EuroHPC Joint Undertaking

Data driven Computational Mechanics at EXascale

DCoMEX

Data driven Computational Mechanics at EXascale

Work program topic: EuroHPC-01-2019
Type of action: Research and Innovation Action (RIA)

DELIVERABLE D4.2

Version No 1

https://www.dcomex.eu
This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking ('the JU'), under Grant Agreement No 956201

DOCUMENTATION SUMMARY INFORMATION

Project Title	**Data driven Computational Mechanics at EXascale**
Project Acronym	DCoMEX
Project No	956201
Call Identifier	EuroHPC-01-2019
Project Start Date	01/04/2021
Related work package	WP 4
Related task(s)	Task 4.2
Lead Organisation	ETHZ
Submission date	27/04/2023
Re-submission date	
Dissemination Level	PU

Quality Control:

	Who	**Affiliation**	**Date**
Checked by internal reviewer	Petros Koumoutsakos		27/04/2023
Checked by WP Leader	Eleni Chatzi	ETHZ	27/04/2023
Checked by Project Coordinator	Vissarion Papadopoulos	NTUA	

Document Change History:

Version	**Date**	**Author(s)**	**Affiliation**	**Comment**
1.0	26/04/2023	Sergey Litvinov, Sebastian Kaltenbach	ETHZ	

Deliverable 4.2

In the previous task and Deliverable 4.1, we developed a Transitional Markov Chain Monte Carlo (TMCMC) algorithm based on [6, 2] and implemented it in Korali, a high-performance framework for uncertainty quantification, optimization, and deep reinforcement learning. This algorithm is particularly suitable for sampling the Bayesian posteriors of hierarchical models. We also created a Python module (`graph.Integral`) that enables the description of complex Bayesian problems. The posterior distribution of the Hierarchical Bayesian Model (HBM) is constructed and available for sampling with any algorithm (tested with Metropolis, Langevin, original TMCMC, and Korali's TMCMC implementation). The graph is dynamic and executed on a define-by-run basis. A module defines the graph simply by running the desired computation, rather than specifying a static graph structure. This allows users to utilize any required Python feature (e.g., arbitrary control flow constructs), an idea which was pioneered by the automatic differentiation module of PyTorch [5]. To produce samples, a well-defined Bayesian posterior and a well-defined forward model are necessary, with no cycles in the dependence of all variables. As a result, we developed another module (`follow`) that conducts a test to verify this crucial condition. To accelerate sampling, `graph.Integral` implements the methodology developed in [7] (also described in [4]). The advantage of this approach is that likelihoods, the most expensive part of the computations, are not re-evaluated for each value of a hyperparameter on the previous level.

The module `follow` can output the graph as a DOT (graph description language) file. We can thus make use of the open-source graph library graphviz [3] to perform various tests and operations on this graph.

This report is structured as follows: we first present details about probabilistic graphical models and the used sampling techniques, before shortly introducing the key implementation aspects of the aforementioned algorithm. More information about the implementation including examples can be found in the attached documentation, which is also publicly accessible via the DCoMEX GitHub page.

Probabilistic Graphical Models

A probabilistic graphical model is a probabilistic model for which a graph expresses the conditional dependence structure between random variables [1]. The vertices of a probabilistic graphical model correspond to random variables, and the (directed) edges are conditional densities. These probabilistic graphical models resemble Bayesian networks in case of an acyclic graph.

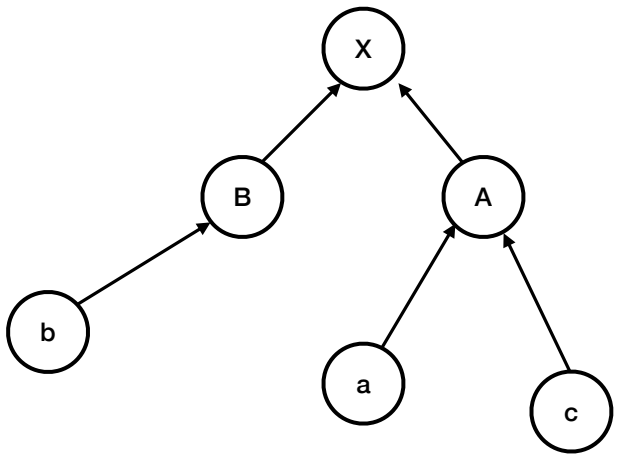


Figure 1: Probabilistic Graphical Model

An example can be found in Figure 1. A possible interpretation of this graph is the following: The data X is conditioned on the variables, A, B which are conditioned on the variables a, b, c. By adding more nodes to the graph, we can incorporate more complicated relations as well as more layers to achieve a hierarchical model. These representations are especially useful for hierarchical models, as the dependencies between different levels can be visualized using the graph structure.
The probabilistic graphical model represents a joint probability density, which can be factorized using the structure of the graph:

$$P(X, A, B, a, b, c) = P(X \mid A, B)P(A \mid a, c)P(a)P(c)P(B \mid b)P(b) \tag{1}$$

An essential prerequisite is that the graphs should be acyclic, as cyclic connections are not admissible. Such connections could potentially result in indeterminate states and disrupt the causal relationship between variables.

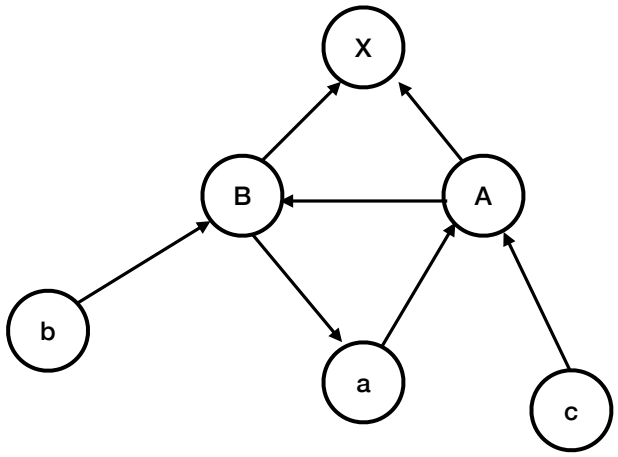


Figure 2: Probabilistic Graphical Model with cyclic structure

An example can be found in Figure 2 which shows the dependence of a on the variable B, which is conditioned on a via A. Given a non-acyclic directed graph as a probabilistic graphical model, we can not factorize the joint probability density anymore, such as for Figure 1.

Sampling from the probabilistic graphical model

As the probabilistic graphical models represent a joint probability density, they can readily be used top generate samples for the quantities of interest. As an example we are again considering the probabilistic graphical model of Figure 1. Following a Bayesian approach we can construct a posterior distribution given the data X:

$$P(A, B, a, b, c \mid X) = \frac{P(X \mid A, B)P(A, B \mid a, b, c)P(a, b, c)}{P(X)} = \frac{P(A, B, a, b, c, X)}{P(X)} \tag{2}$$

Within our framework, we implemented the Metropolis-Algorithm as well as the Metropolis-adjusted-Langevin-Algorithm (MALA) and transitional Markov Chain Monte Carlo to create samples from this posterior. Especially for larger probabilistic graphical models and tMCMC [7], we can make use of the structure of the probabilistic graphical model. In particular, as all dependencies between variables (and thus also all independencies) are given, we can use methods based on importance sampling [4] to deal with different parts of the model individually in case these parts are independent.

Implementation

Details about the implementation and example can be found in the attached documentation of the software, which is also linked online to the GitHub page for the DCoMEX prototype. Here we are shortly summarizing the most important points:

- We implemented a decorator function that allows us to incorporate the new test for acyclic graph directly to the 'old' routines.
- With the help of this decorator function, the graph can be tested for cycles and can be visualized. This is implemented using the open-source library graphviz [3].

Conclusions

We have developed a new module that ensures that a probabilistic graphical model is acyclic. This guarantees that we are able to sample from the posterior of such a model. The implementation has been done in Python and makes use of the open-source library graphviz [3]. Moreover, the implementation is done such that we can readily sample from the probabilistic graphical model using fro instance the sampling algorithm from Korali.
The presented Deliverable 4.2 is important for the DCoMEX project as the presented framework will be used to solve forward

and Bayesian inverse problems for various application areas. Due to relying on a Bayesian network we can readily incorporate uncertainty quantification and compute probabilistic predictions.

References

[1] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[2] Jianye Ching and Yi-Chu Chen. Transitional markov chain monte carlo method for bayesian model updating, model class selection, and model averaging. *Journal of engineering mechanics*, 133(7):816–832, 2007.

[3] John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz and dynagraph—static and dynamic graph drawing tools. *Graph drawing software*, pages 127–148, 2004.

[4] Lina Kulakova, Georgios Arampatzis, Panagiotis Angelikopoulos, Panagiotis Hadjidoukas, Costas Papadimitriou, and Petros Koumoutsakos. Data driven inference for the repulsive exponent of the lennard-jones potential in molecular dynamics simulations. *Scientific reports*, 7(1):16576, 2017.

[5] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[6] Stephen Wu, Panagiotis Angelikopoulos, Costas Papadimitriou, and Petros Koumoutsakos. Bayesian annealed sequential importance sampling (BASIS): an unbiased version of transitional markov chain Monte Carlo. *ASCE-ASME J. Risk Uncertain. Eng. Sys. B*, aug 2017.

[7] Stephen Wu, Panagiotis Angelikopoulos, Gerardo Tauriello, Costas Papadimitriou, and Petros Koumoutsakos. Fusing heterogeneous data for the calibration of molecular dynamics force fields using hierarchical bayesian models. *The Journal of Chemical Physics*, 145(24):244112, 2016.

