parameters

- **discrepancy** (*callable*) – Signature of the discrepancy function is of the form: *discrepancy(summary_1, summary_2, ..., observed)*, where summaries are arrays containing *batch_size* simulated values and observed is a tuple (observed_summary_1, observed_summary_2, ...). The callable object should return a vector of discrepancies between the simulated summaries and the observed summaries.

- ***parents** – Typically the summaries for the discrepancy function.
- ****kwargs** –

See also:

elfi.Distance
creating common distance discrepancies.

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

class elfi.**Distance**(*distance*, **summaries*, ***kwargs*)

A convenience class for the discrepancy node.

Initialize a distance node of an ELFI graph.

This class contains many common distance implementations through scipy.

Parameters

- **distance** (*str, callable*) – If string it must be a valid metric from *scipy.spatial.distance.cdist*.

  Is a callable, the signature must be *distance(X, Y)*, where X is a n x m array containing n simulated values (summaries) in rows and Y is a 1 x m array that contains the observed values (summaries). The callable should return a vector of distances between the simulated summaries and the observed summaries.

- ***summaries** – Summary nodes of the model.
- ****kwargs** – Additional parameters may be required depending on the chosen distance. See the scipy documentation. (The support is not exhaustive.) ELFI-related kwargs are passed on to elfi.Discrepancy.

Examples

```
>>> d = elfi.Distance('euclidean', summary1, summary2...)
```

```
>>> d = elfi.Distance('minkowski', summary, p=1)
```

Notes

Your summaries need to be scalars or vectors for this method to work. The summaries will be first stacked to a single 2D array with the simulated summaries in the rows for every simulation and the distance is taken row wise against the corresponding observed summary vector.

Scipy distances: https://docs.scipy.org/doc/scipy/reference/generated/generated/scipy.spatial.distance.cdist.html # noqa

See also:

elfi.Discrepancy
A general discrepancy node

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

class elfi.**AdaptiveDistance**(**summaries*, ***kwargs*)

Euclidean (2-norm) distance calculation with adaptive scale.

Summary statistics are normalised to vary on similar scales.

References

Prangle D (2017). Adapting the ABC Distance Function. Bayesian Analysis 12(1):289-309, 2017. https://projecteuclid.org/euclid.ba/1460641065

Initialize an AdaptiveDistance.

Parameters

- ***summaries** – Summary nodes of the model.
- ****kwargs** –

Notes

Your summaries need to be scalars or vectors for this method to work. The summaries will be first stacked to a single 2D array with the simulated summaries in the rows for every simulation and the distances are taken row wise against the corresponding observed summary vector.

add_data(**data*)

Add summaries data to update estimated standard deviation.

Parameters
***data** – Summary nodes output data.

Notes

Standard deviation is computed with Welford's online algorithm.

become(*other_node*)

Make this node become the *other_node*.

The children of this node will be preserved.

Parameters
other_node (*NodeReference*) –

generate(*batch_size=1*, *with_values=None*)

Generate output from this node.

Useful for testing.

Parameters

- **batch_size** (*int, optional*) –
- **with_values** (*dict, optional*) –

init_adaptation_round()

Initialise data stores to start a new adaptation round.

init_state()

Initialise adaptive distance state.

nested_distance(*u*, *v*)

Compute distance between simulated and observed summaries.

Parameters

- **u** (*ndarray*) – 2D array with M x (num summaries) observations
- **v** (*ndarray*) – 2D array with 1 x (num summaries) observations

Returns
2D array with M x (num distance functions) distances

Return type
ndarray

property parents

Get all positional parent nodes (inputs) of this node.

Returns
parents – List of positional parents

Return type
list

classmethod reference(*name*, *model*)

Construct a reference for an existing node in the model.

Parameters

- **name** (*string*) – name of the node
- **model** (ElfiModel) –

Return type
NodePointer instance

property state

Return the state dictionary of the node.

update_distance()

Update distance based on accumulated summaries data.

Other

elfi.**new_model**(*name=None*, *set_default=True*)

Create a new `ElfiModel` instance.

In addition to making a new ElfiModel instance, this method sets the new instance as the default for new nodes.

Parameters

- **name** (*str, optional*) –
- **set_default** (*bool, optional*) – Whether to set the newly created model as the current model.

elfi.**load_model**(*name*, *prefix=None*, *set_default=True*)

Load the pickled ElfiModel.

Assumes there exists a file "name.pkl" in the current directory. Also sets the loaded model as the default model for new nodes.

Parameters

- **name** (*str*) – Name of the model file to load (without the .pkl extension).
- **prefix** (*str*) – Path to directory where the model file is located, optional.
- **set_default** (*bool, optional*) – Set the loaded model as the default model. Default is True.

Return type
ElfiModel

elfi.**get_default_model**()

Return the current default `ElfiModel` instance.

New nodes will be added to this model by default.

elfi.**set_default_model**(*model=None*)

Set the current default `ElfiModel` instance.

New nodes will be placed the given model by default.

Parameters
model (*ElfiModel, optional*) – If None, creates a new `ElfiModel`.

elfi.**draw**(*G*, *internal=False*, *param_names=False*, *filename=None*, *format=None*)

Draw the *ElfiModel*.

Parameters

- **G** (*nx.DiGraph or ElfiModel*) – Graph or model to draw
- **internal** (*boolean, optional*) – Whether to draw internal nodes (starting with an underscore)
- **param_names** (*bool, optional*) – Show param names on edges
- **filename** (*str, optional*) – If given, save the dot file into the given filename.
- **format** (*str, optional*) – format of the file

Notes

Requires the optional 'graphviz' library.

Returns
A GraphViz dot representation of the model.

Return type
dot

elfi.**plot_params_vs_node**(*node*, *n_samples=100*, *func=None*, *seed=None*, *axes=None*, ***kwargs*)

Plot some realizations of parameters vs. *node*.

Useful e.g. for exploring how a summary statistic varies with parameters. Currently only nodes with scalar output are supported, though a function *func* can be given to reduce node output. This allows giving the simulator as the *node* and applying a summarizing function without incorporating it into the ELFI graph.

If *node* is one of the model parameters, its histogram is plotted.

Parameters

- **node** (*elfi.NodeReference*) – The node which to evaluate. Its output must be scalar (shape=(batch_size,1)).
- **n_samples** (*int, optional*) – How many samples to plot.
- **func** (*callable, optional*) – A function to apply to node output.
- **seed** (*int, optional*) –
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

Inference API classes

class elfi.**Rejection**(*model*, *discrepancy_name=None*, *output_names=None*, ***kwargs*)

Parallel ABC rejection sampler.

For a description of the rejection sampler and a general introduction to ABC, see e.g. Lintusaari et al. 2016.

References

Lintusaari J, Gutmann M U, Dutta R, Kaski S, Corander J (2016). Fundamentals and Recent Developments in Approximate Bayesian Computation. Systematic Biology. http://dx.doi.org/10.1093/sysbio/syw077.

Initialize the Rejection sampler.

Parameters

- **model** (*ElfiModel or NodeReference*) –
- **discrepancy_name** (*str, NodeReference, optional*) – Only needed if model is an ElfiModel
- **output_names** (*list, optional*) – Additional outputs from the model to be included in the inference result, e.g. corresponding summaries to the acquired samples

- **kwargs** – See ParameterInference

property batch_size

Return the current batch_size.

extract_result()

Extract the result from the current state.

Returns
result

Return type
Sample

property finished

Check whether objective of n_batches have been reached.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

iterate()

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

property parameter_names

Return the parameters to be inferred.

plot_state(***options*)

Plot the current state of the inference algorithm.

This feature is still experimental and only supports 1d or 2d cases.

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

ELFI calls this method before submitting a new batch with an increasing index *batch_index*. This is an optional method to override. Use this if you have a need do do preparations, e.g. in Bayesian optimization algorithm, the next acquisition points would be acquired here.

If you need provide values for certain nodes, you can do so by constructing a batch dictionary and returning it. See e.g. BayesianOptimization for an example.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

sample(*n_samples*, **args*, ***kwargs*)

Sample from the approximate posterior.

See the other arguments from the *set_objective* method.

Parameters

- **n_samples** (*int*) – Number of samples to generate from the (approximate) posterior
- ***args** –
- ****kwargs** –

Returns
result

Return type
Sample

property seed

Return the seed of the inference.

set_objective(*n_samples*, *threshold=None*, *quantile=None*, *n_sim=None*)

Set objective for inference.

Parameters

- **n_samples** (*int*) – number of samples to generate
- **threshold** (*float*) – Acceptance threshold
- **quantile** (*float*) – In between (0,1). Define the threshold as the p-quantile of all the simulations. n_sim = n_samples/quantile.
- **n_sim** (*int*) – Total number of simulations. The threshold will be the n_samples-th smallest discrepancy among n_sim simulations.

update(*batch*, *batch_index*)

Update the inference state with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values

- **batch_index** (*int*) –

class elfi.**SMC**(*model*, *discrepancy_name=None*, *output_names=None*, ***kwargs*)

Sequential Monte Carlo ABC sampler.

Initialize the SMC-ABC sampler.

Parameters

- **model** (*ElfiModel or NodeReference*) –
- **discrepancy_name** (*str, NodeReference, optional*) – Only needed if model is an ElfiModel
- **output_names** (*list, optional*) – Additional outputs from the model to be included in the inference result, e.g. corresponding summaries to the acquired samples
- **kwargs** – See ParameterInference

property batch_size

Return the current batch_size.

property current_population_threshold

Return the threshold for current population.

extract_result()

Extract the result from the current state.

Return type
SmcSample

property finished

Check whether objective of n_batches have been reached.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

iterate()

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

property parameter_names

Return the parameters to be inferred.

plot_state(***kwargs*)

Plot the current state of the algorithm.

Parameters

- **axes** (*matplotlib.axes.Axes (optional)*) –
- **figure** (*matplotlib.figure.Figure (optional)*) –
- **xlim** – x-axis limits
- **ylim** – y-axis limits
- **interactive** (*bool (default False)*) – If true, uses IPython.display to update the cell figure
- **close** – Close figure in the end of plotting. Used in the end of interactive mode.

Return type
None

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

sample(*n_samples*, **args*, ***kwargs*)

Sample from the approximate posterior.

See the other arguments from the *set_objective* method.

Parameters

- **n_samples** (*int*) – Number of samples to generate from the (approximate) posterior
- ***args** –
- ****kwargs** –

Returns
result

Return type
Sample

property seed

Return the seed of the inference.

set_objective(*n_samples*, *thresholds=None*, *quantiles=None*)

Set objective for ABC-SMC inference.

Parameters

- **n_samples** (*int*) – Number of samples to generate
- **thresholds** (*list, optional*) – List of thresholds for ABC-SMC
- **quantiles** (*list, optional*) – List of selection quantiles used to determine sample thresholds

update(*batch*, *batch_index*)

Update the inference state with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

class elfi.**AdaptiveDistanceSMC**(*model*, *discrepancy_name=None*, *output_names=None*, ***kwargs*)

SMC-ABC sampler with adaptive threshold and distance function.

Notes

Algorithm 5 in Prangle (2017)

References

Prangle D (2017). Adapting the ABC Distance Function. Bayesian Analysis 12(1):289-309, 2017. https://projecteuclid.org/euclid.ba/1460641065

Initialize the adaptive distance SMC-ABC sampler.

Parameters

- **model** (*ElfiModel or NodeReference*) –
- **discrepancy_name** (*str, NodeReference, optional*) – Only needed if model is an ElfiModel
- **output_names** (*list, optional*) – Additional outputs from the model to be included in the inference result, e.g. corresponding summaries to the acquired samples
- **kwargs** – See ParameterInference

property batch_size

Return the current batch_size.

property current_population_threshold

Return the threshold for current population.

extract_result()

Extract the result from the current state.

Return type
SmcSample

property finished

Check whether objective of n_batches have been reached.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

iterate()

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

property parameter_names

Return the parameters to be inferred.

plot_state(***kwargs*)

Plot the current state of the algorithm.

Parameters

- **axes** (*matplotlib.axes.Axes (optional)*) –
- **figure** (*matplotlib.figure.Figure (optional)*) –
- **xlim** – x-axis limits
- **ylim** – y-axis limits
- **interactive** (*bool (default False)*) – If true, uses IPython.display to update the cell figure
- **close** – Close figure in the end of plotting. Used in the end of interactive mode.

Return type
None

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

sample(*n_samples*, **args*, ***kwargs*)

Sample from the approximate posterior.

See the other arguments from the *set_objective* method.

Parameters

- **n_samples** (*int*) – Number of samples to generate from the (approximate) posterior
- ***args** –
- ****kwargs** –

Returns
result

Return type
Sample

property seed

Return the seed of the inference.

set_objective(*n_samples*, *rounds*, *quantile=0.5*)

Set objective for adaptive distance ABC-SMC inference.

Parameters

- **n_samples** (*int*) – Number of samples to generate
- **rounds** (*int, optional*) – Number of populations to sample
- **quantile** (*float, optional*) – Selection quantile used to determine sample thresholds

update(*batch*, *batch_index*)

Update the inference state with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

class elfi.**AdaptiveThresholdSMC**(*model*, *discrepancy_name=None*, *output_names=None*, *initial_quantile=0.2*, *q_threshold=0.99*, *densratio_estimation=None*, ***kwargs*)

ABC-SMC sampler with adaptive threshold selection.

References

Simola U, Cisewski-Kehe J, Gutmann M U, Corander J (2021). Adaptive Approximate Bayesian Computation Tolerance Selection. Bayesian Analysis. https://doi.org/10.1214/20-BA1211

Initialize the adaptive threshold SMC-ABC sampler.

Parameters

- **model** (*ElfiModel or NodeReference*) –
- **discrepancy_name** (*str, NodeReference, optional*) – Only needed if model is an ElfiModel
- **output_names** (*list, optional*) – Additional outputs from the model to be included in the inference result, e.g. corresponding summaries to the acquired samples
- **initial_quantile** (*float, optional*) – Initial selection quantile for the first round of adaptive-ABC-SMC
- **q_threshold** (*float, optional*) – Termination criteratia for adaptive-ABC-SMC
- **densratio_estimation** (*DensityRatioEstimation, optional*) – Density ratio estimation object defining parameters for KLIEP
- **kwargs** – See ParameterInference

property batch_size

Return the current batch_size.

property current_population_threshold

Return the threshold for current population.

extract_result()

Extract the result from the current state.

Return type
SmcSample

property finished

Check whether objective of n_batches have been reached.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

`iterate()`

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

`property parameter_names`

Return the parameters to be inferred.

`plot_state(**kwargs)`

Plot the current state of the algorithm.

Parameters

- **axes** (*matplotlib.axes.Axes (optional)*) –
- **figure** (*matplotlib.figure.Figure (optional)*) –
- **xlim** – x-axis limits
- **ylim** – y-axis limits
- **interactive** (*bool (default False)*) – If true, uses IPython.display to update the cell figure
- **close** – Close figure in the end of plotting. Used in the end of interactive mode.

Return type
None

`property pool`

Return the output pool of the inference.

`prepare_new_batch(batch_index)`

Prepare values for a new batch.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

sample(*n_samples*, **args*, ***kwargs*)

Sample from the approximate posterior.

See the other arguments from the *set_objective* method.

Parameters

- **n_samples** (*int*) – Number of samples to generate from the (approximate) posterior
- ***args** –
- ****kwargs** –

Returns
result

Return type
Sample

property seed

Return the seed of the inference.

set_objective(*n_samples*, *max_iter=10*)

Set objective for ABC-SMC inference.

Parameters

- **n_samples** (*int*) – Number of samples to generate
- **thresholds** (*list, optional*) – List of thresholds for ABC-SMC
- **max_iter** (*int, optional*) – Maximum number of iterations

update(*batch*, *batch_index*)

Update the inference state with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

class elfi.**BayesianOptimization**(*model*, *target_name=None*, *bounds=None*, *initial_evidence=None*, *update_interval=10*, *target_model=None*, *acquisition_method=None*, *acq_noise_var=0*, *exploration_rate=10*, *batch_size=1*, *batches_per_acquisition=None*, *async_acq=False*, ***kwargs*)

Bayesian Optimization of an unknown target function.

Initialize Bayesian optimization.

Parameters

- **model** (*ElfiModel or NodeReference*) –
- **target_name** (*str or NodeReference*) – Only needed if model is an ElfiModel
- **bounds** (*dict, optional*) – The region where to estimate the posterior for each parameter in model.parameters: dict('parameter_name':(lower, upper), ...)`. Not used if custom target_model is given.
- **initial_evidence** (*int, dict, optional*) – Number of initial evidence or a precomputed batch dict containing parameter and discrepancy values. Default value depends on the dimensionality.

- **update_interval** (*int, optional*) – How often to update the GP hyperparameters of the target_model
- **target_model** (*GPyRegression, optional*) –
- **acquisition_method** (*Acquisition, optional*) – Method of acquiring evidence points. Defaults to LCBSC.
- **acq_noise_var** (*float or dict, optional*) – Variance(s) of the noise added in the default LCBSC acquisition method. If a dictionary, values should be float specifying the variance for each dimension.
- **exploration_rate** (*float, optional*) – Exploration rate of the acquisition method
- **batch_size** (*int, optional*) – Elfi batch size. Defaults to 1.
- **batches_per_acquisition** (*int, optional*) – How many batches will be requested from the acquisition function at one go. Defaults to max_parallel_batches.
- **async_acq** (*bool, optional*) – Allow acquisitions to be made asynchronously, i.e. do not wait for all the results from the previous acquisition before making the next. This can be more efficient with a large amount of workers (e.g. in cluster environments) but forgoes the guarantee for the exactly same result with the same initial conditions (e.g. the seed). Default False.
- ****kwargs** –

property acq_batch_size

Return the total number of acquisition per iteration.

property batch_size

Return the current batch_size.

extract_result()

Extract the result from the current state.

Return type
OptimizationResult

property finished

Check whether objective of n_batches have been reached.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

`iterate()`

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

`property n_evidence`

Return the number of acquired evidence points.

`property parameter_names`

Return the parameters to be inferred.

`plot_discrepancy(axes=None, **kwargs)`

Plot acquired parameters vs. resulting discrepancy.

Parameters
axes (*plt.Axes or arraylike of plt.Axes*) –

Returns
axes

Return type
np.array of plt.Axes

`plot_gp(axes=None, resol=50, const=None, bounds=None, true_params=None, **kwargs)`

Plot pairwise relationships as a matrix with parameters vs. discrepancy.

Parameters

- **axes** (*matplotlib.axes.Axes, optional*) –
- **resol** (*int, optional*) – Resolution of the plotted grid.
- **const** (*np.array, optional*) – Values for parameters in plots where held constant. Defaults to minimum evidence.
- **bounds** (*list of tuples, optional*) – List of tuples for axis boundaries.
- **true_params** (*dict, optional*) – Dictionary containing parameter names with corresponding true parameter values.

Returns
axes

Return type
np.array of plt.Axes

`plot_state(**options)`

Plot the GP surface.

This feature is still experimental and currently supports only 2D cases.

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

property seed

Return the seed of the inference.

set_objective(*n_evidence=None*)

Set objective for inference.

You can continue BO by giving a larger n_evidence.

Parameters
n_evidence (*int*) – Number of total evidence for the GP fitting. This includes any initial evidence.

update(*batch*, *batch_index*)

Update the GP regression model of the target node with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

class elfi.**BOLFI**(*model*, *target_name=None*, *bounds=None*, *initial_evidence=None*, *update_interval=10*, *target_model=None*, *acquisition_method=None*, *acq_noise_var=0*, *exploration_rate=10*, *batch_size=1*, *batches_per_acquisition=None*, *async_acq=False*, ***kwargs*)

Bayesian Optimization for Likelihood-Free Inference (BOLFI).

Approximates the discrepancy function by a stochastic regression model. Discrepancy model is fit by sampling the discrepancy function at points decided by the acquisition function.

The method implements the framework introduced in Gutmann & Corander, 2016.

References

Gutmann M U, Corander J (2016). Bayesian Optimization for Likelihood-Free Inference of Simulator-Based Statistical Models. JMLR 17(125):147, 2016. http://jmlr.org/papers/v17/15-017.html

Initialize Bayesian optimization.

Parameters

- **model** (*ElfiModel or NodeReference*) –
- **target_name** (*str or NodeReference*) – Only needed if model is an ElfiModel

- **bounds** (*dict, optional*) – The region where to estimate the posterior for each parameter in model.parameters: dict('parameter_name':(lower, upper), ...)`. Not used if custom target_model is given.
- **initial_evidence** (*int, dict, optional*) – Number of initial evidence or a precomputed batch dict containing parameter and discrepancy values. Default value depends on the dimensionality.
- **update_interval** (*int, optional*) – How often to update the GP hyperparameters of the target_model
- **target_model** (*GPyRegression, optional*) –
- **acquisition_method** (*Acquisition, optional*) – Method of acquiring evidence points. Defaults to LCBSC.
- **acq_noise_var** (*float or dict, optional*) – Variance(s) of the noise added in the default LCBSC acquisition method. If a dictionary, values should be float specifying the variance for each dimension.
- **exploration_rate** (*float, optional*) – Exploration rate of the acquisition method
- **batch_size** (*int, optional*) – Elfi batch size. Defaults to 1.
- **batches_per_acquisition** (*int, optional*) – How many batches will be requested from the acquisition function at one go. Defaults to max_parallel_batches.
- **async_acq** (*bool, optional*) – Allow acquisitions to be made asynchronously, i.e. do not wait for all the results from the previous acquisition before making the next. This can be more efficient with a large amount of workers (e.g. in cluster environments) but forgoes the guarantee for the exactly same result with the same initial conditions (e.g. the seed). Default False.
- ****kwargs** –

property acq_batch_size

Return the total number of acquisition per iteration.

property batch_size

Return the current batch_size.

extract_posterior(*threshold=None*)

Return an object representing the approximate posterior.

The approximation is based on surrogate model regression.

Parameters
threshold (*float, optional*) – Discrepancy threshold for creating the posterior (log with log discrepancy).

Returns
posterior

Return type
elfi.methods.posteriors.BolfiPosterior

extract_result()

Extract the result from the current state.

Return type
OptimizationResult

property finished

Check whether objective of n_batches have been reached.

fit(*n_evidence*, *threshold=None*, *bar=True*)

Fit the surrogate model.

Generates a regression model for the discrepancy given the parameters.

Currently only Gaussian processes are supported as surrogate models.

Parameters

- **n_evidence** (*int, required*) – Number of evidence for fitting
- **threshold** (*float, optional*) – Discrepancy threshold for creating the posterior (log with log discrepancy).
- **bar** (*bool, optional*) – Flag to remove (False) the progress bar from output.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

iterate()

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

property n_evidence

Return the number of acquired evidence points.

property parameter_names

Return the parameters to be inferred.

plot_discrepancy(*axes=None*, ***kwargs*)

Plot acquired parameters vs. resulting discrepancy.

Parameters
axes (*plt.Axes or arraylike of plt.Axes*) –

Returns
axes

Return type
np.array of plt.Axes

plot_gp(*axes=None*, *resol=50*, *const=None*, *bounds=None*, *true_params=None*, ***kwargs*)

Plot pairwise relationships as a matrix with parameters vs. discrepancy.

Parameters

- **axes** (*matplotlib.axes.Axes, optional*) –
- **resol** (*int, optional*) – Resolution of the plotted grid.
- **const** (*np.array, optional*) – Values for parameters in plots where held constant. Defaults to minimum evidence.
- **bounds** (*list of tuples, optional*) – List of tuples for axis boundaries.
- **true_params** (*dict, optional*) – Dictionary containing parameter names with corresponding true parameter values.

Returns
axes

Return type
np.array of plt.Axes

plot_state(***options*)

Plot the GP surface.

This feature is still experimental and currently supports only 2D cases.

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

sample(*n_samples*, *warmup=None*, *n_chains=4*, *threshold=None*, *initials=None*, *algorithm='nuts'*, *sigma_proposals=None*, *n_evidence=None*, ***kwargs*)

Sample the posterior distribution of BOLFI.

Here the likelihood is defined through the cumulative density function of the standard normal distribution:

L(theta) propto F((h-mu(theta)) / sigma(theta))

where h is the threshold, and mu(theta) and sigma(theta) are the posterior mean and (noisy) standard deviation of the associated Gaussian process.

The sampling is performed with an MCMC sampler (the No-U-Turn Sampler, NUTS).

Parameters

- **n_samples** (*int*) – Number of requested samples from the posterior for each chain. This includes warmup, and note that the effective sample size is usually considerably smaller.
- **warmpup** (*int, optional*) – Length of warmup sequence in MCMC sampling. Defaults to n_samples//2.
- **n_chains** (*int, optional*) – Number of independent chains.
- **threshold** (*float, optional*) – The threshold (bandwidth) for posterior (give as log if log discrepancy).
- **initials** (*np.array of shape (n_chains, n_params), optional*) – Initial values for the sampled parameters for each chain. Defaults to best evidence points.
- **algorithm** (*string, optional*) – Sampling algorithm to use. Currently 'nuts'(default) and 'metropolis' are supported.
- **sigma_proposals** (*dict, optional*) – Standard deviations for Gaussian proposals of each parameter for Metropolis Markov Chain sampler. Defaults to 1/10 of surrogate model bound lengths.
- **n_evidence** (*int*) – If the regression model is not fitted yet, specify the amount of evidence

Return type
McmcSample

property seed

Return the seed of the inference.

set_objective(*n_evidence=None*)

Set objective for inference.

