# ELFI

*Release 0.6.2*

**Sep 11, 2017**

# 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 1

Currently implemented LFI methods:

- ABC rejection sampler
- Sequential Monte Carlo ABC sampler
- Bayesian Optimization for Likelihood-Free Inference (BOLFI) framework

ELFI also has the following non LFI methods:

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

## Installation

ELFI requires Python 3.5 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.

### Installing Python 3.5

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 3.5. environment with ELFI:

```
conda create -n elfi python=3.5 numpy
source activate elfi
pip install elfi
```

### Optional dependencies

We recommend to install:

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

### 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 3.5 or greater
- In some environments `pip` refers to Python 2.x, and you have to use `pip3` to use the Python 3.x version

### 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.

## 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.

### 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).

### 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.

### 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.

## Quickstart

First ensure you have installed Python 3.5 (or greater) and ELFI. After installation you can start using 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()
```

# API

This file describes the classes and methods available in ELFI.

## 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[, p, w, ...]) | A convenience class for the discrepancy node. |

### 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*. |

## 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.BayesianOptimization*(model[, ...]) | Bayesian Optimization of an unknown target function. |

Continued on next page

Table 1.6 – continued from previous page

| | |
|---|---|
| *elfi.BOLFI*(model[, target_name, bounds, ...]) | Bayesian Optimization for Likelihood-Free Inference (BOLFI). |

#### 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. |
| *BolfiSample*(method_name, chains, ...) | Container for results from BOLFI. |

#### Post-processing

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

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

## 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). |

## 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.

**Returns**

**Return type** *ElfiModel*

**generate**(*batch_size=1*, *outputs=None*, *with_values=None*)
Generate a batch of outputs using the global numpy seed.

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

**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.

**Returns**

**Return type** *ElfiModel*

**name**
Return name of the model.

**observed**
Return the observed data for the nodes in a dictionary.

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference** (*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference**(*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**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*) –

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference**(*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**size**
Return the size of the output from the distribution.

**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*) –

**compile_operation**(*state*)
Compile a callable operation that samples the associated distribution.

**Parameters** **state** (*dict*) –

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference**(*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**size**
Return the size of the output from the distribution.

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference**(*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference**(*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference**(*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**state**
Return the state dictionary of the node.

**class** elfi.**Distance**(*distance*, **summaries*, *p=None*, *w=None*, *V=None*, *VI=None*, ***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
- **p** (*double, optional*) – The p-norm to apply Only for distance Minkowski (*'minkowski'*), weighted and unweighted. Default: 2.
- **w** (*ndarray, optional*) – The weight vector. Only for weighted Minkowski (*'wminkowski'*). Mandatory.

- **V** (*ndarray, optional*) – The variance vector. Only for standardized Euclidean (*'seuclidean'*). Mandatory.
- **VI** (*ndarray, optional*) – The inverse of the covariance matrix. Only for Mahalanobis. Mandatory.

### 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*) –

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

**Returns** **parents** – List of positional parents

**Return type** list

**reference** (*name*, *model*)
Construct a reference for an existing node in the model.

**Parameters**

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

**Returns**

**Return type** NodePointer instance

**state**
Return the state dictionary of the node.

**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.

visualization.**nx_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

## 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 InferenceMethod

**batch_size**
Return the current batch_size.

**extract_result**()
Extract the result from the current state.

**Returns** result

**Return type** *Sample*

**infer**(**args*, *vis=None*, ***kwargs*)
Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

**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.

**Returns**

**Return type** None

**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.

**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*

**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 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 InferenceMethod

**batch_size**
Return the current batch_size.

**current_population_threshold**
Return the threshold for current population.

**extract_result**()
Extract the result from the current state.

**Returns**

**Return type** *SmcSample*

**infer**(**args*, *vis=None*, ***kwargs*)
Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

**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.

**Returns**

**Return type** None

**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.

**Returns**

**Return type** None

**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*

**seed**
Return the seed of the inference.

**set_objective**(*n_samples*, *thresholds*)
Set the objective of the inference.

**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=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 np.array, optional*) – Variance(s) of the noise added in the default LCBSC acquisition method. If an array, should be 1d 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** (*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** –

**acq_batch_size**
Return the total number of acquisition per iteration.