Appendix

The links to access the website and online documentation for the DCoMEX framework are presented herewith. Moreover, a PDF version of the documentation has also been attached.
`https://github.com/DComEX/dcomex-prototype`
`https://dcomex-framework-prototype.readthedocs.io/en/latest`

DComEX Documentation

NTUA, ETHZ, CSCS

Apr 27, 2023

Contents

1 Introduction 1

2 Follow 2

3 Graphs 4

4 Kahan 10

5 Integration 12

6 Examples 14

Index 42

1 Introduction

The EU-funded DComEX project plans to develop numerical methods enhanced by artificial intelligence as well as a scalable software framework that enables exascale computing. A key innovation of DComEX is the development of AI-Solve, a novel scalable library of AI-enhanced algorithms for solving large-scale sparse linear systems, which are fundamental to computational mechanics. Researchers will fuse physics-constrained machine learning with efficient block-iterative methods and incorporate experimental data at multiple levels of fidelity to quantify model uncertainties. Efficient deployment of these methods in exascale supercomputers will offer scientists and engineers unprecedented capabilities for conducting predictive simulations of mechanical systems in applications ranging from bioengineering to manufacturing.

2 Follow

This module provides a decorator follow that allows functions to be “followed” in a call graph, as well as a function graphviz that generates a call graph of the decorated functions. The decorator can be used with an optional label for the wrapped function in the call graph, and the generated graph can be written to a file-like object.

2.1 Summary

follow.follow([label])	Decorator that allows a function to be "followed" in a call graph.
follow.graphviz(buf)	Decorator that generates a call graph of the decorated functions.
follow.loop()	Determines whether there is a loop in the call graph.

2.2 Functions

follow.**follow**(*label=None*)

Decorator that allows a function to be “followed” in a call graph.

Parameters

label

[str, optional] A label for the wrapped function in the call graph. If not provided, the function name will be used as the label.

Returns

function

A wrapped version of the input function that can be used to generate a call graph.

Examples

```
>>> @follow()
... def add(a, b):
...     return a + b
>>> add(2, 3)
5
```

```
>>> @follow(label='Subtract')
... def sub(a, b):
...     return a - b
>>> sub(5, 2)
3
```

```
>>> add = follow()(lambda a, b: a + b)
>>> add(2, 3)
5
```

follow.**graphviz**(*buf*)

Decorator that generates a call graph of the decorated functions.

Parameters

buf

[TextIOWrapper] A file-like object that the graph will be written to.

Returns

None

Examples

```
>>> from io import StringIO
>>> clear()
>>> @follow(label='Addition')
... def add(a, b):
...     return a + b
>>> @follow(label='Subtraction')
... def sub(a, b):
...     return a - b
>>> @follow()
... def foo():
...     add(1, 2)
...     sub(4, 3)
>>> foo()
>>> buf = StringIO()
>>> graphviz(buf)
>>> print(buf.getvalue())
digraph {
  0 [label = "Addition"]
  1 [label = "Subtraction"]
  2 [label = "foo"]
  2 -> 1
  2 -> 0
}
```

follow.**loop**()

Determines whether there is a loop in the call graph.

Returns

bool
True if there is a loop in the call graph, False otherwise.

Examples

```
>>> clear()
>>> @follow()
... def func1(x):
...     if x > 0:
...         return func2(x-1)
...     else:
...         return 0
>>> @follow()
... def func2(x):
...     if x > 0:
...         return func1(x-1)
...     else:
...         return 0
>>> func2(42)
0
>>> has_loop = loop()
>>> print(has_loop)
True
```

```
>>> clear()
>>> @follow()
... def func1(x):
...     return func2(x-1)
>>>
>>> @follow()
... def func2(x):
...     pass
>>> func2(42)
>>> loop()
False
```

3 Graphs

The code is a Python package containing classes and functions for estimating integrals using various sampling algorithms. The Integral class in the package caches samples and evaluates the integral given a hyperparameter psi. It takes two callable arguments: data_given_theta, which is the joint probability of the observed data viewed as a function of parameter, and theta_given_psi, which is the conditional probability of parameters theta given hyperparameter psi. The method parameter specifies the type of sampling algorithm to use, and the options parameter is a dictionary of options for the sampling algorithm.

The package contains several sampling algorithms implemented as functions, including metropolis and langevin. metropolis is a Metropolis sampler, which takes as arguments a function that calculates the unnormalized density or log unnormalized probability, the number of samples to draw, the initial point, the scale of the proposal distribution,

and a boolean indicating whether to assume log-probability. langevin is a Metropolis-adjusted Langevin (MALA) sampler, which takes as arguments a function that calculates the unnormalized density or log unnormalized probability, the number of samples to draw, the initial point, the gradient of the log unnormalized probability, the standard deviation of the proposal distribution, and a boolean indicating whether to assume log-probability.

3.1 Summary

graph.cmaes(fun, x0, sigma, g_max[, trace])	CMA-ES optimization
graph.Integral(data_given_theta, theta_given_psi)	Caches the samples to evaluate the integral several times.
graph.korali(fun, draws, lo, hi[, beta, ...])	Korali TMCMC sampler
graph.metropolis(fun, draws, init, scale[, log])	Metropolis sampler
graph.tmcmc(fun, draws, lo, hi[, beta, ...])	Generates samples from the target distribution using a transitional Markov chain Monte Carlo(TMCMC) algorithm.

3.2 Functions

class graph.**Integral**(*data_given_theta*, *theta_given_psi*, *method='metropolis'*, ***options*)

Caches the samples to evaluate the integral several times.

Parameters

data_given_theta
[callable] The joint probability of the observed data viewed as a function of parameter.

theta_given_psi
[callable] The conditional probability of parameters theta given hyper-parameter psi.

method
[str or callable, optional] The type of the sampling algorithm to sample from *data_given_theta*. Can be one of:

- 'metropolis' (default)
- 'langevin'
- 'tmcmc'
- 'korali'
- 'hamiltonian' (WIP)

options
[dict, optional] A dictionary of options for the sampling algorithm.

Attributes

samples
: [List] A list of samples obtained from the sampling algorithm.

Raises

ValueError
: If the provided sampling method is unknown.

Examples

```
>>> import random
>>> import numpy as np
>>> random.seed(123456)
>>> np.random.seed(123456)
>>> data = np.random.normal(0, 1, size=1000)
>>> data_given_theta = lambda theta: np.prod(np.exp(-0.5 * (data - theta[0]) ** 2))
>>> theta_given_psi = lambda theta, psi: np.exp(-0.5 * (theta[0] - psi[0]) ** 2)
>>> integral = Integral(data_given_theta, theta_given_psi, method='metropolis',
... init=[0], scale=[0.1], draws=1000)
>>> integral([0])  # Evaluate the integral for psi=0
0.9984153240011582
```

`__call__`(*psi*)
: Compute the integral estimate for a given hyperparameter.

Parameters

psi
: [array_like] The hyperparameter values at which to evaluate the integral estimate.

Returns

float
: The estimated value of the integral.

Examples

```
>>> import random
>>> from scipy.stats import norm
>>> random.seed(123456)
>>> np.random.seed(123456)
>>> def data_given_theta(theta):
...     return norm.pdf(theta[0], loc=1, scale=2) * norm.pdf(theta[0], loc=3,␣
↪scale=2)
>>> def theta_given_psi(theta, psi):
...     return norm.pdf(theta[0], loc=psi[0], scale=[1])
```

(continues on next page)

(continued from previous page)

```
>>> integral = Integral(data_given_theta, theta_given_psi, method="metropolis",
...    draws=1000, scale=[1.0], init=[0])
>>> integral([2])
0.2388726795076229
```

graph.**cmaes**(*fun*, *x0*, *sigma*, *g_max*, *trace=False*)

CMA-ES optimization

Parameters

fun
[callable] a target function

x0
[tuple] the initial point

sigma
[double] initial variance

g_max
[int] maximum generation

trace
[bool] return a trace of the algorithm (default: False)

Return

xmin : tuple

graph.**korali**(*fun*, *draws*, *lo*, *hi*, *beta=1*, *return_evidence=False*)

Korali TMCMC sampler

Parameters

fun
[callable] log-probability

draws
[int] the number of samples to draw

lo, hi
[tuples] the bounds of the initial distribution

beta
[float] The coefficient to scale the proposal distribution. Larger values of beta lead to larger proposal steps and potentially faster convergence, but may also increase the likelihood of rejecting proposals (default is 1)

return_evidence
[bool] If True, return a tuple containing the samples and the evidence (the logarithm of the normalization constant). If False (the default), return only the samples

trace
[bool] If True, return a trace of the algorithm, which is a list of tuples containing the current set of samples and the number of accepted proposals at each iteration. If False (the default), do not return a trace.

Return

samples
[list or tuple] a list of samples, a tuple of (samples, log-evidence), or a trace

graph.**metropolis**(*fun*, *draws*, *init*, *scale*, *log=False*)

Metropolis sampler

Parameters

fun
[callable] The unnormalized density or the log unnormalized probability. If *log* is True, *fun* should return the log unnormalized probability. Otherwise, it should return the unnormalized density.

draws
[int] The number of samples to draw.

init
[tuple] The initial point.

scale
[tuple] The scale of the proposal distribution. Should be the same size as *init*.

log
[bool, optional] If True, assume that *fun* returns the log unnormalized probability. Default is False.

Returns

samples
[list] A list of *draws* samples.

Examples

Define a log unnormalized probability function for a standard normal distribution:

```
>>> import math
>>> import random
>>> import statistics
>>> random.seed(12345)
>>> def log_normal(x):
...     return -0.5 * x[0]**2
```

Draw 1000 samples from the distribution starting at 0, with proposal standard deviation 1:

```
>>> samples = list(metropolis(log_normal, 10000, [0], [1], log=True))
```

Check that the mean and standard deviation of the samples are close to 0 and 1, respectively:

```
>>> math.isclose(statistics.fmean(s[0] for s in samples), 0, abs_tol=0.05)
True
>>> math.isclose(statistics.fmean(s[0]**2 for s in samples), 1, abs_tol=0.05)
True
```

graph.**tmcmc**(*fun*, *draws*, *lo*, *hi*, *beta=1*, *return_evidence=False*, *trace=False*)

Generates samples from the target distribution using a transitional Markov chain Monte Carlo(TMCMC) algorithm.