You can continue BO by giving a larger n_evidence.

Parameters
n_evidence (*int*) – Number of total evidence for the GP fitting. This includes any initial evidence.

update(*batch*, *batch_index*)

Update the GP regression model of the target node with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

class elfi.**ROMC**(*model: ElfiModel | NodeReference*, *bounds: List | None = None*, *discrepancy_name: str | None = None*, *output_names: List[str] | None = None*, *custom_optim_class=None*, *parallelize: bool = False*, ***kwargs*)

Robust Optimisation Monte Carlo inference method.

Ikonomov, B., & Gutmann, M. U. (2019). Robust Optimisation Monte Carlo. http://arxiv.org/abs/1904.00670

Class constructor.

Parameters

- **model** (*Model or NodeReference*) – the elfi model or the output node of the graph
- **bounds** (*List[(start,stop), ...]*) – bounds of the n-dim bounding box area containing the mass of the posterior
- **discrepancy_name** (*string, optional*) – the name of the output node (obligatory, only if Model is passed as model)
- **output_names** (*List[string]*) – which node values to store during inference
- **custom_optim_class** (*class*) – Custom OptimizationProblem class provided by the user, to extend the algorithm
- **parallelize** (*bool*) – whether to parallelize all parts of the algorithm
- **kwargs** (*Dict*) – other named parameters

property batch_size

Return the current batch_size.

compute_divergence(*gt_posterior*, *bounds=None*, *step=0.1*, *distance='Jensen-Shannon'*)

Compute divergence between ROMC posterior and ground-truth.

Parameters

- **gt_posterior** (*Callable,*) – ground-truth posterior, must accepted input in a batched fashion (np.ndarray with shape: (BS,D))
- **bounds** (*List[(start, stop)]*) – if bounds are not passed at the ROMC constructor, they can be passed here
- **step** (*float*) –
- **distance** (*str*) – which distance to use. must be in ["Jensen-Shannon", "KL-Divergence"]

Returns
The computed divergence between the distributions

Return type
float

compute_eps(*quantile*)

Return the quantile distance, out of all optimal distance.

Parameters
quantile (*value in [0,1]*) –

Return type
float

compute_ess()

Compute the Effective Sample Size.

Returns
The effective sample size.

Return type
float

compute_expectation(*h*)

Compute an expectation, based on h.

Parameters
h (*Callable*) –

Return type
float or np.array, depending on the return value of the Callable h

distance_hist(*savefig=False*, ***kwargs*)

Plot a histogram of the distances at the optimal point.

Parameters

- **savefig** (*False or str, if str it must be the path to save the figure*) –
- **kwargs** (*Dict with arguments to be passed to the plt.hist()*) –

estimate_regions(*eps_filter*, *use_surrogate=False*, *region_args=None*, *fit_models=True*, *fit_models_args=None*, *eps_region=None*, *eps_cutoff=None*)

Filter solutions and build the N-Dimensional bounding box around the optimal point.

Parameters

- **eps_filter** (*float*) – threshold for filtering the solutions
- **use_surrogate** (*Union[None, bool]*) – whether to use the surrogate model for bulding the bounding box. if None, it will be set based on which optimisation scheme has been used.
- **region_args** (*Union[None, Dict]*) – keyword-arguments that will be passed to the regionConstructor. The arguments "eps_region" and "use_surrogate" are automatically appended, if not defined explicitly.
- **fit_models** (*bool*) – whether to fit a helping model around the optimal point
- **fit_models_args** (*Union[None, Dict]*) – arguments passed for fitting the helping models
- **eps_region** (*Union[None, float]*) – threshold for the bounding box limits. If None, it will be equal to eps_filter.
- **eps_cutoff** (*Union[None, float]*) – threshold for the indicator function. If None, it will be equal to eps_filter.

eval_posterior(*theta*)

Evaluate the normalized posterior. The operation is NOT vectorized.

Parameters
theta (*np.ndarray (BS, D)*) –

Returns
np.array

Return type
(BS,)

eval_unnorm_posterior(*theta*)

Evaluate the unnormalized posterior. The operation is NOT vectorized.

Parameters
theta (*np.ndarray (BS, D)*) – the position to evaluate

Returns
np.array

Return type
(BS,)

`extract_result()`

Extract the result from the current state.

Returns
result

Return type
Sample

`property finished`

Check whether objective of n_batches have been reached.

`fit_posterior(n1, eps_filter, use_bo=False, quantile=None, optimizer_args=None, region_args=None, fit_models=False, fit_models_args=None, seed=None, eps_region=None, eps_cutoff=None)`

Execute all training steps.

Parameters

- **n1** (*integer*) – nof deterministic optimisation problems
- **use_bo** (*Boolean*) – whether to use Bayesian Optimisation
- **eps_filter** (*Union[float, str]*) – threshold for filtering solution or "auto" if defined by through quantile
- **quantile** (*Union[None, float], optional*) – quantile of optimal distances to set as eps if eps="auto"
- **optimizer_args** (*Union[None, Dict]*) – keyword-arguments that will be passed to the optimiser
- **region_args** (*Union[None, Dict]*) – keyword-arguments that will be passed to the regionConstructor
- **seed** (*Union[None, int]*) – seed definition for making the training process reproducible
- **eps_region** – threshold for region construction

`infer(*args, vis=None, bar=True, **kwargs)`

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

`iterate()`

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

`property parameter_names`

Return the parameters to be inferred.

`plot_state(**kwargs)`

Plot the current state of the algorithm.

Parameters

- **axes** (*matplotlib.axes.Axes (optional)*) –
- **figure** (*matplotlib.figure.Figure (optional)*) –
- **xlim** – x-axis limits
- **ylim** – y-axis limits
- **interactive** (*bool (default False)*) – If true, uses IPython.display to update the cell figure
- **close** – Close figure in the end of plotting. Used in the end of interactive mode.

Return type
None

`property pool`

Return the output pool of the inference.

`prepare_new_batch(batch_index)`

Prepare values for a new batch.

ELFI calls this method before submitting a new batch with an increasing index *batch_index*. This is an optional method to override. Use this if you have a need do do preparations, e.g. in Bayesian optimization algorithm, the next acquisition points would be acquired here.

If you need provide values for certain nodes, you can do so by constructing a batch dictionary and returning it. See e.g. BayesianOptimization for an example.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

sample(*n2*, *seed=None*)

Get samples from the posterior.

Parameters

- **n2** (*int*) – number of samples
- **seed** (*int,*) – seed of the sampling procedure

property seed

Return the seed of the inference.

set_objective(**args*, ***kwargs*)

Set the objective of the inference.

This method sets the objective of the inference (values typically stored in the *self.objective* dict).

Return type
None

solve_problems(*n1*, *use_bo=False*, *optimizer_args=None*, *seed=None*)

Define and solve n1 optimisation problems.

Parameters

- **n1** (*integer*) – number of deterministic optimisation problems to solve
- **use_bo** (*Boolean, default: False*) – whether to use Bayesian Optimisation. If False, gradients are used.
- **optimizer_args** (*Union[None, Dict], default None*) – keyword-arguments that will be passed to the optimiser. The argument "seed" is automatically appended to the dict. In the current implementation, all arguments are optional.
- **seed** (*Union[None, int]*) –

update(*batch*, *batch_index*)

Update the inference state with a new batch.

ELFI calls this method when a new batch has been computed and the state of the inference should be updated with it. It is also possible to bypass ELFI and call this directly to update the inference.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

Return type
None

visualize_region(*i*, *force_objective=False*, *savefig=False*)

Plot the acceptance area of the i-th optimisation problem.

Parameters

- **i** (*int,*) – index of the problem
- **savefig** – None or path

class elfi.BSL(*model*, *n_sim_round*, *feature_names=None*, *likelihood=None*, ***kwargs*)

Bayesian Synthetic Likelihood for parameter inference.

For a description of the default BSL see Price et. al. 2018. Sampler implemented using Metropolis-Hastings MCMC.

References

L. F. Price, C. C. Drovandi, A. Lee & D. J. Nott (2018). Bayesian Synthetic Likelihood, Journal of Computational and Graphical Statistics, 27:1, 1-11, DOI: 10.1080/10618600.2017.1302882

Initialize the BSL sampler.

Parameters

- **model** (ElfiModel) – ELFI graph used by the algorithm.
- **n_sim_round** (*int*) – Number of simulations for 1 parametric approximation of the likelihood.
- **feature_names** (*str or list, optional*) – Features used in synthetic likelihood estimation. Defaults to all summary statistics.
- **likelihood** (*callable, optional*) – Synthetic likelihood estimation method. Defaults to gaussian_syn_likelihood.

property batch_size

Return the current batch_size.

property current_params

Return parameter values explored in the current round.

BSL runs simulations with the candidate parameter values stored in method state.

extract_result()

Extract the result from the current state.

Returns
result

Return type
BslSample

property finished

Check whether objective of n_batches have been reached.

infer(**args*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

Initialise a new data collection round if needed.

Returns
result

Return type
Sample

iterate()

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

property parameter_names

Return the parameters to be inferred.

plot_state(***kwargs*)

Plot the current state of the algorithm.

Parameters

- **axes** (*matplotlib.axes.Axes (optional)*) –
- **figure** (*matplotlib.figure.Figure (optional)*) –
- **xlim** – x-axis limits
- **ylim** – y-axis limits
- **interactive** (*bool (default False)*) – If true, uses IPython.display to update the cell figure
- **close** – Close figure in the end of plotting. Used in the end of interactive mode.

Return type
None

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

Parameters
batch_index (*int*) –

Returns
batch

Return type
dict

sample(*n_samples*, *sigma_proposals*, *params0=None*, *param_names=None*, *burn_in=0*, *logit_transform_bound=None*, *tau=0.5*, *w=1*, *max_iter=1000*, ***kwargs*)

Sample from the posterior distribution of BSL.

The specific approximate likelihood estimated depends on the BSL class but generally uses a multivariate normal approximation.

The sampling is performed with a metropolis MCMC sampler, and gamma parameters are sampled with a slice sampler when adjustment for model misspecification is used.

Parameters

- **n_samples** (*int*) – Number of requested samples from the posterior. This includes burn_in.

- **sigma_proposals** (*np.array of shape (k x k) - k = number of parameters*) – Standard deviations for Gaussian proposals of each parameter.
- **params0** (*array_like, optional*) – Initial values for each sampled parameter.
- **param_names** (*list, optional*) – Custom list of parameter names corresponding to the order of parameters in params0 and sigma_proposals.
- **burn_in** (*int, optional*) – Length of burnin sequence in MCMC sampling. These samples are "thrown away". Defaults to 0.
- **logit_transform_bound** (*list, optional*) – Each list element contains the lower and upper bound for the logit transformation of the corresponding parameter.
- **tau** (*float, optional*) – Scale parameter for the prior distribution used by the gamma sampler.
- **w** (*float, optional*) – Step size used by the gamma sampler.
- **max_iter** (*int, optional*) – Maximum number of iterations used by the gamma sampler.

Return type
BslSample

property seed

Return the seed of the inference.

set_objective(*rounds*)

Set objective for inference.

Parameters
rounds (*int*) – Number of data collection rounds.

update(*batch*, *batch_index*)

Update the inference state with a new batch.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

Result objects

class elfi.methods.results.**OptimizationResult**(*x_min*, ***kwargs*)

Base class for results from optimization.

Initialize result.

Parameters

- **x_min** – The optimized parameters
- ****kwargs** – See *ParameterInferenceResult*

property is_multivariate

Check whether the result contains multivariate parameters.

class elfi.methods.results.**Sample**(*method_name*, *outputs*, *parameter_names*, *discrepancy_name=None*, *weights=None*, ***kwargs*)

Sampling results from inference methods.

Initialize result.

Parameters

- **method_name** (*string*) – Name of inference method.
- **outputs** (*dict*) – Dictionary with outputs from the nodes, e.g. samples.
- **parameter_names** (*list*) – Names of the parameter nodes
- **discrepancy_name** (*string, optional*) – Name of the discrepancy in outputs.
- **weights** (*array_like*) –
- ****kwargs** – Other meta information for the result

property dim

Return the number of parameters.

property discrepancies

Return the discrepancy values.

get_sample_covariance()

Return covariance of samples.

property idata

Convert posterior sample to arviz InferenceData object.

property is_multivariate

Check whether the result contains multivariate parameters.

property n_samples

Return the number of samples.

plot_marginals(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, ***kwargs*)

Plot marginal distributions for parameters.

Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

plot_pairs(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, *draw_upper_triagonal=False*, ***kwargs*)

Plot pairwise relationships as a matrix with marginals on the diagonal.

The y-axis of marginal histograms are scaled. Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.

- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

property sample_means

Evaluate weighted averages of sampled parameters.

Return type
OrderedDict

property sample_means_and_95CIs

Construct OrderedDict for mean and 95% credible interval.

property sample_means_array

Evaluate weighted averages of sampled parameters.

Return type
np.array

sample_means_summary()

Print a representation of sample means.

sample_quantiles(*alpha=0.5*)

Evaluate weighted sample quantiles of sampled parameters.

sample_summary()

Print sample mean and 95% credible interval.

property samples_array

Return the samples as an array.

The columns are in the same order as in self.parameter_names.

Return type
list of np.arrays

save(*fname=None*)

Save samples in csv, json or pickle file formats.

Clarification: csv saves only samples, json saves the whole object's dictionary except *outputs* key and pickle saves the whole object.

Parameters
fname (*str, required*) – File name to be saved. The type is inferred from extension ('csv', 'json' or 'pkl').

summary()

Print a verbose summary of contained results.

class elfi.methods.results.**SmcSample**(*method_name*, *outputs*, *parameter_names*, *populations*, **args*, ***kwargs*)

Container for results from SMC-ABC.

Initialize result.

Parameters

- **method_name** (*str*) –

- **outputs** (*dict*) –
- **parameter_names** (*list*) –
- **populations** (*list[Sample]*) – List of Sample objects
- **args** –
- **kwargs** –

property dim

Return the number of parameters.

property discrepancies

Return the discrepancy values.

get_sample_covariance()

Return covariance of samples.

property idata

Convert posterior sample to arviz InferenceData object.

property is_multivariate

Check whether the result contains multivariate parameters.

property n_populations

Return the number of populations.

property n_samples

Return the number of samples.

plot_marginals(*selector=None*, *bins=20*, *axes=None*, *all=False*, ***kwargs*)

Plot marginal distributions for parameters for all populations.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –
- **all** (*bool, optional*) – Plot the marginals of all populations

plot_pairs(*selector=None*, *bins=20*, *axes=None*, *all=False*, ***kwargs*)

Plot pairwise relationships as a matrix with marginals on the diagonal.

The y-axis of marginal histograms are scaled.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –
- **all** (*bool, optional*) – Plot for all populations

property sample_means

Evaluate weighted averages of sampled parameters.

Return type
OrderedDict

property sample_means_and_95CIs

Construct OrderedDict for mean and 95% credible interval.

property sample_means_array

Evaluate weighted averages of sampled parameters.

Return type
np.array

sample_means_summary(*all=False*)

Print a representation of sample means.

Parameters
all (*bool, optional*) – Whether to print the means for all populations separately, or just the final population (default).

sample_quantiles(*alpha=0.5*)

Evaluate weighted sample quantiles of sampled parameters.

sample_summary()

Print sample mean and 95% credible interval.

property samples_array

Return the samples as an array.

The columns are in the same order as in self.parameter_names.

Return type
list of np.arrays

save(*fname=None*)

Save samples in csv, json or pickle file formats.

Clarification: csv saves only samples, json saves the whole object's dictionary except *outputs* key and pickle saves the whole object.

Parameters
fname (*str, required*) – File name to be saved. The type is inferred from extension ('csv', 'json' or 'pkl').

summary(*all=False*)

Print a verbose summary of contained results.

Parameters
all (*bool, optional*) – Whether to print the summary for all populations separately, or just the final population (default).

class elfi.methods.results.McmcSample(*method_name*, *chains*, *parameter_names*, *warmup*, ***kwargs*)

Container for MCMC results.

Initialize result.

Parameters

- **method_name** (*string*) – Name of inference method.

- **chains** (*np.array*) – Chains from sampling, warmup included. Shape: (n_chains, n_samples, n_parameters).
- **parameter_names** (*list : list of strings*) – List of names in the outputs dict that refer to model parameters.
- **warmup** (*int*) – Number of warmup iterations in chains.

property dim

Return the number of parameters.

property discrepancies

Return the discrepancy values.

get_sample_covariance()

Return covariance of samples.

property idata

Convert MCMC chains to arviz InferenceData object.

property is_multivariate

Check whether the result contains multivariate parameters.

property n_samples

Return the number of samples.

plot_marginals(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, ***kwargs*)

Plot marginal distributions for parameters.

Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

plot_pairs(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, *draw_upper_triagonal=False*, ***kwargs*)

Plot pairwise relationships as a matrix with marginals on the diagonal.

The y-axis of marginal histograms are scaled. Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

plot_traces(*selector=None*, *axes=None*, ***kwargs*)

Plot MCMC traces.

property sample_means

Evaluate weighted averages of sampled parameters.

Return type
OrderedDict

property sample_means_and_95CIs

Construct OrderedDict for mean and 95% credible interval.

property sample_means_array

Evaluate weighted averages of sampled parameters.

Return type
np.array

sample_means_summary()

Print a representation of sample means.

sample_quantiles(*alpha=0.5*)

Evaluate weighted sample quantiles of sampled parameters.

sample_summary()

Print sample mean and 95% credible interval.

property samples_array

Return the samples as an array.

The columns are in the same order as in self.parameter_names.

Return type
list of np.arrays

save(*fname=None*)

Save samples in csv, json or pickle file formats.

Clarification: csv saves only samples, json saves the whole object's dictionary except *outputs* key and pickle saves the whole object.

Parameters
fname (*str, required*) – File name to be saved. The type is inferred from extension ('csv', 'json' or 'pkl').

summary()

Print a verbose summary of contained results.

class elfi.methods.results.**RomcSample**(*method_name*, *outputs*, *parameter_names*, *discrepancy_name*, *weights*, ***kwargs*)

Container for results from ROMC.

Class constructor.

Parameters

- **method_name** (*string*) – Name of the inference method
- **outputs** (*Dict*) – Dict where key is the parameter name and value are the samples

- **parameter_names** (*List[string]*) – List of the parameter names
- **discrepancy_name** (*string*) – name of the output (=distance) node
- **weights** (*np.ndarray*) – the weights of the samples
- **kwargs** –

property dim

Return the number of parameters.

property discrepancies

Return the discrepancy values.

get_sample_covariance()

Return covariance of samples.

property idata

Convert posterior sample to arviz InferenceData object.

property is_multivariate

Check whether the result contains multivariate parameters.

property n_samples

Return the number of samples.

plot_marginals(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, ***kwargs*)

Plot marginal distributions for parameters.

Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

plot_pairs(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, *draw_upper_triagonal=False*, ***kwargs*)

Plot pairwise relationships as a matrix with marginals on the diagonal.

The y-axis of marginal histograms are scaled. Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

property sample_means

Evaluate weighted averages of sampled parameters.

Return type
OrderedDict

property sample_means_and_95CIs

Construct OrderedDict for mean and 95% credible interval.

property sample_means_array

Evaluate weighted averages of sampled parameters.

Return type
np.array

sample_means_summary()

Print a representation of sample means.

sample_quantiles(*alpha=0.5*)

Evaluate weighted sample quantiles of sampled parameters.

sample_summary()

Print sample mean and 95% credible interval.

property samples_array

Return the samples as an array.

The columns are in the same order as in self.parameter_names.

Return type
list of np.arrays

samples_cov()

Print the empirical covariance matrix.

Returns
the covariance matrix

Return type
np.ndarray (D,D)

save(*fname=None*)

Save samples in csv, json or pickle file formats.

Clarification: csv saves only samples, json saves the whole object's dictionary except *outputs* key and pickle saves the whole object.

Parameters
fname (*str, required*) – File name to be saved. The type is inferred from extension ('csv', 'json' or 'pkl').

summary()

Print a verbose summary of contained results.

class elfi.methods.results.**BslSample**(*method_name*, *samples_all*, *parameter_names*, *burn_in=0*, *acc_rate=None*, ***kwargs*)

Container for results from BSL.

Initialize result.

Parameters

- **method_name** (*string*) – Name of inference method.
- **samples_all** (*np.ndarray*) – Dictionary with all samples from the MCMC chain, burn in included.
- **parameter_names** (*list*) – Names of the parameter nodes
- **burn_in** (*int*) – Number of samples to discard from start of MCMC chain.
- **acc_rate** (*float*) – The acceptance rate of proposed parameters in the MCMC chain
- ****kwargs** – Other meta information for the result

compute_ess()

Compute the effective sample size of mcmc chain.

Returns
Effective sample size for each paramter

Return type
dict

property dim

Return the number of parameters.

property discrepancies

Return the discrepancy values.

get_sample_covariance()

Return covariance of samples.

property idata

Convert posterior sample to arviz InferenceData object.

property is_multivariate

Check whether the result contains multivariate parameters.

property n_samples

Return the number of samples.

plot_marginals(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, ***kwargs*)

Plot marginal distributions for parameters.

Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

plot_pairs(*selector=None*, *bins=20*, *axes=None*, *reference_value=None*, *draw_upper_triagonal=False*, ***kwargs*)

Plot pairwise relationships as a matrix with marginals on the diagonal.

The y-axis of marginal histograms are scaled. Supports only univariate distributions.

Parameters

- **selector** (*iterable of ints or strings, optional*) – Indices or keys to use from samples. Default to all.
- **bins** (*int, optional*) – Number of bins in histograms.
- **axes** (*one or an iterable of plt.Axes, optional*) –

Returns
axes

Return type
np.array of plt.Axes

plot_traces(*selector=None*, *axes=None*, ***kwargs*)

Plot MCMC traces.

property sample_means

Evaluate weighted averages of sampled parameters.

Return type
OrderedDict

property sample_means_and_95CIs

Construct OrderedDict for mean and 95% credible interval.

property sample_means_array

Evaluate weighted averages of sampled parameters.

Return type
np.array

sample_means_summary()

Print a representation of sample means.

sample_quantiles(*alpha=0.5*)

Evaluate weighted sample quantiles of sampled parameters.

sample_summary()

Print sample mean and 95% credible interval.

property samples_array

Return the samples as an array.

The columns are in the same order as in self.parameter_names.

Return type
list of np.arrays

save(*fname=None*)

Save samples in csv, json or pickle file formats.

Clarification: csv saves only samples, json saves the whole object's dictionary except *outputs* key and pickle saves the whole object.

Parameters
fname (*str, required*) – File name to be saved. The type is inferred from extension ('csv', 'json' or 'pkl').

summary()

Print a verbose summary of contained results.

Post-processing

elfi.**adjust_posterior**(*sample*, *model*, *summary_names*, *parameter_names=None*, *adjustment='linear'*)

Adjust the posterior using local regression.

Note that the summary nodes need to be explicitly included to the sample object with the *output_names* keyword argument when performing the inference.

Parameters

- **sample** (elfi.methods.results.Sample) – a sample object from an ABC algorithm
- **model** (elfi.ElfiModel) – the inference model
- **summary_names** (*list[str]*) – names of the summary nodes
- **parameter_names** (*list[str] (optional)*) – names of the parameters
- **adjustment** (*RegressionAdjustment or string*) – a regression adjustment object or a string specification

  Accepted values for the string specification:

  – 'linear'

Returns
a Sample object with the adjusted posterior

Return type
elfi.methods.results.Sample

Examples

```
>>> import elfi
>>> from elfi.examples import gauss
>>> m = gauss.get_model()
>>> res = elfi.Rejection(m['d'], output_names=['ss_mean', 'ss_var'],
...                      batch_size=10).sample(500, bar=False)
>>> adj = adjust_posterior(res, m, ['ss_mean', 'ss_var'], ['mu'],␣
↪LinearAdjustment())
```

class elfi.methods.post_processing.**LinearAdjustment**(***kwargs*)

Regression adjustment using a local linear model.

adjust()

Adjust the posterior.

Only the non-finite values used to fit the regression model will be adjusted.

Return type
a Sample object containing the adjusted posterior

fit(*sample*, *model*, *summary_names*, *parameter_names=None*)

Fit a regression adjustment model to the posterior sample.

Non-finite values in the summary statistics and parameters will be omitted.

Parameters

- **sample** (*elfi.methods.Sample*) – a sample object from an ABC method
- **model** (elfi.ElfiModel) – the inference model
- **summary_names** (*list[str]*) – a list of names for the summary nodes
- **parameter_names** (*list[str] (optional)*) – a list of parameter names

Diagnostics

class elfi.**TwoStageSelection**(*simulator*, *fn_distance*, *list_ss=None*, *prepared_ss=None*, *max_cardinality=4*, *seed=0*)

Perform the summary-statistics selection proposed by Nunes and Balding (2010).

The user can provide a list of summary statistics as list_ss, and let ELFI to combine them, or provide some already combined summary statistics as prepared_ss.