**batch_size**
Return the current batch_size.

**extract_result**()
Extract the result from the current state.

**Returns**

**Return type** *OptimizationResult*

**infer**(**args*, *vis=None*, ***kwargs*)
Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

**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.

**Returns**

**Return type** None

**n_evidence**
Return the number of acquired evidence points.

**parameter_names**
Return the parameters to be inferred.

**plot_discrepancy**(*axes=None*, ***kwargs*)
Plot acquired parameters vs. resulting discrepancy.

TODO: refactor

**plot_state**(***options*)
Plot the GP surface.

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

**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

**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=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 np.array, optional*) – Variance(s) of the noise added in the default LCBSC acquisition method. If an array, should be 1d 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** (*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** –

**acq_batch_size**
Return the total number of acquisition per iteration.

**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.

**Returns**

**Return type** *OptimizationResult*

**fit**(*n_evidence*, *threshold=None*)
Fit the surrogate model.

Generates a regression model for the discrepancy given the parameters.

Currently only Gaussian processes are supported as surrogate models.

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

**infer**(**args*, *vis=None*, ***kwargs*)
Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

**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.

**Returns**

**Return type** None

**n_evidence**
Return the number of acquired evidence points.

**parameter_names**
Return the parameters to be inferred.

**plot_discrepancy**(*axes=None*, ***kwargs*)
Plot acquired parameters vs. resulting discrepancy.

TODO: refactor

**plot_state**(***options*)
Plot the GP surface.

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

**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'*, *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 only 'nuts' is supported.
- **n_evidence** (*int*) – If the regression model is not fitted yet, specify the amount of evidence

**Returns**

**Return type** *BolfiSample*

**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*) –

**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*

**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

**dim**
Return the number of parameters.

**discrepancies**
Return the discrepancy values.

**n_samples**
Return the number of samples.

**plot_marginals**(*selector=None*, *bins=20*, *axes=None*, ***kwargs*)
Plot marginal distributions for parameters.

**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*, ***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*) –

**Returns** **axes**

**Return type** np.array of plt.Axes

**sample_means**
Evaluate weighted averages of sampled parameters.

**Returns**

**Return type** OrderedDict

**sample_means_array**
Evaluate weighted averages of sampled parameters.

**Returns**

**Return type** np.array

**sample_means_summary**()
Print a representation of sample means.

**samples_array**
Return the samples as an array.

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

**Returns**

**Return type** list of np.arrays

**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** –

**dim**
Return the number of parameters.

**discrepancies**
Return the discrepancy values.

**n_populations**
Return the number of populations.

**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

**sample_means**
Evaluate weighted averages of sampled parameters.

**Returns**

**Return type** OrderedDict

**sample_means_array**
Evaluate weighted averages of sampled parameters.

**Returns**

**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).

**samples_array**
Return the samples as an array.

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

**Returns**

**Return type** list of np.arrays

**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.**BolfiSample**(*method_name*, *chains*, *parameter_names*, *warmup*, ***kwargs*)

Container for results from BOLFI.

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.

**dim**
Return the number of parameters.

**discrepancies**
Return the discrepancy values.

**n_samples**
Return the number of samples.

**plot_marginals**(*selector=None*, *bins=20*, *axes=None*, ***kwargs*)
Plot marginal distributions for parameters.

**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*, ***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*) –

**Returns** **axes**

**Return type** np.array of plt.Axes

**plot_traces**(*selector=None*, *axes=None*, ***kwargs*)
Plot MCMC traces.

**sample_means**
Evaluate weighted averages of sampled parameters.

**Returns**

**Return type** OrderedDict

**sample_means_array**
Evaluate weighted averages of sampled parameters.

**Returns**

**Return type** np.array

**sample_means_summary**()
Print a representation of sample means.

**samples_array**
Return the samples as an array.

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

**Returns**

**Return type** list of np.arrays

**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=['S1', 'S2']).sample(1000)
>>> adj = adjust_posterior(res, m, ['S1', 'S2'], ['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.

**Returns**

**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

### 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*.

**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*) –

**Returns**

**Return type** *ArrayPool*

**output_names**
Return a list of stored names.

**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*.

**has_context**
Check if current pool has context information.

**has_store**(*node*)
Check if *node* is in stores.

**open**(*name*, *prefix=None*)
Open a closed or saved ArrayPool from disk.

**Parameters**

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

**Returns**

**Return type** *ArrayPool*

**output_names**
Return a list of stored names.

**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*)
Set the current ELFI client instance.

**Tools**

tools.**vectorize**(*operation*, *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**(*command*, *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

## 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.

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

## 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
import matplotlib.pyplot as plt
import logging
logging.basicConfig(level=logging.INFO)

%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
np.random.seed(seed)
```