Parameters

fun
: [callable] log-probability

draws
: [int] the number of samples to draw

lo, hi
: [tuples] the bounds of the initial distribution

beta
: [float] The coefficient to scale the proposal distribution. Larger values of beta lead to larger proposal steps and potentially faster convergence, but may also increase the likelihood of rejecting proposals (default is 1)

return_evidence
: [bool] If True, return a tuple containing the samples and the evidence (the logarithm of the normalization constant). If False (the default), return only the samples

trace
: [bool] If True, return a trace of the algorithm, which is a list of tuples containing the current set of samples and the number of accepted proposals at each iteration. If False (the default), do not return a trace.

Return

samples
: [list or tuple] a list of samples, a tuple of (samples, log-evidence), or a trace

Examples

```
>>> import numpy as np
>>> np.random.seed(123)
>>> def log_prob(x):
...     return -0.5 * sum(x**2 for x in x)
>>> samples = tmcmc(log_prob, 10000, [-5, -5], [5, 5])
>>> len(samples)
10000
>>> np.abs(np.mean(samples, axis=0)) < 0.1
array([ True,  True])
```

4 Kahan

The kahan package provides efficient implementations of Kahan's algorithms for numerical summation and other basic statistical calculations. These algorithms are designed to reduce the loss of precision that can occur when adding a large number of floating-point values.

For more information on Kahan's algorithms and their implementation in the kahan package, please see the package documentation.

References:

Kahan, W. (1965). Pracniques: Further remarks on reducing truncation errors. Communications of the ACM, 8(1), 40-41.

4.1 Summary

kahan.cummean(a)	Cumulative mean.
kahan.cumsum(a)	Cumulative sum.
kahan.cumvariance(a)	Cumulative mean and variance.
kahan.mean(a)	Cumulative mean.
kahan.sum(a)	Cumulative sum.

4.2 Functions

kahan.**cummean**(*a*)

Cumulative mean.

Return an iterator over yielding cumulative means of an input sequence

Parameters

a
: [iterable] Input sequence

Returns

iterator
: An iterator that yields cumulative means.

Examples

```
>>> list(cummean([1, 2, 3, 4]))
[1.0, 1.5, 2.0, 2.5]
```

kahan.**cumsum**(*a*)

Cumulative sum.

Return an iterator over yielding cumulative sums of an input sequence

Parameters

a
: [iterable] Input sequence

Returns

iterator
: An iterator that yields cumulative sums.

Examples

```
>>> list(cumsum([1, 2, 3, 4]))
[1.0, 3.0, 6.0, 10.0]
```

kahan.**cumvariance**(*a*)

Cumulative mean and variance.

Return an iterator over yielding pairs of cumulative mean and cumulative variance of an input sequence

Parameters

a
: [iterable] Input sequence

Returns

iterator
: An iterator that yields pairs of cumulative mean and cumulative variance.

Examples

```
>>> list(cumvariance([1, 7, 4]))
[(1.0, 0.0), (4.0, 9.0), (4.0, 6.0)]
```

kahan.**mean**(*a*)

Cumulative mean.

Return the cumulative mean of an input sequence

Parameters

a
: [iterable] Input sequence

Returns

float
The cumulative mean of the input sequence.

Examples

```
>>> mean([1, 2, 3, 4])
2.5
```

kahan.**sum**(*a*)

Cumulative sum.

Return the cumulative sum of an input sequence

Parameters

a
[iterable] Input sequence

Returns

float
The cumulative sum of the input sequence.

Examples

```
>>> sum([1, 2, 3, 4])
10.0
```

5 Integration

This section describes the integration of MSolve with Korali.

5.1 Approach

The integration of MSolve into Korali is performed through files. To execute one MSolve simulation, corresponding to a Korali sample, Korali performs the following:

1. Create a directory unique to the sample.
2. Create an xml configuration file that contains the parameters values. This file is readable by MSolve and describes the simulation.
3. Execute MSolve. Msolve produces an xml output file containing the results of the simulations.
4. Parse the output file and extract the quantities of interest needed by the optimization or sampling process.

5.2 Python module

Here we describe the bridge python module and its functions.

Summary

integration.bridge.run_msolve(xcoords, ...)	Run an instance of Msolve for given parameters.
integration.bridge.write_config_file(...)	Write an xml configuration file readable by MSolve.
integration.bridge.run_msolve_mock(xcoords, ...)	Run a mock version of msolve.

Interface with MSolve

integration.bridge.**run_msolve**(*xcoords*, *ycoords*, *generation: int*, *sample_id: int*, *parameters: list*)

Run an instance of Msolve for given parameters. This function performs the following steps: - Create a directory unique to the sample - Create an xml configuration file readable by Msolve - Run Msolve - Parse the xml file produced by MSolve - Return the parsed results

Parameters

- **xcoords** (*list*) – x coordinates of the temperature measurements
- **ycoords** (*list*) – y coordinates of the temperature measurements
- **generation** (*int*) – the generation index of the korali experiment (used to create a unique directory)
- **sample_id** (*int*) – the index of the korali sample (used to create a unique directory)
- **parameters** (*list*) – position of the heat source

Returns

the coordinates and values of the temperature measurements (x, y, T)

Utilities

integration.bridge.**write_config_file**(*xcoords*, *ycoords*, *parameters: list*, *filename: str*)

Write an xml configuration file readable by MSolve. The configuration describes the problem of a 2D plate heated by a heat source, with temperature measurements at given locations.

Parameters

- **xcoords** (*list*) – x coordinates of the temperature measurements
- **ycoords** (*list*) – y coordinates of the temperature measurements
- **parameters** (*list*) – position of the heat source
- **filename** (*str*) – name of the configuration file to write

integration.bridge.**run_msolve_mock**(*xcoords*, *ycoords*, *generation: int*, *sample_id: int*, *parameters: list*)

Run a mock version of msolve. For debugging and testing purpose.

6 Examples

6.1 analitical.follow.py

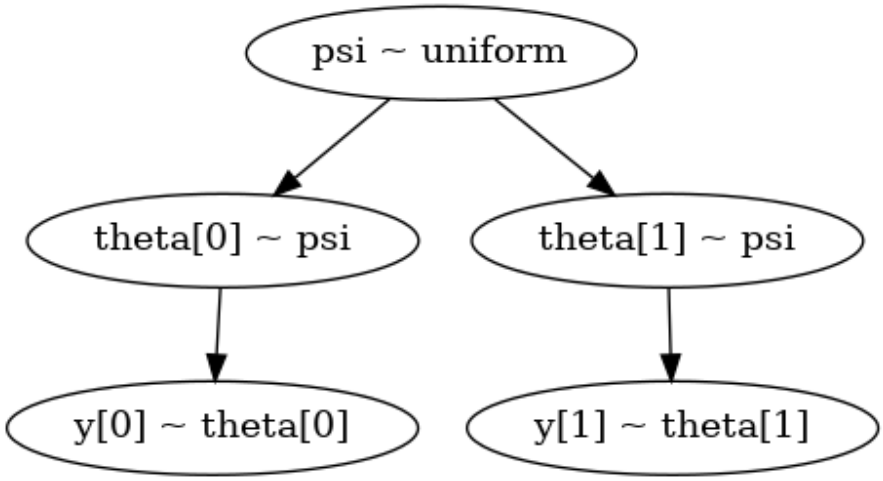
```
import math
import numpy as np
import random
import statistics
import follow
import graph

random.seed(123456)
y = [2, 3]
sigma = 0.1
theta_given_psi = [
    follow.follow("theta[0] ~ psi")(
        lambda theta, psi: statistics.NormalDist(psi, 1).pdf(theta[0])),
    follow.follow("theta[1] ~ psi")(
        lambda theta, psi: statistics.NormalDist(psi, 5).pdf(theta[0])),
]

data_given_theta = [
    follow.follow("y[0] ~ theta[0]")(
        lambda theta: -(theta[0] - y[0])**2 / sigma**2 / 2),
    follow.follow("y[1] ~ theta[1]")(
        lambda theta: -(theta[0] - y[1])**2 / sigma**2 / 2),
]

integral = [
    graph.Integral(data_given_theta[0],
                   theta_given_psi[0],
                   draws=1000,
                   init=[0],
                   scale=[0.1],
                   log=True),
    graph.Integral(data_given_theta[1],
                   theta_given_psi[1],
                   draws=1000,
                   init=[0],
                   scale=[0.1],
                   log=True),
]


@follow.follow(label="psi ~ uniform")
def likelihood(psi):
    return math.prod(fun(psi[0]) for fun in integral)
```

(continues on next page)

(continued from previous page)

```
def prior(psi):
    return 1 if -4 <= psi[0] <= 4 else 0


samples = graph.metropolis(lambda psi: likelihood(psi) * prior(psi),
                           draws=100,
                           init=[0],
                           scale=[1.0])
for s in samples:
    break
print("has loop:", follow.loop())
with open("analitical.follow.gv", "w") as file:
    follow.graphviz(file)
```

```
has loop: False
```



