PyThia

Release 01.02.2021

Nando Hegemann

Sep 08, 2023

CONTENTS

The PyThia UQ toolbox uses polynomial chaos surrogates to efficiently generate a surrogate of any (parametric) forward problem. The surrogate is fast to evaluate, allows analytical differentiation and has a built-in global sensitivity analysis via Sobol indices. Assembling the surrogate is done non-intrusive by least-squares regression, hence only training pairs of parameter realizations and evaluations of the forward problem are required to construct the surrogate. No need to compute any nasty interfaces for lagacy code.

For more information, see the .

CHAPTER

ONE

WHY THE NAME?

Pythia was the title of the high priestess of the temple of Apollo in Delphi. Hence you could say she used her prophetic abilities to quantify which was uncertain. Moreover, the package is written in python, so…

CHAPTER

TWO

HOW TO CITE PYTHIA

There is no official related article to cite PyThia yet. If you make use of PyThia in a publication, please use the meta data from the `CITATION.cff` file or cite it with a BibTeX entry similar to this:

```
@software{pythia,
    author = {Hegemann, Nando and Heidenreich, Sebastian},
    title = {PyThia UQ Toolbox},
    version = {4.0.0},
    url = {https://pythia-uq.readthedocs.io/en/v4.0.0/},
    year = {2023},
    month = {9}
}
```

CHAPTER

THREE

LICENSE

This work is dual-licensed under GNU Lesser General Public License v3.0 or later and Hippocratic License 3.0 or later. You can choose between one of them if you use this work.

```
SPDX-License-Identifier:  LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL
```

CHAPTER

FOUR

FUNDING

The development of PyThia UQ Toolbox vers. 1 and 2 has been funded by the German Central Innovation Program (ZIM) No. ZF4014017RR7. The development of PyThia UQ Toolbox vers. 3 has been funded by the .

CHAPTER

FIVE

LOGO

Access and usage information about the PyThia logo can be found under the following URL: .

CHAPTER

SIX

CONTENTS

6.1 Installation

The installation of PyThia is very quick and easy.

The latest stable version of PyThia can be installed using pip

```
pip install pythia-uq
```

To install PyThia from source, i.e., if you want to work with the latest (and possibly unstable) changes, simply clone the repository and run the setup script to install PyThia to any environment

```
git clone https://gitlab.com/pythia/pythia.git
cd pythia
pip install .
```

PyThia can then be imported from any location with `import pythia`.

6.2 Tutorials

Here you can find some tutorials that explain the basic functionality of the PyThia software package. If you want to run the tutorials on your own machine, make sure that your have *installed PyThia*.

6.2.1 Tutorials - Basics

Tutorial 01 - Approximation of Functions with Polynomial Chaos

In this tutorial we cover the very basic usage of PyThia by approximating a vector valued function depending on one stoachastic parameter.

The function we want to approximate by a polynomial chaos expansion is a simple sine in both components, i.e.,

$$f(x) = \big(\sin(4\pi x) + 2,\ \sin(3\pi x) + 2\big).$$

So we define the target function first.

```
import numpy as np
import pythia as pt
```

(continues on next page)

(continued from previous page)

```
def target_function(x: np.ndarray) -> np.ndarray:
    f1 = np.sin(4 * np.pi * x) + 2
    f2 = -np.sin(3 * np.pi * x) + 2
    return np.column_stack((f1, f2))
```

To utilize the polynomial chaos expansion implemented in PyThia, we need to define the stoachastic parameter. For this tutorial, we consider the parameter x to be uniformly distributed on the interval $[0, 1]$. Other admissible distributions are *normal*, *gamma* and *beta*.

```
param = pt.parameter.Parameter(name="x", domain=[0, 1], distribution="uniform")
```

We need to specify which terms the sparse PC expansion should include, i.e., create a multiindex set with the *IndexSet* class. Here, we will simply limit the maximal polynomial degree and include all expansion terms with total degree smaller then the chosen degree. The *index* module provides diverse function to generate multiindex arrays, e.g., `tensor_set`, `simplex_set`, `lq_bound_set`, `union`, `intersection`` and `set_difference`. But since we only have one variable in this tutorial, we can also create a list using `numpy`.

```
indices = np.arange(6, dtype=int).reshape(-1, 1)  # includes indices 0, ..., 5
index_set = pt.index.IndexSet(indices)  # try 15 for good a approximation
```

Next we generate training data for the linear regression. Here, we use the distribution specified by the parameter to generate samples. Try and see how the surrogate changes, if you use a different number of samples or a different sampling strategy. We also need weights for the linear regression used to compute the polynomial chaos approximation. The integrals are approximated with a standard empirical integration rule in our case. Thus all the weights are equal and are simply 1 over the number of samples we use. Most importantly, however, we need function evaluations. Note that the shape has to be equal to the number of samples in the first and image dimension in the second component.

```
N = 1000
s = pt.sampler.ParameterSampler([param])
x_train = s.sample(N)
w_train = np.ones(x_train.size) / x_train.size
y_train = target_function(x_train)
```

Since we assembled all the data we need to compute our surrogate, we can finally use the `PolynomialChaos` class of the *pythia.chaos* module.

```
surrogate = pt.chaos.PolynomialChaos([param], index_set, x_train, w_train, y_train)
```

Note: The `PolynomialChaos` class expects a list of parameters to be given.

The `PolynomialChaos` object we just created can do a lot of things, but for the moment we are only interested in the approximation of our function. Let us generate some testing data to see how good our approximation is.

```
N = 1000
x_test = s.sample(N)
y_test = target_function(x_test)
y_approx = surrogate.eval(x_test)
```

This concludes the first tutorial.

Complete Script

The complete source code listed below is located in the `tutorials/` subdirectory of the repository root.

```python
"""
File: tutorials/tutorial_01.py
Author: Nando Hegemann
Gitlab: https://gitlab.com/Nando-Hegemann
Description: Tutorial 01 - Approximation of Functions with Polynomial Chaos.
SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL
"""
import numpy as np
import pythia as pt


def target_function(x: np.ndarray) -> np.ndarray:
    """Target function.

    Parameters
    ----------
    x : np.ndarray
    """
    f1 = np.sin(4 * np.pi * x) + 2
    f2 = -np.sin(3 * np.pi * x) + 2
    return np.column_stack((f1, f2))


print("Tutorial 01 - Approximation of Functions with Polynomial Chaos")

param = pt.parameter.Parameter(name="x", domain=[0, 1], distribution="uniform")

print("parameter information:")
print(param)

indices = np.arange(6, dtype=int).reshape(-1, 1)  # includes indices 0, ..., 5
index_set = pt.index.IndexSet(indices)  # try 15 for good a approximation
print("multiindex information:")
print(f"    number of indices: {index_set.shape[0]}")
print(f"    dimension: {index_set.shape[1]}")
print(f"    maximum dimension: {index_set.max}")
print(f"    number of sobol indices: {len(index_set.sobol_tuples)}")

N = 1000
print(f"generate training data ({N})")
s = pt.sampler.ParameterSampler([param])
x_train = s.sample(N)
w_train = np.ones(x_train.size) / x_train.size
y_train = target_function(x_train)

print("compute PC expansion")
surrogate = pt.chaos.PolynomialChaos([param], index_set, x_train, w_train, y_train)

N = 1000
```

(continues on next page)

(continued from previous page)

```
print(f"generate test data ({N})")
x_test = s.sample(N)
y_test = target_function(x_test)
y_approx = surrogate.eval(x_test)

e_L2 = np.sqrt(np.sum((y_test - y_approx) ** 2) / y_test.shape[0])
e_L2_rel = e_L2 / np.sqrt(np.sum((y_test) ** 2) / y_test.shape[0])
e_max = np.max(np.abs(y_test - y_approx), axis=0)
e_max_rel = np.max(np.abs(y_test - y_approx) / np.abs(y_test), axis=0)
print(f"error L2 (abs/rel): {e_L2:4.2e}/{e_L2_rel:4.2e}")
print("error max (abs/rel):")
print(f"    first component:  {e_max[0]:4.2e}/{e_max_rel[0]:4.2e}")
print(f"    second component: {e_max[0]:4.2e}/{e_max_rel[0]:4.2e}")
```

Tutorial 02 - Approximation of n-D functions with Polynomial Chaos

This tutorial covers the extension of *Tutorial 01 - Approximation of Functions with Polynomial Chaos* to an arbitrary number of stochastic parameters as input for the target function.

For reasons of simplicity, we consider the real valued function

$$f(x) = -\sin(4\pi x_1)\sin(3\pi x_2) + 2.$$

as the target function throughout this tutorial, i.e.,

```
import numpy as np
import pythia as pt

def target_function(x: np.ndarray) -> np.ndarray:
    val = -np.sin(4 * np.pi * x[:, 0]) * np.sin(3 * np.pi * x[:, 1]) + 2
    return val.reshape(-1, 1)
```

First, we define the stochastic input parameter $x = (x_1, x_2)$ with $x \sim \mathcal{U}([0,1]^2)$.

```
param1 = pt.parameter.Parameter(name="x_1", domain=[0, 1], distribution="uniform")
param2 = pt.parameter.Parameter(name="x_2", domain=[0, 1], distribution="uniform")
params = [param1, param2]
```

Next, we need to specify which terms the PC expansion should include. For this, we need the *IndexSet* class of PyThia. We will just take all the expansion terms from the zeroth up to a certain polynomial degree, but for different degrees in each component.

```
sdim = [13, 11]  # stochastic dimensions (tensor)
indices = pt.index.tensor_set(sdim)
index_set = pt.index.IndexSet(indices)
```

What remains is to generate training data for the PC expansion. Here, we use a specific strategy to generate samples that are optimal for training. For more detail on the optimality of the sampling strategy see the work of Cohen & Migliorati (2017). Try and see how the surrogate changes, if you use a different number of samples or a different sampling strategy.

```
N = 1000
s = pt.sampler.WLSTensorSampler(params, sdim)
```

(continues on next page)

(continued from previous page)

```
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)
```

We assembled all the data we need to compute our surrogate and can finally use the `PolynomialChaos` class of the PyThia.chaos method.

```
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)
```

The `PolynomialChaos` object we just created can do a lot of things, but for the moment we are only interested in the approximation of our function. Let us generate some testing data to see how good our approximation is.

```
N = 1000
s = pt.sampler.WLSTensorSampler(params, sdim)
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)
```

Note: For testing, we choose sample realizations drawn according to the distribution of the parameters, not with respect to the weighted Least-Squares distribution we used to generate the training data.

This concludes the second tutorial. Below you find the complete script you can use to run on your own system. This script also computes the approximation error of the PC surrogate and plots the approximation against the target.