## 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 $\theta_1, \theta_2$ 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. Above we rely on numpy to carry out the vectorized calculations.

In this case the arguments `t1` and `t2` are going to be vectors of length `batch_size` and the method returns a 2d array with the simulations on the rows. Notice that for convenience, the funtion also works with scalars that 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 developments 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

## 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 $\theta_1, \theta_2$. 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:

```
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. The software is already installed in many unix-like OS.

## 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).

### 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_
↪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.

## 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 2.28 s, sys: 165 ms, total: 2.45 s
Wall time: 2.45 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()
```

```
0.56
```

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 222 ms, sys: 40.3 ms, total: 263 ms
Wall time: 261 ms
Method: Rejection
Number of samples: 1000
Number of simulations: 40000
Threshold: 0.185
Sample means: t1: 0.555, t2: 0.223
```

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

```
Method: Rejection
Number of samples: 1000
Number of simulations: 190000
Threshold: 0.0855
Sample means: t1: 0.561, t2: 0.218
```

```
# 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.

## 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, pool=pool)

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

```
CPU times: user 5.26 s, sys: 37.1 ms, total: 5.3 s
Wall time: 5.3 s
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 1000000
Threshold: 0.036
Sample means: t1: 0.561, t2: 0.227
```

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, pool=pool)

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

```
CPU times: user 636 ms, sys: 1.35 ms, total: 638 ms
Wall time: 638 ms
```

```
Method: Rejection
Number of samples: 1000
Number of simulations: 1000000
Threshold: 0.0452
Sample means: t1: 0.56, t2: 0.228
```

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 1.72 s, sys: 10.6 ms, total: 1.73 s
Wall time: 1.73 s
```

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

```
Number of simulations: 1200000
Threshold: 0.0417
Sample means: t1: 0.561, t2: 0.225
```

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, pool=arraypool)
%time result5 = rej.sample(100, threshold=0.3)
```

```
CPU times: user 25.8 ms, sys: 3.27 ms, total: 29 ms
Wall time: 28.5 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_3521077242 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([ 0.79, -0.01, -1.47, ...,  0.98,  0.18,  0.5 ])
```

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

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

```
arraypool_3521077242
```

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 3 batches
```

You can delete the files with:

```
arraypool.delete()

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

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

```
The directry is removed
```

## 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.

## 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)
        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 quantiles 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: 154 ms, total: 1.87 s
Wall time: 1.56 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.

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

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

## 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)
```

## 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 272 ms, sys: 28 ms, total: 300 ms
Wall time: 2.41 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: 0.0817
Sample means: t1: 0.68, t2: 0.133
```

## 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 --daemon

# 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 3.16 s, sys: 184 ms, total: 3.35 s
Wall time: 13.4 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.