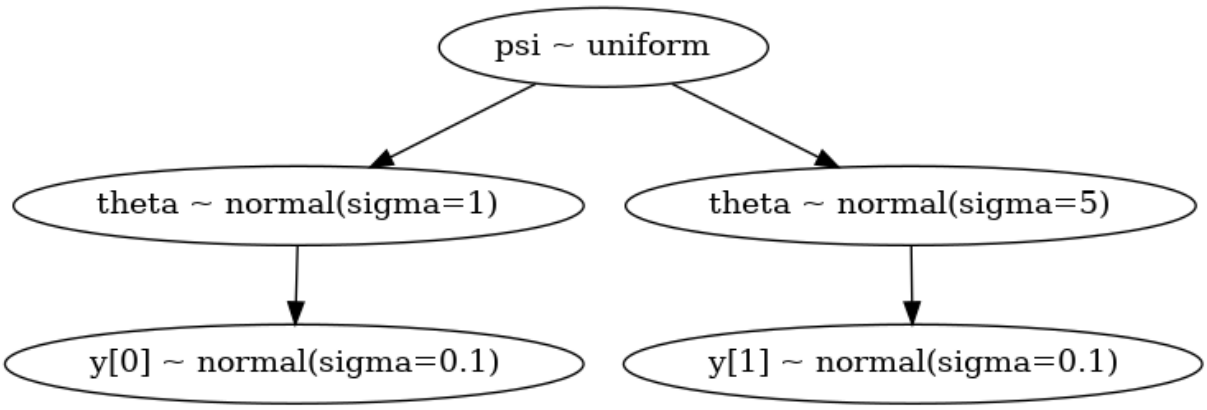
6.2 analitical.py

```
import math
import matplotlib.pylab as plt
import numpy as np
import random
import statistics
import graph

random.seed(123456)
y = [2, 3]
sigma = 0.1
theta_given_psi = [
    lambda theta, psi: statistics.NormalDist(psi, 1).pdf(theta[0]),
    lambda theta, psi: statistics.NormalDist(psi, 5).pdf(theta[0]),
]
```

(continues on next page)

(continued from previous page)

```
data_given_theta = [
    lambda theta: -(theta[0] - y[0])**2 / sigma**2 / 2,
    lambda theta: -(theta[0] - y[1])**2 / sigma**2 / 2,
]

integral = [
    graph.Integral(data_given_theta[0],
                   theta_given_psi[0],
                   draws=1000,
                   init=[0],
                   scale=[0.1],
                   log=True),
    graph.Integral(data_given_theta[1],
                   theta_given_psi[1],
                   draws=1000,
                   init=[0],
                   scale=[0.1],
                   log=True),
]


def likelihood(psi):
    return math.prod(fun(psi[0]) for fun in integral)


def prior(psi):
    return 1 if -4 <= psi[0] <= 4 else 0


def fpost(psi):
    return 0.4225878520215124 * math.exp(-0.5150415081492156 * psi**2 +
                                         2.100150038994303 * psi -
                                         2.160126048590465)


samples = graph.metropolis(lambda psi: likelihood(psi) * prior(psi),
                           draws=5000,
                           init=[0],
                           scale=[1.0])
psi = np.linspace(-4, 4, 100)
post = [fpost(e) for e in psi]
plt.yticks([])
plt.xlim(-2, 5)
plt.hist([e[0] for e in samples],
         50,
         density=True,
         histtype='step',
         linewidth=2)
plt.plot(psi, post)
plt.savefig("analitical.vis.png")
```



6.3 cmaes0.py

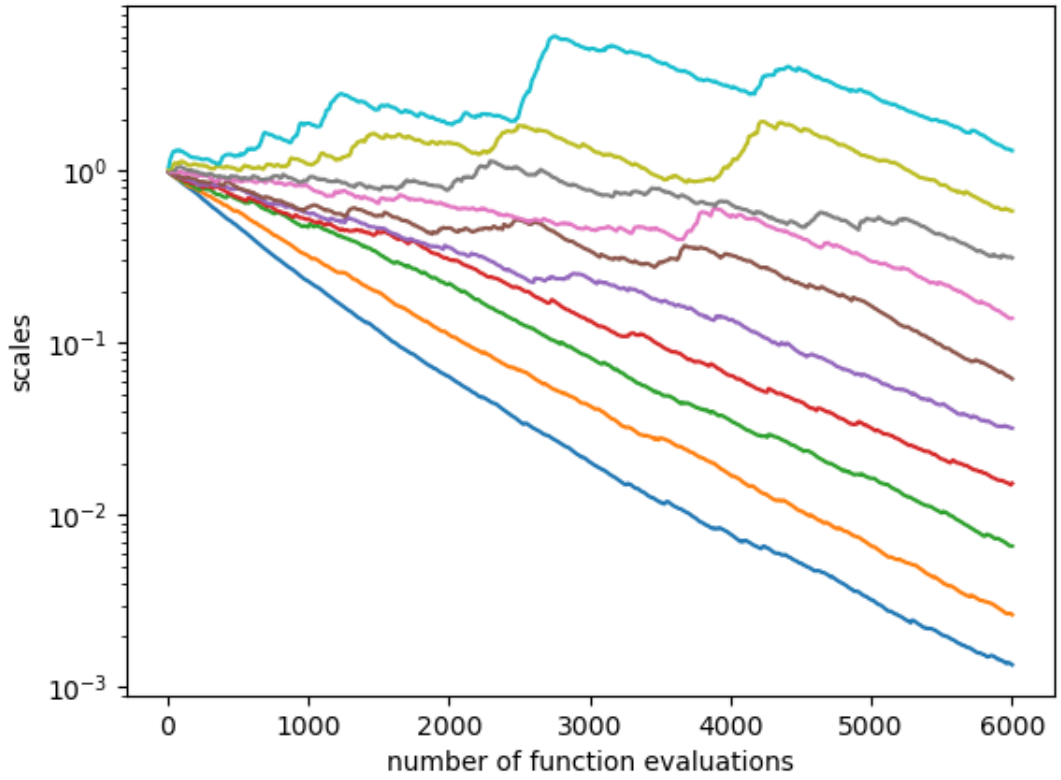
```
import math
import graph
import matplotlib.pyplot as plt
import random
import sys
import scipy.linalg


def print0(l):
    for i in l:
        sys.stdout.write("%+7.2e " % i)
    sys.stdout.write("\n")


def frandom(x):
    return random.uniform(0, 1)


def sphere(x):
    return math.fsum(e**2 for e in x)


def flower(x):
    a, b, c = 1, 1, 4
    return a * math.sqrt(x[0]**2 + x[1]**2) + b * math.sin(
        c * math.atan2(x[1], x[0]))


def elli(x):
    n = len(x)
    return math.fsum(1e6**(i / (n - 1)) * x[i]**2 for i in range(n))


def rosen(x):
    alpha = 100
    return sum(alpha * (x[:-1]**2 - x[1:])**2 + (1 - x[:-1])**2)
```

(continues on next page)

(continued from previous page)

```
def cigar(x):
    return x[0]**2 + 1e6 * math.fsum(e**2 for e in x[1:])


random.seed(1234)
print0(graph.cmaes(elli, (2, 2, 2, 2), 1, 167))
print0(graph.cmaes(sphere, 8 * [1], 1, 100))
print0(graph.cmaes(cigar, 8 * [1], 1, 300))
print0(graph.cmaes(rosen, 8 * [0], 0.5, 439))
print0(graph.cmaes(flower, (1, 1), 1, 200))

trace = graph.cmaes(elli, 10 * [0.1], 0.1, 600, trace=True)
nfev, fmin, xmin, sigma, C, *rest = zip(*trace)
plt.figure()
plt.yscale("log")
plt.xlabel("number of function evaluations")
plt.ylabel("fmin")
plt.plot(nfev, fmin)

plt.figure()
plt.xlabel("number of function evaluations")
plt.ylabel("object variables")
for x in zip(*xmin):
    plt.plot(nfev, x)

plt.figure()
plt.xlabel("number of function evaluations")
plt.ylabel("sigma")
plt.yscale("log")
plt.plot(nfev, sigma)

ratio, scale = [], []
for c in C:
    w = [math.sqrt(e) for e in scipy.linalg.eigvalsh(c)]
    scale.append(w)
    ratio.append(w[-1] / w[0])

plt.figure()
plt.xlabel("number of function evaluations")
plt.ylabel("axis ratio")
plt.yscale("log")
plt.plot(nfev, ratio)

plt.figure()
plt.xlabel("number of function evaluations")
plt.ylabel("scales")
plt.yscale("log")
for s in zip(*scale):
    plt.plot(nfev, s)
plt.savefig("cmaes0.png")
```

```
-1.80e-01 +2.13e-02 -1.89e-03 +2.55e-04
-1.14e-04 -5.98e-06 +8.51e-05 +8.02e-05 -1.39e-05 +3.30e-05 +6.01e-05 -5.29e-06
+2.97e-04 +4.54e-07 +5.48e-07 -5.03e-07 +1.02e-07 -2.24e-07 +9.78e-08 +5.27e-07
+1.00e+00 +1.00e+00 +1.00e+00 +1.00e+00 +1.00e+00 +1.00e+00 +1.00e+00 +1.00e+00
-5.22e-09 +2.16e-09
```



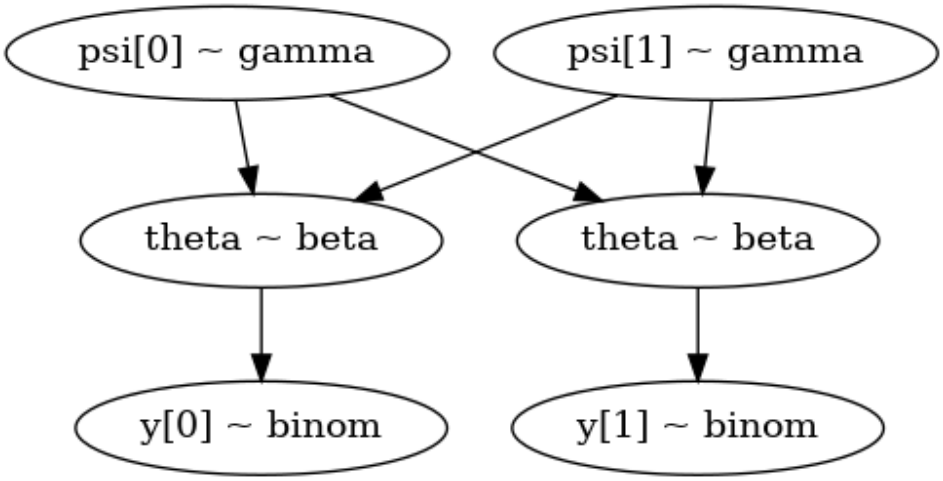
6.4 coins.py

```
import graph
import math
import numpy as np
import pickle
import random
import scipy.stats
import statistics
import sys

random.seed(123456)
n = 250
y = [140, 110]
theta_given_psi = [
```

