PyThia

Release 01.02.2021

Nando Farchmin

Dec 10, 2022

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 PyThia homepage.

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

CONTENTS

2.1 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://gitlab1.ptb.de/pythia/pythia.git
cd pythia
pip install .
```

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

2.2 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*.

2.2.1 List of Tutorials

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
def target_function(x):
    f1 = np.sin(4*np.pi*x).reshape(-1, 1) + 2
```

(continues on next page)

(continued from previous page)

```
    f2 = -np.sin(3*np.pi*x).reshape(-1, 1) + 2
    return np.concatenate([f1, f2], axis=1)
```

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*.

```
import pythia as pt
param = pt.parameter.Parameter(
    index=1, 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 also provides diverse function to generate multiindex arrays, e.g., `tensor`, `simplex`, `add_indices` and `subtract_indices`. But since we only have one variable in this tutorial, we only need one of them for now.

```
indices = pt.index.tensor([6])  # 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.

```
s = pt.sampler.ParameterSampler([param])
x_train = s.sample(1000)
w_train = np.ones(x_train.shape[0]) / x_train.shape[0]
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.

```
x_test = s.sample(1000)
y_test = target_function(x_test)
y_approx = surrogate.eval(x_test)  # evaluate PC expansion in test data
```

This concludes the first 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 function.

Complete Script

```
import os
import matplotlib.pyplot as plt
import numpy as np
import pythia as pt


def target_function(y):
    f1 = np.sin(4*np.pi*y).reshape(-1, 1) + 2
    f2 = -np.sin(3*np.pi*y).reshape(-1, 1) + 2
    return np.concatenate([f1, f2], axis=1)


print("TUTORIAL 01 - 1D approximation with PC expansion")

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

print(f"[{pt.misc.now()}] parameter information:")
print(param)

indices = pt.index.tensor([6])  # try 15 for good a approximation
index_set = pt.index.IndexSet(indices)  # we only have one parameter (y,)
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}] 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(f"[{pt.misc.now()}] compute PC expansion")
surrogate = pt.chaos.PolynomialChaos([param], index_set, x_train, w_train, y_train)

N = 1000
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}] error L2 (abs/rel): {e_L2:4.2e}/{e_L2_rel:4.2e}")
print(f"[{pt.misc.now()}] error max (abs/rel):")
```

(continues on next page)

(continued from previous page)

```
print(f"[{pt.misc.now()}]     first component:  {e_max[0]:4.2e}/{e_max_rel[0]:4.2e}")
print(f"[{pt.misc.now()}]     second component: {e_max[0]:4.2e}/{e_max_rel[0]:4.2e}")

PATH = "./img/"
os.makedirs(PATH, exist_ok=True)
yy = np.linspace(0, 1, 200).reshape(-1, 1)
plt.figure()
plt.title("Approximation of 1D function with PC")
plt.plot(yy, target_function(yy)[:, 0], color="blue", label="target")
plt.plot(yy, target_function(yy)[:, 1], color="red")
plt.plot(yy, surrogate.eval(yy)[:, 0], "--", color="blue", label="surrogate")
plt.plot(yy, surrogate.eval(yy)[:, 1], "--", color="red")
plt.legend()
plt.grid()
plt.savefig(PATH+"tutorial_01.png")
print(f"[{pt.misc.now()}] save plot to: {PATH}")
```

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
def target_function(x):
    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(
   index=1, name="x_1", domain=[0, 1], distribution='uniform'
)
param2 = pt.parameter.Parameter(
   index=2, 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(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.

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

```
test_sampler = pt.sampler.ParameterSampler(params)
x_test = test_sampler.sample(1000)
y_test = target_function(x_test)
y_approx = surrogate.eval(x_test)
```

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

```
import os
import matplotlib.pyplot as plt
import numpy as np
import pythia as pt


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


print("[{}] run TUTORIAL 02 - 2D approximation with pc".format(pt.misc.now()))

print(f"[{pt.misc.now()}] set parameters")
param1 = pt.parameter.Parameter(
   index=1, name="x_1", domain=[0, 1], distribution='uniform'
)
param2 = pt.parameter.Parameter(
   index=2, name="x_2", domain=[0, 1], distribution='uniform'
)
params = [param1, param2]

sdim = [13, 11]
indices = pt.index.tensor(sdim)
```

(continues on next page)

(continued from previous page)

```
index_set = pt.index.IndexSet(indices)
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}] 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(f"[{pt.misc.now()}] compute pc expansion")
surrogate = pt.chaos.PolynomialChaos(
   params, index_set, x_train, w_train, y_train)