The rationale of the Two Stage procedure procedure is the following:

- First, the module computes or accepts the combinations of the candidate summary statistics.
- In Stage 1, each summary-statistics combination is evaluated using the Minimum Entropy algorithm.
- In Stage 2, the minimum-entropy combination is selected, and the 'closest' datasets are identified.
- Further in Stage 2, for each summary-statistics combination, the mean root sum of squared errors (MRSSE) is calculated over all 'closest datasets', and the minimum-MRSSE combination is chosen as the one with the optimal performance.

References

[1] Nunes, M. A., & Balding, D. J. (2010). On optimal selection of summary statistics for approximate Bayesian computation. Statistical applications in genetics and molecular biology, 9(1). [2] Blum, M. G., Nunes, M. A., Prangle, D., & Sisson, S. A. (2013). A comparative review of dimension reduction methods in approximate Bayesian computation. Statistical Science, 28(2), 189-208.

Initialise the summary-statistics selection for the Two Stage Procedure.

Parameters

- **simulator** (*elfi.Node*) – Node (often elfi.Simulator) for which the summary statistics will be applied. The node is the final node of a coherent ElfiModel (i.e. it has no child nodes).
- **fn_distance** (*str or callable function*) – Distance metric, consult the elfi.Distance documentation for calling as a string.
- **list_ss** (*List of callable functions, optional*) – List of candidate summary statistics.
- **prepared_ss** (*List of lists of callable functions, optional*) – List of prepared combinations of candidate summary statistics. No other combinations will be evaluated.
- **max_cardinality** (*int, optional*) – Maximum cardinality of a candidate summary-statistics combination.
- **seed** (*int, optional*) –

run(*n_sim*, *n_acc=None*, *n_closest=None*, *batch_size=1*, *k=4*)

Run the Two Stage Procedure for identifying relevant summary statistics.

Parameters

- **n_sim** (*int*) – Number of the total ABC-rejection simulations.
- **n_acc** (*int, optional*) – Number of the accepted ABC-rejection simulations.
- **n_closest** (*int, optional*) – Number of the 'closest' datasets (i.e., the closest n simulation datasets w.r.t the observations).
- **batch_size** (*int, optional*) – Number of samples per batch.
- **k** (*int, optional*) – Parameter for the kth-nearest-neighbour search performed in the minimum-entropy step (in Nunes & Balding, 2010 it is fixed to 4).

Returns
Summary-statistics combination showing the optimal performance.

Return type
array_like

Acquisition methods

class elfi.methods.bo.acquisition.**LCBSC**(**args*, *delta=None*, ***kwargs*)

Lower Confidence Bound Selection Criterion.

Srinivas et al. call this GP-LCB.

LCBSC uses the parameter delta which is here equivalent to 1/exploration_rate.

Parameter delta should be in (0, 1) for the theoretical results to hold. The theoretical upper bound for total regret in Srinivas et al. has a probability greater or equal to 1 - delta, so values of delta very close to 1 or over it do not make much sense in that respect.

Delta is roughly the exploitation tendency of the acquisition function.

References

N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In Proc. International Conference on Machine Learning (ICML), 2010

E. Brochu, V.M. Cora, and N. de Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. arXiv:1012.2599, 2010.

Notes

The formula presented in Brochu (pp. 15) seems to be from Srinivas et al. Theorem 2. However, instead of having t**(d/2 + 2) in beta_t, it seems that the correct form would be t**(2d + 2).

Initialize LCBSC.

Parameters
delta (*float, optional*) – In between (0, 1). Default is 1/exploration_rate. If given, overrides the exploration_rate.

acquire(*n*, *t=None*)

Return the next batch of acquisition points.

Gaussian noise ~N(0, self.noise_var) is added to the acquired points.

Parameters

- **n** (*int*) – Number of acquisition points to return.
- **t** (*int*) – Current acq_batch_index (starting from 0).

Returns
x – The shape is (n, input_dim)

Return type
np.ndarray

property delta

Return the inverse of exploration rate.

evaluate(*x*, *t=None*)

Evaluate the Lower confidence bound selection criterion.

Parameters

- **x** (*np.ndarray*) –
- **t** (*int, optional*) – Current iteration (starting from 0).

Return type
np.ndarray

evaluate_gradient(*x*, *t=None*)

Evaluate the gradient of the lower confidence bound selection criterion.

Parameters

- **x** (*np.ndarray*) –
- **t** (*int, optional*) – Current iteration (starting from 0).

Return type
np.ndarray

class elfi.methods.bo.acquisition.**MaxVar**(*model*, *prior*, *quantile_eps=0.01*, ***opts*)

The maximum variance acquisition method.

The next evaluation point is acquired in the maximiser of the variance of the unnormalised approximate posterior.

$$\theta_{t+1} = \arg\max \mathrm{Var}(p(\theta) \cdot p_a(\theta)),$$

where the unnormalised likelihood p_a is defined using the CDF of normal distribution, Φ, as follows:

$$p_a(\theta) = \Phi((\epsilon - \mu_{1:t}(\theta))/\sqrt{v_{1:t}(\theta) + \sigma_n^2}),$$

where epsilon is the ABC threshold, $\mu_{1:t}$ and $v_{1:t}$ are determined by the Gaussian process, σ_n^2 is the noise.

References

Järvenpää et al. (2019). Efficient Acquisition Rules for Model-Based Approximate Bayesian Computation. Bayesian Analysis 14(2):595-622, 2019 https://projecteuclid.org/euclid.ba/1537258134

Initialise MaxVar.

Parameters

- **model** (*elfi.GPyRegression*) – Gaussian process model used to calculate the unnormalised approximate likelihood.

- **prior** (*scipy-like distribution*) – Prior distribution.
- **quantile_eps** (*int, optional*) – Quantile of the observed discrepancies used in setting the ABC threshold.

acquire(*n*, *t=None*)

Acquire a batch of acquisition points.

Parameters

- **n** (*int*) – Number of acquisitions.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Coordinates of the yielded acquisition points.

Return type
array_like

evaluate(*theta_new*, *t=None*)

Evaluate the acquisition function at the location theta_new.

Parameters

- **theta_new** (*array_like*) – Evaluation coordinates.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Variance of the approximate posterior.

Return type
array_like

evaluate_gradient(*theta_new*, *t=None*)

Evaluate the acquisition function's gradient at the location theta_new.

Parameters

- **theta_new** (*array_like*) – Evaluation coordinates.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Gradient of the variance of the approximate posterior

Return type
array_like

class elfi.methods.bo.acquisition.**RandMaxVar**(*model*, *prior*, *quantile_eps=0.01*, *sampler='nuts'*, *n_samples=50*, *warmup=None*, *limit_faulty_init=1000*, *init_from_prior=False*, *sigma_proposals=None*, ***opts*)

The randomised maximum variance acquisition method.

The next evaluation point is sampled from the density corresponding to the variance of the unnormalised approximate posterior (The MaxVar acquisition function).

$$\theta_{t+1} \sim q(\theta),$$

where $q(\theta) \propto \mathrm{Var}(p(\theta) \cdot p_a(\theta))$ and the unnormalised likelihood p_a is defined using the CDF of normal distribution, Φ, as follows:

$$p_a(\theta) = \Phi((\epsilon - \mu_{1:t}(\theta))/\sqrt{v_{1:t}(\theta) + \sigma_n^2}),$$

where ϵ is the ABC threshold, $\mu_{1:t}$ and $v_{1:t}$ are determined by the Gaussian process, σ_n^2 is the noise.

References

Järvenpää et al. (2019). Efficient Acquisition Rules for Model-Based Approximate Bayesian Computation. Bayesian Analysis 14(2):595-622, 2019 https://projecteuclid.org/euclid.ba/1537258134

Initialise RandMaxVar.

Parameters

- **model** (*elfi.GPyRegression*) – Gaussian process model used to calculate the unnormalised approximate likelihood.
- **prior** (*scipy-like distribution*) – Prior distribution.
- **quantile_eps** (*int, optional*) – Quantile of the observed discrepancies used in setting the ABC threshold.
- **sampler** (*string, optional*) – Name of the sampler (options: metropolis, nuts).
- **n_samples** (*int, optional*) – Length of the sampler's chain for obtaining the acquisitions.
- **warmup** (*int, optional*) – Number of samples discarded as warmup. Defaults to n_samples/2.
- **limit_faulty_init** (*int, optional*) – Limit for the iterations used to obtain the sampler's initial points.
- **init_from_prior** (*bool, optional*) – Controls whether the sampler's initial points are sampled from the prior or a uniform distribution within model bounds. Defaults to model bounds.
- **sigma_proposals** (*dict, optional*) – Standard deviations for Gaussian proposals of each parameter for Metropolis Markov Chain sampler. Defaults to 1/10 of surrogate model bound lengths.

acquire(*n*, *t=None*)

Acquire a batch of acquisition points.

Parameters

- **n** (*int*) – Number of acquisitions.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Coordinates of the yielded acquisition points.

Return type
array_like

evaluate(*theta_new*, *t=None*)

Evaluate the acquisition function at the location theta_new.

Parameters

- **theta_new** (*array_like*) – Evaluation coordinates.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Variance of the approximate posterior.

Return type
array_like

evaluate_gradient(*theta_new*, *t=None*)

Evaluate the acquisition function's gradient at the location theta_new.

Parameters

- **theta_new** (*array_like*) – Evaluation coordinates.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Gradient of the variance of the approximate posterior

Return type
array_like

class elfi.methods.bo.acquisition.**ExpIntVar**(*model*, *prior*, *quantile_eps=0.01*, *integration='grid'*, *d_grid=0.2*, *n_samples_imp=100*, *iter_imp=2*, *sampler='nuts'*, *n_samples=2000*, *sigma_proposals=None*, ***opts*)

The Expected Integrated Variance (ExpIntVar) acquisition method.

Essentially, we define a loss function that measures the overall uncertainty in the unnormalised ABC posterior over the parameter space. The value of the loss function depends on the next simulation and thus the next evaluation location θ^* is chosen to minimise the expected loss.

$$\theta_{t+1} = argmin_{\theta^* \in \Theta} L_{1:t}(\theta^*),$$

where Θ is the parameter space, and L is the expected loss function approximated as follows:

$$L_{1:t}(\theta^*) \approx 2 * \sum_{i=1}^{s} (\omega^i \cdot p^2(\theta^i) \cdot w_{1:t+1}(\theta^i, \theta^*),$$

where ω^i is an importance weight, $p^2(\theta^i)$ is the prior squared, and $w_{1:t+1}(\theta^i, \theta^*)$ is the expected variance of the unnormalised ABC posterior at theta^i after running the simulation model with parameter θ^*

References

Järvenpää et al. (2019). Efficient Acquisition Rules for Model-Based Approximate Bayesian Computation. Bayesian Analysis 14(2):595-622, 2019 https://projecteuclid.org/euclid.ba/1537258134

Initialise ExpIntVar.

Parameters

- **model** (*elfi.GPyRegression*) – Gaussian process model used to calculate the approximate unnormalised likelihood.
- **prior** (*scipy-like distribution*) – Prior distribution.
- **quantile_eps** (*int, optional*) – Quantile of the observed discrepancies used in setting the discrepancy threshold.
- **integration** (*str, optional*) – Integration method. Options: - grid (points are taken uniformly): more accurate yet computationally expensive in high dimensions; - importance (points are taken based on the importance weight): less accurate though applicable in high dimensions.
- **d_grid** (*float, optional*) – Grid tightness.

- **n_samples_imp** (*int, optional*) – Number of importance samples.
- **iter_imp** (*int, optional*) – Gap between acquisition iterations in performing importance sampling.
- **sampler** (*string, optional*) – Sampler for generating random numbers from the proposal distribution for IS. (Options: metropolis, nuts.)
- **n_samples** (*int, optional*) – Chain length for the sampler that generates the random numbers from the proposal distribution for IS.
- **sigma_proposals** (*dict, optional*) – Standard deviations for Gaussian proposals of each parameter for Metropolis Markov Chain sampler. Defaults to 1/10 of surrogate model bound lengths.

acquire(*n*, *t*)

Acquire a batch of acquisition points.

Parameters

- **n** (*int*) – Number of acquisitions.
- **t** (*int*) – Current iteration.

Returns
Coordinates of the yielded acquisition points.

Return type
array_like

evaluate(*theta_new*, *t=None*)

Evaluate the acquisition function at the location theta_new.

Parameters

- **theta_new** (*array_like*) – Evaluation coordinates.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Expected loss's term dependent on theta_new.

Return type
array_like

evaluate_gradient(*theta_new*, *t=None*)

Evaluate the acquisition function's gradient at the location theta_new.

Parameters

- **theta_new** (*array_like*) – Evaluation coordinates.
- **t** (*int, optional*) – Current iteration, (unused).

Returns
Gradient of the variance of the approximate posterior

Return type
array_like

class elfi.methods.bo.acquisition.**UniformAcquisition**(*model*, *prior=None*, *n_inits=10*, *max_opt_iters=1000*, *noise_var=None*, *exploration_rate=10*, *seed=None*, *constraints=None*)

Acquisition from uniform distribution.

Initialize AcquisitionBase.

Parameters

- **model** (*an object with attributes*) –

  input_dim
  [int] bounds : tuple of length 'input_dim' of tuples (min, max)

  and methods
  evaluate(x) : function that returns model (mean, var, std)

- **prior** (*scipy-like distribution, optional*) – By default uniform distribution within model bounds.
- **n_inits** (*int, optional*) – Number of initialization points in internal optimization.
- **max_opt_iters** (*int, optional*) – Max iterations to optimize when finding the next point.
- **noise_var** (*float or np.array, optional*) – Acquisition noise variance for adding noise to the points near the optimized location. If array, must be 1d specifying the variance for different dimensions. Default: no added noise.
- **exploration_rate** (*float, optional*) – Exploration rate of the acquisition function (if supported)
- **seed** (*int, optional*) – Seed for getting consistent acquisition results. Used in getting random starting locations in acquisition function optimization.
- **constraints** (*{Constraint, dict} or List of {Constraint, dict}, optional*) – Additional model constraints.

acquire(*n*, *t=None*)

Return random points from uniform distribution.

Parameters

- **n** (*int*) – Number of acquisition points to return.
- **t** (*int, optional*) – (unused)

Returns
x – The shape is (n, input_dim)

Return type
np.ndarray

evaluate(*x*, *t=None*)

Evaluate the acquisition function at 'x'.

Parameters

- **x** (*numpy.array*) –
- **t** (*int*) – current iteration (starting from 0)

evaluate_gradient(*x*, *t=None*)

Evaluate the gradient of acquisition function at 'x'.

Parameters

- **x** (*numpy.array*) –

- **t** (*int*) – Current iteration (starting from 0).

Model selection

elfi.**compare_models**(*sample_objs*, *model_priors=None*)

Find posterior probabilities for different models.

The algorithm requires elfi.Sample objects from prerun inference methods. For example the output from elfi.Rejection.sample is valid. The portion of samples for each model in the top discrepancies are adjusted by each models acceptance ratio and prior probability.

The discrepancies (including summary statistics) must be comparable so that it is meaningful to sort them!

Parameters

- **sample_objs** (*list of elfi.Sample*) – Resulting Sample objects from prerun inference models. The objects must include a valid *discrepancies* attribute.
- **model_priors** (*array_like, optional*) – Prior probability of each model. Defaults to 1 / n_models.

Returns
Posterior probabilities for the considered models.

Return type
np.array

Other

Data pools

class elfi.**OutputPool**(*outputs=None*, *name=None*, *prefix=None*)

Store node outputs to dictionary-like stores.

The default store is a Python dictionary.

Notes

Saving the store requires that all the stores are pickleable.

Arbitrary objects that support simple array indexing can be used as stores by using the *elfi.store.ArrayObjectStore* class.

See the *elfi.store.StoreBase* interfaces if you wish to implement your own ELFI compatible store. Basically any object that fulfills the Pythons dictionary api will work as a store in the pool.

Initialize OutputPool.

Depending on the algorithm, some of these values may be reused after making some changes to *ElfiModel* thus speeding up the inference significantly. For instance, if all the simulations are stored in Rejection sampling, one can change the summaries and distances without having to rerun the simulator.

Parameters

- **outputs** (*list, dict, optional*) – List of node names which to store or a dictionary with existing stores. The stores are created on demand.
- **name** (*str, optional*) – Name of the pool. Used to open a saved pool from disk.
- **prefix** (*str, optional*) – Path to directory under which *elfi.ArrayPool* will place its folder. Default is a relative path ./pools.

Returns
instance

Return type
OutputPool

add_batch(*batch*, *batch_index*)

Add the outputs from the batch to their stores.

add_store(*node*, *store=None*)

Add a store object for the node.

Parameters

- **node** (*str*) –
- **store** (*dict, StoreBase, optional*) –

clear()

Remove all data from the stores.

close()

Save and close the stores that support it.

The pool will not be usable afterwards.

delete()

Remove all persisted data from disk.

flush()

Flush all data from the stores.

If the store does not support flushing, do nothing.

get_batch(*batch_index*, *output_names=None*)

Return a batch from the stores of the pool.

Parameters

- **batch_index** (*int*) –
- **output_names** (*list*) – which outputs to include to the batch

Returns
batch

Return type
dict

get_store(*node*)

Return the store for *node*.

property has_context

Check if current pool has context information.

has_store(*node*)

Check if *node* is in stores.

classmethod open(*name*, *prefix=None*)

Open a closed or saved ArrayPool from disk.

Parameters

- **name** (*str*) –
- **prefix** (*str, optional*) –

Return type
ArrayPool

property output_names

Return a list of stored names.

property path

Return the path to the pool.

remove_batch(*batch_index*)

Remove the batch from all stores.

remove_store(*node*)

Remove and return a store from the pool.

Parameters
node (*str*) –

Returns
The removed store

Return type
store

save()

Save the pool to disk.

This will use pickle to store the pool under self.path.

set_context(*context*)

Set the context of the pool.

The pool needs to know the batch_size and the seed.

Notes

Also sets the name of the pool if not set already.

Parameters
context (*elfi.ComputationContext*) –

class elfi.**ArrayPool**(*outputs=None*, *name=None*, *prefix=None*)

OutputPool that uses binary .npy files as default stores.

The default store medium for output data is a NumPy binary *.npy* file for NumPy array data. You can however also add other types of stores as well.

Notes

The default store is implemented in elfi.store.NpyStore that uses NpyArrays as stores. The NpyArray is a wrapper over NumPy .npy binary file for array data and supports appending the .npy file. It uses the .npy format 2.0 files.

Initialize OutputPool.

Depending on the algorithm, some of these values may be reused after making some changes to *ElfiModel* thus speeding up the inference significantly. For instance, if all the simulations are stored in Rejection sampling, one can change the summaries and distances without having to rerun the simulator.

Parameters

- **outputs** (*list, dict, optional*) – List of node names which to store or a dictionary with existing stores. The stores are created on demand.
- **name** (*str, optional*) – Name of the pool. Used to open a saved pool from disk.
- **prefix** (*str, optional*) – Path to directory under which *elfi.ArrayPool* will place its folder. Default is a relative path ./pools.

Returns
instance

Return type
OutputPool

add_batch(*batch*, *batch_index*)

Add the outputs from the batch to their stores.

add_store(*node*, *store=None*)

Add a store object for the node.

Parameters

- **node** (*str*) –
- **store** (*dict, StoreBase, optional*) –

clear()

Remove all data from the stores.

close()

Save and close the stores that support it.

The pool will not be usable afterwards.

delete()

Remove all persisted data from disk.

flush()

Flush all data from the stores.

If the store does not support flushing, do nothing.

get_batch(*batch_index*, *output_names=None*)

Return a batch from the stores of the pool.

Parameters

- **batch_index** (*int*) –
- **output_names** (*list*) – which outputs to include to the batch

Returns
batch

Return type
dict

`get_store`(*node*)

Return the store for *node*.

`property has_context`

Check if current pool has context information.

`has_store`(*node*)

Check if *node* is in stores.

`classmethod open`(*name*, *prefix=None*)

Open a closed or saved ArrayPool from disk.

Parameters

- **name** (*str*) –
- **prefix** (*str, optional*) –

Return type
ArrayPool

`property output_names`

Return a list of stored names.

`property path`

Return the path to the pool.

`remove_batch`(*batch_index*)

Remove the batch from all stores.

`remove_store`(*node*)

Remove and return a store from the pool.

Parameters
node (*str*) –

Returns
The removed store

Return type
store

`save`()

Save the pool to disk.

This will use pickle to store the pool under self.path.

`set_context`(*context*)

Set the context of the pool.

The pool needs to know the batch_size and the seed.

Notes

Also sets the name of the pool if not set already.

Parameters
context (*elfi.ComputationContext*) –

Module functions

elfi.**get_client**()

Get the current ELFI client instance.

elfi.**set_client**(*client=None*, ***kwargs*)

Set the current ELFI client instance.

Parameters
client (*ClientBase or str*) – Instance of a client from ClientBase, or a string from ['native', 'multiprocessing', 'ipyparallel']. If string, the respective constructor is called with *kwargs*.

Tools

tools.**vectorize**(*constants=None*, *dtype=None*)

Vectorize an operation.

Helper for cases when you have an operation that does not support vector arguments. This tool is still experimental and may not work in all cases.

Parameters

- **operation** (*callable*) – Operation to vectorize.
- **constants** (*tuple, list, optional*) – A mask for constants in inputs, e.g. (0, 2) would indicate that the first and third positional inputs are constants. The constants will be passed as they are to each operation call.
- **dtype** (*np.dtype, bool[False], optional*) – If None, numpy converts a list of outputs automatically. In some cases this produces non desired results. If you wish to keep the outputs as they are with no conversion, specify dtype=False. This results into a 1d object numpy array with outputs as they were returned.

Notes

This is a convenience method that uses a for loop internally for the vectorization. For best performance, one should aim to implement vectorized operations (by using e.g. numpy functions that are mostly vectorized) if at all possible.

Examples

```
# This form works in most cases
vectorized_simulator = elfi.tools.vectorize(simulator)

# Tell that the second and third argument to the simulator will be a constant
vectorized_simulator = elfi.tools.vectorize(simulator, [1, 2])
elfi.Simulator(vectorized_simulator, prior, constant_1, constant_2)

# Tell the vectorizer that it should not do any conversion to the outputs
vectorized_simulator = elfi.tools.vectorize(simulator, dtype=False)
```

tools.**external_operation**(*process_result=None*, *prepare_inputs=None*, *sep=' '*, *stdout=True*, *subprocess_kwargs=None*)

Wrap an external command as a Python callable (function).

The external command can be e.g. a shell script, or an executable file.

Parameters

- **command** (*str*) – Command to execute. Arguments can be passed to the executable by using Python's format strings, e.g. *"myscript.sh {0} {batch_size} –seed {seed}"*. The command is expected to write to stdout. Since *random_state* is python specific object, a *seed* keyword argument will be available to operations that use *random_state*.
- **process_result** (*callable, np.dtype, str, optional*) – Callable result handler with a signature *output = callable(result, *inputs, **kwinputs)*. Here the *result* is either the stdout or *subprocess.CompletedProcess* depending on the stdout flag below. The inputs and kwinputs will come from ELFI. The default handler converts the stdout to numpy array with *array = np.fromstring(stdout, sep=sep)*. If *process_result* is *np.dtype* or a string, then the stdout data is casted to that type with *stdout = np.fromstring(stdout, sep=sep, dtype=process_result)*.
- **prepare_inputs** (*callable, optional*) – Callable with a signature *inputs, kwinputs = callable(*inputs, **kwinputs)*. The inputs will come from elfi.
- **sep** (*str, optional*) – Separator to use with the default *process_result* handler. Default is a space ' '. If you specify your own callable to *process_result* this value has no effect.
- **stdout** (*bool, optional*) – Pass the *process_result* handler the stdout instead of the *subprocess.CompletedProcess* instance. Default is true.
- **subprocess_kwargs** (*dict, optional*) – Options for Python's *subprocess.run* that is used to run the external command. Defaults are *shell=True, check=True*. See the *subprocess* documentation for more details.

Examples

```
>>> import elfi
>>> op = elfi.tools.external_operation('echo 1 {0}', process_result='int8')
>>>
>>> constant = elfi.Constant(123)
>>> simulator = elfi.Simulator(op, constant)
>>> simulator.generate()
array([  1, 123], dtype=int8)
```

Returns
operation – ELFI compatible operation that can be used e.g. as a simulator.

Return type
callable

1.5 Frequently Asked Questions

Below are answers to some common questions asked about ELFI.

Q: My uniform prior `elfi.Prior('uniform', 1, 2)` *does not seem to be right as it produces outputs from the interval (1, 3).*

A: The distributions defined by strings are those from `scipy.stats` and follow their definitions. There the uniform distribution uses the location/scale definition, so the first argument defines the starting point of the interval and the second its length.

Q: What is vectorization in ELFI?

A: Looping is relatively inefficient in Python, and so whenever possible, you should *vectorize* your operations. This means that repetitive computations are performed on a batch of data using precompiled libraries (typically NumPy), which effectively runs the loops in faster, compiled C-code. ELFI supports vectorized operations, and due to the potentially huge saving in CPU-time it is recommended to vectorize all user-code whenever possible.