(continues on next page)

(continued from previous page)

```
    lambda theta, psi: scipy.stats.beta.pdf(theta[0], a=psi[0], b=psi[1]),
    lambda theta, psi: scipy.stats.beta.pdf(theta[0], a=psi[0], b=psi[1]),
]

data_given_theta = [
    lambda theta: scipy.stats.binom.pmf(y[0], n, theta[0]),
    lambda theta: scipy.stats.binom.pmf(y[1], n, theta[0]),
]

init = [y[0] / n, y[1] / n]
scale = 0.05
draws = 100
integral = [
    graph.Integral(data_given_theta[0],
                   theta_given_psi[0],
                   draws=draws,
                   init=[init[0]],
                   scale=[scale]),
    graph.Integral(data_given_theta[1],
                   theta_given_psi[1],
                   draws=draws,
                   init=[init[1]],
                   scale=[scale]),
]


def prior(psi):
    return scipy.stats.gamma.pdf(psi[0], 4, scale=2) * scipy.stats.gamma.pdf(
        psi[1], 4, scale=2)


def likelihood(psi):
    return integral[0](psi) * integral[1](psi)


samples = graph.metropolis(lambda psi: likelihood(psi) * prior(psi),
                           draws=50000,
                           init=[0.1, 0.1],
                           scale=[1.5, 1.5])
with open("coins.samples.pkl", "wb") as f:
    pickle.dump(list(samples), f)
'''
https://allendowney.github.io/BayesianInferencePyMC/04_hierarchical.html#going-
↪hierarchical
'''
```



6.5 coins.vis.py

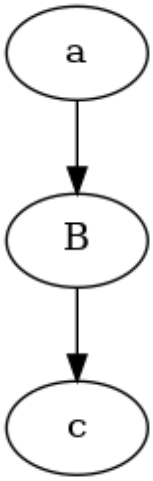
```
import pickle
import numpy as np
import scipy.integrate
import matplotlib.pylab as plt
import statistics
with open("coins.samples.pkl", "rb") as f:
    samples = pickle.load(f)
print(statistics.fmean(e[0] for e in samples))
print(statistics.fmean(e[1] for e in samples))
plt.ylim((0, 0.15))
plt.yticks([])
plt.hist([e[0] for e in samples],
         100,
         density=True,
         histtype='step',
         linewidth=2)
plt.hist([e[1] for e in samples],
         100,
         density=True,
         histtype='step',
         linewidth=2)
plt.savefig("coins.post.png")
```

```
8.476043542775162
8.434431620632731
```

6.6 follow0.py

```
import follow


@follow.follow()
def a():
    b()


@follow.follow(label="B")
def b():
    c(0)


c = follow.follow(label="c")(lambda i: i, )
a()

print("has loop:", follow.loop())
with open("follow0.gv", "w") as file:
    follow.graphviz(file)
```

```
has loop: False
```



6.7 follow1.py

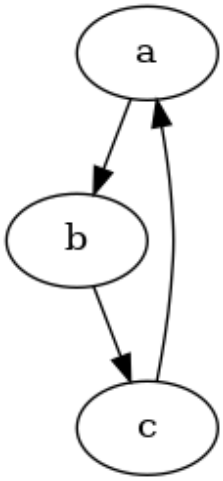
```
import follow


@follow.follow()
def a(i):
    if i > 0:
        b(i - 1)
```

(continues on next page)

(continued from previous page)

```
@follow.follow()
def b(i):
    c(i)


@follow.follow()
def c(i):
    a(i)


a(3)

print("has loop:", follow.loop())
with open("follow1.gv", "w") as file:
    follow.graphviz(file)
```

```
has loop: True
```



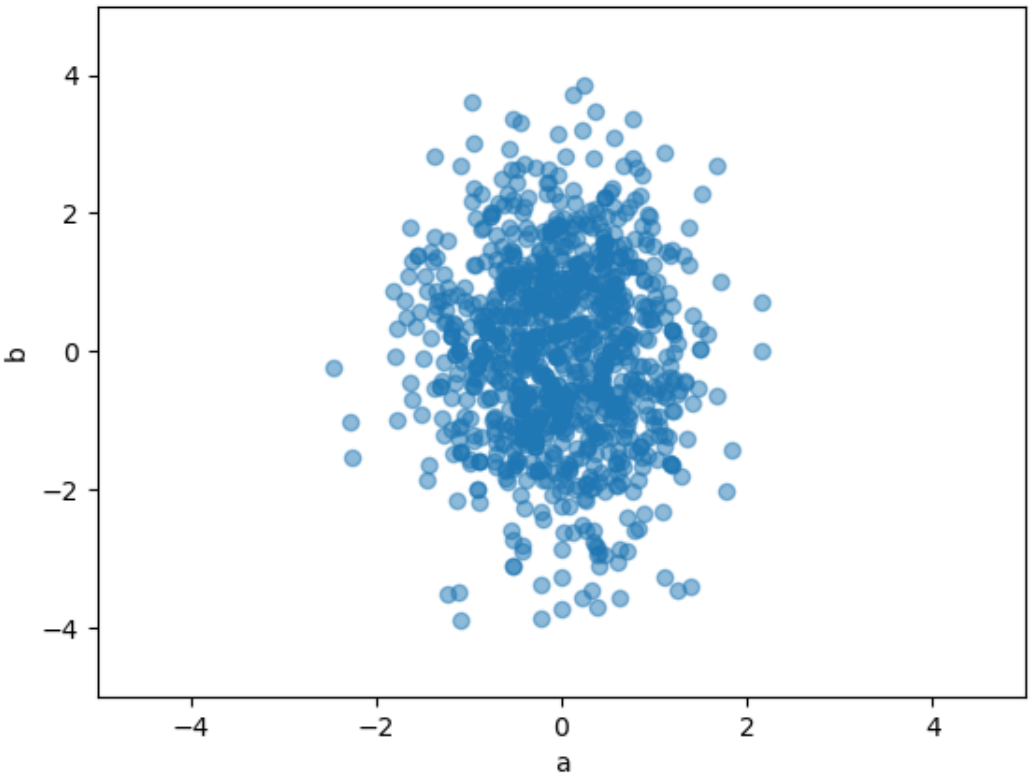
6.8 korali0.py

```
import graph
import matplotlib.pylab as plt

def fun(x):
    a, b = x
    return -a**2 - (b / 2)**2


samples, S = graph.korali(fun, 1000, [-5, -4], [5, 4], return_evidence=True)
print("log evidence: ", S)
plt.plot(*zip(*samples), 'o', alpha=0.5)
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.xlabel("a")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("b")
plt.savefig("korali0.png")
```



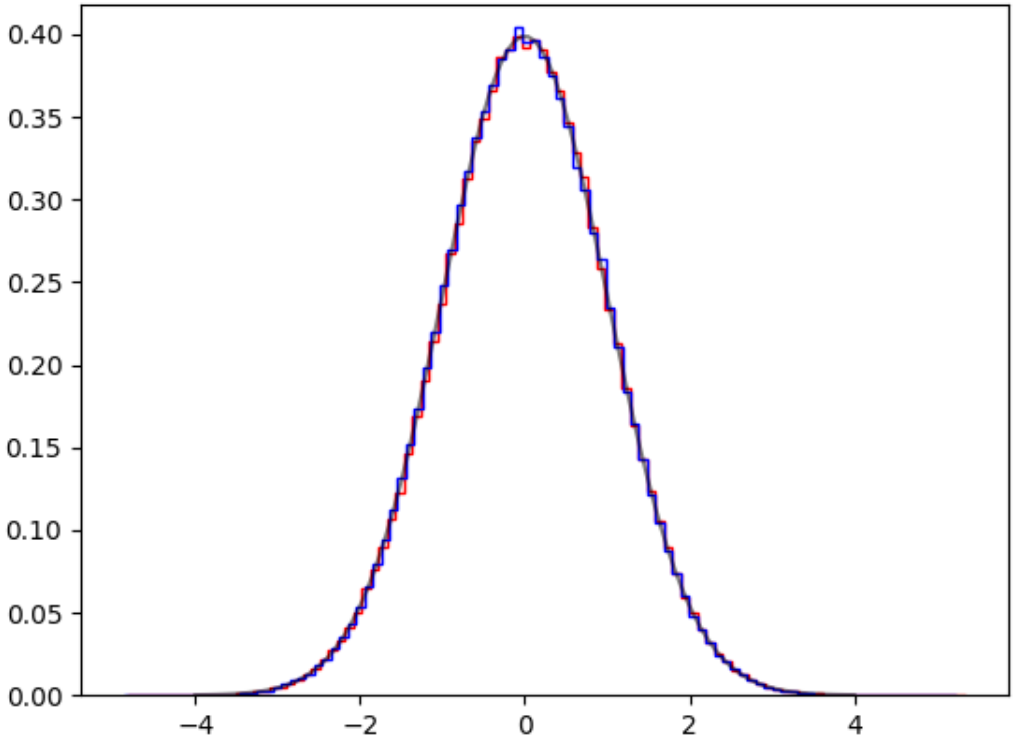
6.9 langevin0.py

```
import math
import random
import matplotlib.pylab as plt
import graph
import numpy as np


def fun(x):
    return math.exp(-x[0]**2 / 2)


def dfun(x):
    return [-x[0]]


random.seed(123456)
```

(continues on next page)