Complete Script

The complete source code listed below is located in the `tutorials/` subdirectory of the repository root.

```
"""
File: tutorials/tutorial_02.py
Author: Nando Hegemann
Gitlab: https://gitlab.com/Nando-Hegemann
Description: Tutorial 02 - Approximation of n-D functions with Polynomial Chaos.
SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL
"""
import numpy as np
import pythia as pt


def target_function(x: np.ndarray) -> np.ndarray:
    """Target function.

    Parameters
    ----------
    x : np.ndarray
    """
    val = -np.sin(4 * np.pi * x[:, 0]) * np.sin(3 * np.pi * x[:, 1]) + 2
    return val.reshape(-1, 1)


```

(continues on next page)

(continued from previous page)

```
print("Tutorial 02 - Approximation of n-D functions with Polynomial Chaos")

print("set parameters")
param1 = pt.parameter.Parameter(name="x_1", domain=[0, 1], distribution="uniform")
param2 = pt.parameter.Parameter(name="x_2", domain=[0, 1], distribution="uniform")
params = [param1, param2]

sdim = [13, 11]  # stochastic dimensions (tensor)
indices = pt.index.tensor_set(sdim)
index_set = pt.index.IndexSet(indices)
print("multiindex information:")
print(f"    number of indices: {index_set.shape[0]}")
print(f"    dimension: {index_set.shape[1]}")
print(f"    maximum dimension: {index_set.max}")
print(f"    number of sobol indices: {len(index_set.sobol_tuples)}")

N = 1000
print(f"generate training data ({N})")
s = pt.sampler.WLSTensorSampler(params, sdim)
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)

print("compute pc expansion")
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)

N = 1000
print(f"generate test data ({N})")
test_sampler = pt.sampler.ParameterSampler(params)
x_test = test_sampler.sample(N)
y_test = target_function(x_test)
y_approx = surrogate.eval(x_test)

# error computation
e_L2 = np.sqrt(np.sum((y_test - y_approx) ** 2) / x_test.shape[0])
e_L2_rel = e_L2 / np.sqrt(np.sum((y_test) ** 2) / x_test.shape[0])
e_max = np.max(np.abs(y_test - y_approx), axis=0)
e_max_rel = np.max(np.abs(y_test - y_approx) / np.abs(y_test), axis=0)
print(f"error L2 (abs/rel):  {e_L2:4.2e}/{e_L2_rel:4.2e}")
print(f"error max (abs/rel): {e_max[0]:4.2e}/{e_max_rel[0]:4.2e}")
```

Tutorial 03 - Computation of Sobol Indices

This tutorial covers the approximation of the Sobol indices of a target function, which are used to infer information about the global parameter sensitivity of the model. To verify the results we compute with PyThia, we use the Sobol function as the object of interest, as the Sobol indices are explicitly known for this function. The Sobol function is given by

$$f(x) = \prod_{j=1}^{M} \frac{|4x_j - 2| + a_j}{1 + a_j} \qquad \text{for} \quad a_1, \ldots, a_M \geq 0.$$

Note: The larger a_j, the less is the influence, i.e. the sensitivity, of parameter x_j.

The mean of the Sobol function is $\mathbb{E}[f] = 1$ and the variance reads

$$\mathrm{Var}[f] = \prod_{j=1}^{M}(1 + c_j) - 1 \qquad \text{for} \qquad c_j = \frac{1}{3(1 + a_j)^2}.$$

With this, we can easily compute the Sobol indices by

$$\mathrm{Sob}_{i_1,\ldots,i_s} = \frac{1}{\mathrm{Var}[f]} \prod_{k=1}^{s} c_{i_k}.$$

An implementation of the Sobol function method `sobol_function()` and the analytical Sobol indices method `sobol_sc()` is included in the complete script at the end of this tutorial.

First, we choose some values for the coefficients $a_1, \ldots, a_M$ and with this the target function.

```
import numpy as np
import pythia as pt

a = np.array([1, 2, 3])
def target_function(x: np.ndarray) -> np.ndarray:
    return sobol_function(x, a=a)
```

Additionally, we compute the analytical Sobol indices.

```
sobol_dict = sobol_sc(a=a, dim=len(a))[0]
sobol_coefficients = np.array(list(sobol_dict.values())).reshape(-1, 1)
```

Next we define the necessary quantities for the PC expansion. For more information see *Tutorial 01 - Approximation of Functions with Polynomial Chaos*. As stochastic parameters we need to choose uniformly distributed variables x_j on $[0, 1]$ according to the number of coefficients $a_1, \ldots, a_M$, i.e.,

```
params = [
    pt.parameter.Parameter(name=f"x_{j+1}", domain=[0, 1], distribution="uniform")
    for j in range(a.size)
]
```

As expansion terms we choose multivariate Legendre polynomials of total polynomial degree less then 11

```
max_dim = 11
# limit total polynomial degree of expansion terms to 10
indices = pt.index.simplex_set(len(params), max_dim - 1)
index_set = pt.index.IndexSet(indices)
```

The last thing we need are training data. We generate the training pairs again by an optimal weighted distribution, see *Tutorial 02 - Approximation of n-D functions with Polynomial Chaos* for more detail.

```
N = 10_000
s = pt.sampler.WLSTensorSampler(params, [max_dim - 1] * len(params))
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)
```

With this, we compute the PC expansion of the `target_function` with PyThia

```
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)
```

As the approximative Sobol indices can be easily derived from the PC expansion coefficients, the `PolynomialChaos` class computes them automatically upon initialization. They can be accessed via `surrogate.sobol`, which is an array ordered according to `index_set.sobol_tuples`. Hence it is easy to verify, if the approximation of the Sobol indices was done correctly.

```
print("Comparison of Sobol indices")
print(f" {'sobol_tuple':<12} {'exact':<8}  {'approx':<8}  {'abs error':<9}")
print("-" * 44)
for j, sdx in enumerate(sobol_dict.keys()):
    print(
        f" {str(sdx):<11} ",  # Sobol index subscripts
        f"{sobol_coefficients[j, 0]:<4.2e} ",
        f"{surrogate.sobol[j, 0]:<4.2e} ",
        f"{np.abs(sobol_coefficients[j, 0] - surrogate.sobol[j, 0]):<4.2e}",
    )
```

This concludes this tutorial.

Complete Script

The complete source code listed below is located in the `tutorials/` subdirectory of the repository root.

```
"""
File: tutorials/tutorial_03.py
Author: Nando Hegemann
Gitlab: https://gitlab.com/Nando-Hegemann
Description: Tutorial 03 - Computation of Sobol Indices.
SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL
"""
import numpy as np
import pythia as pt


def sobol_function(x: np.ndarray, a: np.ndarray | None = None, **kwargs) -> np.ndarray:
    """Sobol function.

    Parameters
    ----------
    x : np.ndarray
        Input values.
    a : np.ndarray | None, optional
        Coefficients, by default None

    Returns
    -------
    :
        Sobol function values.

    Raises
```

(continues on next page)

(continued from previous page)

```
    ------
    ValueError
        Wrong dimension for input `x`.
    ValueError
        Wrong shape of Coefficients `a`.
    """
    if not 0 < x.ndim < 3:
        raise ValueError("Wrong ndim: {}".format(x.ndim))
    if x.ndim == 1:
        x.shape = 1, -1
    if a is None:
        a = np.zeros(x.shape[1])
    elif not a.shape == (x.shape[1],):
        raise ValueError("Wrong shape: {}".format(a.shape))
    return np.prod((abs(4.0 * x - 2.0) + a) / (1.0 + a), axis=1).reshape(-1, 1)


def sobol_sc(a: np.ndarray, dim: int = 1, **kwargs) -> tuple | np.ndarray:
    """Sobol function Sobol indices.

    Parameters
    ----------
    a : np.ndarray
        Coefficients.
    dim : int, optional
        Parameter dimension, by default 1.

    Returns
    -------
    :
        Sobol indices of Sobol function.
    """
    sobol = {}
    beta = (1.0 + a) ** (-2) / 3
    var = np.prod(1.0 + beta) - 1.0
    sobol_tuples = pt.index.IndexSet(pt.index.tensor_set([1, 1, 1])).sobol_tuples
    for sdx in sobol_tuples:
        sobol[sdx] = 1.0 / var
        for k in sdx:
            sobol[sdx] *= beta[k - 1]
    if dim > 1:
        return np.array([sobol for _ in range(dim)])
    else:
        return sobol


print("Tutorial 03 - 2D approximation with PC")

# target function definition
a = np.array([1, 2, 3])


```

(continues on next page)

(continued from previous page)

```python
def target_function(x: np.ndarray) -> np.ndarray:
    """Target function.

    Parameters
    ----------
    x : np.ndarray
    """
    return sobol_function(x, a=a)


# analytical sobol coefficients
sobol_dict = sobol_sc(a=a, dim=len(a))[0]
sobol_coefficients = np.array(list(sobol_dict.values())).reshape(-1, 1)

# setup pc surrogate
params = [
    pt.parameter.Parameter(name=f"x_{j+1}", domain=[0, 1], distribution="uniform")
    for j in range(a.size)
]

max_dim = 11
# limit total polynomial degree of expansion terms to 10
indices = pt.index.simplex_set(len(params), max_dim - 1)
index_set = pt.index.IndexSet(indices)
print("multiindex information:")
print(f"    number of indices: {index_set.shape[0]}")
print(f"    dimension: {index_set.shape[1]}")
print(f"    number of sobol indices: {len(index_set.sobol_tuples)}")

N = 10_000
print(f"generate training data ({N})")
s = pt.sampler.WLSTensorSampler(params, [max_dim - 1] * len(params))
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)

print("compute pc expansion")
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)

# test PC approximation
N = 1000
print(f"generate test data ({N})")
s_test = pt.sampler.ParameterSampler(params)
x_test = s_test.sample(N)
y_test = target_function(x_test)
y_approx = surrogate.eval(x_test)

error_L2 = np.sqrt(np.sum((y_test - y_approx) ** 2) / N)
error_L2_rel = error_L2 / np.sqrt(np.sum((y_test) ** 2) / N)
error_max = np.max(np.abs(y_test - y_approx))
error_max_rel = np.max(np.abs(y_test - y_approx) / np.abs(y_test))

```

(continues on next page)

(continued from previous page)

```
print(f"test error L2 (abs/rel): {error_L2:4.2e} / {error_L2_rel:4.2e}")
print(f"test error max (abs/rel): {error_max:4.2e} / {error_max_rel:4.2e}")

# compare Sobol indices
print("Comparison of Sobol indices")
print(f" {'sobol_tuple':<12} {'exact':<8}  {'approx':<8}  {'abs error':<9}")
print("-" * 44)
for j, sdx in enumerate(sobol_dict.keys()):
    print(
        f" {str(sdx):<11} ",  # Sobol index subscripts
        f"{sobol_coefficients[j, 0]:<4.2e} ",
        f"{surrogate.sobol[j, 0]:<4.2e} ",
        f"{np.abs(sobol_coefficients[j, 0] - surrogate.sobol[j, 0]):<4.2e}",
    )
```

Tutorial 04 - Automatic choice of expansion terms

This tutorial covers the automatic generation of sparse PC expansion indices based on a crude approximation of the Sobol indices. To verify the results we compute with PyThia, we use the Sobol function as the object of interest, as the Sobol indices are explicitly known for this function. The Sobol function is given by

$$f(x) = \prod_{j=1}^{M} \frac{|4x_j - 2| + a_j}{1 + a_j} \qquad \text{for} \quad a_1, \ldots, a_M \geq 0.$$

Note: The larger a_j, the less is the influence, i.e., the sensitivity, of parameter x_j.

The mean of the Sobol function is $\mathbb{E}[f] = 1$ and the variance reads

$$\mathrm{Var}[f] = \prod_{j=1}^{M}(1 + c_j) - 1 \qquad \text{for} \qquad c_j = \frac{1}{3(1 + a_j)^2}.$$

With this, we can easily compute the Sobol indices by

$$\mathrm{Sob}_{i_1,\ldots,i_s} = \frac{1}{\mathrm{Var}[f]} \prod_{k=1}^{s} c_{i_k}.$$

An implementation of the Sobol function method `sobol_function()` and the analytical Sobol indices method `sobol_sc()` is included in the complete script at the end of this tutorial.