For example, imagine you have a simulator that depends on a scalar parameter and produces a vector of 5 values. When this is used in ELFI with `batch_size` set to 1000, ELFI draws 1000 values from the parameter's prior distribution and gives this *vector* to the simulator. Ideally, the simulator should efficiently process all 1000 parameter cases in one go and output an array of shape (1000, 5). When using vectorized operations in ELFI, the length (i.e. the first dimension) of all output arrays should equal `batch_size`. Note that because of this the evaluation of summary statistics, distances etc. should bypass the first dimension (e.g. with NumPy functions using `axis=1` in this case).

See `elfi.examples` for tips on how to vectorize simulators and work with ELFI. In case you are unable to vectorize your simulator, you can use elfi.tools.vectorize to mimic vectorized behaviour, though without the performance benefits. Finally, for simplicity vectorization is not assumed (`batch_size=1` by default).

This tutorial is generated from a Jupyter notebook that can be found here.

1.6 ELFI tutorial

This tutorial covers the basics of using ELFI, i.e. how to make models, save results for later use and run different inference algorithms.

Let's begin by importing libraries that we will use and specify some settings.

```
import time

import numpy as np
import scipy.stats
import matplotlib.pyplot as plt
import logging
logging.basicConfig(level=logging.INFO)  # sometimes this is required to enable logging␣
↪inside Jupyter

%matplotlib inline
%precision 2

# Set an arbitrary seed and a global random state to keep the randomly generated␣
↪quantities the same between runs
seed = 20170530  # this will be separately given to ELFI
np.random.seed(seed)
```

1.6.1 Inference with ELFI: case MA(2) model

Throughout this tutorial we will use the 2nd order moving average model MA(2) as an example. MA(2) is a common model used in univariate time series analysis. Assuming zero mean it can be written as

$$y_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2},$$

where $\theta_1, \theta_2 \in \mathbb{R}$ and $(w_k)_{k\in\mathbb{Z}} \sim N(0,1)$ represents an independent and identically distributed sequence of white noise.

The observed data and the inference problem

In this tutorial, our task is to infer the parameters θ_1, θ_2 given a sequence of 100 observations y that originate from an MA(2) process. Let's define the MA(2) simulator as a Python function:

```
def MA2(t1, t2, n_obs=100, batch_size=1, random_state=None):
    # Make inputs 2d arrays for numpy broadcasting with w
    t1 = np.asanyarray(t1).reshape((-1, 1))
    t2 = np.asanyarray(t2).reshape((-1, 1))
    random_state = random_state or np.random

    w = random_state.randn(batch_size, n_obs+2)  # i.i.d. sequence ~ N(0,1)
    x = w[:, 2:] + t1*w[:, 1:-1] + t2*w[:, :-2]
    return x
```

Above, `t1`, `t2`, and `n_obs` are the arguments specific to the MA2 process. The latter two, `batch_size` and `random_state` are ELFI specific keyword arguments. The `batch_size` argument tells how many simulations are needed. The `random_state` argument is for generating random quantities in your simulator. It is a `numpy.RandomState` object that has all the same methods as `numpy.random` module has. It is used for ensuring consistent results and handling random number generation in parallel settings.

Vectorization

What is the purpose of the `batch_size` argument? In ELFI, operations are vectorized, meaning that instead of simulating a single MA2 sequence at a time, we simulate a *batch* of them. A vectorized function takes vectors as inputs, and computes the output for each element in the vector. Vectorization is a way to make operations efficient in Python.

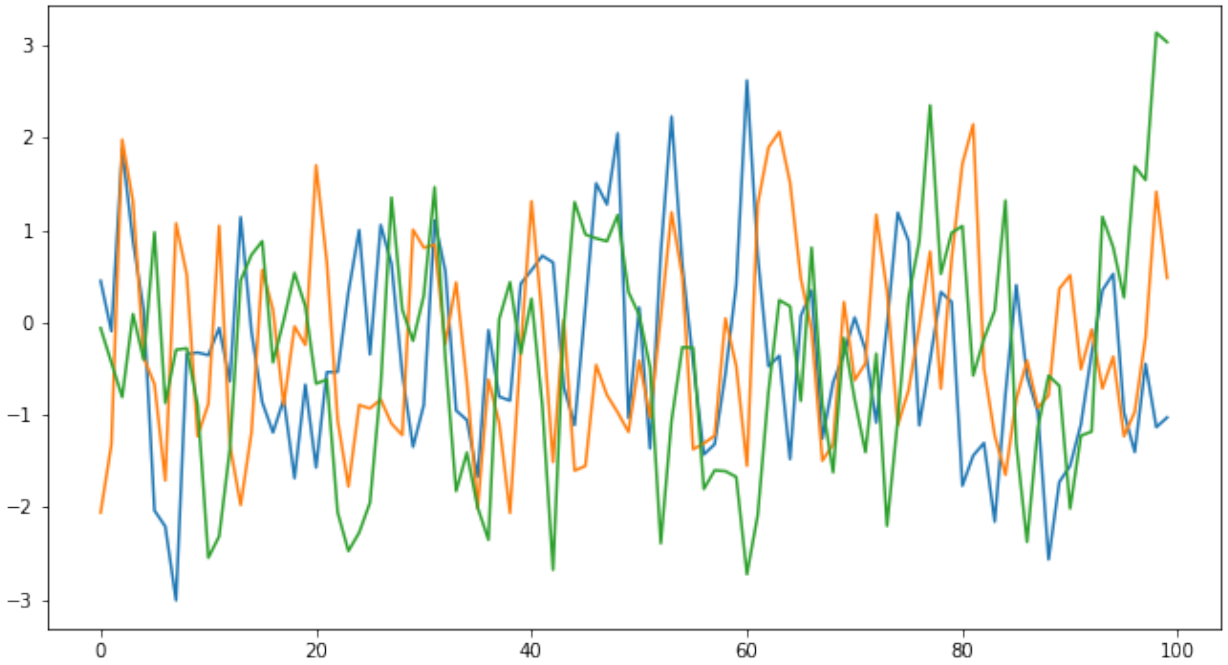
In the MA2 simulator above we rely on numpy to carry out the vectorized calculations. The arguments `t1` and `t2` are going to be **vectors** of length `batch_size` and the function returns a 2D array of shape `(batch_size, n_obs)` with each row corresponding to a single argument pair. Notice that for convenience, the funtion also works with scalars as they are first converted to vectors.

Note: There is a built-in tool (*elfi.tools.vectorize*) in ELFI to vectorize operations that are not vectorized. It is basically a for loop wrapper.

Important: In order to guarantee a consistent state of pseudo-random number generation, the simulator must have *random_state* as a keyword argument for reading in a *numpy.RandomState* object.

Let's now use this simulator to create toy observations. We will use parameter values $\theta_1 = 0.6, \theta_2 = 0.2$ as in Marin et al. (2012) and then try to infer these parameter values back based on the toy observed data alone.

```
# true parameters
t1_true = 0.6
t2_true = 0.2

y_obs = MA2(t1_true, t2_true)

# Plot the observed sequence
plt.figure(figsize=(11, 6));
plt.plot(y_obs.ravel());

# To illustrate the stochasticity, let's plot a couple of more observations with the␣
↪same true parameters:
plt.plot(MA2(t1_true, t2_true).ravel());
plt.plot(MA2(t1_true, t2_true).ravel());
```



Approximate Bayesian Computation

Standard statistical inference methods rely on the use of the *likelihood* function. Given a configuration of the parameters, the likelihood function quantifies how likely it is that values of the parameters produced the observed data. In our simple example case above however, evaluating the likelihood is difficult due to the unobserved latent sequence (variable `w` in the simulator code). In many real world applications the likelihood function is not available or it is too expensive to evaluate preventing the use of traditional inference methods.

One way to approach this problem is to use Approximate Bayesian Computation (ABC) which is a statistically based method replacing the use of the likelihood function with a simulator of the data. Loosely speaking, it is based on the intuition that similar data is likely to have been produced by similar parameters. Looking at the picture above, in essence we would keep simulating until we have found enough sequences that are similar to the observed sequence. Although the idea may appear inapplicable for the task at hand, you will soon see that it does work. For more information about ABC, please see e.g.

- Lintusaari, J., Gutmann, M. U., Dutta, R., Kaski, S., and Corander, J. (2016). Fundamentals and recent develop-

ments in approximate Bayesian computation. Systematic Biology, doi: 10.1093/sysbio/syw077.

- Marin, J.-M., Pudlo, P., Robert, C. P., and Ryder, R. J. (2012). Approximate Bayesian computational methods. Statistics and Computing, 22(6):1167–1180.
- https://en.wikipedia.org/wiki/Approximate_Bayesian_computation

1.6.2 Defining the model

ELFI includes an easy to use generative modeling syntax, where the generative model is specified as a directed acyclic graph (DAG). This provides an intuitive means to describe rather complex dependencies conveniently. Often the target of the generative model is a distance between the simulated and observed data. To start creating our model, we will first import ELFI:

```
import elfi
```

As is usual in Bayesian statistical inference, we need to define *prior* distributions for the unknown parameters θ_1, θ_2. In ELFI the priors can be any of the continuous and discrete distributions available in `scipy.stats` (for custom priors, see *below*). For simplicity, let's start by assuming that both parameters follow `Uniform(0, 2)`.

```
# a node is defined by giving a distribution from scipy.stats together with any␣
↪arguments (here 0 and 2)
t1 = elfi.Prior(scipy.stats.uniform, 0, 2)

# ELFI also supports giving the scipy.stats distributions as strings
t2 = elfi.Prior('uniform', 0, 2)
```

Next, we define the *simulator* node with the `MA2` function above, and give the priors to it as arguments. This means that the parameters for the simulations will be drawn from the priors. Because we have the observed data available for this node, we provide it here as well:

```
Y = elfi.Simulator(MA2, t1, t2, observed=y_obs)
```

But how does one compare the simulated sequences with the observed sequence? Looking at the plot of just a few observed sequences above, a direct pointwise comparison would probably not work very well: the three sequences look quite different although they were generated with the same parameter values. Indeed, the comparison of simulated sequences is often the most difficult (and ad hoc) part of ABC. Typically one chooses one or more summary statistics and then calculates the discrepancy between those.

Here, we will apply the intuition arising from the definition of the MA(2) process, and use the autocovariances with lags 1 and 2 as the summary statistics. Note that since the rows of `x` correspond to independent simulations, we have to tell this numpy function to take row-wise means by the keyword argument `axis=1`:

```
def autocov(x, lag=1):
    C = np.mean(x[:,lag:] * x[:,:-lag], axis=1)
    return C
```

As is familiar by now, a `Summary` node is defined by giving the autocovariance function and the simulated data (which includes the observed as well):

```
S1 = elfi.Summary(autocov, Y)
S2 = elfi.Summary(autocov, Y, 2)  # the optional keyword lag is given the value 2
```

Here, we choose the discrepancy as the common Euclidean L2-distance. ELFI can use many common distances directly from `scipy.spatial.distance` like this:

```
# Finish the model with the final node that calculates the squared distance (S1_sim-S1_
↪obs)**2 + (S2_sim-S2_obs)**2
d = elfi.Distance('euclidean', S1, S2)
```

One may wish to use a distance function that is unavailable in `scipy.spatial.distance`. ELFI supports defining a custom distance/discrepancy functions as well (see the documentation for `elfi.Distance` and `elfi.Discrepancy`).

Now that the inference model is defined, ELFI can visualize the model as a DAG.

```
elfi.draw(d)  # just give it a node in the model, or the model itself (d.model)
```

Note: You will need the Graphviz software as well as the graphviz Python package (https://pypi.python.org/pypi/graphviz) for drawing this.

1.6.3 Modifying the model

Although the above definition is perfectly valid, let's use the same priors as in Marin et al. (2012) that guarantee that the problem will be identifiable (loosely speaking, the likelihood willl have just one mode). Marin et al. used priors for which $-2 < \theta_1 < 2$ with $\theta_1 + \theta_2 > -1$ and $\theta_1 - \theta_2 < 1$ i.e. the parameters are sampled from a triangle (see below).

Note: By default all created nodes (even independent ones) will belong to the same ELFI model. It's good practice to always check with *elfi.draw* that the result looks as intended. A new default model can be started and set with the call *elfi.new_model()*. One can also create a new model with *my_model = elfi.ElfiModel()* and pass this as a keyword argument *model=my_model* when creating new nodes. Several ELFI models can coexist.

Custom priors

In ELFI, custom distributions can be defined similar to distributions in `scipy.stats` (i.e. they need to have at least the `rvs` method implemented for the simplest algorithms). To be safe they can inherit `elfi.Distribution` which defines the methods needed. In this case we only need these for sampling, so implementing a static `rvs` method suffices. As was in the context of simulators, it is important to accept the keyword argument `random_state`, which is needed for ELFI's internal book-keeping of pseudo-random number generation. Also the `size` keyword is needed (which in the simple cases is the same as the `batch_size` in the simulator definition).

```
# define prior for t1 as in Marin et al., 2012 with t1 in range [-b, b]
class CustomPrior_t1(elfi.Distribution):
    def rvs(b, size=1, random_state=None):
        u = scipy.stats.uniform.rvs(loc=0, scale=1, size=size, random_state=random_state)
        t1 = np.where(u<0.5, np.sqrt(2.*u)*b-b, -np.sqrt(2.*(1.-u))*b+b)
        return t1

# define prior for t2 conditionally on t1 as in Marin et al., 2012, in range [-a, a]
class CustomPrior_t2(elfi.Distribution):
    def rvs(t1, a, size=1, random_state=None):
        locs = np.maximum(-a-t1, t1-a)
        scales = a - locs
        t2 = scipy.stats.uniform.rvs(loc=locs, scale=scales, size=size, random_
```

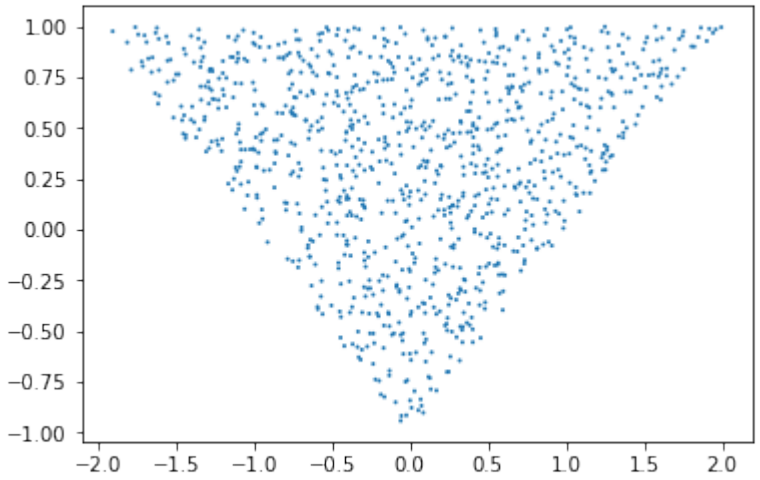
(continues on next page)

(continued from previous page)

```
↪state=random_state)
        return t2
```

These indeed sample from a triangle:

```
t1_1000 = CustomPrior_t1.rvs(2, 1000)
t2_1000 = CustomPrior_t2.rvs(t1_1000, 1, 1000)
plt.scatter(t1_1000, t2_1000, s=4, edgecolor='none');
# plt.plot([0, 2, -2, 0], [-1, 1, 1, -1], 'b')  # outlines of the triangle
```



Let's change the earlier priors to the new ones in the inference model:

```
t1.become(elfi.Prior(CustomPrior_t1, 2))
t2.become(elfi.Prior(CustomPrior_t2, t1, 1))

elfi.draw(d)
```

Note that `t2` now depends on `t1`. Yes, ELFI supports hierarchy.

1.6.4 Inference with rejection sampling

The simplest ABC algorithm samples parameters from their prior distributions, runs the simulator with these and compares them to the observations. The samples are either accepted or rejected depending on how large the distance is. The accepted samples represent samples from the approximate posterior distribution.

In ELFI, ABC methods are initialized either with a node giving the distance, or with the `ElfiModel` object and the name of the distance node. Depending on the inference method, additional arguments may be accepted or required.

A common optional keyword argument, accepted by all inference methods, `batch_size` defines how many simulations are performed in each passing through the graph.

Another optional keyword is the seed. This ensures that the outcome will be always the same for the same data and model. If you leave it out, a random seed will be taken.

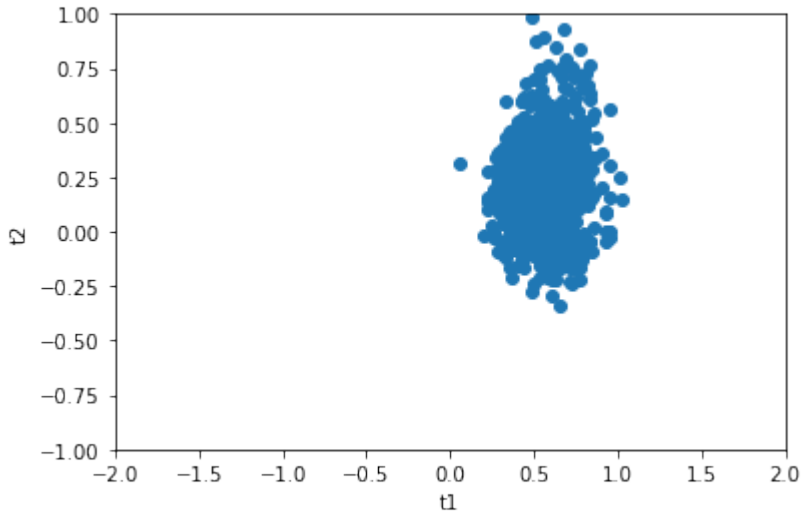
```
rej = elfi.Rejection(d, batch_size=10000, seed=seed)
```

Note: In Python, doing many calculations with a single function call can potentially save a lot of CPU time, depending on the operation. For example, here we draw 10000 samples from *t1*, pass them as input to *t2*, draw 10000 samples from *t2*, and then use these both to run 10000 simulations and so forth. All this is done in one passing through the graph and hence the overall number of function calls is reduced 10000-fold. However, this does not mean that batches should be as big as possible, since you may run out of memory, the fraction of time spent in function call overhead becomes insignificant, and many algorithms operate in multiples of *batch_size*. Furthermore, the *batch_size* is a crucial element for efficient parallelization (see the notebook on parallelization).

After the ABC method has been initialized, samples can be drawn from it. By default, rejection sampling in ELFI works in `quantile` mode i.e. a certain quantile of the samples with smallest discrepancies is accepted. The `sample` method requires the number of output samples as a parameter. Note that the simulator is then run `(N/quantile)` times. (Alternatively, the same behavior can be achieved by saying `n_sim=1000000`.)

The IPython magic command `%time` is used here to give you an idea of runtime on a typical personal computer. We will turn interactive visualization on so that if you run this on a notebook you will see the posterior forming from a prior distribution. In this case most of the time is spent in drawing.

```
N = 1000

vis = dict(xlim=[-2,2], ylim=[-1,1])

# You can give the sample method a `vis` keyword to see an animation how the prior␣
↪transforms towards the
# posterior with a decreasing threshold.
%time result = rej.sample(N, quantile=0.01, vis=vis)
```




```
CPU times: user 1.89 s, sys: 173 ms, total: 2.06 s
Wall time: 2.13 s
```

The `sample` method returns a `Sample` object, which contains several attributes and methods. Most notably the attribute `samples` contains an `OrderedDict` (i.e. an ordered Python dictionary) of the posterior numpy arrays for all the model parameters (`elfi.Priors` in the model). For rejection sampling, other attributes include e.g. the `threshold`, which is the threshold value resulting in the requested quantile.

```
result.samples['t1'].mean()
```

```
0.55600915483879665
```

The `Sample` object includes a convenient `summary` method:

```
result.summary()
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 100000
Threshold: 0.117
Sample means: t1: 0.556, t2: 0.219
```

Rejection sampling can also be performed with using a threshold or total number of simulations. Let's define here threshold. This means that all draws from the prior for which the generated distance is below the threshold will be accepted as samples. Note that the simulator will run as long as it takes to generate the requested number of samples.

```
%time result2 = rej.sample(N, threshold=0.2)

print(result2)  # the Sample object's __str__ contains the output from summary()
```

```
CPU times: user 221 ms, sys: 40 ms, total: 261 ms
Wall time: 278 ms
Method: Rejection
Number of samples: 1000
Number of simulations: 40000
Threshold: 0.185
Sample means: t1: 0.555, t2: 0.223
```

1.6.5 Iterative advancing

Often it may not be practical to wait to the end before investigating the results. There may be time constraints or one may wish to check the results at certain intervals. For this, ELFI provides an iterative approach to advance the inference. First one sets the objective of the inference and then calls the `iterate` method.

Below is an example how to run the inference until the objective has been reached or a maximum of one second of time has been used.

```
# Request for 1M simulations.
rej.set_objective(1000, n_sim=1000000)

# We only have 1 sec of time and we are unsure if we will be finished by that time.
# So lets simulate as many as we can.

time0 = time.time()
time1 = time0 + 1
while not rej.finished and time.time() < time1:
    rej.iterate()
    # One could investigate the rej.state or rej.extract_result() here
    # to make more complicated stopping criterions

# Extract and print the result as it stands. It will show us how many simulations were␣
```

(continues on next page)

(continued from previous page)

```
↪generated.
print(rej.extract_result())
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 130000
Threshold: 0.104
Sample means: t1: 0.558, t2: 0.219
```

```
# We will see that it was not finished in 1 sec
rej.finished
```

```
False
```

We could continue from this stage just by continuing to call the `iterate` method. The `extract_result` will always give a proper result even if the objective was not reached.

Next we will look into how to store all the data that was generated so far. This allows us to e.g. save the data to disk and continue the next day, or modify the model and reuse some of the earlier data if applicable.

1.6.6 Storing simulated data

As the samples are already in numpy arrays, you can just say e.g. `np.save('t1_data.npy', result.samples['t1'])` to save them. However, ELFI provides some additional functionality. You may define a *pool* for storing all outputs of any node in the model (not just the accepted samples). Let's save all outputs for `t1`, `t2`, `S1` and `S2` in our model:

```
pool = elfi.OutputPool(['t1', 't2', 'S1', 'S2'])
rej = elfi.Rejection(d, batch_size=10000, pool=pool)

%time result3 = rej.sample(N, n_sim=1000000)
result3
```

```
CPU times: user 6.13 s, sys: 1.15 s, total: 7.29 s
Wall time: 8.57 s
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 1000000
Threshold: 0.0364
Sample means: t1: 0.557, t2: 0.211
```

The benefit of the pool is that you may reuse simulations without having to resimulate them. Above we saved the summaries to the pool, so we can change the distance node of the model without having to resimulate anything. Let's do that.

```
# Replace the current distance with a cityblock (manhattan) distance and recreate the␣
↪inference
d.become(elfi.Distance('cityblock', S1, S2, p=1))
rej = elfi.Rejection(d, batch_size=10000, pool=pool)
```

(continues on next page)

(continued from previous page)

```
%time result4 = rej.sample(N, n_sim=1000000)
result4
```

```
CPU times: user 161 ms, sys: 2.84 ms, total: 163 ms
Wall time: 167 ms
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 1000000
Threshold: 0.0456
Sample means: t1: 0.56, t2: 0.214
```

Note the significant saving in time, even though the total number of considered simulations stayed the same.

We can also continue the inference by increasing the total number of simulations and only have to simulate the new ones:

```
%time result5 = rej.sample(N, n_sim=1200000)
result5
```

```
CPU times: user 1.14 s, sys: 185 ms, total: 1.33 s
Wall time: 1.34 s
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 1200000
Threshold: 0.0415
Sample means: t1: 0.56, t2: 0.216
```

Above the results were saved into a python dictionary. If you store a lot of data to dictionaries, you will eventually run out of memory. ELFI provides an alternative pool that, by default, saves the outputs to standard numpy .npy files:

```
arraypool = elfi.ArrayPool(['t1', 't2', 'Y', 'd'])
rej = elfi.Rejection(d, batch_size=10000, pool=arraypool)
%time result5 = rej.sample(100, threshold=0.3)
```

```
CPU times: user 66.6 ms, sys: 70.3 ms, total: 137 ms
Wall time: 175 ms
```

This stores the simulated data in binary `npy` format under `arraypool.path`, and can be loaded with `np.load`.

```
# Let's flush the outputs to disk (alternatively you can save or close the pool) so that␣
↪we can read the .npy files.
arraypool.flush()

import os
print('Files in', arraypool.path, 'are', os.listdir(arraypool.path))
```

```
Files in pools/arraypool_4290044000 are ['d.npy', 't1.npy', 't2.npy', 'Y.npy']
```

Now lets load all the parameters `t1` that were generated with numpy:

```
np.load(arraypool.path + '/t1.npy')
```

```
array([ 0.42, -1.15,  1.3 , ...,  0.64,  1.06, -0.47])
```

We can also close (or save) the whole pool if we wish to continue later:

```
arraypool.close()
name = arraypool.name
print(name)
```

```
arraypool_4290044000
```

And open it up later to continue where we were left. We can open it using its name:

```
arraypool = elfi.ArrayPool.open(name)
print('This pool has', len(arraypool), 'batches')

# This would give the contents of the first batch
# arraypool[0]
```

```
This pool has 1 batches
```

You can delete the files with:

```
arraypool.delete()

# verify the deletion
try:
    os.listdir(arraypool.path)

except FileNotFoundError:
    print("The directory is removed")
```

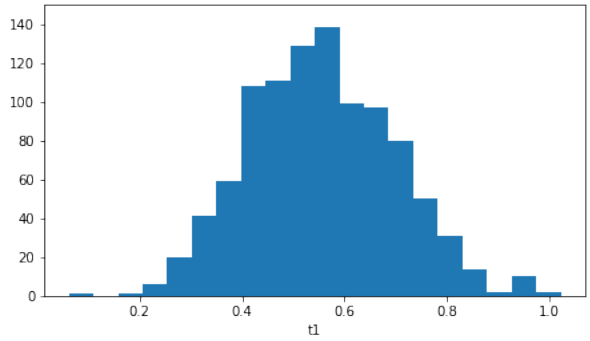
```
The directory is removed
```

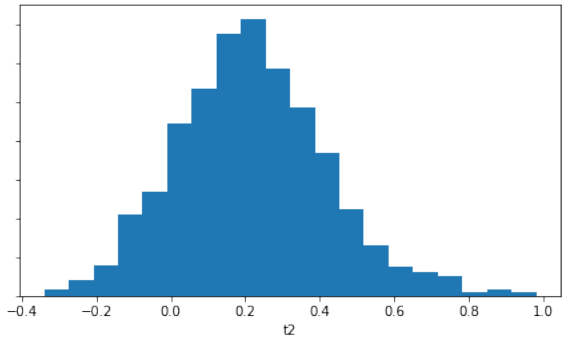
1.6.7 Visualizing the results

Instances of `Sample` contain methods for some basic plotting (these are convenience methods to plotting functions defined under `elfi.visualization`).