## 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: 0.0136
Sample means: t1: 0.676, t2: 0.129
```

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
```

```
2017-07-19 16:20:58.662 [IPClusterStop] Stopping cluster [pid=21020] with [signal=<Signals.SIGINT: 2>]
```

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

# 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)
```

## 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=[0.1, 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
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.

## 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)
```

## 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...
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.

## 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

%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.

### 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 $\alpha$, the second column has constant zeros, and the last one lists the seeds provided to the command by ELFI.

## 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:

- $\alpha$ - (birth rate) the rate at which any infectious host transmits the disease.
- $\delta$ - (death rate) the rate at which any existing infectious hosts either recovers or dies.
- $\tau$ - (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 $\alpha$ 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))

    # 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]
 [19  1  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]]
```

### 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.
```

## 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)
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
```

## 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()
```

## 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.


## 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.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.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

        # 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.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)

        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
```

## 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:

### `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.

### `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.parameter_inference.`**`ParameterInference`**(*model*, *output_names*, *batch_size=1000*, *seed=None*, *pool=None*, *max_parallel_batches=None*)

A base class for parameter inference methods.

**`model`**
*elfi.ElfiModel* – The ELFI graph used by the algorithm

**`output_names`**
*list* – Names of the nodes whose outputs are included in the batches

**client**
*elfi.client.ClientBase* – The batches are computed in the client

**max_parallel_batches**
*int*

**state**
*dict* – Stores any changing data related to achieving the objective. Must include a key `n_batches` for determining when the inference is finished.

**objective**
*dict* – 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.

**batches**
*elfi.client.BatchHandler* – Helper class for submitting batches to the client and keeping track of their indexes.

**pool**
*elfi.store.OutputPool* – Pool object for storing and reusing node outputs.

Construct the inference algorithm object.

If you are implementing your own algorithm do not forget to call *super*.

**Parameters**

- **model** (`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*) –
- **seed** (*int, optional*) – Seed for the data generation from the ElfiModel
- **pool** (`OutputPool`, *optional*) – OutputPool both stores and provides precomputed 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

**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

**infer**(**args*, *vis=None*, ***kwargs*)
Set the objective and start the iterate loop until the inference is finished.

See the other arguments from the *set_objective* method.

**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.

**Returns**

**Return type** None

**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.

**Returns**

**Return type** None

**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

**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).

**Returns**

**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*) –

**Returns**

**Return type** None

# 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.

## 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.

### 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

### 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.

## 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:

## 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 :)

## Get Started!

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:

```
$ git clone git@github.com:your_name_here/elfi.git
```

3. Install your local copy and the development requirements into a conda environment:

```
$ conda create -n elfi python=3.5 numpy
$ source activate elfi
$ cd elfi
$ make dev
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. Follow the *Style Guidelines*

6. When you're done making changes, check that your changes pass flake8 and the tests:

```
$ make lint
$ make test
```

   Also make sure that the docstrings of your code are formatted properly:

```
$ make docs
```

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

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

8. Submit a pull request through the GitHub website.

## 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.

## 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 Travis-CI.
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.rst.
3. The pull request should work for Python 3.5 and later. Check https://travis-ci.org/elfi-dev/elfi/pull_requests and make sure that the tests pass for all supported Python versions.

## Tips

To run a subset of tests:

```
$ py.test tests.test_elfi
```

# CHAPTER 2

## Citation

If you wish to cite ELFI, please use the paper in arXiv:

```
@misc{1708.00707,
Author = {Jarno Lintusaari and Henri Vuollekoski and Antti Kangasrääsiö and Kusti␣
↪Skytén and Marko Järvenpää and Michael Gutmann and Aki Vehtari and Jukka Corander␣
↪and Samuel Kaski},
Title = {ELFI: Engine for Likelihood Free Inference},
Year = {2017},
Eprint = {arXiv:1708.00707},
}
```

# Index

## G



## H



## I



## L



## M



## N



## O



## P

## R



## S

## U



## V