First, we choose some values for the coefficients $a_1, \ldots, a_M$ and with this the target function.

```
import numpy as np
import pythia as pt

a = np.array([1, 2, 4, 8])
def target_function(x: np.ndarray) -> np.ndarray:
    return sobol_function(x, a=a)
```

Additionally, we compute the analytical Sobol indices.

```
sobol_dict = sobol_sc(a=a, dim=len(a))[0]
sobol_coefficients = np.array(list(sobol_dict.values())).reshape(-1, 1)
```

Next we define the necessary quantities for the PC expansion. For more information see *Tutorial 01 - Approximation of Functions with Polynomial Chaos*. As stochastic parameters we need to choose uniformly distributed variables x_j on $[0, 1]$ according to the number of coefficients $a_1, \ldots, a_M$, i.e.,

```
params = [
    pt.parameter.Parameter(name=f"x_{j+1}", domain=[0, 1], distribution="uniform")
    for j in range(a.size)
]
```

We want to compare the approximation results of the PC expansion using automatically generated expansion indices with a choice done by hand. For this we compute a reference PC expansion using multivariate Legendre polynomials of total degree less then 7.

```
dim = 7
indices = pt.index.simplex_set(len(params), dim - 1)
index_set = pt.index.IndexSet(indices)
```

The last thing we need are training data. We generate the training pairs again by an optimal weighted distribution, see *Tutorial 02 - Approximation of n-D functions with Polynomial Chaos* for more detail.

```
N = 10_000
s = pt.sampler.WLSTensorSampler(params, [dim] * len(params))
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)
```

With this, we can compute a reference PC expansion for our forward model with

```
surrogate_ref = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)
```

With this out of the way we now focus on choosing a more sparse set of PC expansion indices automatically. To fit our needs, we specify the maximal number of expansion terms we want the PC expansion to have as well as the truncation threshold. Computing the "optimal" multiindices can then be done using `pt.chaos.find_optimal_indices`. To show the efficiency of the automatic multiindex computation, we choose an expansion with half the number of terms (~ 100) as the reference PC (~ 200) and set the truncation threshold to 10^{-3}.

```
max_terms = index_set.shape[0] // 2
threshold = 1.0e-03
indices, sC = pt.chaos.find_optimal_indices(
    params, x_train, w_train, y_train, max_terms=max_terms, threshold=threshold
)
index_set = pt.index.IndexSet(indices)
```

For a detailed explaination on the workings of `pt.chaos.find_optimal_indices` we refer to the module documentation. However, we want to provide a small explaination of the `threshold` input. In principle `find_optimal_indices` computes a very inaccurate PC expansion with far too many terms for the given amount of training data to obtain a very crude approximation of as many Sobol indices as possible. The expansion is chosen so that at least one expansion term is computed for each Sobol index. Depending on the `threshold` the function decides which Sobol indices, i.e., parameter combinations are most relevant. This means, that the function will exclude all multiindices associated to Sobol indices that have a (combined) contribution of less then `threshold`. The multiindices are then distributed according to the magnitude of the crude Sobol index computation. We use this option here only for showcasing the results later on.

Computing the PC expansion with the automatically chosen multiindices can now be done as before.

```
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)
```

What remains is to check if the approximation of the Sobol function using only half the expansion terms is comparable in both the approximation error as well as the approximate Sobol indices. If everything went right you should see that the approximation error of both PC expansions is of the same order of magnitude and that the Sobol indices coincide where they are greater then 10^{-3}.

```
error_L2_ref = np.sqrt(np.sum((f_test - f_approx_ref) ** 2) / x_test.shape[0])
error_L2 = np.sqrt(np.sum((f_test - f_approx) ** 2) / x_test.shape[0])
print(f"test error L2 auto: {error_L2:4.2e}")
print(f"test error L2 ref:  {error_L2_ref:4.2e}")

# print Sobol coefficients
print("Comparison of Sobol indices")
print(
    f" {'sobol_tuples':<13} {'exact':<8}  {'pc-auto':<8}  {'(crude)':<8}  {'pc-ref':<8}"
)
print("-" * 54)
for j, sdx in enumerate(index_set.sobol_tuples):
    print(
        f" {str(sdx):<12} ",  # Sobol index subscripts
        f"{sobol_coefficients[j, 0]:<4.2e} ",
        f"{surrogate.sobol[j, 0]:<4.2e} ",
        f"{sC[j][0]:<4.2e} ",
        f"{surrogate_ref.sobol[j, 0]:<4.2e} ",
    )
```

This concludes this tutorial.

Complete Script

The complete source code listed below is located in the `tutorials/` subdirectory of the repository root.

```
"""
File: tutorials/tutorial_03.py
Author: Nando Hegemann
Gitlab: https://gitlab.com/Nando-Hegemann
Description: Automatic choice of expansion terms.
SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL
"""
import numpy as np
import pythia as pt


def sobol_function(x: np.ndarray, a: np.ndarray | None = None, **kwargs) -> np.ndarray:
    """Sobol function.

    Parameters
    ----------
    x : np.ndarray
        Input values.
```

(continues on next page)

(continued from previous page)

```
    a : np.ndarray | None, optional
        Coefficients, by default None

    Returns
    -------
    :
        Sobol function values.

    Raises
    ------
    ValueError
        Wrong dimension for input `x`.
    ValueError
        Wrong shape of Coefficients `a`.
    """
    if not 0 < x.ndim < 3:
        raise ValueError("Wrong ndim: {}".format(x.ndim))
    if x.ndim == 1:
        x.shape = 1, -1
    if a is None:
        a = np.zeros(x.shape[1])
    elif not a.shape == (x.shape[1],):
        raise ValueError("Wrong shape: {}".format(a.shape))
    return np.prod((abs(4.0 * x - 2.0) + a) / (1.0 + a), axis=1).reshape(-1, 1)


def sobol_sc(a: np.ndarray, dim: int = 1, **kwargs) -> tuple | np.ndarray:
    """Sobol function Sobol indices.

    Parameters
    ----------
    a : np.ndarray
        Coefficients.
    dim : int, optional
        Parameter dimension, by default 1.

    Returns
    -------
    :
        Sobol indices of Sobol function.
    """
    sobol = {}
    beta = (1.0 + a) ** (-2) / 3
    var = np.prod(1.0 + beta) - 1.0
    sobol_tuples = pt.index.IndexSet(pt.index.tensor_set([1] * len(a))).sobol_tuples
    for sdx in sobol_tuples:
        sobol[sdx] = 1.0 / var
        for k in sdx:
            sobol[sdx] *= beta[k - 1]
    if dim > 1:
        return np.array([sobol for _ in range(dim)])
    else:
```

(continues on next page)

(continued from previous page)

```
        return sobol


print("Tutorial 04 - Automatic choice of expansion terms")

# function definitions
a = np.array([1, 2, 4, 8])


def target_function(x: np.ndarray) -> np.ndarray:
    """Target function.

    Parameters
    ----------
    x : np.ndarray
    """
    return sobol_function(x, a=a)


# analytical Sobol coefficients
sobol_dict = sobol_sc(a=a, dim=len(a))[0]
sobol_coefficients = np.array(list(sobol_dict.values())).reshape(-1, 1)

# setup reference PC surrogate
params = [
    pt.parameter.Parameter(name=f"x_{j+1}", domain=[0, 1], distribution="uniform")
    for j in range(a.size)
]

dim = 7
indices = pt.index.simplex_set(len(params), dim - 1)
index_set = pt.index.IndexSet(indices)
print("multiindex information:")
print(f"    number of indices: {index_set.shape[0]}")
print(f"    dimension: {index_set.shape[1]}")
print(f"    number of sobol indices: {len(index_set.sobol_tuples)}")

N = 10_000
print(f"generate training data ({N})")
s = pt.sampler.WLSTensorSampler(params, [dim] * len(params))
x_train = s.sample(N)
w_train = s.weight(x_train)
y_train = target_function(x_train)

print("compute pc expansion")
surrogate_ref = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)

# auto-generate PC multiindices
max_terms = index_set.shape[0] // 2
threshold = 1.0e-03
print("compute optimal mdx")
indices, sC = pt.chaos.find_optimal_indices(
```

(continues on next page)

(continued from previous page)

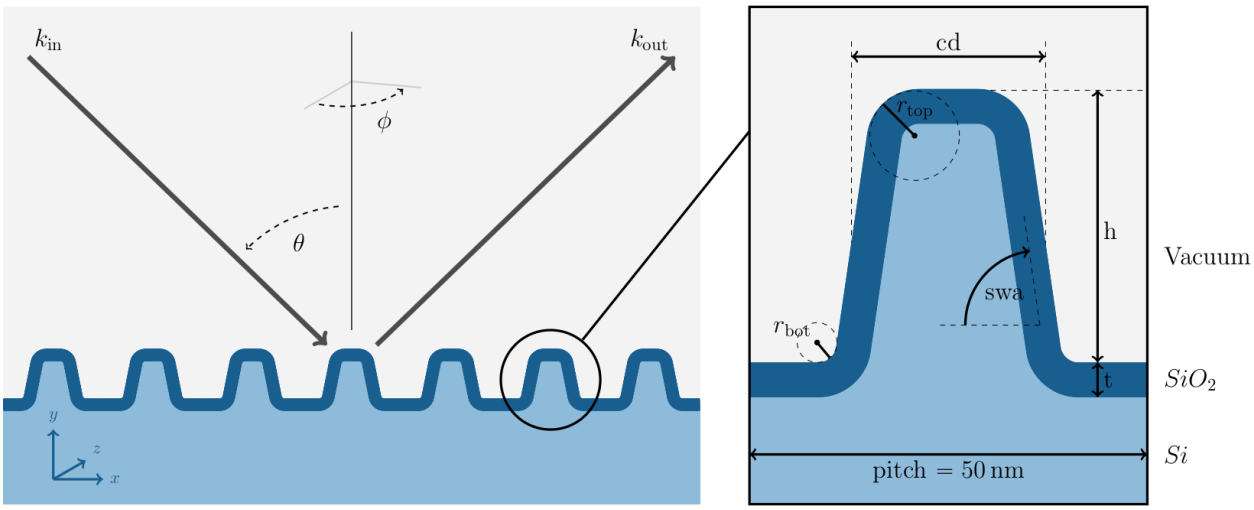
```
    params, x_train, w_train, y_train, max_terms=max_terms, threshold=threshold
)
index_set = pt.index.IndexSet(indices)
print("automatic multiindex information:")
print(f"    number of indices: {index_set.shape[0]}")
print(f"    dimension: {index_set.shape[1]}")
print(f"    number of sobol indices: {len(index_set.sobol_tuples)}")

print("compute optimal pc expansion")
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)

# test PC approximation
N = 1000
print(f"generate test data ({N})")
s_test = pt.sampler.ParameterSampler(params)
x_test = s_test.sample(N)
f_test = target_function(x_test)
f_approx = surrogate.eval(x_test)
f_approx_ref = surrogate_ref.eval(x_test)

error_L2_ref = np.sqrt(np.sum((f_test - f_approx_ref) ** 2) / x_test.shape[0])
error_L2 = np.sqrt(np.sum((f_test - f_approx) ** 2) / x_test.shape[0])
print(f"test error L2 auto: {error_L2:4.2e}")
print(f"test error L2 ref:  {error_L2_ref:4.2e}")

# print Sobol coefficients
print("Comparison of Sobol indices")
print(
    f" {'sobol_tuples':<13} {'exact':<8}  {'pc-auto':<8}  {'(crude)':<8}  {'pc-ref':<8}"
)
print("-" * 54)
for j, sdx in enumerate(index_set.sobol_tuples):
    print(
        f" {str(sdx):<12} ",  # Sobol index subscripts
        f"{sobol_coefficients[j, 0]:<4.2e} ",
        f"{surrogate.sobol[j, 0]:<4.2e} ",
        f"{sC[j][0]:<4.2e} ",
        f"{surrogate_ref.sobol[j, 0]:<4.2e} ",
    )
```

6.2.2 Tutorials - Applications

Tutorial - Shape Reconstruction via Scatterometry

This tutorial shows how PyThia can be used to approximate the scatterometry measurement process and conduct a global sensitivity analysis of the model.