For example one can plot the marginal distributions:

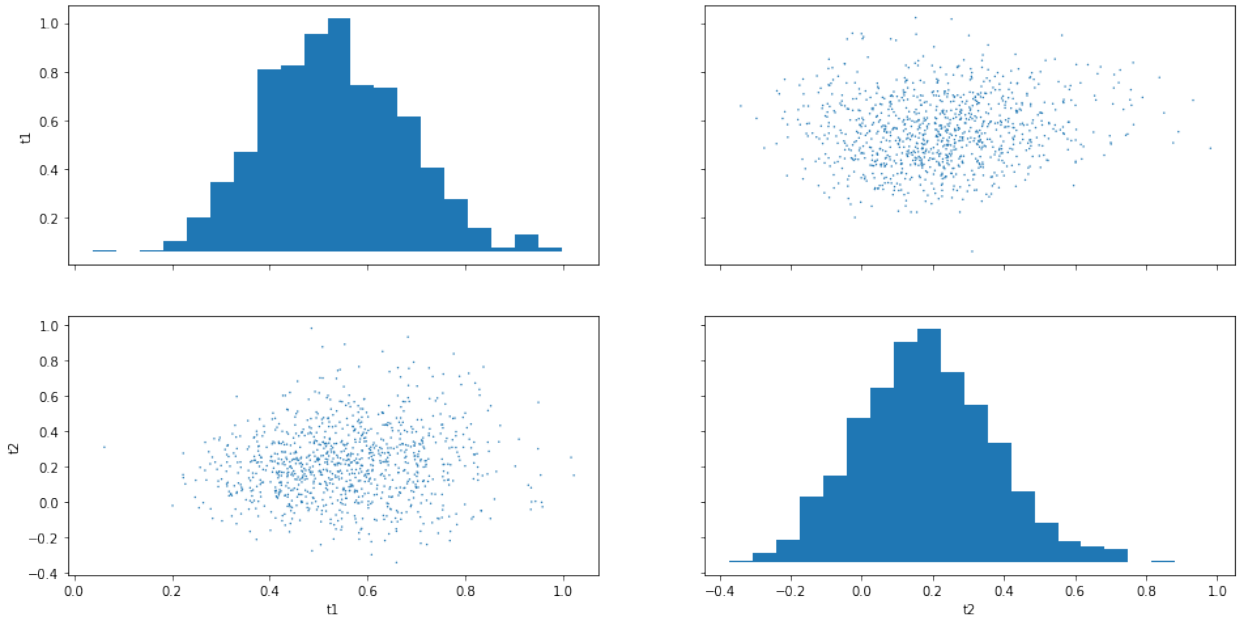
```
result.plot_marginals();
```





Often "pairwise relationships" are more informative:

```
result.plot_pairs();
```



Note that if working in a non-interactive environment, you can use e.g. `plt.savefig('pairs.png')` after an ELFI plotting command to save the current figure to disk.

1.6.8 Sequential Monte Carlo ABC

Rejection sampling is quite inefficient, as it does not learn from its history. The sequential Monte Carlo (SMC) ABC algorithm does just that by applying importance sampling: samples are *weighed* according to the resulting discrepancies and the next *population* of samples is drawn near to the previous using the weights as probabilities.

For evaluating the weights, SMC ABC needs to be able to compute the probability density of the generated parameters. In our MA2 example we used custom priors, so we have to specify a `pdf` function by ourselves. If we used standard priors, this step would not be needed. Let's modify the prior distribution classes:

```
# define prior for t1 as in Marin et al., 2012 with t1 in range [-b, b]
class CustomPrior_t1(elfi.Distribution):
    def rvs(b, size=1, random_state=None):
        u = scipy.stats.uniform.rvs(loc=0, scale=1, size=size, random_state=random_state)
```

(continues on next page)

(continued from previous page)

```
        t1 = np.where(u<0.5, np.sqrt(2.*u)*b-b, -np.sqrt(2.*(1.-u))*b+b)
        return t1

    def pdf(x, b):
        p = 1./b - np.abs(x) / (b*b)
        p = np.where(p < 0., 0., p)  # disallow values outside of [-b, b] (affects␣
↪weights only)
        return p


# define prior for t2 conditionally on t1 as in Marin et al., 2012, in range [-a, a]
class CustomPrior_t2(elfi.Distribution):
    def rvs(t1, a, size=1, random_state=None):
        locs = np.maximum(-a-t1, t1-a)
        scales = a - locs
        t2 = scipy.stats.uniform.rvs(loc=locs, scale=scales, size=size, random_
↪state=random_state)
        return t2

    def pdf(x, t1, a):
        locs = np.maximum(-a-t1, t1-a)
        scales = a - locs
        p = scipy.stats.uniform.pdf(x, loc=locs, scale=scales)
        p = np.where(scales>0., p, 0.)  # disallow values outside of [-a, a] (affects␣
↪weights only)
        return p


# Redefine the priors
t1.become(elfi.Prior(CustomPrior_t1, 2, model=t1.model))
t2.become(elfi.Prior(CustomPrior_t2, t1, 1))
```

Run SMC ABC

In ELFI, one can setup a SMC ABC sampler just like the Rejection sampler:

```
smc = elfi.SMC(d, batch_size=10000, seed=seed)
```

For sampling, one has to define the number of output samples, the number of populations and a *schedule* i.e. a list of thresholds to use for each population. In essence, a population is just refined rejection sampling.

```
N = 1000
schedule = [0.7, 0.2, 0.05]
%time result_smc = smc.sample(N, schedule)
```

```
INFO:elfi.methods.parameter_inference:---------------- Starting round 0 ----------------
INFO:elfi.methods.parameter_inference:---------------- Starting round 1 ----------------
INFO:elfi.methods.parameter_inference:---------------- Starting round 2 ----------------
```

```
CPU times: user 1.72 s, sys: 183 ms, total: 1.9 s
Wall time: 1.65 s
```

We can have summaries and plots of the results just like above:

```
result_smc.summary(all=True)
```

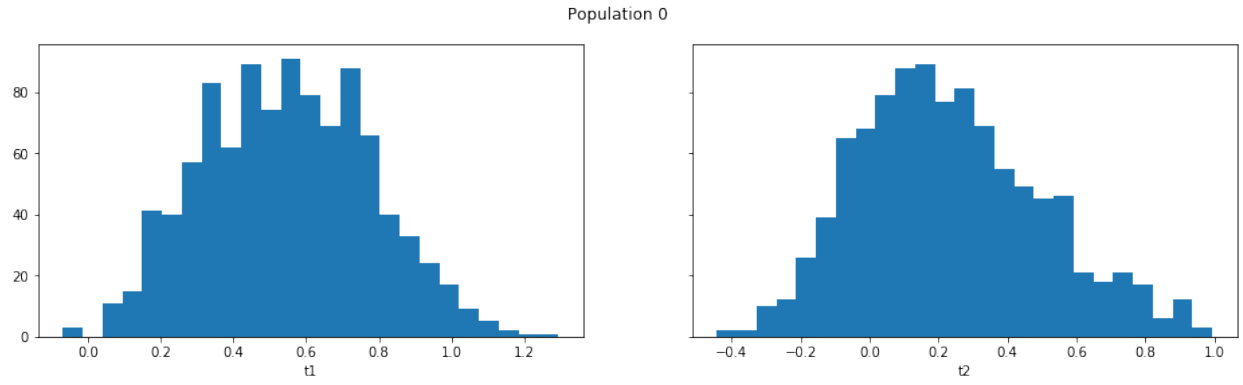
```
Method: SMC
Number of samples: 1000
Number of simulations: 170000
Threshold: 0.0493
Sample means: t1: 0.554, t2: 0.229

Population 0:
Method: Rejection within SMC-ABC
Number of samples: 1000
Number of simulations: 10000
Threshold: 0.488
Sample means: t1: 0.547, t2: 0.232

Population 1:
Method: Rejection within SMC-ABC
Number of samples: 1000
Number of simulations: 20000
Threshold: 0.172
Sample means: t1: 0.562, t2: 0.22

Population 2:
Method: Rejection within SMC-ABC
Number of samples: 1000
Number of simulations: 140000
Threshold: 0.0493
Sample means: t1: 0.554, t2: 0.229
```

Or just the means:

```
result_smc.sample_means_summary(all=True)
```

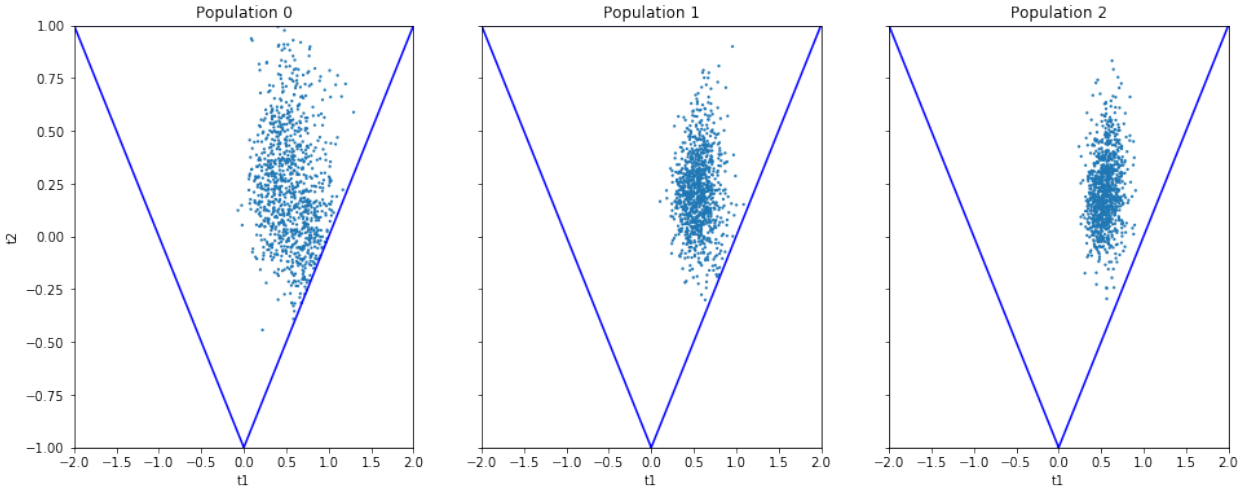
```
Sample means for population 0: t1: 0.547, t2: 0.232
Sample means for population 1: t1: 0.562, t2: 0.22
Sample means for population 2: t1: 0.554, t2: 0.229
```

```
result_smc.plot_marginals(all=True, bins=25, figsize=(8, 2), fontsize=12)
```



Obviously one still has direct access to the samples as well, which allows custom plotting:

```
n_populations = len(schedule)
fig, ax = plt.subplots(ncols=n_populations, sharex=True, sharey=True, figsize=(16,6))

for i, pop in enumerate(result_smc.populations):
    s = pop.samples
    ax[i].scatter(s['t1'], s['t2'], s=5, edgecolor='none');
    ax[i].set_title("Population {}".format(i));
    ax[i].plot([0, 2, -2, 0], [-1, 1, 1, -1], 'b')
    ax[i].set_xlabel('t1');
ax[0].set_ylabel('t2');
ax[0].set_xlim([-2, 2])
ax[0].set_ylim([-1, 1]);
```



It can be seen that the populations iteratively concentrate more and more around the true parameter values. Note, however, that samples from SMC are weighed, and the weights should be accounted for when interpreting the results. ELFI does this automatically when computing the mean, for example.

1.6.9 What next?

If you want to play with different ABC algorithms, no need to repeat the simulator definitions etc. from this notebook. ELFI provides a convenient way to quickly get you going:

```
from elfi.examples import ma2
ma2_model = ma2.get_model()
```

This constructs the same ELFI graph discussed in this tutorial. The example is self-contained and includes implementations of all relevant operations.

```
elfi.draw(ma2_model)
```

```
elfi.Rejection(ma2_model['d'], batch_size=10000).sample(1000)
```

```
Method: Rejection
Number of samples: 1000
```

(continues on next page)

(continued from previous page)

```
Number of simulations: 100000
Threshold: 0.128
Sample means: t1: 0.719, t2: 0.412
```

ELFI ships with many other common example cases from ABC literature, and they all include the `get_model` method mentioned above. The source codes of these examples are also good for learning more about best practices with ELFI.

```
!ls {elfi.examples.__path__[0] + '/*.py'} | xargs basename
```

```
__init__.py
bdm.py
bignk.py
gauss.py
gnk.py
lotka_volterra.py
ma2.py
ricker.py
```

That's it! See the other documentation for more advanced topics on e.g. BOLFI, external simulators and parallelization.

1.7 Adaptive distance

ABC provides means to sample an approximate posterior distribution over unknown parameters based on comparison between observed and simulated data. This comparison is often based on distance between features that summarise the data and are informative about the parameter values.

Here we assume that the summaries calculated based on observed and simulated data are compared based on weighted distance with weight $w_i = 1/\sigma_i$ calculated based on their standard deviation σ_i. This ensures that the selected summaries to have an equal contribution in the distance between observed and simulated data.

This notebook studies adaptive distance SMC-ABC where σ_i and w_i are recalculated between SMC iterations as proposed in [*1*].

```
import numpy as np
import scipy.stats as ss
import matplotlib.pyplot as plt
%matplotlib inline
```

```
import elfi
```

1.7.1 Example 1:

Assume we have an unknown parameter with prior distribution $\theta \sim U(0, 50)$ and two simulator outputs $S_1 \sim N(\theta, 1)$ and $S_2 \sim N(\theta, 100)$ whose observed values are 20.

```
def simulator(mu, batch_size=1, random_state=None):

    batches_mu = np.asarray(mu).reshape((-1,1))

    obs_1 = ss.norm.rvs(loc=batches_mu, scale=1, random_state=random_state).reshape((-1,
```

(continues on next page)

(continued from previous page)

```
↪1))
    obs_2 = ss.norm.rvs(loc=batches_mu, scale=100, random_state=random_state).reshape((-
↪1,1))

    return np.hstack((obs_1, obs_2))
```

```
observed_data = np.array([20,20])[None,:]
```

Here the simulator outputs are both informative about the unknown model parameter, but S_2 has more observation noise than S_1. We do not calculate separate summaries in this example, but compare observed and simulated data based on these two variables.

Euclidean distance between observed and simulated outputs or summaries can be used to find parameter values that could produce the observed data. Here we describe dependencies between the unknown parameter value and observed distances as an ELFI model `m` and sample the approximate posterior distribution with the rejection sampler.

```
m = elfi.new_model()
theta = elfi.Prior(ss.uniform, 0, 50, model=m)
sim = elfi.Simulator(simulator, theta, observed=observed_data)
d = elfi.Distance('euclidean', sim)
```

```
rej = elfi.Rejection(d, batch_size=10000, seed=123)
```

Let us sample 100 parameters with `quantile=0.01`. This means that we sample 10000 candidate parameters from the prior distribution and take the 100 parameters that produce simulated data closest to the observed data.

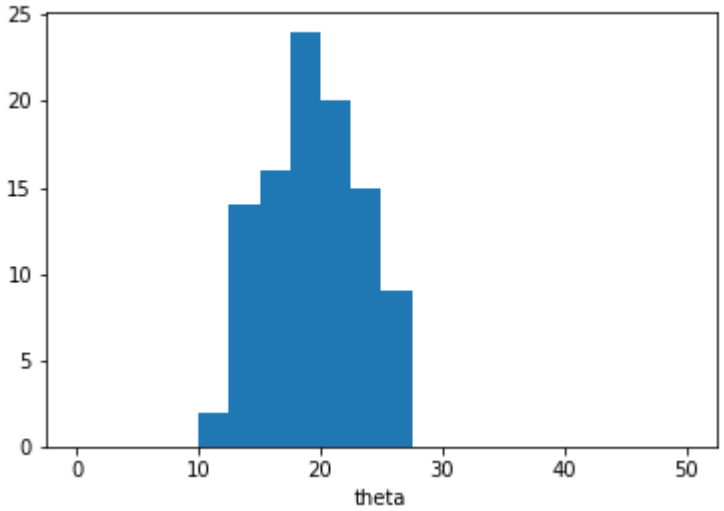
```
sample = rej.sample(100, quantile=0.01)
```

```
Progress [==================================================] 100.0% Complete
```

```
sample
```

```
Method: Rejection
Number of samples: 100
Number of simulations: 10000
Threshold: 6.66
Sample means: theta: 19.6
```

```
plt.hist(sample.samples_array,range=(0,50),bins=20)
plt.xlabel('theta');
```



The approximate posterior sample is concentrated around $\theta = 20$ as expected in this example. However the sample distribution is much wider than we would observe in case the sample was selected based on S_1 alone.

Now let us test adaptive distance in the same example.

First we switch the distance node `d` to an adaptive distance node and initialise adaptive distance SMC-ABC. Initialisation is identical to the rejection sampler, and here we use the same batch size and seed as earlier, so that the methods are presented with the exact same candidate parameters.

```
d.become(elfi.AdaptiveDistance(sim))
```

```
ada_smc = elfi.AdaptiveDistanceSMC(d, batch_size=10000, seed=123)
```

Since this is an iterative method, we must decide both sample size (`n_samples`) and how many populations are sampled (`rounds`). In addition we can decide the α quantile (`quantile`) used in estimation.

Each population with `n_samples` parameter values is sampled as follows: 1. `n_samples/quantile` parameters are sampled from the current proposal distribution with acceptance threshold determined based on the previous population and 2. the distance measure is updated based on the observed sample and `n_samples` with the smallest distance are selected as the new population. The first population is sampled from the prior distribution and all samples are accepted in step 1.

Here we sample one population with `quantile=0.01`. This means that the total simulation count will be the same as with the rejection sampler, but now the distance function is updated based on the 10000 simulated observations, and the 100 parameters included in the posterior sample are selected based on the new distance measure.

```
sample_ada = ada_smc.sample(100, 1, quantile=0.01)
```

```
ABC-SMC Round 1 / 1
Progress [==================================================] 100.0% Complete
```

```
sample_ada
```

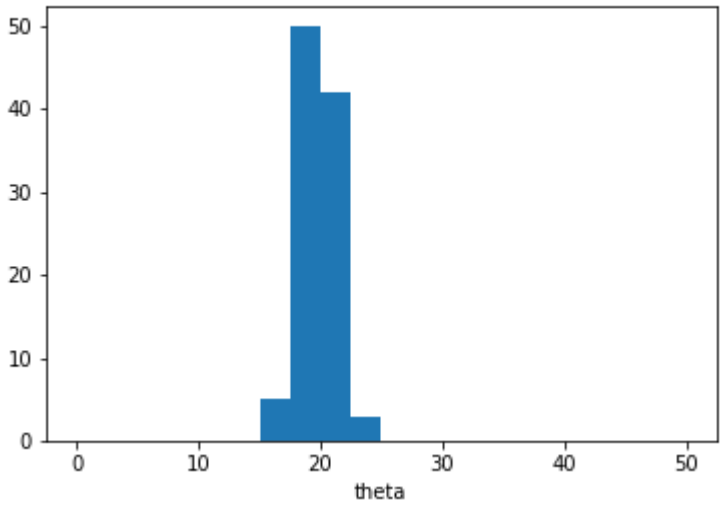
```
Method: AdaptiveDistanceSMC
Number of samples: 100
Number of simulations: 10000
```

(continues on next page)

(continued from previous page)

```
Threshold: 0.462
Sample means: theta: 19.8
```

```
plt.hist(sample_ada.samples_array,range=(0,50),bins=20)
plt.xlabel('theta');
```



We see that the posterior distribution over unknown parameter values is narrower than in the previous example. This is because the simulator outputs are now normalised based on their estimated standard deviation.

We can see w_1 and w_2:

```
sample_ada.adaptive_distance_w
```

```
[array([0.06940134, 0.0097677 ])]
```

1.7.2 Example 2:

This is the normal distribution example presented in [*1*].

Here we have an unknown parameter with prior distribution $\theta \sim N(0, 100)$ and two simulator outputs $S_1 \sim N(\theta, 0.1)$ and $S_2 \sim N(1, 1)$ whose observed values are 0.

```
def simulator(mu, batch_size=1, random_state=None):

    batches_mu = np.asarray(mu).reshape((-1,1))

    obs_1 = ss.norm.rvs(loc=batches_mu, scale=0.1, random_state=random_state).reshape((-
↪1,1))
    obs_2 = ss.norm.rvs(loc=1, scale=1, size=batch_size, random_state=random_state).
↪reshape((-1,1))

    return np.hstack((obs_1, obs_2))
```

```
observed_data = np.array([0,0])[None,:]
```

S_1 is now informative and S_2 uninformative about the unknown parameter value, and we note that between the two output variables, S_1 has larger variance under the prior predictive distribution. This means that normalisation estimated based on output data observed in the initial round or based on a separate sample would not work well in this example.

Let us define a new model and initialise adaptive distance SMC-ABC.

```
m = elfi.new_model()
theta = elfi.Prior(ss.norm, 0, 100, model=m)
sim = elfi.Simulator(simulator, theta, observed=observed_data)
d = elfi.AdaptiveDistance(sim)
```

```
ada_smc = elfi.AdaptiveDistanceSMC(d, batch_size=2000, seed=123)
```

Next we sample 1000 parameter values in 5 rounds with the default `quantile=0.5` which is recommended in sequential estimation [*1*]:

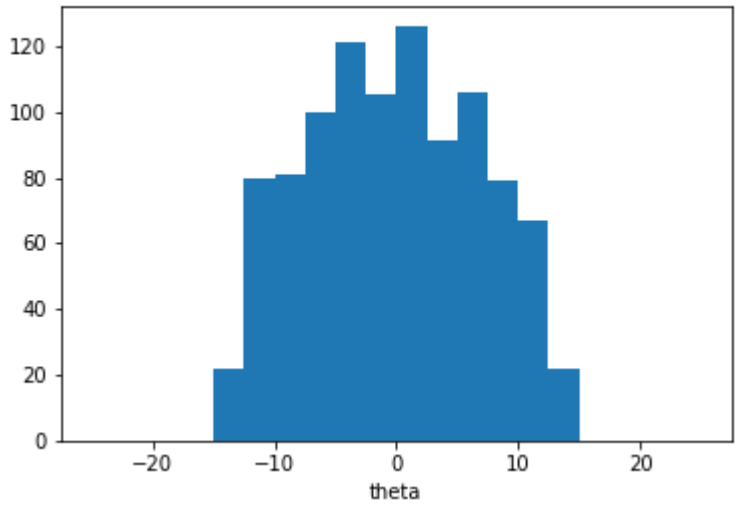
```
sample_ada = ada_smc.sample(1000, 5)
```

```
ABC-SMC Round 1 / 5
Progress [==================================================] 100.0% Complete
ABC-SMC Round 2 / 5
Progress [==================================================] 100.0% Complete
ABC-SMC Round 3 / 5
Progress [==================================================] 100.0% Complete
ABC-SMC Round 4 / 5
Progress [==================================================] 100.0% Complete
ABC-SMC Round 5 / 5
Progress [==================================================] 100.0% Complete
```

```
sample_ada
```

```
Method: AdaptiveDistanceSMC
Number of samples: 1000
Number of simulations: 32000
Threshold: 0.925
Sample means: theta: -0.195
```

```
plt.hist(sample_ada.samples_array, range=(-25,25), bins=20)
plt.xlabel(theta);
```



The sample distribution is concentrated around $\theta = 0$ but wider than could be expected. However we can continue the iterative estimation process. Here we sample two more populations:

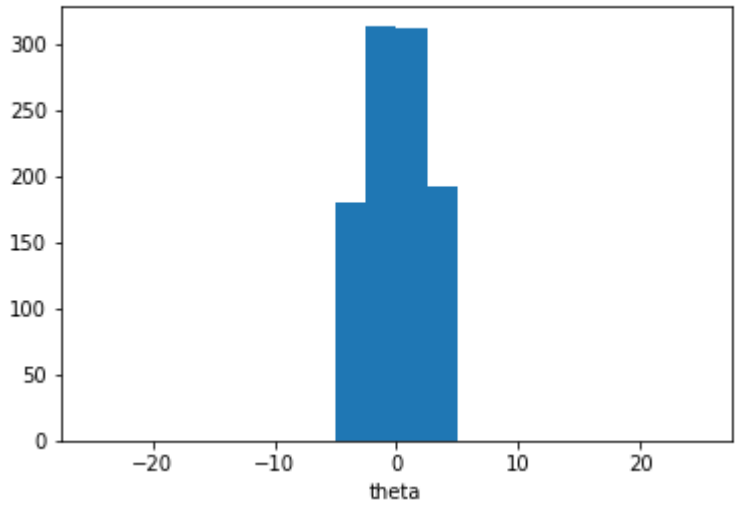
```
sample_ada = ada_smc.sample(1000, 2)
```

```
ABC-SMC Round 6 / 7
Progress [==================================================] 100.0% Complete
ABC-SMC Round 7 / 7
Progress [==================================================] 100.0% Complete
```

```
sample_ada
```

```
Method: AdaptiveDistanceSMC
Number of samples: 1000
Number of simulations: 48000
Threshold: 0.868
Sample means: theta: 0.0183
```

```
plt.hist(sample_ada.samples_array, range=(-25,25), bins=20)
plt.xlabel('theta');
```



We observe that the sample mean is now closer to zero and the sample distribution is narrower.