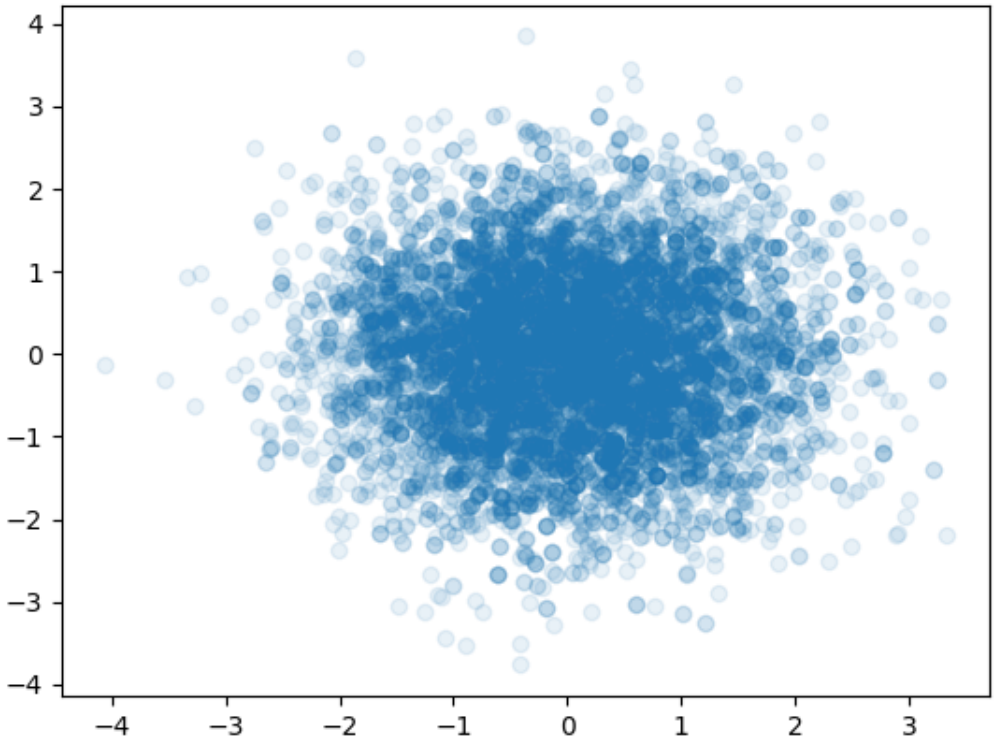
(continued from previous page)

```
log = False
draws = 1000000
sigma = math.sqrt(3)
step = [1.5]
S0 = graph.langevin(fun, draws, [0], dfun, sigma, log=log)
S1 = graph.metropolis(fun, draws, [0], step, log=log)
x = np.linspace(-4, 4, 100)
y = [fun([x]) / math.sqrt(2 * math.pi) for x in x]
plt.hist([e for (e, ) in S0], 100, histtype='step', density=True, color='red')
plt.hist([e for (e, ) in S1], 100, histtype='step', density=True, color='blue')
plt.plot(x, y, '-k', alpha=0.5)
plt.savefig('langevin0.png')
```



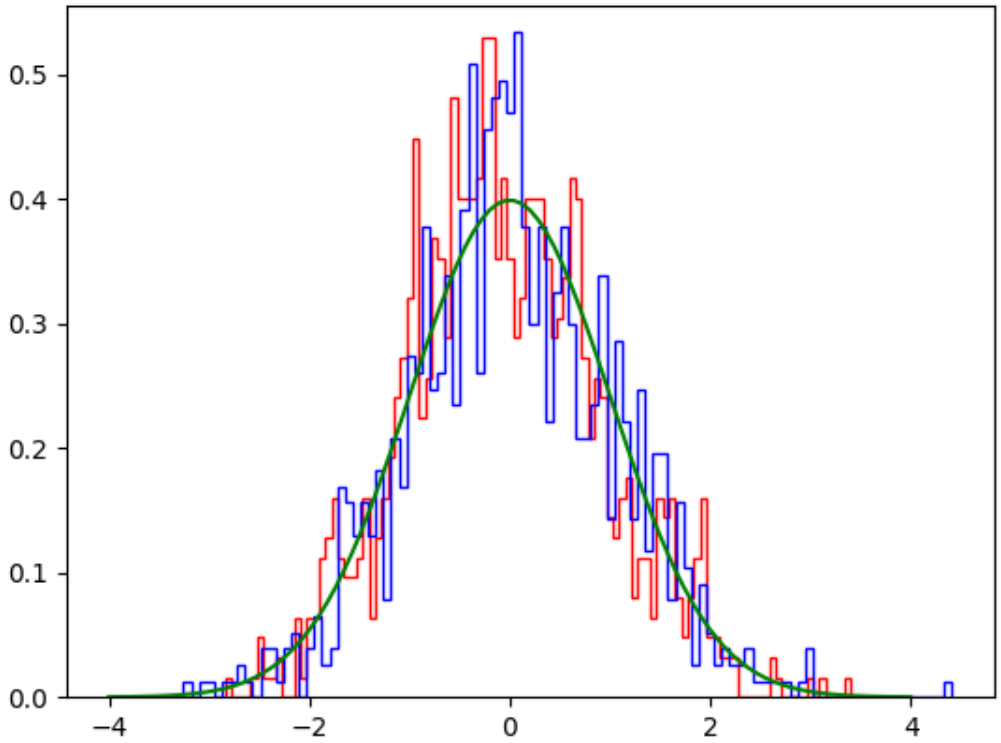
6.10 langevin1.py

```
import math
import random
import matplotlib.pylab as plt
import graph
import numpy as np


def fun(x):
    return -1 / 2 * (x[0]**2 + x[1]**2)


def dfun(x):
    return [-x[0], -x[1]]


random.seed(123456)
draws = 10000
sigma = math.sqrt(3)
S0 = graph.langevin(fun, draws, [0, 0], dfun, sigma, log=True)
plt.scatter(*zip(*S0), alpha=0.1)
plt.savefig("langevin1.png")
```



6.11 langevin2.py

```
import graph
import math
import matplotlib.pyplot as plt
import numpy as np
import tensorflow.compat.v2 as tf
import tensorflow_probability as tfp


def lprob(x):
    return target.log_prob(x)


def fun(x):
    return -1 / 2 * x[0]**2


def dfun(x):
    return [-x[0]]


tfd = tfp.distributions
dtype = np.float32
step_size = 0.75
sigma = math.sqrt(2 * step_size)
target = tfd.Normal(loc=dtype(0), scale=dtype(1))
draws = 1000
S0 = tfp.mcmc.sample_chain(num_results=draws,
                           current_state=dtype(1),
                           kernel=tfp.mcmc.MetropolisAdjustedLangevinAlgorithm(
                               lprob, step_size=step_size),
                           trace_fn=None,
                           seed=42)
S1 = graph.langevin(fun, draws, [0], dfun, sigma, log=True)
x = np.linspace(-4, 4, 100)
y = [math.exp(-x**2 / 2) / math.sqrt(2 * math.pi) for x in x]
plt.hist(S0, 100, histtype='step', density=True, color='red')
plt.hist([e for (e, ) in S1], 100, histtype='step', density=True, color='blue')
plt.plot(x, y, '-g')
plt.savefig('langevin2.png')
```



6.12 langevin3.py

```
import graph
import math
import matplotlib.pyplot as plt
import numpy as np
import tensorflow.compat.v2 as tf
import tensorflow_probability as tfp
from tensorflow_probability.python.distributions import mvn_tril
from tensorflow_probability.python.mcmc import sample
from tensorflow_probability.python.mcmc import langevin


def target_log_prob(z):
    return target.log_prob(z)


dtype = np.float32
true_mean = dtype([1, 2, 7])
true_cov = dtype([[1, 0.25, 0.25], [0.25, 1, 0.25], [0.25, 0.25, 1]])
num_results = 500
num_chains = 500
```

(continues on next page)

(continued from previous page)

```
chol = np.linalg.cholesky(true_cov)
target = mvn_tril.MultivariateNormalTriL(loc=true_mean, scale_tril=chol)
init_state = [
    np.ones([num_chains, 3], dtype=dtype),
]
states = sample.sample_chain(
    num_results=num_results,
    current_state=init_state,
    kernel=langevin.MetropolisAdjustedLangevinAlgorithm(
        target_log_prob_fn=target_log_prob, step_size=.1),
    num_burnin_steps=200,
    num_steps_between_results=1,
    trace_fn=None,
    seed=123456)
states = tf.concat(states, axis=-1)
sample_mean = tf.reduce_mean(states, axis=[0, 1])
x = (states - sample_mean)[..., tf.newaxis]
sample_cov = tf.reduce_mean(tf.matmul(x, tf.transpose(a=x, perm=[0, 1, 3, 2])),
                            axis=[0, 1])
print(sample_mean.numpy())
print(sample_cov.numpy())
```