In this experiment a silicon line grating is irradiated by an electromagnetic wave under different angles of incidence θ and azimuthal orientations ϕ. The aim of this experiment is to build a surrogate for the map from the six shape parameters $x = (h, cd, swa, t, r_{\text{top}}, r_{\text{bot}})$ to the detected scattered far field of the electromagnetic wave. The experimental setup is visualized in the image below.



Note: If you want to run the source code for this tutorial yourself, you need to download the . This tutorial assumes that the data are located in the same directory as the source code.

Surrogate

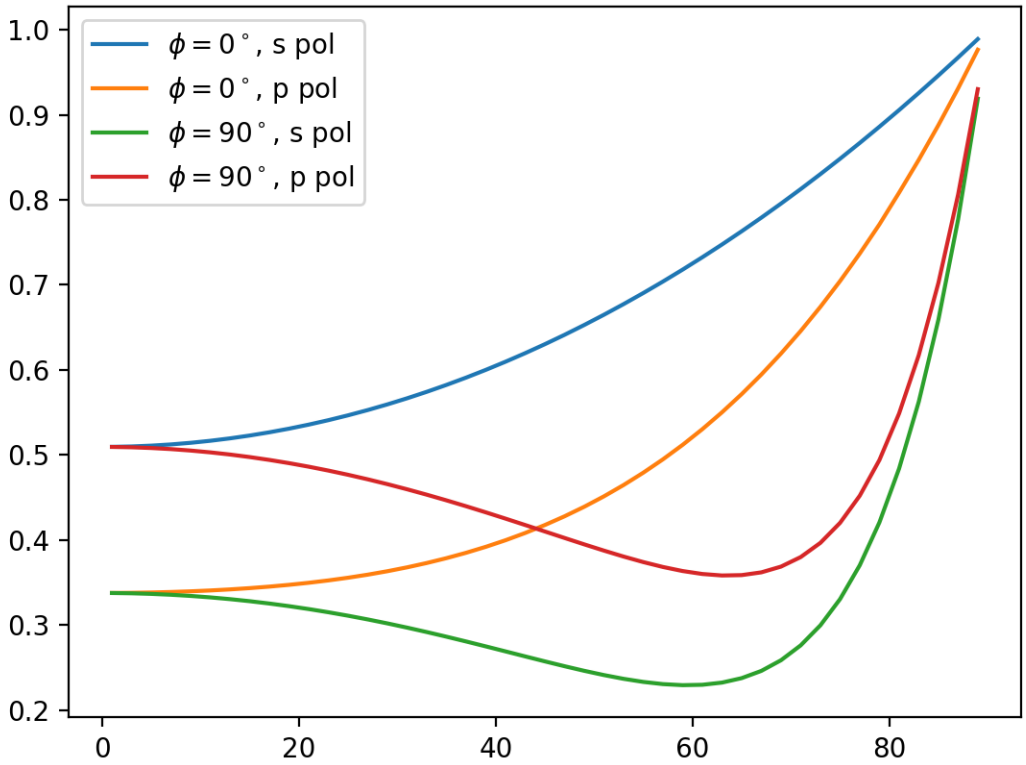
We start by loading the `numpy` and `pythia` packages.

```
import numpy as np
import pythia as pt
```

Generating training data from the forward problem requires solving Maxwell's equations with finite elements, which is very intricate and time consuming. We assume the training data we rely on in this tutorial have been previously generated in an "offline" step, on a different machine using the software. Hence we can simply load the data.

```
x_data = np.load("x_train.npy")  # (10_000, 6)
y_data = np.load("y_train.npy")  # (90, 10_000, 2)
w_data = np.load("w_train.npy")  # (10_000, )
```

For the simulations we assumed Beta distributed parameters in reasonable parameter domains, which is also the density we drew the samples from. In particular, this implies that the weights are simply $w_i = 1/N$, where N is the number of samples (i.e., 10.000 in this case). The scattered intensity signal data consist of 4 curves (2 different azimuthal angles ϕ with s- and p- polarized wave signals each) where each curve is evaluated in 45 points (angles of incidence θ). The continuous curves look similar to the image below.



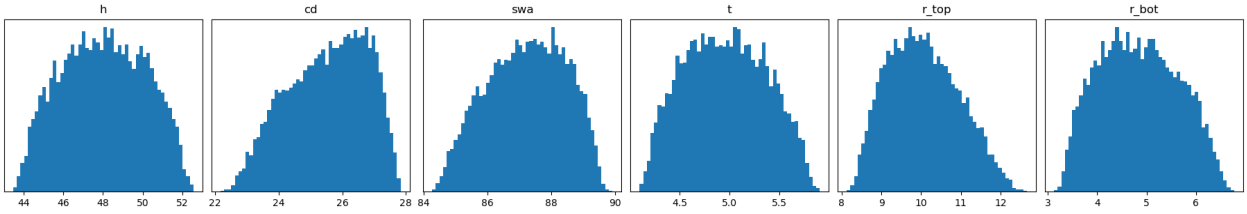
As a first step to compute a polynomial chaos expansion, we need to define the parameters using PyThias *Parameter* class. We can use the `parameter.npy` file to access the parameter information.

```
params = []
for dct in np.load("parameter.npy", allow_pickle=True):
    params.append(
        pt.parameter.Parameter(
            name=dct["_name"],
            domain=dct["_dom"],
            distribution=dct["_dist"],
            alpha=dct["_alpha"],
            beta=dct["_beta"],
        )
    )
```

Below you can find a table and an image listing the relevant parameter information.

Table 1: Parameter Information

#	Name	distribution	α	β
1	h	$\text{Beta}_{\alpha,\beta}([43, 53])$	10.0	9.67
2	cd	$\text{Beta}_{\alpha,\beta}([22, 28])$	10.0	7.76
3	swa	$\text{Beta}_{\alpha,\beta}([84, 90])$	10.0	10.13
4	t	$\text{Beta}_{\alpha,\beta}([4, 6])$	10.0	11.29
5	r_{top}	$\text{Beta}_{\alpha,\beta}([8, 13])$	10.0	11.10
6	r_{bot}	$\text{Beta}_{\alpha,\beta}([3, 7])$	10.0	12.35



Next, we split the available data into a training and a test set, using roughly 8.000 data points for the training.

```
# split training/test dataset
split = int(0.8 * x_data.shape[0])

# training data
x_train = x_data[:split]
y_train = y_data[:split]
w_train = w_data[:split]

# test data
x_test = x_data[split:]
y_test = y_data[split:]
```

Now that we have our parameters defined and the training data in place, we only need to define which expansion terms we want to include into our expansion. In this case, we choose a simplex index set to bound the total polynomial degree of the expansion by four. This leaves us with 210 polynomial chaos terms in our expansion.

```
indices = pt.index.simplex_set(dimension=len(params), maximum=4)
index_set = pt.index.IndexSet(indices)
```

Finally, we have all the ingredients to compute our polynomial chaos surrogate, which requires only one line of code.

```
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)
```

Evaluating the surrogate on our test set with

```
y_approx = surrogate.eval(x_test)
```

allows us compute an empirical approximation of the L^2-error (MSE), which is approximately at $5 * 10^{-3}$. This is ok, since the measurement error of the observation we will investigate for the inverse problem will be at least one order of magnitude larger.

Sensitivity analysis

There are essentially two different types of sensitivity analyses, local and global. Local sensitivity analysis typicalls considers a change in the signal by a small variation in the input around a specific point. Given the polynomial chaos surrogate, computation of the local sensitivities is of course very easy as derivatives of the polynomial are given analytically. To compute the local sensitivities of the second parameter (cd) in all points of our test set, i.e. $\partial / \partial x_{cd}\ f(x_{\text{test}})$, we can use the optional `partial` argument of the `eval()` function

```
local_sensitivity = surrogate.eval(x_test, partial=[0, 1, 0, 0, 0, 0])
```

However, computing local sensitivities yields only a very limited image of the signal dependency on change in the individual parameter or combinations thereof. When it comes to estimating whether the reconstruction of the hidden parameters will be possible it is often more reliable to consider the overall dependence of the signal on the parameters. There are various types of global sensitivity analysis methods, but essentially all of them rely on integrating the target function over the high-dimensional parameter space, which is typically not feasible. The polynomial chaos expansion, however, is closely connected to the so called Sobol indices, which allows us to compute these indices by squaring and summing different subsets of our expansion coefficients. The Sobol coefficients are automatically computed with the polynomial chaos approximation and can be accessed via the `surrogate.sobol` attribute, which is ordered according to the list of Sobol tuples in `index_set.sobol_tuples`. As the signal data consist of multiple points on different curves, we get the Sobol indices for each point of our signal. As this is hard to visualize however, we can have a look on the maximal value each Sobol index takes over the signal points. The 10 most significant Sobol indices are listed in the table below.

Table 2: Most relevant Sobol indices

#	Sobol tuple	Parameter combinations	Sobol index value (max)
1	$(2,)$	$(cd,)$	0.9853
2	$(4,)$	$(t,)$	0.8082
3	$(1,)$	$(h,)$	0.3906
4	$(3,)$	$(swa,)$	0.2462
5	$(2,5)$	(cd, r_{top})	0.0332
6	$(5,)$	$(r_{\text{top}},)$	0.0240
7	$(3,5)$	(swa, r_{top})	0.0137
8	$(1,2)$	(h, cd)	0.0065
9	$(6,)$	$(r_{\text{bot}},)$	0.0059
10	$(2,3,5)$	$(cd, swa, r_{\text{top}})$	0.0030

We see that all single parameter Sobol indices are among the top ten, implying that changes in single parameters are among the most influential. Moreover, looking at the actual (maximal) values, we can assume that the first 4 parameters (h, cd, swa, t) can be reconstructed very well, whereas the sensitivities of the corner roundings r_{top} and r_{bot} imply that we will get large uncertainties in the reconstruction.

This concludes the tutorial.