Let us examine w_1 and w_2:

```
sample_ada.adaptive_distance_w
```

```
[array([0.01023228, 1.00584519]),
 array([0.00921258, 0.99287166]),
 array([0.01201937, 0.99365522]),
 array([0.02217631, 0.98925365]),
 array([0.04355987, 1.00076738]),
 array([0.07863284, 0.9971017 ]),
 array([0.13892778, 1.00929049])]
```

We can see that w_2 (second column) is constant across iterations whereas w_1 increases as the method learns more about possible parameter values and the proposal distribution becomes more concentrated around $\theta = 0$.

1.7.3 Notes

The adaptive distance SMC-ABC method demonstrated in this notebook normalises simulator outputs or summaries calculated based on simulator output based on their estimated standard deviation under the proposal distribution in each iteration. This ensures that all outputs or summaries have an equal contribution to the distance between simulated and observed data in all iterations.

It is important to note that the method does not evaluate whether outputs or summaries are needed or informative. In both examples studied in this notebook, results would improve if inference was carried out based on S_1 alone. Hence one should choose the summaries used in adaptive distance SMC-ABC with the usual care. ELFI tools that aid in the selection process are discussed in the diagnostics notebook available here.

1.7.4 Reference

[1] Prangle D (2017). Adapting the ABC Distance Function. Bayesian Analysis 12(1): 289-309, 2017. https://projecteuclid.org/euclid.ba/1460641065

1.8 Adaptive Approximate Bayesian Computation Tolerance Selection

To use ABC-SMC one has to choose the tolerance threshold or acceptance quantiles beforehand to run the method. However, Simola et al. [1] describe an adaptive strategy for selecting the threshold based on density ratios of rejection ABC posteriors. This tutorial teaches you how to use the adaptive threshold selection method.

```
import numpy as np
# from scipy.stats import norm
import scipy.stats as ss
import elfi
import logging
import matplotlib
import matplotlib.pyplot as plt

from scipy.stats import gaussian_kde

%matplotlib inline

# Set an arbitrary global seed to keep the randomly generated quantities the same
seed = 10
np.random.seed(seed)
```

We reproduce the Example 1 from [1] as a test case for `AdaptiveThresholdSMC`.

```
def gaussian_mixture(theta, batch_size=1, random_state=None):
    sigma1 = 1
    sigma2 = np.sqrt(0.01)
    sigmas = np.array((sigma1, sigma2))
    mixture_prob = 0.5
    random_state = random_state or np.random

    scale_array = random_state.choice(sigmas,
                                      size=batch_size,
                                      replace=True,
                                      p=np.array((mixture_prob, 1-mixture_prob)))
    observation = ss.norm.rvs(loc=theta,
                              scale=scale_array,
                              size=batch_size,
                              random_state=random_state)

    return observation
```

```
yobs = 0
model = elfi.ElfiModel()
elfi.Prior('uniform', -10, 20, name='theta', model=model)
```

(continues on next page)

(continued from previous page)

```
elfi.Simulator(gaussian_mixture, model['theta'], observed=yobs, name='GM')
elfi.Distance('euclidean', model['GM'], name='d');
```

```
smc = elfi.SMC(model['d'], batch_size=500, seed=1)
thresholds = [1., 0.5013, 0.2519, 0.1272, 0.0648, 0.0337, 0.0181, 0.0102, 0.0064, 0.0025]
smc_samples = smc.sample(1000, thresholds=thresholds)
```

```
ABC-SMC Round 1 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 2 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 3 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 4 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 5 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 6 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 7 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 8 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 9 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 10 / 10
Progress [==================================================] 100.0% Complete
```

Adaptive threshold selection ABC (`elfi.AdaptiveThresholdSMC`) can be used in similar fashion as `elfi.SMC`. One does not need to provide a list of thresholds but user can set densityratio-based termination condition (`q_threshold`) and a limit for the number of iterations (`max_iter`).

```
adaptive_smc = elfi.AdaptiveThresholdSMC(model['d'], batch_size=500, seed=2, q_
↪threshold=0.995)
adaptive_smc_samples = adaptive_smc.sample(1000, max_iter=10)
```

```
ABC-SMC Round 1 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 2 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 3 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 4 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 5 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 6 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 7 / 10
Progress [==================================================] 100.0% Complete
ABC-SMC Round 8 / 10
Progress [==================================================] 100.0% Complete
```

We compare visually the approximated posterior and the true posterior, which in this case is available.

```
def gaussian_mixture_density(theta, sigma_1=1, sigma_2=0.1):
    y = 0.5 * ss.norm.pdf(theta, loc=0, scale=sigma_1) + 0.5 * ss.norm.pdf(theta, loc=0,␣
↪scale=sigma_2)
    return y
```

```
print(smc_samples)
print(adaptive_smc_samples)
```

```
Method: SMC
Number of samples: 1000
Number of simulations: 1352000
Threshold: 0.0025
Sample means: theta: 0.0181

Method: AdaptiveThresholdSMC
Number of samples: 1000
Number of simulations: 49500
Threshold: 0.236
Sample means: theta: -0.042
```

We compute Kernel density estimates of the posteriors based on the approximate posterior samples and visualise them in a density plot.

```
smc_posteriorpdf = gaussian_kde(smc_samples.samples_array[:,0])
adaptive_smc_posteriorpdf = gaussian_kde(adaptive_smc_samples.samples_array[:,0])

reference_posteriorpdf = gaussian_mixture_density

xs = np.linspace(-3,3,200)
smc_posteriorpdf.covariance_factor = lambda : .25
smc_posteriorpdf._compute_covariance()
adaptive_smc_posteriorpdf.covariance_factor = lambda : .25
adaptive_smc_posteriorpdf._compute_covariance()
plt.figure(figsize=(16,10))
plt.plot(xs,smc_posteriorpdf(xs))
plt.plot(xs,adaptive_smc_posteriorpdf(xs))
plt.plot(xs,reference_posteriorpdf(xs))
plt.legend(('abc-smc', 'adaptive abc-smc', 'reference'));
```

[1] Simola, U., Cisewski-Kehe, J., Gutmann, M.U. and Corander, J. Adaptive Approximate Bayesian Computation Tolerance Selection, Bayesian Analysis 1(1):1-27, 2021

This tutorial is generated from a Jupyter notebook that can be found here.

1.9 Parallelization

Behind the scenes, ELFI can automatically parallelize the computational inference via different clients. Currently ELFI includes three clients:

- `elfi.clients.native` (activated by default): does not parallelize but makes it easy to test and debug your code.
- `elfi.clients.multiprocessing`: basic local parallelization using Python's built-in multiprocessing library
- `elfi.clients.ipyparallel`: ipyparallel based client that can parallelize from multiple cores up to a distributed cluster.

A client is activated by giving the name of the client to `elfi.set_client`.

This tutorial shows how to activate and use the `multiprocessing` or `ipyparallel` client with ELFI. The `ipyparallel` client supports parallelization from local computer up to a cluster environment. For local parallelization however, the `multiprocessing` client is simpler to use. Let's begin by importing ELFI and our example MA2 model from the tutorial.

```
import elfi
from elfi.examples import ma2
```

Let's get the model and plot it (requires graphviz)

```
model = ma2.get_model()
elfi.draw(model)
```

1.9.1 Multiprocessing client

The multiprocessing client allows you to easily use the cores available in your computer. You can activate it simply by

```
elfi.set_client('multiprocessing')
```

Any inference instance created **after** you have set the new client will automatically use it to perform the computations. Let's try it with our MA2 example model from the tutorial. When running the next command, take a look at the system monitor of your operating system; it should show that all of your cores are doing heavy computation simultaneously.

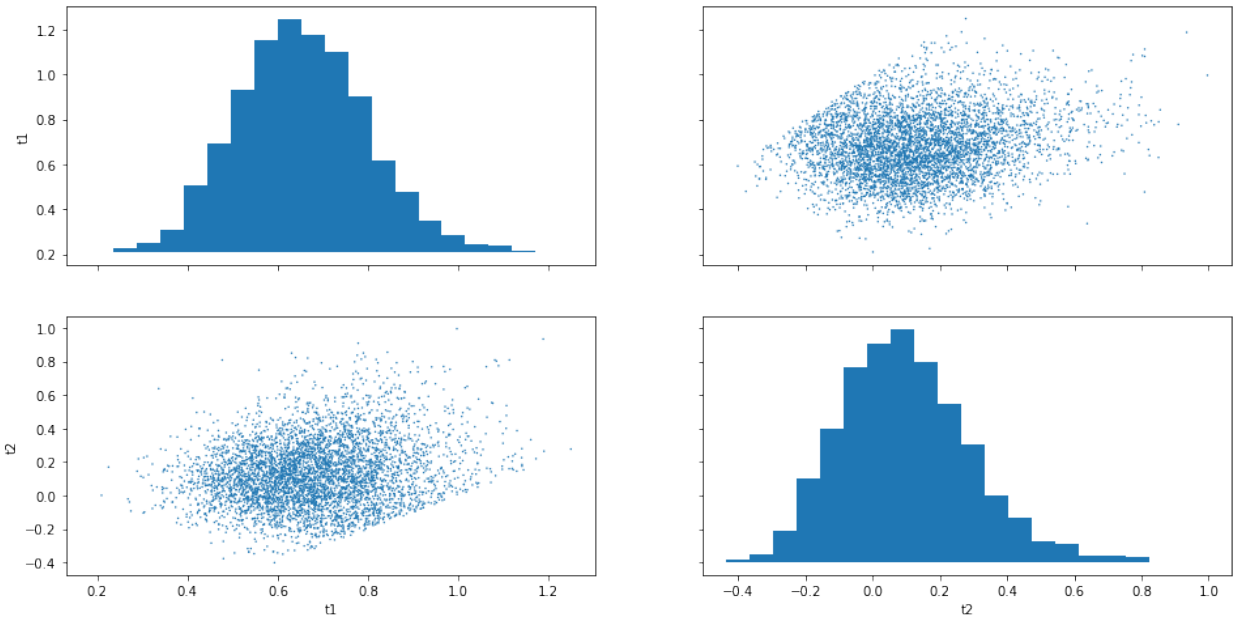
```
rej = elfi.Rejection(model, 'd', batch_size=10000, seed=20170530)
%time result = rej.sample(5000, n_sim=int(1e6))  # 1 million simulations
```

```
CPU times: user 298 ms, sys: 25.7 ms, total: 324 ms
Wall time: 3.93 s
```

And that is it. The result object is also just like in the basic case:

```
# Print the summary
result.summary()

import matplotlib.pyplot as plt
result.plot_pairs();
plt.show()
```

```
Method: Rejection
Number of samples: 5000
Number of simulations: 1000000
Threshold: 0.0826
Sample means: t1: 0.694, t2: 0.226
```



Note that for reproducibility a reference to the activated client is saved in the inference instance:

```
rej.client
```

```
<elfi.clients.multiprocessing.Client at 0x1a19c2f128>
```

If you want to change the client for an existing inference instance, you have to do something like this:

```
elfi.set_client('native')
rej.client = elfi.get_client()
rej.client
```

```
<elfi.clients.native.Client at 0x1a1d2a5cf8>
```

By default the multiprocessing client will use all cores on your system. This is not always desirable, as the operating system may prioritize some other process, leaving ELFI queuing for the promised resources. You can define some other number of processes like so:

```
elfi.set_client(elfi.clients.multiprocessing.Client(num_processes=3))
```

Note: The `multiprocessing` library may require additional care under Windows. If you receive a RuntimeError mentioning `freeze_support`, please include a call to `multiprocessing.freeze_support()`, see documentation.

1.9.2 Ipyparallel client

The `ipyparallel` client allows you to parallelize the computations to cluster environments. To use the `ipyparallel` client, you first have to create an `ipyparallel` cluster. Below is an example of how to start a local cluster to the background using 4 CPU cores:

```
!ipcluster start -n 4 --daemonize

# This is here just to ensure that ipcluster has enough time to start properly before␣
↪continuing
import time
time.sleep(10)
```

Note: The exclamation mark above is a Jupyter syntax for executing shell commands. You can run the same command in your terminal without the exclamation mark.

Tip: Please see the ipyparallel documentation (https://ipyparallel.readthedocs.io/en/latest/intro.html#getting-started) for more information and details for setting up and using ipyparallel clusters in different environments.

Running parallel inference with ipyparallel

After the cluster has been set up, we can proceed as usual. ELFI will take care of the parallelization from now on:

```
# Let's start using the ipyparallel client
elfi.set_client('ipyparallel')

rej = elfi.Rejection(model, 'd', batch_size=10000, seed=20170530)
%time result = rej.sample(5000, n_sim=int(5e6))  # 5 million simulations
```

```
CPU times: user 3.47 s, sys: 288 ms, total: 3.76 s
Wall time: 18.1 s
```

To summarize, the only thing that needed to be changed from the basic scenario was creating the `ipyparallel` cluster and enabling the `ipyparallel` client.

1.9.3 Working interactively with ipyparallel

If you are using the `ipyparallel` client from an interactive environment (e.g. jupyter notebook) there are some things to take care of. All imports and definitions must be visible to all `ipyparallel` engines. You can ensure this by writing a script file that has all the definitions in it. In a distributed setting, this file must be present in all remote workers running an `ipyparallel` engine.

However, you may wish to experiment in an interactive session, using e.g. a jupyter notebook. `ipyparallel` makes it possible to interactively define functions for ELFI model and send them to workers. This is especially useful if you work from a jupyter notebook. We will show a few examples. More information can be found from `` `ipyparallel`` documentation <http://ipyparallel.readthedocs.io/>`__.

In interactive sessions, you can change the model with built-in functionality without problems:

```
d2 = elfi.Distance('cityblock', model['S1'], model['S2'], p=1)

rej2 = elfi.Rejection(d2, batch_size=10000)
result2 = rej2.sample(1000, quantile=0.01)
```

But let's say you want to use your very own distance function in a jupyter notebook:

```
def my_distance(x, y):
    # Note that interactively defined functions must use full module names, e.g. numpy␣
↪instead of np
    return numpy.sum((x-y)**2, axis=1)

d3 = elfi.Distance(my_distance, model['S1'], model['S2'])
rej3 = elfi.Rejection(d3, batch_size=10000)
```

This function definition is not automatically visible for the `ipyparallel` engines if it is not defined in a physical file. The engines run in different processes and will not see interactively defined objects and functions. The below would therefore fail:

```
# This will fail if you try it!
# result3 = rej3.sample(1000, quantile=0.01)
```

Ipyparallel provides a way to manually `push` the new definition to the scopes of the engines from interactive sessions. Because `my_distance` also uses `numpy`, that must be imported in the engines as well:

```
# Get the ipyparallel client
ipyclient = elfi.get_client().ipp_client

# Import numpy in the engines (note that you cannot use "as" abbreviations, but must use␣
↪plain imports)
with ipyclient[:].sync_imports():
    import numpy

# Then push my_distance to the engines
ipyclient[:].push({'my_distance': my_distance});
```

```
importing numpy on engine(s)
```

The above may look a bit cumbersome, but now this works:

```
rej3.sample(1000, quantile=0.01)  # now this works
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 100000
Threshold: 0.0146
Sample means: t1: 0.693, t2: 0.233
```

However, a simpler solution to cases like this may be to define your functions in external scripts (see `elfi.examples.ma2`) and have the module files be available in the folder where you run your ipyparallel engines.

Remember to stop the ipcluster when done

```
!ipcluster stop
```

```
2018-04-24 19:14:56.997 [IPClusterStop] Stopping cluster [pid=39639] with [signal=
↪<Signals.SIGINT: 2>]
```

This tutorial is generated from a Jupyter notebook that can be found here.

1.10 BOLFI

In practice inference problems often have a complicated and computationally heavy simulator, and one simply cannot run it for millions of times. The Bayesian Optimization for Likelihood-Free Inference BOLFI framework is likely to prove useful in such situation: a statistical model (usually Gaussian process, GP) is created for the discrepancy, and its minimum is inferred with Bayesian optimization. This approach typically reduces the number of required simulator calls by several orders of magnitude.

This tutorial demonstrates how to use BOLFI to do LFI in ELFI.

```
import numpy as np
import scipy.stats
import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline
%precision 2

import logging
logging.basicConfig(level=logging.INFO)

# Set an arbitrary global seed to keep the randomly generated quantities the same
seed = 1
np.random.seed(seed)

import elfi
```

Although BOLFI is best used with complicated simulators, for demonstration purposes we will use the familiar MA2 model introduced in the basic tutorial, and load it from ready-made examples:

```
from elfi.examples import ma2
model = ma2.get_model(seed_obs=seed)
elfi.draw(model)
```

1.10.1 Fitting the surrogate model

Now we can immediately proceed with the inference. However, when dealing with a Gaussian process, it may be beneficial to take a logarithm of the discrepancies in order to reduce the effect that high discrepancies have on the GP. (Sometimes you may want to add a small constant to avoid very negative or even -Inf distances occurring especially if it is likely that there can be exact matches between simulated and observed data.) In ELFI such transformed node can be created easily:

```
log_d = elfi.Operation(np.log, model['d'])
```

As BOLFI is a more advanced inference method, its interface is also a bit more involved as compared to for example rejection sampling. But not much: Using the same graphical model as earlier, the inference could begin by defining a Gaussian process (GP) model, for which ELFI uses the GPy library. This could be given as an `elfi.GPyRegression` object via the keyword argument `target_model`. In this case, we are happy with the default that ELFI creates for us when we just give it each parameter some `bounds` as a dictionary.

Other notable arguments include the `initial_evidence`, which gives the number of initialization points sampled straight from the priors before starting to optimize the acquisition of points, `update_interval` which defines how often the GP hyperparameters are optimized, and `acq_noise_var` which defines the diagonal covariance of noise added to the acquired points. Note that in general BOLFI does not benefit from a `batch_size` higher than one, since the acquisition surface is updated after each batch (especially so if the noise is 0!).

```
bolfi = elfi.BOLFI(log_d, batch_size=1, initial_evidence=20, update_interval=10,
                   bounds={'t1':(-2, 2), 't2':(-1, 1)}, acq_noise_var={'t1':0.1, 't2':0.
↪1}, seed=seed)
```

Sometimes you may have some samples readily available. You could then initialize the GP model with a dictionary of previous results by giving `initial_evidence=result.outputs`.

The BOLFI class can now try to `fit` the surrogate model (the GP) to the relationship between parameter values and the resulting discrepancies. We'll request only 100 evidence points (including the `initial_evidence` defined above).

```
%time post = bolfi.fit(n_evidence=200)
```

```
INFO:elfi.methods.parameter_inference:BOLFI: Fitting the surrogate model...
INFO:elfi.methods.posteriors:Using optimized minimum value (-1.6146) of the GP␣
↪discrepancy mean function as a threshold
```

```
CPU times: user 1min 48s, sys: 1.29 s, total: 1min 50s
Wall time: 1min
```

(More on the returned `BolfiPosterior` object *below*.)

Note that in spite of the very few simulator runs, fitting the model took longer than any of the previous methods. Indeed, BOLFI is intended for scenarios where the simulator takes a lot of time to run.

The fitted `target_model` uses the GPy library, and can be investigated further:

```
bolfi.target_model
```

```
Name : GP regression
Objective : 151.86636065302943
Number of Parameters : 4
Number of Optimization Parameters : 4
Updates : True
```

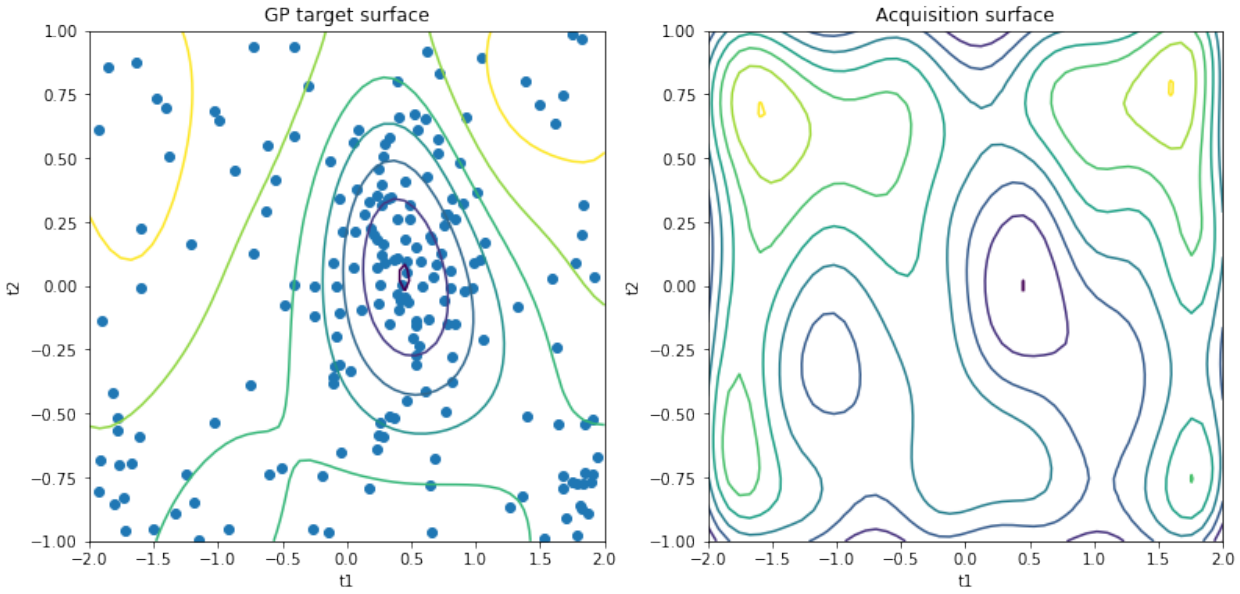
(continues on next page)

(continued from previous page)

```
Parameters:
  [1mGP_regression.          [0;0m  |           value  |  constraints  |     priors
  [1msum.rbf.variance        [0;0m  |  0.321697451372  |      +ve      |  Ga(0.024, 1)
  [1msum.rbf.lengthscale     [0;0m  |  0.541352150083  |      +ve      |   Ga(1.3, 1)
  [1msum.bias.variance       [0;0m  |  0.021827430988  |      +ve      |  Ga(0.006, 1)
  [1mGaussian_noise.variance[0;0m  |  0.183562040169  |      +ve      |
```

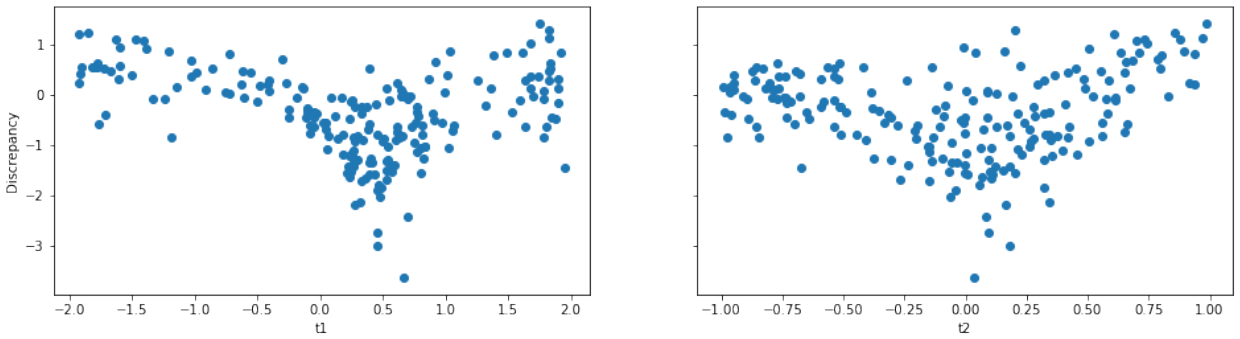
```
bolfi.plot_state();
```

```
<matplotlib.figure.Figure at 0x11b2b2ba8>
```



It may be useful to see the acquired parameter values and the resulting discrepancies:

```
bolfi.plot_discrepancy();
```



There could be an unnecessarily high number of points at parameter bounds. These could probably be decreased by lowering the covariance of the noise added to acquired points, defined by the optional `acq_noise_var` argument for the BOLFI constructor. Another possibility could be to add virtual derivative observations at the borders, though not yet implemented in ELFI.