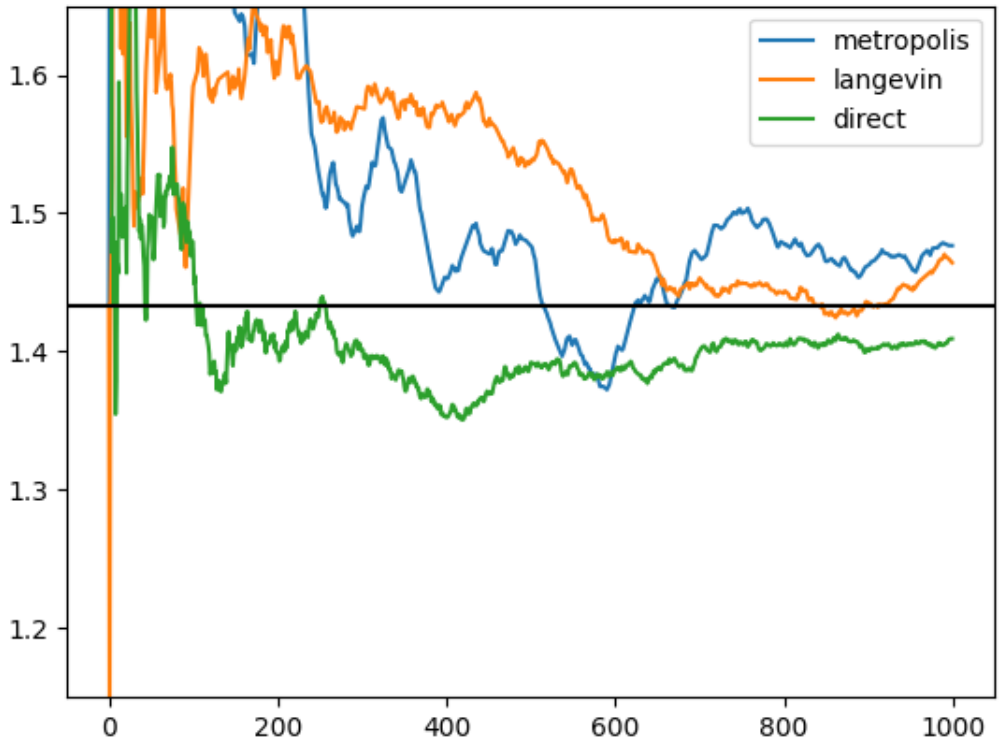
6.13 langevin4.py

```
import graph
import math
import matplotlib.pylab as plt
import random
import kahan


def fun(x):
    a, b = x
    return -1 / 2 * (a**2 * 5 / 4 + b**2 * 5 / 4 + a * b / 2 - (a + b) * y)


def dfun(x):
    a, b = x
    return (2 * y - b - 5 * a) / 4, (2 * y - 5 * b - a) / 4


def direct(draws):
    for i in range(draws):
        a = random.gauss(y / 3, math.sqrt(5 / 6))
        yield [random.gauss((2 * y - a) / 5, math.sqrt(4 / 5))]


random.seed(123456)
y = 4.3
init = (y / 3, y / 3)
```

(continues on next page)

(continued from previous page)

```
sigma = 1.46
scale = (1, 1)
draws = 1000
for samples, label in ((graph.metropolis(fun, draws, init, scale, log=True),
                        "metropolis"), (graph.langevin(fun,
                                                       draws,
                                                       init,
                                                       dfun,
                                                       sigma,
                                                       log=True), "langevin"),
                       (direct(draws), "direct")):
    mean = kahan.cummean(a for (a, *rest) in samples)
    plt.plot(list(mean), label=label)
    plt.ylim(1.15, 1.65)
plt.axhline(init[0], color='k')
plt.legend()
plt.savefig("langevin4.png")
```



6.14 metropolis0.py

```
import graph
import kahan
import scipy.stats
import statistics


def fun(x):
    return scipy.stats.multivariate_normal.logpdf(x, mean, cov)


mean = (1, 2, 7)
cov = ((1, 0.25, 0.25), (0.25, 1, 0.25), (0.25, 0.25, 1))
init = mean
scale = (0.75, 0.75, 0.75)
draws = 40000
S0 = list(graph.metropolis(fun, draws, init, scale, log=True))
x0 = kahan.mean(x for x, y, z in S0)
y0 = kahan.mean(y for x, y, z in S0)
z0 = kahan.mean(z for x, y, z in S0)

xx = kahan.mean((x - x0) * (x - x0) for x, y, z in S0)
xy = kahan.mean((x - x0) * (y - y0) for x, y, z in S0)
xz = kahan.mean((x - x0) * (z - z0) for x, y, z in S0)
yy = kahan.mean((y - y0) * (y - y0) for x, y, z in S0)
yz = kahan.mean((y - y0) * (z - z0) for x, y, z in S0)
zz = kahan.mean((z - z0) * (z - z0) for x, y, z in S0)

print("%.2f %.2f %.2f" % (x0, y0, z0))
print("%.2f %.2f %.2f" % (xx, xy, xz))
print("     %.2f %.2f" % (yy, yz))
print("          %.2f" % zz)
```

```
0.97 2.00 6.98
1.03 0.26 0.27
     1.02 0.27
          0.98
```

6.15 three.follow.py

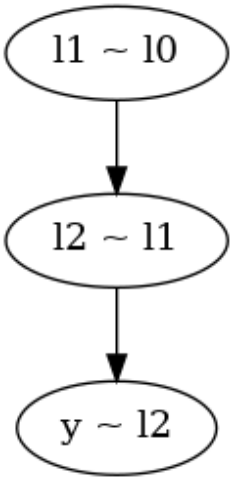
```
import scipy.stats
import graph
import numpy as np
import random
import math
import follow


def prior(psi):
    return 1 if 1 < psi[0] < 40 else 0
```

(continues on next page)

(continued from previous page)

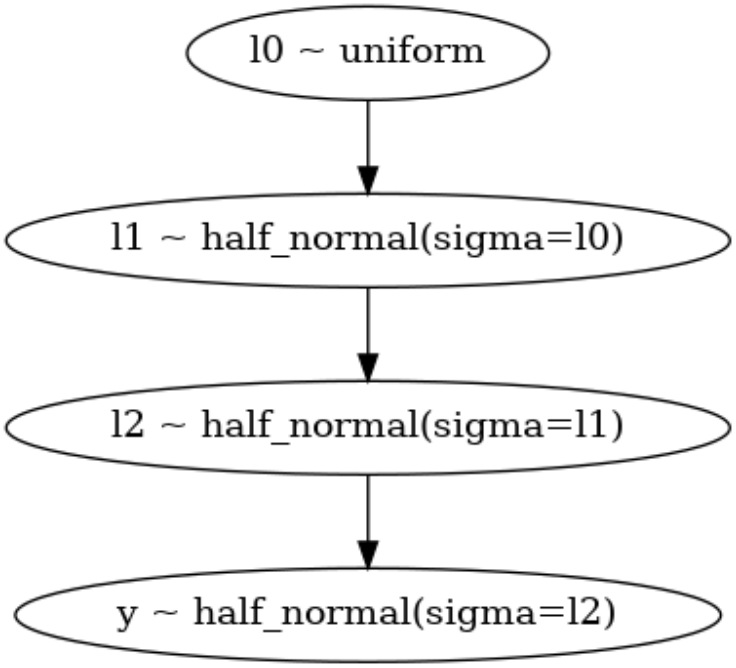
```
seed = 123456
np.random.seed(seed)
random.seed(seed)
data = [0.28, 1.51, 1.14]
l1_given_l0 = follow.follow("l1 ~ l0")(
    lambda theta, psi: scipy.stats.halfnorm.pdf(theta[0], scale=psi[0]))
l2_given_l1 = follow.follow("l2 ~ l1")(
    lambda theta, psi: scipy.stats.halfnorm.pdf(theta[0], scale=psi[0]))
data_given_l2 = follow.follow("y ~ l2")(lambda theta: math.prod(
    scipy.stats.halfnorm.pdf(e, scale=theta[0]) for e in data))
l1 = graph.Integral(data_given_l2,
                    l2_given_l1,
                    draws=10,
                    init=[1],
                    scale=[1.5])
l0 = graph.Integral(l1, l1_given_l0, draws=1000, init=[50], scale=[20])
samples = graph.metropolis(lambda psi: l0(psi) * prior(psi),
                           draws=500,
                           init=[50],
                           scale=[20])
print("has loop:", follow.loop())
with open("three.follow.gv", "w") as file:
    follow.graphviz(file)
```

```
has loop: False
```



6.16 three.py

```
import scipy.stats
import graph
import numpy as np
import random
import math
import pickle


def prior(psi):
    return 1 if 1 < psi[0] < 40 else 0


seed = 123456
np.random.seed(seed)
random.seed(seed)
data = [0.28, 1.51, 1.14]
l1_given_l0 = lambda theta, psi: scipy.stats.halfnorm.pdf(theta[0],
                                                          scale=psi[0])
l2_given_l1 = lambda theta, psi: scipy.stats.halfnorm.pdf(theta[0],
                                                          scale=psi[0])
data_given_l2 = lambda theta: math.prod(
    scipy.stats.halfnorm.pdf(e, scale=theta[0]) for e in data)
l1 = graph.Integral(data_given_l2,
                    l2_given_l1,
                    draws=1000,
                    init=[1],
                    scale=[1.5])
l0 = graph.Integral(l1, l1_given_l0, draws=1000, init=[50], scale=[20])
samples = graph.metropolis(lambda psi: l0(psi) * prior(psi),
                           draws=50000,
                           init=[50],
                           scale=[20])
with open("samples.pkl", "wb") as f:
    pickle.dump(list(samples), f)
```



6.17 three.vis.py

```
import pickle
import numpy as np
import scipy.integrate
import matplotlib.pylab as plt

with open("samples.pkl", "rb") as f:
    samples = pickle.load(f)
D = np.loadtxt("three.dat")
x = D[:, 0]
y0 = D[:, 1]
I0 = scipy.integrate.trapz(y0, x)
plt.yticks([])
plt.hist([e[0] for e in samples],
         40,
         density=True,
         histtype='step',
         linewidth=2)
plt.plot(x, y0 / I0, '-')
plt.savefig("three.vis.png")
```

6.18 tmcmc0.py

```
import graph
import math
import random
import scipy.stats
import statistics
import sys


def fun(x, a, b, w, dev):
    coeff = -1 / (2 * dev**2)
    u = coeff * statistics.fsum((e - d)**2 for e, d in zip(x, a))
    v = coeff * statistics.fsum((e - d)**2 for e, d in zip(x, b))
    return scipy.special.logsumexp((u + math.log(w), v + math.log(1 - w)))


sampler = graph.tmcmc
# sampler = graph.korali

random.seed(12345)
D = (
    ("I", 2, 0.5, 0.5),
    ("II", 2, 0.1, 0.9),
    ("III", 4, 0.5, 0.5),
    ("IV", 4, 0.1, 0.9),
    ("V", 6, 0.5, 0.5),
    ("VI", 6, 0.3, 0.5),
    ("VII", 6, 0.1, 0.5),
    ("VIII", 6, 0.1, 0.9),
)
N = 1000
M = 50
beta = 1.0
print("beta = %g" % beta)
for name, d, dev, w in D:
    a = [0.5] * d
    b = [-0.5] * d
    first_peak = []
    smax = []
    logev = []
    for t in range(M):
        x, S = sampler(lambda x: fun(x, a, b, w, dev),
                       N, [-2] * d, [2] * d,
                       beta=beta,
                       return_evidence=True)
        cnt = 0
        for e in x:
            da = statistics.fsum((u - v)**2 for u, v in zip(e, a))
            db = statistics.fsum((u - v)**2 for u, v in zip(e, b))
            if da < db:
                cnt += 1
        first_peak.append(cnt / N)
```