Complete Script

The complete source code listed below is located in the `tutorials/` subdirectory of the repository root.

```
"""
File: tutorials/tutorial_scat.py
Author: Nando Hegemann
Gitlab: https://gitlab.com/Nando-Hegemann
Description: Tutorial - Shape Reconstruction via Scatterometry.
SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL
"""
import numpy as np
import pythia as pt

angles = np.load("angles.npy")  # (90, 2)

# reshape
angles = np.concatenate([angles, angles], axis=0)  # (180, 2)
angle_idxs = [range(45), range(45, 90), range(90, 135), range(135, 180)]
angle_labels = [
    "$\\phi=0^\\circ$, s pol",
    "$\\phi=0^\\circ$, p pol",
    "$\\phi=90^\\circ$, s pol",
    "$\\phi=90^\\circ$, p pol",
]
print("angle info:")
print(f"    azimuth phi: {np.unique(angles[:,0])}")
print(f"    incidence theta: {np.unique(angles[:,1])}")

# load data
x_data = np.load("x_train.npy")  # (10_000, 6)
y_data = np.load("y_train.npy")  # (90, 10_000, 2)
w_data = np.load("w_train.npy")  # (10_000, )

# reshape y_data
y_data = np.concatenate(
    [y_data[:45, :, 0], y_data[:45, :, 1], y_data[45:, :, 0], y_data[45:, :, 1]], axis=0
).T  # (10_000, 180)
print("data info:")
print(f"    x_data.shape: {x_data.shape}")
print(f"    w_data.shape: {w_data.shape}")
print(f"    y_data.shape: {y_data.shape}")

# load parameters
params = []
for dct in np.load("parameter.npy", allow_pickle=True):
    params.append(
        pt.parameter.Parameter(
            name=dct["_name"],
            domain=dct["_dom"],
            distribution=dct["_dist"],
            alpha=dct["_alpha"],
            beta=dct["_beta"],
```

(continues on next page)

(continued from previous page)

```
        )
    )

print("parameter info:")
print("    parameter  dist    domain    alpha   beta")
print("    " + "-" * 42)
for param in params:
    print(
        f"    {param.name:<8}  ",
        f"{param.distribution:<5}  ",
        f"{str(np.array(param.domain, dtype=int)):<7}  ",
        f"{param.alpha:<4.2f}  ",
        f"{param.beta:<4.2f}",
    )

# split training/test dataset
split = int(0.8 * x_data.shape[0])

# training data
x_train = x_data[:split]
y_train = y_data[:split]
w_train = w_data[:split]

# test data
x_test = x_data[split:]
y_test = y_data[split:]

# define multiindex set
indices = pt.index.simplex_set(dimension=len(params), maximum=4)
index_set = pt.index.IndexSet(indices)

print("Multiindex information:")
print(f"    number of indices: {index_set.shape[0]}")
print(f"    dimension: {index_set.shape[1]}")
print(f"    maximum dimension: {index_set.max}")
print(f"    number of sobol indices: {len(index_set.sobol_tuples)}")

print("compute PC surrogate")
# run pc approximation
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)
# test approximation error
y_approx = surrogate.eval(x_test)

# L2-L2 error
err_abs = np.sum(np.linalg.norm(y_test - y_approx, axis=1)) / y_test.shape[0]
err_rel = np.sum(np.linalg.norm((y_test - y_approx) / y_test, axis=1)) / y_test.shape[0]
print(f"    L2-L2 error: {err_abs:4.2e} (abs), {err_rel:4.2e} (rel)")

# C0-L2 error
err_abs = np.sum(np.max(np.abs(y_test - y_approx), axis=1)) / y_test.shape[0]
err_rel = (
    np.sum(np.max(np.abs(y_test - y_approx) / np.abs(y_test), axis=1)) / y_test.shape[0]
```

(continues on next page)

(continued from previous page)

```
)
print(f"    L2-C0 error: {err_abs:4.2e} (abs), {err_rel:4.2e} (rel)")

# global sensitivity analysis
print("maximal Sobol indices:")
max_vals = np.max(surrogate.sobol, axis=1)
l2_vals = np.linalg.norm(surrogate.sobol, axis=1)
sobol_max = [
    list(reversed([x for _, x in sorted(zip(max_vals, index_set.sobol_tuples))]))
]
sobol_L2 = [
    list(reversed([x for _, x in sorted(zip(l2_vals, index_set.sobol_tuples))]))
]
print(f"    {'#':>2}  {'max':<10}  {'L2':<10}")
print("    " + "-" * 25)
for j in range(10):
    print(f"    {j+1:>2}  {str(sobol_max[0][j]):<10}  {str(sobol_L2[0][j]):<10}")
```

6.3 pythia

6.3.1 pythia package

pythia.basis module

File: pythia/basis.py Author: Nando Hegemann Gitlab: https://gitlab.com/Nando-Hegemann Description: Assemble sparse univariate and multivariate basis polynomials. SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL

pythia.basis.**multivariate_basis**(*univariate_bases*, *indices*, *partial=None*)

Assemble multivariate polynomial basis.

Set the (partial derivative of the) multivariate (product) polynomial basis functions.

Return type
list[Callable]

Parameters

- **univariate_bases** (*list of list of callable*) – Univariate basis functions for parameters. Is called by *univariate_bases[paramIdx][deg]()*.
- **indices** (*array_like*) – Array of multiindices for multivariate basis functions.
- **partial** (*list of int*) – Number of partial derivatives for each dimension. Length is same as *univariate_bases*.

Returns
List of multivariate product polynomials with univariate degrees as specified in *indices*.

pythia.basis.**normalize_polynomial**(*weight*, *basis*, *param*)

Normalize orthogonal polynomials.

Normalize a polynomial of an orthogonal system with respect to the scalar product :rtype: list[Callable]

$$a(u,v)_{\mathrm{pdf}} = \int u(p)v(p)\mathrm{pdf}(p)\mathrm{d}p.$$

The normalized polynomial ϕ_j for any given polynomial P_j is given by $\phi_j = P_j/\sqrt{c_j}$ for the constant $c_j = \int \mathrm{pdf}(p) * P_j(p)^2 \mathrm{d}p$.

Parameters

- **weight** (*callable*) – Probability density function.
- **basis** (list of *numpy.polynomial.Polynomial*) – Polynomials to normalize w.r.t. weight.
- **param** (*pythia.parameter.Parameter*) – Parameter used for distribution and domain information.

Returns
List of normalized univariate polynomials.

pythia.basis.**set_hermite_basis**(*param*, *deg*)

Generate list of probabilists Hermite polynomials.

Generate the Hermite Polynomials up to certain degree according to the mean and variance of the specified parameter.

Return type
list[Callable]

Parameters

- **param** (*pythia.parameters.Parameter*) – Parameter for basis function. Needs to be normal distributed.
- **deg** (*int*) – Maximum degree for polynomials.

Returns
List of probabilists Hermite polynomials up to (including) degree specified in *deg*.

pythia.basis.**set_jacobi_basis**(*param*, *deg*)

Generate list of Jacobi polynomials.

Generate the Jacobi Polynomials up to certain degree on the interval and DoFs specified by the parameter. :rtype: list[Callable]

Note: The Jacobi polynomials have leading coefficient 1.

Parameters

- **param** (*pythia.parameters.Parameter*) – Parameter for basis function. Needs to be Beta-distributed.
- **deg** (*int*) – Maximum degree for polynomials.

Returns
List of Jacobi polynomials up to (including) degree specified in *deg*.

pythia.basis.**set_laguerre_basis**(*param*, *deg*)

Generate list of Leguerre polynomials.

Generate the generalized Laguerre polynomials up to certain degree on the interval and DoFs specified by the parameter.

Return type
list[Callable]

Parameters

- **param** (*pythia.parameters.Parameter*) – Parameter for basis function. Needs to be Gamma-distributed.
- **deg** (*int*) – Maximum degree for polynomials.

Returns
List of Laguerre polynomials up to (including) degree specified in *deg*.

pythia.basis.**set_legendre_basis**(*param*, *deg*)

Generate list of the Legendre Polynomials.

Generate the Legendre Polynomials up to certain degree on the interval specified by the parameter.

Return type
list[Callable]

Parameters

- **param** (*pythia.parameters.Parameter*) – Parameter for basis function. Needs to be uniformly distributed.
- **deg** (*int*) – Maximum degree for polynomials.

Returns
List of Legendre polynomials up to (including) degree specified in *deg*.

pythia.basis.**univariate_basis**(*params*, *degs*)

Assemble a univariate polynomial basis.

Set polynomial basis up to deg for each parameter in *params* according to the parameter distribution and area of definition.

Return type
list[list[Callable]]

Parameters

- **params** (list of *pythia.parameter.Parameter*) – Parameters to compute univariate basis function for.
- **degs** (*array_like*) – Max. degrees of univariate polynomials for each parameter.

Returns
List of normalized univariate polynomials w.r.t. parameter domain and distribution up to specified degree for each parameter in *params*.

pythia.chaos module

File: pythia/chaos.py Author: Nando Hegemann Gitlab: https://gitlab.com/Nando-Hegemann Description: Sample-based computation of polynomial chaos expansion. SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL

class pythia.chaos.**PolynomialChaos**(*params*, *index_set*, *x_train*, *w_train*, *y_train*, *coefficients=None*)

Bases: object

Computation of sparse polynomial chaos expansion.

Parameters

- **params** (list of *pt.parameter.Parameter*) – List of stochastic parameters.
- **index_set** (*pt.index.IndexSet*) – Index set for sparse polynomial chaos expansion.
- **x_train** (*array_like*) – Parameter realizations for training.

- **weights** (*array_like*) – Regression weights for training.
- **fEval** (*array_like*) – Function evaluation for training.
- **coefficients** (*array_like, default=None*) – Polynomial expansion coefficients. If given, the coefficients are not computed during initiation. This can be used to load a chaos expansion.

eval(*x*, *partial=None*)

Evaluate the (partial derivative of the) PC approximation.

Return type
ndarray

Parameters

- **x** (*np.ndarray*) – Parameter realizations in which the approximation is evaluated.
- **partial** (*list[int] | dict | None, optional*) – Number of derivatives for each parameter component. If a list is given, length has to be the number of parameters. Ordering of list is according to `self.parameters`. If a dict is given, keys have to be subset of parameter names.

Returns
Evaluation of polynomial expansion in x values.

Examples

Given two parameters x_1 and x_2

```
>>> param1 = pt.parameter.Parameter("x1", [-1, 1], "uniform")
>>> param2 = pt.parameter.Parameter("x2", [-1, 1], "uniform")
```

a corresponding polynomial chaos approximation for a function $f: (x_1, x_1) \mapsto y$

```
>>> surrogate = pt.chaos.PolynomialChaos([param1, param2], ...)
```

and an array the surrogate should be evaluated in

```
>>> x_test = np.random.uniform(-1, 1, (1000, 2))
```

we can evaluate the surrogate with

```
>>> y_approx = surrogate.eval(x_test)
```

To obtain partial a partial derivative of the approximation, e.g., $\frac{\partial^2 f}{\partial x_2^2}$, specify a list

```
>>> y_approx = surrogate.eval(x_test, partial=[0, 2])
```

or a dictionary with parameter names and number of partial derivates

```
>>> y_approx = surrogate.eval(x_test, partial={'x2':2})
```

property mean: ndarray

Mean of the PC expansion.

property std: ndarray

Standard deviation of the PC expansion.

property var: ndarray

Variance of the PC expansion.

pythia.chaos.**assemble_indices**(*enum_idx*, *sobol_tuples*, *max_terms*)

Compute automatic choice of multiindices.

Return type
ndarray

Parameters

- **enum_idx** (*np.ndarray*) – Sorted enumeration indices according to magnitude of Sobol indices.
- **sobol_tuples** (*list of tuple*) – List of Sobol subscript tuples.
- **max_terms** (*int*) – Maximum number of expansion terms.

Returns
indices – Array of (sparse) optimal multiindices.

Return type
np.ndarray

pythia.chaos.**find_optimal_indices**(*params*, *x_train*, *w_train*, *y_train*, *max_terms=0*, *threshold=0.001*)

Compute optimal multiindices of polynomial chaos expansion.

Heuristical approach to compute almost optimal multiindices for a polynomial chaos expansion based on an estimate of the Sobol index values.

Return type
tuple[ndarray, ndarray]

Parameters

- **params** (*list of pythia.Parameters.Parameter*) – Random parameters of the problem.
- **x_train** (*array_like*) – Sample points for training
- **w_train** (*array_like*) – Weights for training.
- **y_train** (*array_like*) – Function evaluations for training.
- **max_terms** (*int, default=0*) – Maximum number of expansion terms. Number of expansion terms is chosen automatically for *max_terms=0*.
- **threshold** (*float, default=1e-03*) – Truncation threshold for Sobol indices. Smallest Sobol values with sum less then `threshold` are ignored.