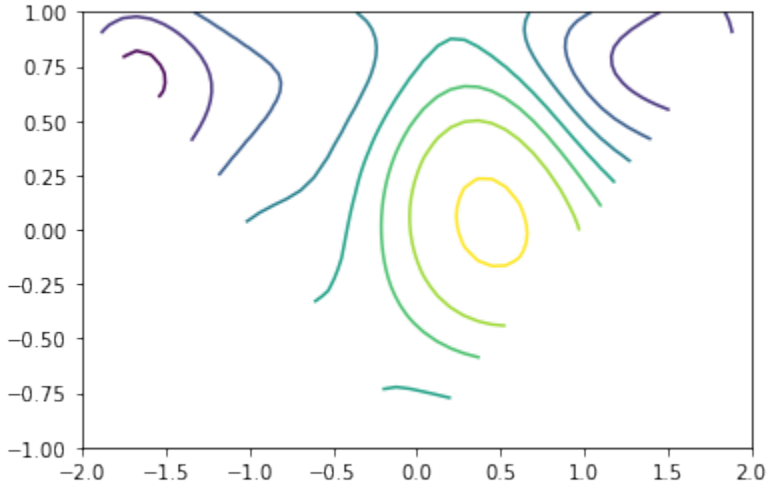
1.10.2 BOLFI Posterior

Above, the `fit` method returned a `BolfiPosterior` object representing a BOLFI posterior (please see the paper for details). The `fit` method accepts a threshold parameter; if none is given, ELFI will use the minimum value of discrepancy estimate mean. Afterwards, one may request for a posterior with a different threshold:

```
post2 = bolfi.extract_posterior(-1.)
```

One can visualize a posterior directly (remember that the priors form a triangle):

```
post.plot(logpdf=True)
```



1.10.3 Sampling

Finally, samples from the posterior can be acquired with an MCMC sampler. By default it runs 4 chains, and half of the requested samples are spent in adaptation/warmup. Note that depending on the smoothness of the GP approximation, the number of priors, their gradients etc., **this may be slow**.

```
%time result_BOLFI = bolfi.sample(1000, info_freq=1000)
```

```
INFO:elfi.methods.posteriors:Using optimized minimum value (-1.6146) of the GP␣
↪discrepancy mean function as a threshold
INFO:elfi.methods.mcmc:NUTS: Performing 1000 iterations with 500 adaptation steps.
INFO:elfi.methods.mcmc:NUTS: Adaptation/warmup finished. Sampling...
INFO:elfi.methods.mcmc:NUTS: Acceptance ratio: 0.423. After warmup 68 proposals were␣
↪outside of the region allowed by priors and rejected, decreasing acceptance ratio.
INFO:elfi.methods.mcmc:NUTS: Performing 1000 iterations with 500 adaptation steps.
INFO:elfi.methods.mcmc:NUTS: Adaptation/warmup finished. Sampling...
INFO:elfi.methods.mcmc:NUTS: Acceptance ratio: 0.422. After warmup 71 proposals were␣
↪outside of the region allowed by priors and rejected, decreasing acceptance ratio.
INFO:elfi.methods.mcmc:NUTS: Performing 1000 iterations with 500 adaptation steps.
INFO:elfi.methods.mcmc:NUTS: Adaptation/warmup finished. Sampling...
INFO:elfi.methods.mcmc:NUTS: Acceptance ratio: 0.419. After warmup 65 proposals were␣
↪outside of the region allowed by priors and rejected, decreasing acceptance ratio.
INFO:elfi.methods.mcmc:NUTS: Performing 1000 iterations with 500 adaptation steps.
INFO:elfi.methods.mcmc:NUTS: Adaptation/warmup finished. Sampling...
```

(continues on next page)

(continued from previous page)

```
INFO:elfi.methods.mcmc:NUTS: Acceptance ratio: 0.439. After warmup 66 proposals were␣
↪outside of the region allowed by priors and rejected, decreasing acceptance ratio.
```

```
4 chains of 1000 iterations acquired. Effective sample size and Rhat for each parameter:
t1 2222.1197791 1.00106816947
t2 2256.93599184 1.0003364409
CPU times: user 1min 45s, sys: 1.29 s, total: 1min 47s
Wall time: 55.1 s
```

The sampling algorithms may be fine-tuned with some parameters. The default No-U-Turn-Sampler is a sophisticated algorithm, and in some cases one may get warnings about diverged proposals, which are signs that something may be wrong and should be investigated. It is good to understand the cause of these warnings although they don't automatically mean that the results are unreliable. You could try rerunning the `sample` method with a higher target probability `target_prob` during adaptation, as its default 0.6 may be inadequate for a non-smooth posteriors, but this will slow down the sampling.

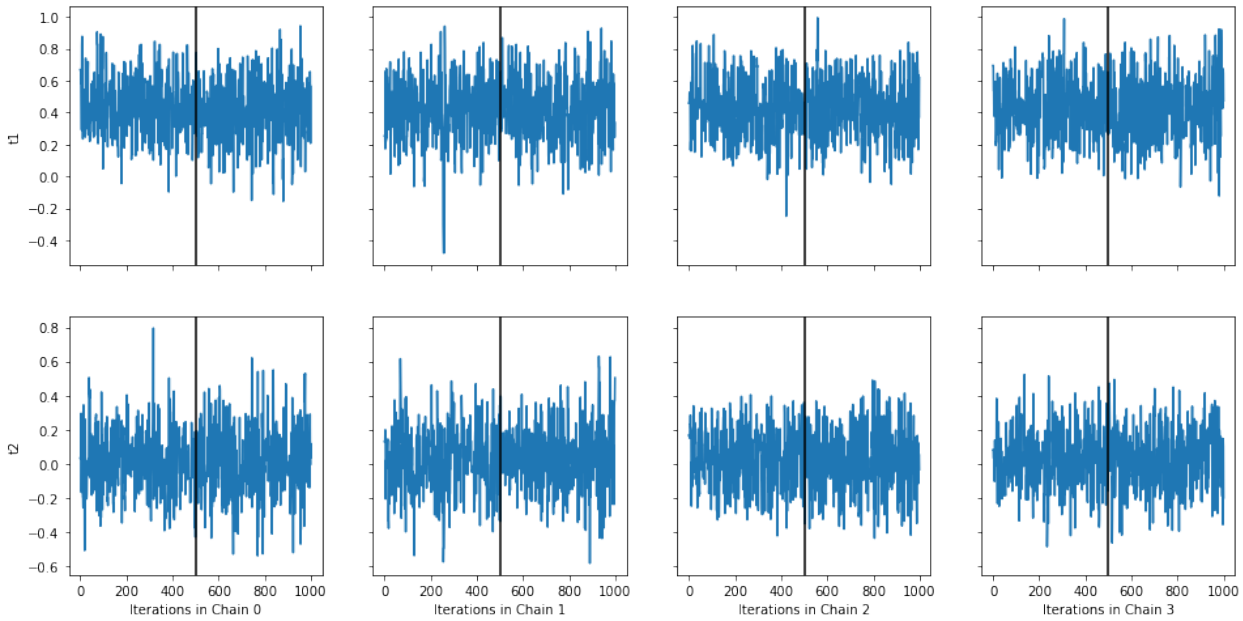
Note also that since MCMC proposals outside the region allowed by either the model priors or GP bounds are rejected, a tight domain may lead to suboptimal overall acceptance ratio. In our MA2 case the prior defines a triangle-shaped uniform support for the posterior, making it a good example of a difficult model for the NUTS algorithm.

Now we finally have a `Sample` object again, which has several convenience methods:

```
result_BOLFI
```

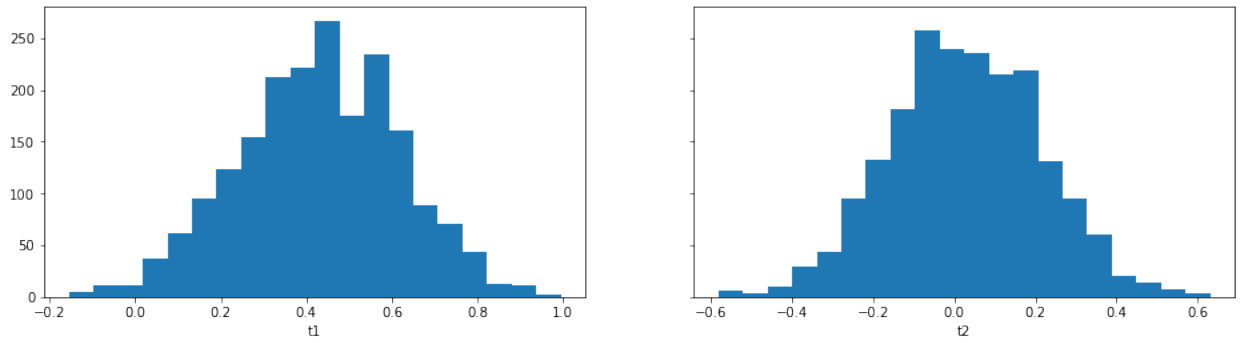
```
Method: BOLFI
Number of samples: 2000
Number of simulations: 200
Threshold: -1.61
Sample means: t1: 0.429, t2: 0.0277
```

```
result_BOLFI.plot_traces();
```



The black vertical lines indicate the end of warmup, which by default is half of the number of iterations.

```
result_BOLFI.plot_marginals();
```



This tutorial is generated from a Jupyter notebook that can be found here.

1.11 Using non-Python operations

If your simulator or other operations are implemented in a programming language other than Python, you can still use ELFI. This notebook briefly demonstrates how to do this in three common scenarios:

- External executable (written e.g. in C++ or a shell script)
- R function
- MATLAB function

Let's begin by importing some libraries that we will be using:

```
import os
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import scipy.io as sio
import scipy.stats as ss

import elfi
import elfi.examples.bdm
import elfi.examples.ma2

%matplotlib inline
```

Note: To run some parts of this notebook you need to either compile the simulator, have R or MATLAB installed and install their respective wrapper libraries.

1.11.1 External executables

ELFI supports using external simulators and other operations that can be called from the command-line. ELFI provides some tools to easily incorporate such operations to ELFI models. This functionality is introduced in this tutorial.

We demonstrate here how to wrap executables as ELFI nodes. We will first use `elfi.tools.external_operation` tool to wrap executables as a Python callables (function). Let's first investigate how it works with a simple shell `echo` command:

```
# Make an external command. {0} {1} are positional arguments and {seed} a keyword␣
↪argument `seed`.
command = 'echo {0} {1} {seed}'
echo_sim = elfi.tools.external_operation(command)

# Test that `echo_sim` can now be called as a regular python function
echo_sim(3, 1, seed=123)
```

```
array([   3.,    1.,  123.])
```

The placeholders for arguments in the command string are just Python's `format strings <https://docs.python.org/3/library/string.html#formatstrings>`__.

Currently `echo_sim` only accepts scalar arguments. In order to work in ELFI, `echo_sim` needs to be vectorized so that we can pass to it a vector of arguments. ELFI provides a handy tool for this as well:

```
# Vectorize it with elfi tools
echo_sim_vec = elfi.tools.vectorize(echo_sim)

# Make a simple model
m = elfi.ElfiModel(name='echo')
elfi.Prior('uniform', .005, 2, model=m, name='alpha')
elfi.Simulator(echo_sim_vec, m['alpha'], 0, name='echo')

# Test to generate 3 simulations from it
m['echo'].generate(3)
```

```
array([[  1.93678222e+00,   0.00000000e+00,   7.43529055e+08],
       [  9.43846120e-01,   0.00000000e+00,   7.43529055e+08],
       [  2.67626618e-01,   0.00000000e+00,   7.43529055e+08]])
```

So above, the first column draws from our uniform prior for α, the second column has constant zeros, and the last one lists the seeds provided to the command by ELFI.

1.11.2 Complex external operations – case BDM

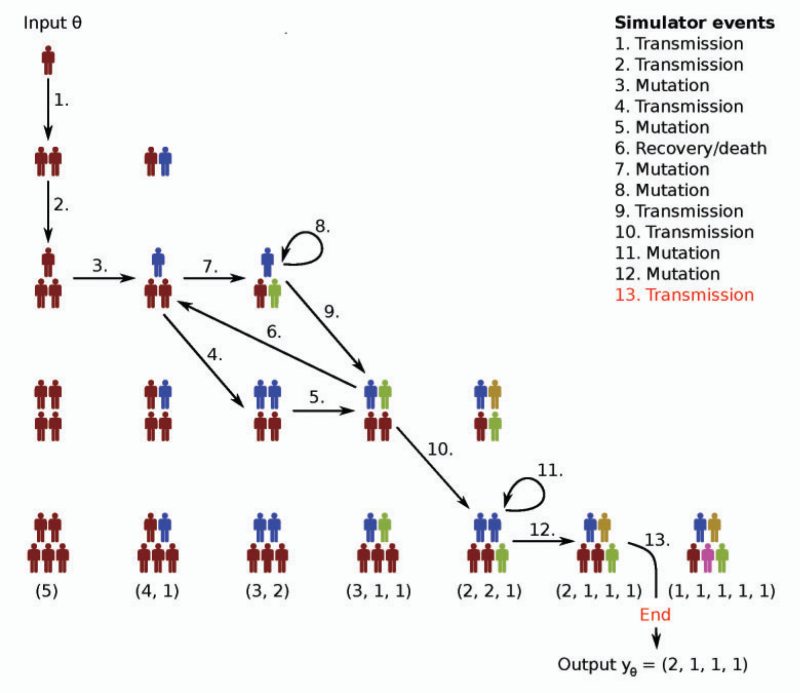
To provide a more realistic example of external operations, we will consider the Birth-Death-Mutation (BDM) model used in Lintusaari at al 2016 *[1]*.

Birth-Death-Mutation process

We will consider here the Birth-Death-Mutation process simulator introduced in *Tanaka et al 2006 [2]* for the spread of Tuberculosis. The simulator outputs a count vector where each of its elements represents a "mutation" of the disease and the count describes how many are currently infected by that mutation. There are three rates and the population size:

- α - (birth rate) the rate at which any infectious host transmits the disease.
- δ - (death rate) the rate at which any existing infectious hosts either recovers or dies.
- τ - (mutation rate) the rate at which any infectious host develops a new unseen mutation of the disease within themselves.
- N - (population size) the size of the simulated infectious population

It is assumed that the susceptible population is infinite, the hosts carry only one mutation of the disease and transmit that mutation onward. A more accurate description of the model can be found from the original paper or e.g. Lintusaari at al 2016 *[1]*.



This simulator cannot be implemented effectively with vectorized operations so we have implemented it with C++ that handles loops efficiently. We will now reproduce Figure 6(a) in Lintusaari at al 2016 *[2]* with ELFI. Let's start by defining some constants:

```
# Fixed model parameters
delta = 0
tau = 0.198
N = 20

# The zeros are to make the observed population vector have length N
y_obs = np.array([6, 3, 2, 2, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=
↪'int16')
```

Let's build the beginning of a new model for the birth rate α as the only unknown

```
m = elfi.ElfiModel(name='bdm')
elfi.Prior('uniform', .005, 2, model=m, name='alpha')
```

```
Prior(name='alpha', 'uniform')
```

```
# Get the BDM source directory
sources_path = elfi.examples.bdm.get_sources_path()

# Compile (unix-like systems)
!make -C $sources_path

# Move the executable in to the working directory
!mv $sources_path/bdm .
```

```
g++ bdm.cpp --std=c++0x -O -Wall -o bdm
```

Note: The source code for the BDM simulator comes with ELFI. You can get the directory with *elfi.examples.bdm.get_source_directory()*. Under unix-like systems it can be compiled with just typing *make* to console in the source directory. For windows systems, you need to have some C++ compiler available to compile it.

```
# Test the executable (assuming we have the executable `bdm` in the working directory)
sim = elfi.tools.external_operation('./bdm {0} {1} {2} {3} --seed {seed} --mode 1')
sim(1, delta, tau, N, seed=123)
```

```
array([ 19.,   1.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
         0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.])
```

The BDM simulator is actually already internally vectorized if you provide it an input file with parameters on the rows. This is more efficient than looping in Python (`elfi.tools.vectorize`), because one simulation takes very little time and we wish to generate tens of thousands of simulations. We will also here redirect the output to a file and then read the file into a numpy array.

This is just one possibility among the many to implement this. The most efficient would be to write a native Python module with C++ but it's beyond the scope of this article. So let's work through files which is a fairly common situation especially with existing software.

```
# Assuming we have the executable `bdm` in the working directory
command = './bdm {filename} --seed {seed} --mode 1 > {output_filename}'


# Function to prepare the inputs for the simulator. We will create filenames and write␣
↪an input file.
def prepare_inputs(*inputs, **kwinputs):
    alpha, delta, tau, N = inputs
    meta = kwinputs['meta']

    # Organize the parameters to an array. The broadcasting works nicely with constant␣
↪arguments here.
    param_array = np.row_stack(np.broadcast(alpha, delta, tau, N))
```

(continues on next page)

(continued from previous page)

```
    # Prepare a unique filename for parallel settings
    filename = '{model_name}_{batch_index}_{submission_index}.txt'.format(**meta)
    np.savetxt(filename, param_array, fmt='%.4f %.4f %.4f %d')

    # Add the filenames to kwinputs
    kwinputs['filename'] = filename
    kwinputs['output_filename'] = filename[:-4] + '_out.txt'

    # Return new inputs that the command will receive
    return inputs, kwinputs


# Function to process the result of the simulation
def process_result(completed_process, *inputs, **kwinputs):
    output_filename = kwinputs['output_filename']

    # Read the simulations from the file.
    simulations = np.loadtxt(output_filename, dtype='int16')

    # Clean up the files after reading the data in
    os.remove(kwinputs['filename'])
    os.remove(output_filename)

    # This will be passed to ELFI as the result of the command
    return simulations


# Create the python function (do not read stdout since we will work through files)
bdm = elfi.tools.external_operation(command,
                                    prepare_inputs=prepare_inputs,
                                    process_result=process_result,
                                    stdout=False)
```

Now let's replace the echo simulator with this. To create unique but informative filenames, we ask ELFI to provide the operation some meta information. That will be available under the `meta` keyword (see the `prepare_inputs` function above):

```
# Create the simulator
bdm_node = elfi.Simulator(bdm, m['alpha'], delta, tau, N, observed=y_obs, name='sim')

# Ask ELFI to provide the meta dict
bdm_node.uses_meta = True

# Draw the model
elfi.draw(m)
```

```
# Test it
data = bdm_node.generate(3)
print(data)
```

```
[[13  1  4  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
```

(continues on next page)

(continued from previous page)

```
 [19  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [14  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]]
```

Completing the BDM model

We are now ready to finish up the BDM model. To reproduce Figure 6(a) in Lintusaari at al 2016 *[2]*, let's add different summaries and discrepancies to the model and run the inference for each of them:

```
def T1(clusters):
    clusters = np.atleast_2d(clusters)
    return np.sum(clusters > 0, 1)/np.sum(clusters, 1)

def T2(clusters, n=20):
    clusters = np.atleast_2d(clusters)
    return 1 - np.sum((clusters/n)**2, axis=1)

# Add the different distances to the model
elfi.Summary(T1, bdm_node, name='T1')
elfi.Distance('minkowski', m['T1'], p=1, name='d_T1')

elfi.Summary(T2, bdm_node, name='T2')
elfi.Distance('minkowski', m['T2'], p=1, name='d_T2')

elfi.Distance('minkowski', m['sim'], p=1, name='d_sim')
```

```
Distance(name='d_sim')
```

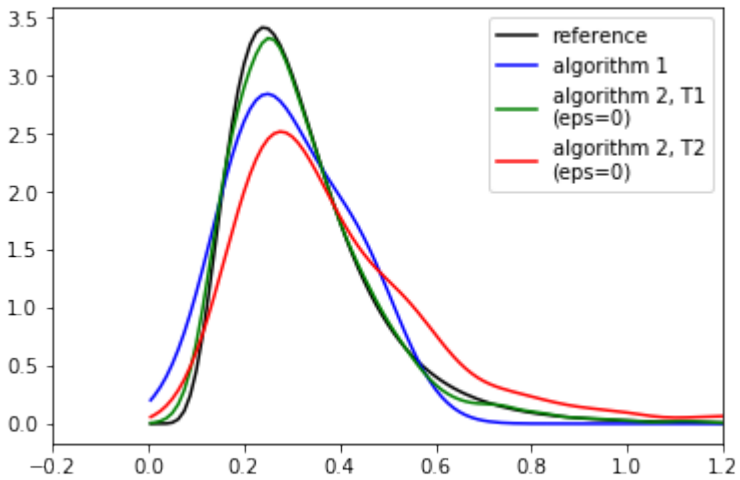
```
elfi.draw(m)
```

```
# Save parameter and simulation results in memory to speed up the later inference
pool = elfi.OutputPool(['alpha', 'sim'])
# Fix a seed
seed = 20170511

rej = elfi.Rejection(m, 'd_T1', batch_size=10000, pool=pool, seed=seed)
%time T1_res = rej.sample(5000, n_sim=int(1e5))

rej = elfi.Rejection(m, 'd_T2', batch_size=10000, pool=pool, seed=seed)
%time T2_res = rej.sample(5000, n_sim=int(1e5))

rej = elfi.Rejection(m, 'd_sim', batch_size=10000, pool=pool, seed=seed)
%time sim_res = rej.sample(5000, n_sim=int(1e5))
```

```
CPU times: user 3.11 s, sys: 143 ms, total: 3.26 s
Wall time: 5.56 s
CPU times: user 29.9 ms, sys: 1.45 ms, total: 31.3 ms
Wall time: 31.2 ms
CPU times: user 33.8 ms, sys: 500 µs, total: 34.3 ms
Wall time: 34 ms
```

```
# Load a precomputed posterior based on an analytic solution (see Lintusaari et al 2016)
matdata = sio.loadmat('./resources/bdm.mat')
x = matdata['likgrid'].reshape(-1)
posterior_at_x = matdata['post'].reshape(-1)

# Plot the reference
plt.figure()
plt.plot(x, posterior_at_x, c='k')

# Plot the different curves
for res, d_node, c in ([sim_res, 'd_sim', 'b'], [T1_res, 'd_T1', 'g'], [T2_res, 'd_T2',
↪'r']):
    alphas = res.outputs['alpha']
    dists = res.outputs[d_node]
    # Use gaussian kde to make the curves look nice. Note that this tends to benefit the␣
↪algorithm 1
    # a lot as it ususally has only a very few accepted samples with 100000 simulations
    kde = ss.gaussian_kde(alphas[dists<=0])
    plt.plot(x, kde(x), c=c)

plt.legend(['reference', 'algorithm 1', 'algorithm 2, T1\n(eps=0)', 'algorithm 2, T2\
↪n(eps=0)'])
plt.xlim([-.2, 1.2]);
print('Results after 100000 simulations. Compare to figure 6(a) in Lintusaari et al.␣
↪2016.')
```

```
Results after 100000 simulations. Compare to figure 6(a) in Lintusaari et al. 2016.
```



1.11.3 Interfacing with R

It is possible to run R scripts in command line for example with Rscript. However, in Python it may be more convenient to use rpy2, which allows convenient access to the functionality of R from within Python. You can install it with `pip install rpy2`.

Here we demonstrate how to calculate the summary statistics used in the ELFI tutorial (autocovariances) using R's `acf` function for the MA2 model.

```
import rpy2.robjects as robj
from rpy2.robjects import numpy2ri as np2ri

# Converts numpy arrays automatically
np2ri.activate()
```

Note: See this issue if you get a *undefined symbol: PC* error in the import after installing rpy2 and you are using Anaconda.