(continues on next page)

(continued from previous page)

```
        smax.append(statistics.fmean(max(e) for e in x))
        logev.append(S)
    cv = lambda a: (statistics.mean(a), 100 * abs(scipy.stats.variation(a)))
    print("%4s %.2f (%.1f%%) %4.2f (%.1f%%) %4.2f (%.1f%%)" %
          (name, *cv(first_peak), *cv(smax), *cv(logev)))
'''
Example 2: Mixture of Two Gaussians
Table 3.Summary of the Analysis Results for Example 2

[0] Ching, J., & Chen, Y. C. (2007). Transitional Markov chain Monte Carlo
method for Bayesian model updating, model class selection, and model
averaging. Journal of engineering mechanics, 133(7), 816-832.

'''
```

```
beta = 1
   I 0.50 (4.5%) 0.29 (9.9%) -2.36 (1.7%)
  II 0.90 (2.6%) 0.46 (5.4%) -5.81 (2.5%)
 III 0.48 (7.4%) 0.52 (9.3%) -4.76 (1.9%)
  IV 0.87 (11.0%) 0.48 (20.2%) -11.68 (5.3%)
   V 0.51 (12.0%) 0.66 (10.7%) -7.18 (2.5%)
  VI 0.47 (36.2%) 0.36 (52.1%) -10.63 (5.3%)
 VII 0.55 (54.5%) 0.19 (158.9%) -18.28 (8.6%)
VIII 0.81 (29.4%) 0.45 (53.0%) -18.56 (9.1%)
```

6.19 tmcmc1.py

```
import math
import statistics
import graph
import random
import sys


def gauss(x):
    return -statistics.fsum(e**2 for e in x) / 2


random.seed(123456)
sampler = graph.tmcmc
beta = 0.1
N = 2000
M = 50
d = 3
mean = []
var0 = []
var1 = []
var2 = []
lo = -5
hi = 5
```

(continues on next page)

(continued from previous page)

```
for t in range(M):
    x = sampler(gauss, N, d * [lo], d * [hi], beta=beta)
    mean.append(statistics.fmean(e[0] for e in x))
    var0.append(statistics.variance(e[0] for e in x))
    var1.append(statistics.variance(e[1] for e in x))
    var2.append(statistics.variance(e[2] for e in x))
print("%.3f %.4f %.4f %.4f" % (statistics.fmean(mean), statistics.fmean(var0),
                               statistics.fmean(var1), statistics.fmean(var2)))
```

```
beta = 0.1
0.011 0.9969 0.9960 1.0063
```

6.20 tmcmc2.py

```
import graph
import math
import random
import scipy.stats
import statistics
import sys
import matplotlib.pylab as plt


def fun(x, a, b, w, dev):
    coeff = -1 / (2 * dev**2)
    u = coeff * statistics.fsum((e - d)**2 for e, d in zip(x, a))
    v = coeff * statistics.fsum((e - d)**2 for e, d in zip(x, b))
    return scipy.special.logsumexp((u + math.log(w), v + math.log(1 - w)))


random.seed(12345)
beta = float(sys.argv[1])
dev = 0.1
w = 0.9
draws = 500
trace = graph.tmcmc(lambda x: fun(x, [0.5, 0.5], [-0.5, -0.5], 0.9, 0.1),
                    draws, [-2, -2], [2, 2],
                    beta=beta,
                    trace=True)
for i, (x, accept) in enumerate(trace):
    plt.xlim(-2.1, 2.1)
    plt.ylim(-2.1, 2.1)
    plt.gca().set_ymargin(0)
    plt.gca().set_xmargin(0)
    plt.gca().set_axis_off()
    plt.gca().set_aspect('equal')
    plt.subplots_adjust(hspace=0.0, wspace=0.0)
    plt.scatter(*zip(*x), alpha=0.1, edgecolor='none', color='k')
    plt.scatter((0.5, -0.5), (0.5, -0.5), marker='x', color='r')
    plt.title("accept ratio: %6.2f" % (accept / draws))
```

(continues on next page)

(continued from previous page)

```
    plt.savefig("%03d.png" % i)
    plt.close()
```

6.21 smtmcmc.py

```
import sys
import statistics
import random
import math
import scipy.special
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats.distributions


def kahan_cumsum(a):
    ans = []
    s = 0.0
    c = 0.0
    for e in a:
        y = e - c
```

(continues on next page)

(continued from previous page)

```
        t = s + y
        c = (t - s) - y
        s = t
        ans.append(s)
    return ans


def kahan_sum(a):
    s = 0.0
    c = 0.0
    for e in a:
        y = e - c
        t = s + y
        c = (t - s) - y
        s = t
    return s


def inside(x):
    for l, h, e in zip(lo, hi, x):
        if e < l or e > h:
            return False
    return True


def fun0(theta, x, y):
    alpha, beta, sigma = theta
    sigma2 = sigma * sigma
    sigma3 = sigma2 * sigma
    M = len(x)
    dif = [y - alpha * x - beta for x, y in zip(x, y)]
    sumsq = statistics.fsum(dif**2 for dif in dif)
    res = -0.5 * M * (math.log(2 * math.pi) +
                      2 * math.log(sigma)) - 0.5 * sumsq / sigma2
    sx = statistics.fsum(x)
    xx = statistics.fsum(x * x for x in x)
    FIM = [[xx, sx, 0], [sx, M, 0], [0, 0, 2 * M]]
    inv_FIM = np.linalg.inv(FIM) * sigma2
    D, V = np.linalg.eig(inv_FIM)
    dd = statistics.fsum(dif)
    dx = statistics.fsum(dif * x for dif, x in zip(dif, x))
    gradient = [dx / sigma2, dd / sigma2, -M / sigma + sumsq / sigma3]
    return res, gradient, inv_FIM, V, D


def fun(theta):
    return fun0(theta, xd, yd)


seed = 123456
np.random.seed(seed)
random.seed(seed)
```

(continues on next page)

(continued from previous page)

```
alpha = 2
beta = -2
sigma = 2
xd = np.linspace(1, 10, 40)
yd = [alpha * xd + beta + random.gauss(0, sigma) for xd in xd]
N = 100
eps = 0.04
d = 3
lo = (-5, -5, 0)
hi = (5, 5, 10)
conf = 0.68
chi = scipy.stats.distributions.chi2.ppf(conf, d)
x = [[random.uniform(l, h) for l, h in zip(lo, hi)] for i in range(N)]
f = [None] * N
out = [None] * N
for i in range(N):
    f[i], *out[i] = fun(x[i])
x2 = [[None] * d for i in range(N)]
f2 = [None] * N
out2 = [None] * N
gen = 1
p = 0
End = False
cov = [[None] * d for i in range(d)]
while True:
    old_p, plo, phi = p, p, 2
    while phi - plo > eps:
        p = (plo + phi) / 2
        temp = [(p - old_p) * f for f in f]
        M1 = scipy.special.logsumexp(temp) - math.log(N)
        M2 = scipy.special.logsumexp(2 * temp) - math.log(N)
        if M2 - 2 * M1 > math.log(2):
            phi = p
        else:
            plo = p
    if p > 1:
        p = 1
        End = True
    dp = p - old_p
    weight = scipy.special.softmax([dp * f for f in f])
    mu = [kahan_sum(w * e[k] for w, e in zip(weight, x)) for k in range(d)]
    x0 = [[a - b for a, b in zip(e, mu)] for e in x]
    for l in range(d):
        for k in range(l, d):
            cov[k][l] = cov[l][k] = beta * beta * kahan_sum(
                w * e[k] * e[l] for w, e in zip(weight, x0))
    ind = random.choices(range(N), cum_weights=kahan_cumsum(weight), k=N)
    ind.sort()
    delta = np.random.multivariate_normal([0] * d, cov=cov, size=N)
    for i, j in enumerate(ind):
        xp = [a + b for a, b in zip(x[j], delta[i])]
        if inside(xp):
```

(continues on next page)

(continued from previous page)

```
            fp, *rest = fun(xp)
            if fp > f[j] or p * fp > p * f[j] + math.log(random.uniform(0, 1)):
                x[j] = xp[:]
                f[j] = fp
        x2[i] = x[j][:]
        f2[i] = f[j]
    if End:
        break
    x2, x, f2, f, out2, out = x, x2, f, f2, out, out2

xmean = [statistics.fmean(x[i] for x in x2) for i in range(d)]
print(xmean)
# print(fun((1.8577, -1.7009, 1.8435), xd, yd))
```

Index

Symbols

__call__() (*graph.Integral method*), 6

C

cmaes() (*in module graph*), 7
cummean() (*in module kahan*), 10
cumsum() (*in module kahan*), 10
cumvariance() (*in module kahan*), 11

F

follow() (*in module follow*), 2

G

graphviz() (*in module follow*), 2

I

Integral (*class in graph*), 5

K

korali() (*in module graph*), 7

L

loop() (*in module follow*), 3

M

mean() (*in module kahan*), 11
metropolis() (*in module graph*), 8

R

run_msolve() (*in module integration.bridge*), 13
run_msolve_mock() (*in module integration.bridge*), 14

S

sum() (*in module kahan*), 12

T

tmcmc() (*in module graph*), 8

W

write_config_file() (*in module integration.bridge*), 13