Returns

- *indices* – Array with multiindices.
- *sobol* – Crude intermediate approximation of Sobol indices.

Notes

To find reasonable candidates for the sparse polynomial chaos expansion, first an expansion with a large simplex index set is computed. The simplex index set uses the same maximum dimension in each component and is designed to have at least `max_terms` many elements. With this index set a polynomial chaos expansion is computed. The computed Sobol indices are then ordered and the largest contributions are collected by a Dörfler marking strategy. Then a new index set is assembled by including a downward closed subset of polynomial chaos coefficient indices for each selected Sobol index tuple. The number of chosen indices for each selected Sobol index tuple is weighted by the respective Sobol index value.

pythia.chaos.**get_gram_batchsize**(*dim*, *save_memory=538445312.5*)

Compute memory allocation batch sizes for information matrix.

Compute the maximal number of samples in each batch when assembling the information matrix to be maximally memory efficient and avoid OutOfMemory errors.

Return type
`int`

Parameters

- **dim** (*int*) – Number of rows/columns of information matrix.
- **save_memory** (*int, default=3*1025/2*) – Memory (in bytes), that should be kept free. The default is equivalent to 512 MB.

Returns
Batchsize for assembling of information matrix.

pythia.index module

File: pythia/index.py Author: Nando Hegemann Gitlab: https://gitlab.com/Nando-Hegemann Description: Create, manipulate and store information about multiindices. SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL

class pythia.index.**IndexSet**(*indices*)

Bases: `object`

Generate index set object for sparse PC expansion.

A general polynomial chaos expansion of a function $f\colon \Gamma \subset \mathbb{R}^M \to \mathbb{R}^J$ with $y \sim \pi$ is given by

$$f(y) = \sum_{\mu \in \mathbb{N}_0^M} \mathbf{f}[\mu] P_\mu(y) \quad \text{for} \quad \mathbf{f}[\mu] = \int_\Gamma f(y) P_\mu(y) \, \mathrm{d}y,$$

where μ is a multiindex, $\mathbf{f}[\mu] \in \mathbb{R}^J$ is a coefficient vector and $\{P_\mu\}_{\mu \in \mathbb{N}_0^M}$ is an orthonormal basis in $L^2(\Gamma, \pi)$. To approximate the infinite expansion choose an index set $\Lambda \subset \mathbb{N}_0^M$ of multiindices and consider

$$f(y) \approx \sum_{\mu \in \Lambda} \mathbf{f}[\mu] P_\mu(y),$$

Parameters
indices (*np.ndarray*) – Array of multiindices with shape (#indices, param dim).

Examples

Create the sparse index set

$$\Lambda = \{(0,0),(1,0),(2,0),(0,1)\} \subset \mathbb{N}_0^2$$

```
>>> import pythia as pt
>>> indices = np.array([[0, 0], [1, 0], [2, 0], [0, 1]], dtype=int)
>>> index_set = pt.index.IndexSet(indices)
```

get_index_number(*indices*)

Get enumeration number of indices.

Get the row indices of the given multiindices such that *self.indices[rows] = indices*.

Return type
ndarray

Parameters
indices (*np.ndarray*) – Indices to get the number of.

Returns
Array containing the enumeration numbers of the indices.

get_sobol_tuple_number(*sobol_tuples*)

Get enumeration indices of Sobol tuples.

Return type
ndarray

Parameters
sobol_tuples (*list of tuple*) – List of Sobol tuples.

Returns
Array containing the enumeration number of the Sobol tuples.

index_to_sobol_tuple(*indices*)

Map array of indices to their respective Sobol tuples.

Return type
list[tuple]

Parameters
indices (*np.ndarray*) – Array of multiindices.

Returns
List of Sobol tuples.

sobol_tuple_to_indices(*sobol_tuples*)

Map Sobol tuples to their respective indices.

Return type
list[ndarray]

Parameters
sobol_tuples (*tuple or list of tuple*) – List of Sobol tuples.

Returns
List of index arrays for each given Sobol tuple.

pythia.index.**intersection**(*index_list*)

Intersect list of multiindex sets.

Given sparse index sets $\Lambda_1, \ldots, \Lambda_N$, compute $\Lambda = \Lambda_1 \cap \cdots \cap \Lambda_N$.

Return type
ndarray

Parameters
index_list (*list[np.ndarray]*) – List of index sets.

Returns
Intersection of index sets.

pythia.index.**lq_bound_set**(*dimensions*, *bound*, *q=1*)

Create set of multiindices with bounded ℓ^q-norm.

For given dimensions $d \in \mathbb{N}^M$, bound $b \in \mathbb{R}_{>0}$ and norm factor $q \in \mathbb{R}_{>0}$, the ℓ^q-norm index set is given by
:rtype: ndarray

$$\Lambda = \{\mu \in [d_1] \times \cdots \times [d_M] \mid \|\mu\|_{\ell^q} \leq b\},$$

where $[d_m] = \{0, \ldots, d_m - 1\}$ and

$$\|\mu\|_{\ell^q} = \Big(\sum_{m=1}^{M} \mu_m^q\Big)^{\frac{1}{q}}.$$

Parameters

- **dimensions** (*list[int] | tuple[int] | np.ndarray*) – Dimensions for each component, i.e., indices from `0` to `dimension-1`.
- **bound** (*float*) – Bound for the ℓ^q-norm.
- **q** (*float, optional*) – Norm factor.

Returns
Array with all possible multiindices with bounded ℓ^q-norm.

See also:

pythia.index.tensor_set, *pythia.index.simplex_set*

Examples

```
>>> pt.index.lq_bound_set([5, 5], 4, 0.5)
array([[0, 0],
       [0, 1],
       [1, 0],
       [0, 2],
       [1, 1],
       [2, 0],
       [0, 3],
       [3, 0],
       [0, 4],
       [4, 0]])
```

pythia.index.**set_difference**(*indices*, *subtract*)

Set difference of two index arrays.

Given two sparse index sets Λ_1 and Λ_2, compute $\Lambda = \Lambda_1 \setminus \Lambda_2$.

Return type
ndarray

Parameters

- **indices** (*np.ndarray*) – Index array multiindices are taken out of.
- **subtract** (*np.ndarray*) – Indices that are taken out of the original set.

Returns
Set difference of both index arrays.

pythia.index.**simplex_set**(*dimension*, *maximum*)

Create a simplex index set.

For given dimension $M \in \mathbb{N}$ and maximum $d \in \mathbb{N}$ the simplex index set is given by :rtype: ndarray

$$\Lambda = \{\mu \in \mathbb{N}_0^M \mid \sum_{m=1}^{M} \mu_m \leq d\}.$$

Notes

Limiting the absolute value of the multiindices creates a simplex in $\mathbb{N}_0^M$, which motivates the name of the function. As an example, in two dimensions this gives us points inside a triangle limited by the axes and the line $x_1 + x_2 = d$.

Parameters

- **dimension** (*int*) – Dimension of the multiindices.
- **maximum** (*int*) – Maximal sum value for the multiindices.

Returns
Array with all possible multiindices in simplex set.

See also:

pythia.index.lq_bound_set, *pythia.index.tensor_set*

Examples

```
>>> pt.index.simplex(2, 2)
array([[0, 0],
       [0, 1],
       [1, 0],
       [0, 2],
       [1, 1],
       [2, 0]])
```

pythia.index.**sort_index_array**(*indices*)

Sort multiindices and remove duplicates.

Sort rows of *indices* by sum of multiindex and remove duplicate multiindices.

Return type
ndarray

Parameters
indices (*np.ndarray*) – Index list before sorting.

Returns
Sorted index array.

pythia.index.**tensor_set**(*shape*, *lower=None*)

Create a tensor index set.

For given upper and lower bounds $0 \leq \ell_m < u_m \in \mathbb{N}_0$ with $m = 1, \ldots, M \in \mathbb{N}$, the tensor index set (n-D cube) is given by :rtype: ndarray

$$\Lambda = \{\mu \in \mathbb{N}_0^M \mid \ell_m \leq \mu_m \leq u_m \text{ for } m = 1, \ldots, M\}.$$

Parameters

- **shape** (*array_like*) – Shape of the tensor, enumeration starting from 0.
- **lower** (*array_like, default = None*) – Starting values for each dimension of the tensor set. If None, all dimensions start with 0.

Returns
Array with all possible multiindices in tensor set.

See also:

pythia.misc.cart_prod, *pythia.index.lq_bound_set*, *pythia.index.simplex_set*

Examples

Create the tensor product multiindices $\{0, 1\} \times \{0, 1\}$

```
>>> pt.index.tensor_set([2, 2])
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

Create 3D univariate multiindices $\{0\} \times \{1, \ldots, 4\} \times \{0\}$

```
>>> pt.index.tensor_set([1, 5, 1], [0, 1, 0])
array([[0, 1, 0],
       [0, 2, 0],
       [0, 3, 0],
       [0, 4, 0]])
```

Create 1D indices similar to `np.arange(1, 5, dtype=int).reshape(-1, 1)`