Let's create a Python function that wraps the R commands (please see the documentation of rpy2 for details):

```
robj.r('''
    # create a function `f`
    f <- function(x, lag=1) {
        ac = acf(x, plot=FALSE, type="covariance", lag.max=lag, demean=FALSE)
        ac[['acf']][lag+1]
    }
    ''')

f = robj.globalenv['f']

def autocovR(x, lag=1):
    x = np.atleast_2d(x)
    apply = robj.r['apply']
    ans = apply(x, 1, f, lag=lag)
    return np.atleast_1d(ans)
```

```
# Test it
autocovR(np.array([[1,2,3,4], [4,5,6,7]]), 1)
```

```
array([  5.,  23.])
```

Load a ready made MA2 model:

```
ma2 = elfi.examples.ma2.get_model(seed_obs=4)
elfi.draw(ma2)
```

Replace the summaries S1 and S2 with our R autocovariance function.

```
# Replace with R autocov
S1 = elfi.Summary(autocovR, ma2['MA2'], 1)
S2 = elfi.Summary(autocovR, ma2['MA2'], 2)
ma2['S1'].become(S1)
```

(continues on next page)

(continued from previous page)

```
ma2['S2'].become(S2)

# Run the inference
rej = elfi.Rejection(ma2, 'd', batch_size=1000, seed=seed)
rej.sample(100)
```

```
Method: Rejection
Number of samples: 100
Number of simulations: 10000
Threshold: 0.111
Sample means: t1: 0.599, t2: 0.177
```

1.11.4 Interfacing with MATLAB

There are a number of options for running MATLAB (or Octave) scripts from within Python. Here, evaluating the distance is demonstrated with a MATLAB function using the official MATLAB Python cd API. (Tested with MATLAB 2016b.)

```
import matlab.engine
```

A MATLAB session needs to be started (and stopped) separately:

```
eng = matlab.engine.start_matlab()  # takes a while...
```

Similarly as with R, we have to write a piece of code to interface between MATLAB and Python:

```
def euclidean_M(x, y):
    # MATLAB array initialized with Python's list
    ddM = matlab.double((x-y).tolist())

    # euclidean distance
    dM = eng.sqrt(eng.sum(eng.power(ddM, 2.0), 2))

    # Convert back to numpy array
    d = np.atleast_1d(dM).reshape(-1)
    return d
```

```
# Test it
euclidean_M(np.array([[1,2,3], [6,7,8], [2,2,3]]), np.array([2,2,2]))
```

```
array([ 1.41421356,  8.77496439,  1.        ])
```

Load a ready made MA2 model:

```
ma2M = elfi.examples.ma2.get_model(seed_obs=4)
elfi.draw(ma2M)
```

Replace the summaries S1 and S2 with our R autocovariance function.

```
# Replace with Matlab distance implementation
d = elfi.Distance(euclidean_M, ma2M['S1'], ma2M['S2'])
ma2M['d'].become(d)

# Run the inference
rej = elfi.Rejection(ma2M, 'd', batch_size=1000, seed=seed)
rej.sample(100)
```

```
Method: Rejection
Number of samples: 100
Number of simulations: 10000
Threshold: 0.113
Sample means: t1: 0.602, t2: 0.178
```

Finally, don't forget to quit the MATLAB session:

```
eng.quit()
```

1.11.5 Verdict

We showed here a few examples of how to incorporate non Python operations to ELFI models. There are multiple other ways to achieve the same results and even make the wrapping more efficient.

Wrapping often introduces some overhead to the evaluation of the generative model. In many cases however this is not an issue since the operations are usually expensive by themselves making the added overhead insignificant.

References

- [1] Jarno Lintusaari, Michael U. Gutmann, Ritabrata Dutta, Samuel Kaski, Jukka Corander; Fundamentals and Recent Developments in Approximate Bayesian Computation. Syst Biol 2017; 66 (1): e66-e82. doi: 10.1093/sysbio/syw077
- [2] Tanaka, Mark M., et al. "Using approximate Bayesian computation to estimate tuberculosis transmission parameters from genotype data." Genetics 173.3 (2006): 1511-1520.

1.12 Implementing a new inference method

This tutorial provides the fundamentals for implementing custom parameter inference methods using ELFI. ELFI provides many features out of the box, such as parallelization or random state handling. In a typical case these happen "automatically" behind the scenes when the algorithms are built on top of the provided interface classes.

The base class for parameter inference classes is the *ParameterInference* interface which is found from the `elfi.methods.inference.parameter_inference` module. Among the methods in the interface, those that must be implemented raise a `NotImplementedError`. In addition, you probably also want to override at least the `update` and `__init__` methods.

Let's create an empty skeleton for a custom method that includes just the minimal set of methods to create a working algorithm in ELFI:

```
from elfi.methods.inference.parameter_inference import ParameterInference

class CustomMethod(ParameterInference):

    def __init__(self, model, output_names, **kwargs):
        super(CustomMethod, self).__init__(model, output_names, **kwargs)

    def set_objective(self):
        # Request 3 batches to be generated
        self.objective['n_batches'] = 3

    def extract_result(self):
        return self.state
```

The method `extract_result` is called by ELFI in the end of inference and should return a `ParameterInferenceResult` object (`elfi.methods.result` module). For illustration we will however begin by returning the member `state` dictionary. It stores all the current state information of the inference. Let's make an instance of our method and run it:

```
import elfi.examples.ma2 as ma2

# Get a ready made MA2 model to test our inference method with
m = ma2.get_model()

# We want the outputs from node 'd' of the model `m` to be available
custom_method = CustomMethod(m, ['d'])

# Run the inference
custom_method.infer()  # {'n_batches': 3, 'n_sim': 3000}
```

Running the above returns the state dictionary. We will find a few keys in it that track some basic properties of the state, such as the `n_batches` telling how many batches has been generated and `n_sim` that tells the number of total simulations contained in those batches. It should be `n_batches` times the current batch size (`custom_method.batch_size` which was 1000 here by default).

You will find that the `n_batches` in the state dictionary had a value 3. This is because in our `CustomMethod.set_objective` method, we set the `n_batches` key of the objective dictionary to that value. Every *ParameterInference* instance has a Python dictionary called `objective` that is a counterpart to the `state` dictionary. The objective defines the conditions when the inference is finished. The default controlling key in that dictionary is the string `n_batches` whose value tells ELFI how many batches we need to generate in total from the provided generative `ElfiModel` model. Inference is considered finished when the `n_batches` in the `state` matches or exceeds that in the `objective`. The generation of batches is automatically parallelized in the background, so we don't have to worry about it.

Note: A batch in ELFI is a dictionary that maps names of nodes of the generative model to their outputs. An output in the batch consists of one or more runs of it's operation stored to a numpy array. Each batch has an index, and the outputs in the same batch are guaranteed to be the same if you recompute the batch.

The algorithm, however, does nothing else at this point besides generating the 3 batches. To actually do something with the batches, we can add the `update` method that allows us to update the state dictionary of the inference with any custom values. It takes in the generated `batch` dictionary and it's index and is called by ELFI every time a new batch is received. Let's say we wish to filter parameters by a threshold (as in ABC Rejection sampling) from the total number of simulations:

```
class CustomMethod(ParameterInference):
    def __init__(self, model, output_names, **kwargs):
        super(CustomMethod, self).__init__(model, output_names, **kwargs)

        # Hard code a threshold and discrepancy node name for now
        self.threshold = .1
        self.discrepancy_name = output_names[0]

        # Prepare lists to push the filtered outputs into
        self.state['filtered_outputs'] = {name: [] for name in output_names}

    def update(self, batch, batch_index):
        super(CustomMethod, self).update(batch, batch_index)

        # Make a filter mask (logical numpy array) from the distance array
        filter_mask = batch[self.discrepancy_name] <= self.threshold

        # Append the filtered parameters to their lists
        for name in self.output_names:
            values = batch[name]
            self.state['filtered_outputs'][name].append(values[filter_mask])

    ... # other methods as before

m = ma2.get_model()
custom_method = CustomMethod(m, ['d'])
custom_method.infer()  # {'n_batches': 3, 'n_sim': 3000, 'filtered_outputs': ...}
```

After running this you should have in the returned state dictionary the `filtered_outputs` key containing filtered distances for node `d` from the 3 batches.

Note: The reason for the imposed structure in `ParameterInference` is to encourage a design where one can advance the inference iteratively using the `iterate` method. This makes it possible to stop at any point, check the current state and to be able to continue. This is important as there are usually many moving parts, such as summary statistic choices or deciding a good discrepancy function.

Now to be useful, we should allow the user to set the different options - the 3 batches is not going to take her very far. The user also probably thinks in terms of simulations rather than batches. ELFI allows you to replace the `n_batches` with `n_sim` key in the objective to spare you from turning `n_sim` to `n_batches` in the code. Just note that the `n_sim` in the state will always be in multiples of the `batch_size`.

Let's modify the algorithm so, that the user can pass the threshold, the name of the discrepancy node and the number of simulations. And let's also add the parameters to the outputs:

```
class CustomMethod(ParameterInference):
    def __init__(self, model, discrepancy_name, threshold, **kwargs):
        # Create a name list of nodes whose outputs we wish to receive
        output_names = [discrepancy_name] + model.parameter_names
        super(CustomMethod, self).__init__(model, output_names, **kwargs)

        self.threshold = threshold
        self.discrepancy_name = discrepancy_name
```

(continues on next page)

(continued from previous page)

```
        # Prepare lists to push the filtered outputs into
        self.state['filtered_outputs'] = {name: [] for name in output_names}

    def set_objective(self, n_sim):
        self.objective['n_sim'] = n_sim

    ... # other methods as before

# Run it
custom_method = CustomMethod(m, 'd', threshold=.1, batch_size=1000)
custom_method.infer(n_sim=2000)  # {'n_batches': 2, 'n_sim': 2000, 'filtered_outputs': ...}
```

Calling the inference method now returns the state dictionary that has also the filtered parameters in it from each of the batches. Note that any arguments given to the `infer` method are passed to the `set_objective` method.

Now due to the structure of the algorithm the user can immediately continue from this state:

```
# Continue inference from the previous state (with n_sim=2000)
custom_method.infer(n_sim=4000) # {'n_batches': 4, 'n_sim': 4000, 'filtered_outputs': ...}

# Or use it iteratively
custom_method.set_objective(n_sim=6000)

custom_method.iterate()
assert custom_method.finished == False
# Investigate the current state
custom_method.extract_result()  # {'n_batches': 5, 'n_sim': 5000, 'filtered_outputs': ...}

self.iterate()
assert custom_method.finished
custom_method.extract_result()  # {'n_batches': 6, 'n_sim': 6000, 'filtered_outputs': ...}
```

This works, because the state is stored into the `custom_method` instance, and we only change the objective. Also ELFI calls `iterate` internally in the `infer` method.

The last finishing touch to our algorithm is to convert the `state` dict to a more user friendly format in the `extract_result` method. First we want to convert the list of filtered arrays from the batches to a numpy array. We will then wrap the result to a `elfi.methods.results.Sample` object and return it instead of the `state` dict. Below is the final complete implementation of our inference method class:

```
import numpy as np

from elfi.methods.inference.parameter_inference import ParameterInference
from elfi.methods.results import Sample


class CustomMethod(ParameterInference):
    def __init__(self, model, discrepancy_name, threshold, **kwargs):
        # Create a name list of nodes whose outputs we wish to receive
        output_names = [discrepancy_name] + model.parameter_names
        super(CustomMethod, self).__init__(model, output_names, **kwargs)
```

(continues on next page)

(continued from previous page)

```
        self.threshold = threshold
        self.discrepancy_name = discrepancy_name

        # Prepare lists to push the filtered outputs into
        self.state['filtered_outputs'] = {name: [] for name in output_names}

    def set_objective(self, n_sim):
        self.objective['n_sim'] = n_sim

    def update(self, batch, batch_index):
        super(CustomMethod, self).update(batch, batch_index)

        # Make a filter mask (logical numpy array) from the distance array
        filter_mask = batch[self.discrepancy_name] <= self.threshold

        # Append the filtered parameters to their lists
        for name in self.output_names:
            values = batch[name]
            self.state['filtered_outputs'][name].append(values[filter_mask])

    def extract_result(self):
        filtered_outputs = self.state['filtered_outputs']
        outputs = {name: np.concatenate(filtered_outputs[name]) for name in self.output_
↪names}

        return Sample(
            method_name='CustomMethod',
            outputs=outputs,
            parameter_names=self.parameter_names,
            discrepancy_name=self.discrepancy_name,
            n_sim=self.state['n_sim'],
            threshold=self.threshold
            )
```

Running the inference with the above implementation should now produce an user friendly output:

```
Method: CustomMethod
Number of posterior samples: 82
Number of simulations: 10000
Threshold: 0.1
Posterior means: t1: 0.687, t2: 0.152
```

1.12.1 Where to go from here

When implementing your own method it is advisable to read the documentation of the *ParameterInference* class. In addition we recommend reading the `Rejection`, `SMC` and/or `BayesianOptimization` class implementations from the source for some more advanced techniques. These methods feature e.g. how to inject values from outside into the ELFI model (acquisition functions in BayesianOptimization), how to modify the user provided model to get e.g. the pdf:s of the parameters (SMC) and so forth.

Good to know

ELFI guarantees that computing a batch with the same index will always produce the same output given the same model and `ComputationContext` object. The `ComputationContext` object holds the batch size, seed for the PRNG, the pool object of precomputed batches of nodes. If your method uses random quantities in the algorithm, please make sure to use the seed attribute of `ParameterInference` so that your results will be consistent.

If you want to provide values for outputs of certain nodes from outside the generative model, you can return them from `prepare_new_batch` method. They will replace any default value or operation in that node. This is used e.g. in `BOLFI` where values from the acquisition function replace values coming from the prior in the Bayesian optimization phase.

The *ParameterInference* instance has also the following helper classes:

1.12.2 `BatchHandler`

ParameterInference class instantiates a `elfi.client.BatchHandler` helper class that is set as the `self.batches` member variable. This object is in essence a wrapper to the `Client` interface making it easier to work with batches that are in computation. Some of the duties of `BatchHandler` is to keep track of the current batch_index and of the status of the batches that have been submitted. You often don't need to interact with it directly.

1.12.3 `OutputPool`

`elfi.store.OutputPool` serves a dual purpose: 1. It stores all the computed outputs of selected nodes 2. It provides those outputs when a batch is recomputed saving the need to recompute them.

Note however that reusing the values is not always possible. In sequential algorithms that decide their next parameter values based on earlier results, modifications to the ELFI model will invalidate the earlier data. On the other hand, Rejection sampling for instance allows changing any of the summaries or distances and still reuse e.g. the simulations. This is because all the parameter values will still come from the same priors.

Parameter inference base class

class elfi.methods.inference.parameter_inference.**ParameterInference**(*model*, *output_names*, *batch_size=1*, *seed=None*, *pool=None*, *max_parallel_batches=None*)

A base class for parameter inference methods.

model

The ELFI graph used by the algorithm

Type
elfi.ElfiModel

`output_names`

Names of the nodes whose outputs are included in the batches

Type
list

`client`

The batches are computed in the client

Type
elfi.client.ClientBase

`max_parallel_batches`

Type
int

`state`

Stores any changing data related to achieving the objective. Must include a key `n_batches` for determining when the inference is finished.

Type
dict

`objective`

Holds the data for the algorithm to internally determine how many batches are still needed. You must have a key `n_batches` here. By default the algorithm finished when the `n_batches` in the state dictionary is equal or greater to the corresponding objective value.

Type
dict

`batches`

Helper class for submitting batches to the client and keeping track of their indexes.

Type
elfi.client.BatchHandler

`pool`

Pool object for storing and reusing node outputs.

Type
elfi.store.OutputPool

Construct the inference algorithm object.

If you are implementing your own algorithm do not forget to call *super*.

Parameters

- **model** (*elfi.ElfiModel*) – Model to perform the inference with.
- **output_names** (*list*) – Names of the nodes whose outputs will be requested from the ELFI graph.
- **batch_size** (*int, optional*) – The number of parameter evaluations in each pass through the ELFI graph. When using a vectorized simulator, using a suitably large batch_size can provide a significant performance boost.
- **seed** (*int, optional*) – Seed for the data generation from the ElfiModel
- **pool** (*elfi.store.OutputPool, optional*) – OutputPool both stores and provides pre-computed values for batches.

- **max_parallel_batches** (*int, optional*) – Maximum number of batches allowed to be in computation at the same time. Defaults to number of cores in the client

property batch_size

Return the current batch_size.

extract_result()

Prepare the result from the current state of the inference.

ELFI calls this method in the end of the inference to return the result.

Returns
result

Return type
elfi.methods.result.Result

property finished

Check whether objective of n_batches have been reached.

infer(**args*, *vis=None*, *bar=True*, ***kwargs*)

Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

Parameters

- **vis** (*dict, optional*) – Plotting options. More info in self.plot_state method
- **bar** (*bool, optional*) – Flag to remove (False) or keep (True) the progress bar from/in output.

Returns
result

Return type
Sample

iterate()

Advance the inference by one iteration.

This is a way to manually progress the inference. One iteration consists of waiting and processing the result of the next batch in succession and possibly submitting new batches.

Notes

If the next batch is ready, it will be processed immediately and no new batches are submitted.

New batches are submitted only while waiting for the next one to complete. There will never be more batches submitted in parallel than the *max_parallel_batches* setting allows.

Return type
None

property parameter_names

Return the parameters to be inferred.

plot_state(***kwargs*)

Plot the current state of the algorithm.

Parameters

- **axes** (*matplotlib.axes.Axes (optional)*) –
- **figure** (*matplotlib.figure.Figure (optional)*) –
- **xlim** – x-axis limits
- **ylim** – y-axis limits
- **interactive** (*bool (default False)*) – If true, uses IPython.display to update the cell figure
- **close** – Close figure in the end of plotting. Used in the end of interactive mode.

Return type
None

property pool

Return the output pool of the inference.

prepare_new_batch(*batch_index*)

Prepare values for a new batch.

ELFI calls this method before submitting a new batch with an increasing index *batch_index*. This is an optional method to override. Use this if you have a need do do preparations, e.g. in Bayesian optimization algorithm, the next acquisition points would be acquired here.

If you need provide values for certain nodes, you can do so by constructing a batch dictionary and returning it. See e.g. BayesianOptimization for an example.

Parameters
batch_index (*int*) – next batch_index to be submitted

Returns
batch – Keys should match to node names in the model. These values will override any default values or operations in those nodes.

Return type
dict or None

property seed

Return the seed of the inference.

set_objective(**args*, ***kwargs*)

Set the objective of the inference.

This method sets the objective of the inference (values typically stored in the *self.objective* dict).

Return type
None

update(*batch*, *batch_index*)

Update the inference state with a new batch.

ELFI calls this method when a new batch has been computed and the state of the inference should be updated with it. It is also possible to bypass ELFI and call this directly to update the inference.

Parameters

- **batch** (*dict*) – dict with *self.outputs* as keys and the corresponding outputs for the batch as values
- **batch_index** (*int*) –

Return type
None

1.13 ELFI architecture

Here we explain the internal representation of the ELFI model. This representation contains everything that is needed to generate data, but is separate from e.g. the inference methods or the data storages. This information is aimed for developers and is not essential for using ELFI. We assume the reader is quite familiar with Python and has perhaps already read some of ELFI's source code.

The low level representation of the ELFI model is a `networkx.DiGraph` with node names as the nodes. The representation of the node is stored to the corresponding attribute dictionary of the `networkx.DiGraph`. We call this attribute dictionary the node *state* dictionary. The `networkx.DiGraph` representation can be found from `ElfiModel.source_net`. Before the ELFI model can be ran, it needs to be compiled and loaded with data (e.g. observed data, precomputed data, batch index, batch size etc). The compilation and loading of data is the responsibility of the `Client` implementation and makes it possible in essence to translate `ElfiModel` to any kind of computational backend. Finally the class `Executor` is responsible for running the compiled and loaded model and producing the outputs of the nodes.

A user typically creates this low level representation by working with subclasses of `NodeReference`. These are easy to use UI classes of ELFI such as the `elfi.Simulator` or `elfi.Prior`. Under the hood they create proper node state dictionaries stored into the `source_net`. The callables such as simulators or summaries that the user provides to these classes are called operations.

1.13.1 The model graph representation

The `source_net` is a directed acyclic graph (DAG) and holds the state dictionaries of the nodes and the edges between the nodes. An edge represents a dependency. For example and edge from a prior node to the simulator node represents that the simulator requires a value from the prior to be able to run. The edge name corresponds to a parameter name for the operation, with integer names interpreted as positional parameters.

In the standard compilation process, the `source_net` is augmented with additional nodes such as batch_size or random_state, that are then added as dependencies for those operations that require them. In addition the state dicts will be turned into either a runnable operation or a precomputed value.

The execution order of the nodes in the compiled graph follows the topological ordering of the DAG (dependency order) and is guaranteed to be the same every time. Note that because the default behaviour is that nodes share a random state, changing a node that uses a shared random state will affect the result of any later node in the ordering using the same random state, even if they would be independent based on the graph topology.

1.13.2 State dictionary

The state of a node is a Python dictionary. It describes the type of the node and any other relevant state information, such as the user provided callable operation (e.g. simulator or summary statistic) and any additional parameters the operation needs to be provided in the compilation.

The following are reserved keywords of the state dict that serve as instructions for the ELFI compiler. They begin with an underscore. Currently these are:

_operation
: [callable] Operation of the node producing the output. Can not be used if _output is present.

_output
: [variable] Constant output of the node. Can not be used if _operation is present.

_class
: [class] The subclass of `NodeReference` that created the state.

_stochastic
: [bool, optional] Indicates that the node is stochastic. ELFI will provide a random_state argument for such nodes,

which contains a RandomState object for drawing random quantities. This node will appear in the computation graph. Using ELFI provided random states makes it possible to have repeatable experiments in ELFI.

_observable
[bool, optional] Indicates that there is observed data for this node or that it can be derived from the observed data. ELFI will create a corresponding observed node into the compiled graph. These nodes are dependencies of discrepancy nodes.

_uses_batch_size
[bool, optional] Indicates that the node operation requires `batch_size` as input. A corresponding edge from batch_size node to this node will be added to the compiled graph.

_uses_meta
[bool, optional] Indicates that the node operation requires meta information dictionary about the execution. This includes, model name, batch index and submission index. Useful for e.g. creating informative and unique file names. If the operation is vectorized with `elfi.tools.vectorize`, then also `index_in_batch` will be added to the meta information dictionary.

_uses_observed
[bool, optional] Indicates that the node requires the observed data of its parents in the source_net as input. ELFI will gather the observed values of its parents to a tuple and link them to the node as a named argument observed.

_parameter
[bool, optional] Indicates that the node is a parameter node

1.13.3 The compilation and data loading phases

The compilation of the computation graph is separated from the loading of the data for making it possible to reuse the compiled model. The subclasses of the `Loader` class take responsibility of injecting data to the nodes of the compiled model. Examples of injected data are precomputed values from the `OutputPool`, the current `random_state` and so forth.

1.14 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

1.14.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/elfi-dev/elfi/issues.

If you are reporting a bug, please include:

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

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/elfi-dev/elfi/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

1.14.2 Get Started!

ELFI is a project with dozens of collaborators, so organization is key to making our contributions effective and avoid reword. Thus, in addition to the recommendations below we strongly recommend reading our Wiki to see what is the suggested git workflow procedure for your type of contribution.

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

1. Fork the *elfi* repo on GitHub.
2. Clone your fork locally and add the base repository as a remote:

```
$ git clone git@github.com:your_github_handle_here/elfi.git
$ cd elfi
$ git remote add upstream git@github.com:elfi-dev/elfi.git
```

3. Make sure you have Python 3 and Anaconda Distribution installed on your machine. Check your conda and Python versions. Currently supported Python versions are 3.9, 3.10, 3.11, 3.12:

```
$ conda -V
$ python -V
```

4. Install your local copy and the development requirements into a conda environment. You may need to replace "3.9" in the first line with the python version printed in the previous step:

```
$ conda create -n elfi python=3.9 numpy
$ source activate elfi
$ cd elfi
$ make dev
```

5. Create a branch for local development:

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

   Now you can make your changes locally.

6. Follow the *Style Guidelines*

7. When you're done making changes, check that your changes pass flake8 and the tests:

   ```
   $ make lint
   $ make test
   ```

   You may run `make test-notslow` instead of `make test` *as long as your proposed changes are unrelated to BOLFI*.

   Also make sure that the docstrings of your code are formatted properly:

   ```
   $ make docs
   ```

8. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   ```

9. **After committing your changes, you may sync with the base repository if there has been changes::**
   $ git fetch upstream $ git rebase upstream/dev

10. Push the changes:: $ git push origin name-of-your-bugfix-or-feature

11. Submit a pull request through the GitHub website.

1.14.3 Style Guidelines

The Python code in ELFI mostly follows PEP8, which is considered the de-facto code style guide for Python. Lines should not exceed 100 characters.

Docstrings follow the NumPy style.

1.14.4 Pull Request Guidelines

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

1. The pull request should include tests that will be run automatically using Github Actions.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.md.
3. The pull request should work for Python 3.9 and later. Check https://github.com/elfi-dev/elfi/actions/workflows/pytest.yml and make sure that the tests pass for all supported Python versions.

1.14.5 Tips

To run a subset of tests:

```
$ py.test tests.test_elfi
```

CHAPTER

TWO

CITATION

If you wish to cite ELFI, please use the paper in JMLR:

```
@article{JMLR:v19:17-374,
author  = {Jarno Lintusaari and Henri Vuollekoski and Antti Kangasr{\"a}{\"a}si{\"o} and␣
↪Kusti Skyt{\'e}n and Marko J{\"a}rvenp{\"a}{\"a} and Pekka Marttinen and Michael U.␣
↪Gutmann and Aki Vehtari and Jukka Corander and Samuel Kaski},
title   = {ELFI: Engine for Likelihood-Free Inference},
journal = {Journal of Machine Learning Research},
year    = {2018},
volume  = {19},
number  = {16},
pages   = {1-7},
url     = {http://jmlr.org/papers/v19/17-374.html}
}
```

INDEX

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V