```
>>> pt.index.tensor_set([5], [1])
array([[1],
       [2],
       [3],
       [4]])
```

pythia.index.**union**(*index_list*)

Build union of multiindex sets.

Given sparse index sets $\Lambda_1, \ldots, \Lambda_N$, compute $\Lambda = \Lambda_1 \cup \cdots \cup \Lambda_N$.

Return type
ndarray

Parameters
index_list (*list of np.ndarray*) – List of multiindex arrays.

Returns
Array with all multiindices.

pythia.misc module

File: pythia/misc.py Author: Nando Hegemann Gitlab: https://gitlab.com/Nando-Hegemann Description: Miscellaneous functions to support PyThia core functionality. SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL

pythia.misc.**batch**(*iterable*, *n=1*)

Split iterable into different batches of batchsize n.

Return type
Iterator

Parameters

- **iterable** (*array_like*) – Iterable to split.
- **n** (*int, default=1*) – Batch size.

Yields
Iterable for different batches.

pythia.misc.**cart_prod**(*array_list*)

Compute the outer product of two or more arrays.

Assemble an array containing all possible combinations of the elements of the input vectors $v_1, \ldots, v_n$.

Return type
ndarray

Parameters
array_list (*list of array_like*) – List of vectors $v_1, \ldots, v_n$.

Returns
Cartesian product array.

pythia.misc.**confidence_interval**(*samples*, *rate=0.95*, *resolution=500*)

Compute confidence intervals of samples.

Compute the confidence intervals of the 1D marginals of the samples (slices). The confidence interval of a given rate is the interval around the median (not mean) of the samples containing roughly *rate* percent of the total mass. This is computed for the left and right side of the median independently.

Return type
ndarray

Parameters

- **samples** (*array_like, ndim < 3*) – Array containing the (multidimensional) samples.

- **rate** (*float, default=0.95*) – Fraction of the total mass the interval should contain.
- **resolution** (*int, default=500*) – Number of bins used in histogramming the samples.

Returns
Confidence intervals for each component.

pythia.misc.**doerfler_marking**(*values*, *idx=None*, *threshold=0.9*)

Dörfler marking for arbitrary values.

Return type
tuple[ndarray, ndarray, int]

Parameters

- **values** (*array_like*) – Values for the Dörfler marking.
- **idx** (*list of int, optional*) – List of indices associated with the entries of *values*. If *None*, this is set to `range(len(values))`.
- **threshold** (*float, default=0.9*) – Threshold paramter for Dörfler marking.

Returns

- *idx_reordered* – Reordered indices given by *idx*. Ordered from largest to smallest value.
- *ordered_values* – Reordered values. Ordered from largest to smallest.
- *marker* – Threshold marker such that `sum(values[:marker]) > threshold * sum(values)`.

pythia.misc.**format_time**(*dt*)

Converts time (seconds) to time format string.

Return type
str

Parameters
dt (*float*) – Time in seconds.

Returns
Formatted time string.

pythia.misc.**gelman_rubin_condition**(*chains*)

Compute Gelman-Rubin criterion.

Implementation of the Gelman-Rubin convergence criterion for multiple parameters. A Markov chain is said to be in its convergence, if the final ration is close to one.

Return type
ndarray

Parameters
chains (*array_like, ndim=3*) – Array containing the Markov chains of each parameter. All chains are equal in length, the assumed shape is `(#chains, chain length, #params)`.

Returns
Values computed by Gelman-Rubin criterion for each parameter.

pythia.misc.**is_contained**(*val*, *domain*)

Check if a given value (vector) is contained in a domain.

Checks if each component of the vector lies in the one dimensional interval of the corresponding component of the domain.

Return type
bool

Parameters

- **val** (*array_like*) – Vector to check containment in domain
- **domain** (*array_like*) – Product domain of one dimensional intervals.

Returns
Bool stating if value is contained in domain.

pythia.misc.**line**(*indicator*, *message=None*)

Print a line of 80 characters by repeating indicator.

An additional message can be given.

Return type
str

Parameters

- **indicator** (*string*) – Indicator the line consists of, e.g. '-', '+' or '+-'.
- **message** (*string, optional*) – Message integrated in the line.

Returns
String of 80 characters length.

pythia.misc.**load**(*filename*)

Alias for numpy.load().

Return type
ndarray

pythia.misc.**now**()

Get string of current machine date and time.

Return type
str

Returns
Formatted date and time string.

pythia.misc.**save**(*filename*, *data*, *path='./'*)

Wrapper for numpy save.

Assures path directory is created if necessary and backup old data if existent.

Return type
None

Parameters

- **name** (*str*) – Filename to save data to.
- **data** (*array_like*) – Data to save as .npy file.
- **path** (*str, default='./'*) – Path under which the file should be created.

pythia.misc.**shift_coord**(*x*, *S*, *T*)

Shift *x* in interval *S* to interval *T*.

Use an affine transformation to shift points x from the source interval $S = [t_0, t_1]$ to the target interval $T = [a, b]$.

Return type
ndarray

Parameters

- **x** (*array_like*) – Points in interval S.
- **S** (*array_like*) – Source interval.
- **T** (*array_like*) – Target interval.

Returns
Shifted values for *x*.

pythia.misc.**str2iter**(*string*, *iterType=<class 'list'>*, *dataType=<class 'int'>*)

Cast *str(iterable)* to *iterType* of *dataType*.

Cast a string of lists, tuples, etc to the specified iterable and data type, i.e., for *iterType=tuple* and *dataType=float* cast `str([1,2,3]) -> (1.0, 2.0, 3.0)`.

Return type
Sequence

Parameters

- **string** (*str*) – String representation of iterable.
- **iterType** (*iterable, default=list*) – Iterable type the string is converted to.
- **dataType** (*type, default=int*) – Data type of entries of iterable, e.g. *int* or *float*.

pythia.misc.**wls_sampling_bound**(*m*, *c=4*)

Compute the weighted Least-Squares sampling bound.

The number of samples n is chosen such that :rtype: `int`

$$\frac{n}{\log(n)} \geq cm,$$

where m is the dimension of the Gramian matrix (number of PC expansion terms) and c is an arbitrary constant. In Cohen & Migliorati 2017 the authors observed that the coice $c = 4$ yields a well conditioned Gramian with high probability.

Parameters

- **m** (*int*) – Dimension of Gramian matrix.
- **c** (*float, default=4*) – Scaling constant.

Returns
Number of required wLS samples.

pythia.parameter module

File: pythia/parameter.py Author: Nando Hegemann Gitlab: https://gitlab.com/Nando-Hegemann Description: PyThia classes containing Parameter information. SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL

class pythia.parameter.**Parameter**(*name*, *domain*, *distribution*, *mean=None*, *var=None*, *alpha=None*, *beta=None*)

Bases: `object`

Class used for stochasic parameters.

Parameters

- **name** (*str*) – Parameter name.
- **domain** (*array_like*) – Supported domain of the parameter distribution.
- **distribution** (*str*) – Distribution identifier of the parameter.
- **mean** (*float, default=None*) – Mean of parameter probability.
- **var** (*float, default=None*) – Variance of parameter probability.
- **alpha** (*float, default=None*) – Alpha value of Beta and Gamma distribution.
- **beta** (*float, default=None*) – Beta value of Beta and Gamma distribution.

alpha: Optional[float] = None

beta: Optional[float] = None

distribution: str

domain: list | tuple | ndarray

mean: Optional[float] = None

name: str

var: Optional[float] = None

pythia.sampler module

File: pythia/sampler.py Author: Nando Hegemann Gitlab: https://gitlab.com/Nando-Hegemann Description: Sampler classes for generating in random samples and PDF evaluations. SPDX-License-Identifier: LGPL-3.0-or-later OR Hippocratic-3.0-ECO-MEDIA-MIL

class pythia.sampler.**BetaSampler**(*domain*, *alpha*, *beta*)

Bases: *Sampler*

Sampler for univariate Beta distributed samples on given domain.

Parameters

- **domain** (*array_like*) – Supported domain of distribution.
- **alpha** (*float*) – Parameter for Beta distribution.
- **beta** (*float*) – Parameter for Beta distribution.

property cov: float

(Co)Variance of the distribution.

property dimension

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF. :rtype: ndarray

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

`hess_x_log_pdf(x)`

Evaluate Hessian of log-PDF. :rtype: `ndarray`

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

`log_pdf(x)`

Evaluate log-PDF.

Return type
`ndarray`

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

`property mass`

Mass of the PDF.

`property maximum`

Maximum value of the PDF.

The maximum of the Beta distribution is given by

$$\max_{x\in[a,b]} f(x) = \begin{cases} \infty & \text{if } 0<\alpha<1 \text{ or } 0<\beta<1, \\ \frac{1}{(b-a)B(\alpha,\beta)} & \text{if } \alpha=1 \text{ or } \beta=1, \\ \frac{(\alpha-1)^{\alpha-1}(\beta-1)^{\beta-1}}{(\alpha+\beta-2)^{\alpha+\beta-2}(b-a)B(\alpha,\beta)} & \text{if } \alpha>1 \text{ and } \beta>1, \end{cases}$$

where $B(\alpha, \beta)$ denotes the Beta-function.

`property mean: float`

Mean value of the distribution.

`pdf(x)`

Evaluate PDF.

Return type
`ndarray`

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

sample(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

property std: float

Standard deviation of the distribution.

property var: float

Variance of the distribution.

class pythia.sampler.**GammaSampler**(*domain*, *alpha*, *beta*)

Bases: *Sampler*

Sampler for univariate Gamma distributed samples on given domain.

Parameters

- **domain** (*array_like*) – Supported domain of distribution.
- **alpha** (*float*) – Parameter for Gamma distribution.
- **beta** (*float*) – Parameter for Gamma distribution.

property cov: float

(Co)Variance of the distribution.

property dimension: float

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF. :rtype: ndarray

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF. :rtype: ndarray

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

`log_pdf`(*x*)

Evaluate log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

`property mass: float`

Mass of the PDF.

`property maximum: float`

Maximum value of the PDF.

The maximum of the Gamma distribution is given by

$$\max_{x\in[a,\infty)} f(x) = \begin{cases} \infty & \text{if } 0 < \alpha < 1 \\ \frac{\beta^\alpha}{\Gamma(\alpha)} & \text{if } \alpha = 1 \\ \frac{\beta^\alpha}{\Gamma(\alpha)} \left(\frac{\alpha-1}{\beta}\right)^{\alpha-1} e^{1-\alpha} & \text{if } \alpha > 1 \end{cases}$$

`property mean: float`

Mean value of the distribution.

`pdf`(*x*)

Evaluate PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

`sample`(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

`property std: float`

Standard deviation of the distribution.

`property var: float`

Variance of the distribution.

class pythia.sampler.**NormalSampler**(*mean*, *var*)

Bases: *Sampler*

Sampler for univariate normally distributed samples.

Parameters

- **mean** (*float*) – Mean of the Gaussian distribution.
- **var** (*float*) – Variance of the Gaussian distribution.

property cov: float

(Co)Variance of the distribution.

property dimension: float

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

log_pdf(*x*)

Evaluate log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

property mass: float

Mass of the PDF.

property maximum: float

Maximum value of the PDF.

property mean: float

Mean value of the distribution.

pdf(*x*)

Evaluate PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

sample(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

property std: float

Standard deviation.

class pythia.sampler.**ParameterSampler**(*params*)

Bases: *Sampler*

Product sampler of given parameters.

Parameters
params (list of *pythia.parameter.Parameter*) – list containing information of parameters.

property cov: ndarray

Covariance of the PDF.

property dimension: int

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

`log_pdf`(*x*)

Evaluate log-PDF.

The log-PDF is given by the sum of the univariate log-PDFs.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

`property mass: float`

Mass of the PDF.

`property maximum: float`

Maximum value of the PDF.

`property mean: ndarray`

Mean of the PDF.

`pdf`(*x*)

Evaluate PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

`sample`(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

`weight`(*x*)

Weights of the parameter product PDF.

Return type
ndarray

Parameters
x (*np.ndarray*) – Evaluation points.

Returns
Array of uniform weights for samples.

class pythia.sampler.**ProductSampler**(*sampler_list*)

Bases: *Sampler*

Tensor sampler for independent parameters.

Sampler for cartesian product samples of a list of (independent) univariate samplers.

Parameters
sampler_list (list of *pythia.sampler.Sampler*) – list of (univariate) Sampler objects.

property cov: ndarray

Covariance of the PDF.

property dimension: int

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

log_pdf(*x*)

Evaluate log-PDF.

The log-PDF is given by the sum of the univariate log-PDFs.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

property mass: float

Mass of the PDF.

property maximum: float

Maximum value of the PDF.

property mean: ndarray

Mean of the PDF.

pdf(*x*)

Evaluate PDF.

The PDF is given by the product of the univariate PDFs.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

sample(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

weight(*x*)

Weights of the product PDF.

Return type
ndarray

Parameters
x (*np.ndarray*) – Evaluation points.

Returns
Array of uniform weights for samples.

class pythia.sampler.**Sampler**

Bases: ABC

Base class for all continuous samplers.

abstract property cov: float | numpy.ndarray

(Co)Variance of the pdf.

abstract property dimension: int

Dimension of the ambient space.

domain: ndarray

abstract grad_x_log_pdf(*x*)

Gradient of log-density of the samplers distribution.

Computes the gradient of the log-density of the samplers underlying distribution at the given points *x*.

Return type
ndarray

Parameters
x (*array_like of shape (..., D)*) – list of points or single point. *D* is the objects dimension.

Returns
Gradient values of the log-density at the points with shape (..., D).

abstract hess_x_log_pdf(*x*)

Hessian of log-density of the samplers distribution.

Computes the Hessian of the log-density of the samplers underlying distribution at the given points *x*.

Return type
ndarray

Parameters
x (*array_like of shape (..., D)*) – list of points or single point. *D* is the objects dimension.

Returns
Hessian values of the log-density at the points with shape (..., D, D).

abstract log_pdf(*x*)

Log-density of the samplers distribution.

Computes the log-density of the samplers underlying distribution at the given points *x*.

Return type
ndarray

Parameters
x (*array_like of shape (..., D)*) – list of points or single point. *D* is the objects dimension.

Returns
Log-density values at the points.

abstract property mass

Mass of the sampler distribution.

The integral of the sampler distribution over the domain of definition. If the density is normalised this value should be one.

abstract property maximum: float

Maximum of the pdf.

abstract property mean: float | numpy.ndarray

Mean value of the pdf.

abstract pdf(*x*)

Density of the samplers distribution.

Computes the density of the samplers underlying distribution at the given points *x*.

Return type
ndarray

Parameters
x (*array_like of shape (..., D)*) – list of points or single point. *D* is the objects dimension.

Returns
Density values at the points.

abstract sample(*shape*)

Random values in a given shape.

Create an array of the given shape and populate it with random samples from the samplers distribution.

Return type
ndarray

Parameters
shape (*array_like, optional*) – The dimensions of the returned array, should all be positive. If no argument is given a single Python float is returned.

Returns
Random values of specified shape.

class pythia.sampler.**UniformSampler**(*domain*)

Bases: *Sampler*

Sampler for univariate uniformly distributed samples on given domain.

Parameters
domain (*array_like*) – Interval of support of distribution.

property cov: float

(Co)Variance of the distribution.

property dimension: int

Parameter dimension.

grad_x_log_pdf(*x*)

Evaluate gradient of uniform log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of uniform log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

log_pdf(*x*)

Evaluate uniform log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

property mass: float

Mass of the PDF.

property maximum: float

Maximum value of the PDF.

property mean: float

Mean value of the distribution.

pdf(*x*)

Evaluate uniform PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

sample(*shape*)

Draw samples from uniform distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

property std: float

Standard deviation of the distribution.

property var: float

Variance of the distribution.

class pythia.sampler.**WLSSampler**(*params*, *basis*, *tsa=True*, *trial_sampler=None*, *bulk=None*)

Bases: *Sampler*

Weighted Least-Squares sampler.

Given a stochastic variable $y \in \Gamma \subset \mathbb{R}^M$ with $y \sim \pi$, a set of multiindices $\Lambda \subset \mathbb{N}_0^M$ and a finite subset $\{P_\alpha\}_{\alpha\in\Lambda}$ of an orthonormal polynomial basis of $L^2(\Gamma, \pi)$, the optimal weighted least-squares sampling distribution for a function $u \in \mathrm{span}\{P_\alpha \mid \alpha \in \Lambda\}$ reads

$$\mathrm{d}\mu = w^{-1}\mathrm{d}\pi \qquad \text{with weight} \qquad w^{-1}(y) = \frac{1}{|\Lambda|}\sum_{\alpha\in\Lambda}|P_\alpha(y)|^2,$$

where $|\Lambda|$ denotes the number of elements in Λ.

Parameters

- **params** (list of *pythia.parameter.Parameter*) – list of parameters.
- **basis** (*list*) – list of basis functions.

- **tsa** (*bool, default=False*) – Trial sampler adaptation. If True, a trial sampler is chosen on the distributions of parameters, if false a uniform trial sampler is used.
- **trial_sampler** (*pythia.sampler.Sampler, default=None*) – Trial sampler for rejection sampling. If *tsa* is true and either *trial_sampler* or *bulk* are *None*, the trial sampler is chosen automatically.
- **bulk** (*float, defaul=None*) – Scaling for trial sampler. If *tsa* is true and either *trial_sampler* or *bulk* are *None*, the trial sampler is chosen automatically.

Notes

To generate samples from the weighted least-squares distribution rejection sampling is used. For certain basis functions it is possible to choose a well-suited trial sampler for the rejection sampling, which can be enabled via setting `tsa=True`.

See also:

pythia.sampler.WLSUnivariateSampler, *pythia.sampler.WLSTensorSampler*

References

The optimal weighted least-squares sampling is based on the results of Cohen & Migliorati[1].

property cov: ndarray

Covariance of the PDF.

property dimension: int

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

log_pdf(*x*)

Evaluate log-PDF.

The log-PDF is given by the sum of the univariate log-PDFs.

[1] Cohen, A. and Migliorati, G., "Optimal weighted least-squares methods", SMAI Journal of Computational Mathematics 3, 181-203 (2017).

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

property mass

Mass of the PDF.

property maximum: float

Maximum value of the PDF.

property mean: ndarray

Mean of the PDF.

pdf(*x*)

Evaluate PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

sample(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

weight(*x*)

Weights for the PDF.

Return type
ndarray

Parameters
x (*array_like*) – Points the weight function is evaluated in.

Returns
weights of evaluation points *x*.

class pythia.sampler.**WLSTensorSampler**(*params*, *deg*, *tsa=True*)

Bases: *Sampler*

Weighted least-squares sampler for tensor multivariate basis.

Given a stochastic variable $y \in \Gamma \subset \mathbb{R}^M$ with $y \sim \pi = \prod_{m=1}^{M} \pi_m$ for one dimensional densities π_m, a tensor set of multiindices $\Lambda = [d_1] \times \cdots \times [d_M] \subset \mathbb{N}_0^M$, where $[d_m] = \{0, \ldots, d_m - 1\}$, and a finite subset $\{P_\alpha\}_{\alpha \in \Lambda}$

of an orthonormal product polynomial basis of $L^2(\Gamma, \pi)$, i.e., $P_\alpha(y) = \prod_{m=1}^{M} P_{\alpha_m}(y_m)$, the optimal weighted least-squares sampling distribution for a function $u \in \mathrm{span}\{P_\alpha \mid \alpha \in \Lambda\}$ reads

$$\mathrm{d}\mu = w^{-1}\mathrm{d}\pi \qquad \text{with weight} \qquad w^{-1}(y) = \prod_{m=1}^{M} \frac{1}{d_m} \sum_{\alpha_m \in [d_m]} |P_{\alpha_m}(y_m)|^2.$$

Parameters

- **params** (list of *pythia.parameter.Parameter*) – Parameter list.
- **deg** (*list of int*) – Polynomial degree of each component (same for all).
- **tsa** (*bool, default=True*) – Trial sampler adaptation. If True, a trial sampler is chosen on the distributions of parameters, if false a uniform trial sampler is used.

Notes

To generate samples from the weighted least-squares distribution rejection sampling is used. For certain basis functions it is possible to choose a well-suited trial sampler for the rejection sampling, which can be enabled via setting `tsa=True`.

See also:

pythia.sampler.WLSUnivariateSampler

References

The optimal weighted least-squares sampling is based on the results in Cohen & Migliorati[Page 61, 1].

property cov: ndarray

Covariance of the PDF.

property dimension: int

Dimension of the parameters.

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

`log_pdf`(*x*)

Evaluate log-PDF.

The log-PDF is given by the sum of the univariate log-PDFs.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

`property mass: float`

Mass of the PDF.

`property maximum: float`

Maximum value of the PDF.

`property mean: ndarray`

Mean of the PDF.

`pdf`(*x*)

Evaluate PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

`sample`(*shape*)

Draw samples from distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

`weight`(*x*)

Weights for the PDF.

Return type
ndarray

Parameters
x (*array_like*) – Points the weight function is evaluated in.

Returns
Weights of evaluation points *x*.

class pythia.sampler.**WLSUnivariateSampler**(*param*, *deg*, *tsa=True*)

Bases: *Sampler*

Sampler for univariate optimally distributed samples on given domain.

Given a stochastic variable $y \in \Gamma \subset \mathbb{R}$ with $y \sim \pi$ and a finite subset $\{P_j\}_{j=0}^{d-1}$ of an orthonormal polynomial basis of $L^2(\Gamma, \pi)$, the optimal weighted least-squares sampling distribution for a function $u \in \operatorname{span}\{P_j \mid j = 0, \ldots, d-1\}$ reads

$$\mathrm{d}\mu = w^{-1}\mathrm{d}\pi \qquad \text{with weight} \qquad w^{-1}(y) = \frac{1}{d}\sum_{j=0}^{d-1} |P_j(y)|^2.$$

Parameters
domain (*array_like*) – Interval of support of distribution.

Notes

To generate samples from the weighted least-squares distribution rejection sampling is used. For certain basis functions it is possible to choose a well-suited trial sampler for the rejection sampling, which can be enabled via setting `tsa=True`.

See also:

pythia.sampler.WLSTensorSampler

References

The optimal weighted least-squares sampling is based on the results in Cohen & Migliorati[Page 61, 1].

property cov: float
(Co)Variance of the distribution.

property dimension: int
Parameter dimension.

grad_x_log_pdf(*x*)
Evaluate gradient of uniform log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of gradient (vector valued) of log-PDF evaluated in *x*.

hess_x_log_pdf(*x*)
Evaluate Hessian of uniform log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of Hessian (matrix valued) of log-PDF evaluated in *x*.

`log_pdf`(*x*)

Evaluate uniform log-PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of log-PDF evaluated in *x*.

`property mass: float`

Mass of the PDF.

`property maximum: float`

Maximum value of the PDF.

`property mean: float`

Mean value of the distribution.

`pdf`(*x*)

Evaluate uniform PDF.

Return type
ndarray

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

`sample`(*shape*)

Draw samples from weighted least-squares parameter distribution.

Return type
ndarray

Parameters
shape (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

`property std: float`

Standard deviation of the distribution.

`property var: float`

Variance of the distribution.

`weight`(*x*)

Weights for the pdf.

Return type
ndarray

Parameters
x (*np.ndarray*) – Points the weight function is evaluated in.

Returns
w – Weights of evaluation points *x*.

Return type
array_like

pythia.sampler.**assign_sampler**(*param*)

Assign a univariate sampler to the given parameter.

Return type
Sampler

Parameters
param (pythia.parameter.Parameter) –

Returns
Univariate sampler.

pythia.sampler.**constraint_sampling**(*sampler*, *constraints*, *shape*)

Draw samples according to algebraic constraints.

Draw samples from target distribution and discard samples that do not satisfy the constraints.

Return type
ndarray

Parameters

- **sampler** (Sampler) – Sampler to sample from.
- **constraints** (*list of callable*) – list of functions that return True if sample point satisfies the constraint.

Returns
Samples drawn from sampler satisfying the constraints.

Notes

The constaints may lead to a non-normalized density function.

pythia.sampler.**get_maximum**(*f*, *domain*, *n=1000*)

Compute essential maximum of function by point evaluations.

Return type
float

Parameters

- **f** (*callable*) – Function to evaluate. Needs to map from n-dim space to 1-dim space.
- **domain** (*array_like*) – Domain to evaluate function on.
- **n** (*int, default=1000*) – Number of function evaluations. Evaluations are done on a uniform grid in domain. Actual number of points may thus be a little greater.

Returns
Approximation of maximum of function *f*.

pythia.sampler.**rejection_sampling**(*pdf*, *trial_sampler*, *scale*, *dimension*, *shape*)

Draw samples from pdf by rejection sampling.

Return type
ndarray

Parameters

- **pdf** (*Callable*) – Probability density the samples are generated from.
- **trial_sampler** (Sampler) – Trial sampler proposal samples are drawn from.
- **scale** (*float*) – Threshold parameter with `pdf <= scale * trialSampler.pdf`
- **dimension** (*int*) – Dimension of the (input of the) pdf.
- **shape** (*array_like*) – Shape of the samples.

Returns
Random samples of specified shape.

Module contents

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

H

I

L

M

N

P

R

S

T

U

V

W