N = 1000
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}] error L2 (abs/rel):  {e_L2:4.2e}/{e_L2_rel:4.2e}")
print(f"[{pt.misc.now()}] error max (abs/rel): {e_max[0]:4.2e}/{e_max_rel[0]:4.2e}")

# PLOT APPROXIMATION
path = "./img/"
os.makedirs(path, exist_ok=True)

x1 = np.linspace(0, 1, 200).reshape(-1, 1)
x2 = np.linspace(0, 1, 200).reshape(-1, 1)
xx = pt.misc.cartProd([x1, x2])  # cartesian product of two vectors
y_target = target_function(xx).reshape(x1.size, -1).T
y_approx = surrogate.eval(xx).reshape(x1.size, -1).T

fig, ax = plt.subplots(figsize=(10, 4), nrows=1, ncols=3)
fig.suptitle("Approximation of 2D function with PyThia")
im0 = ax[0].contourf(y_target, 15, extent=[0, 1, -3, 3])
ax[0].set_title("target function")
im1 = ax[1].contourf(y_approx, 15, extent=[0, 1, -3, 3])
ax[1].set_title("surrogate")
im2 = ax[2].contourf(y_target-y_approx, 15, extent=[0, 1, -3, 3])
ax[2].set_title("target - surrogate")
fig.subplots_adjust(right=0.8)
```

(continues on next page)

(continued from previous page)

```
cbar_ax = fig.add_axes([0.85, 0.15, 0.05, 0.7])
fig.colorbar(im2, cax=cbar_ax)
plt.savefig(path+"tutorial_02.png")

print(f"[{pt.misc.now()}] save plot to: {path}")
```

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

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

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

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

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 $a_1, \dots, a_M$ and with this the target function.

```
import numpy as np
a = np.array([1, 2, 3])
def target_function(x):
    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 $a_1, \dots, a_M$, i.e.,

```
import pythia as pt
params = [pt.parameter.Parameter(
    index=j, 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(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.

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

Complete Script

```
import numpy as np
import pythia as pt


def sobol_function(x, a=None, **kwargs):
    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)
```

(continues on next page)

(continued from previous page)

```
def sobol_sc(a, dim=1, **kwargs):
    sobol = {}
    beta = (1.0+a)**(-2) / 3
    var = np.prod(1.0 + beta) - 1.0
    sobol_tuples = pt.index.IndexSet(pt.index.tensor([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


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


def target_function(x):
    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(
    index=j, 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(len(params), max_dim-1)
index_set = pt.index.IndexSet(indices)
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}] 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(f"[{pt.misc.now()}] compute pc expansion")
surrogate = pt.chaos.PolynomialChaos(
    params, index_set, x_train, w_train, y_train)
```

(continues on next page)

(continued from previous page)

```
# test PC approximation
N = 1000
print(f"[{pt.misc.now()}] 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))

print(f"[{pt.misc.now()}]     test error L2 (abs/rel):",
      f" {error_L2:4.2e} / {error_L2_rel:4.2e}")
print(f"[{pt.misc.now()}]     test error max (abs/rel):",
      f" {error_max:4.2e} / {error_max_rel:4.2e}")

# compare Sobol indices
print(f"[{pt.misc.now()}] 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
a = np.array([1, 2, 4, 8])
def target_function(x):
    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.,

```
import pythia as pt
params = [pt.parameter.Parameter(
    index=j, 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(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.

```
s = pt.sampler.WLSTensorSampler(params, [dim]*len(params))
x_train = s.sample(10_000)
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"[{pt.misc.now()}]     test error L2 auto: {error_L2:4.2e}")
print(f"[{pt.misc.now()}]     test error L2 ref:  {error_L2_ref:4.2e}")

print(f"[{pt.misc.now()}] 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. Below you find the complete script you can use to run on your own system.

Complete Script

```
import numpy as np
import pythia as pt


def sobol_function(x, a=None, **kwargs):
    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)
```

(continues on next page)

(continued from previous page)

```
def sobol_sc(a, dim=1, **kwargs):
    sobol = {}
    beta = (1.0+a)**(-2) / 3
    var = np.prod(1.0 + beta) - 1.0
    sobol_tuples = pt.index.IndexSet(pt.index.tensor([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:
        return sobol

# function definitions
a = np.array([1, 2, 4, 8])
def target_function(x):
    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(
    index=j, name=f"x_{j+1}", domain=[0, 1], distribution='uniform')
    for j in range(a.size)]

dim = 7
indices = pt.index.simplex(len(params), dim-1)
index_set = pt.index.IndexSet(indices)
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}] 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(f"[{pt.misc.now()}] 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
```

(continues on next page)

(continued from previous page)

```
print(f"[{pt.misc.now()}] compute optimal mdx")
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)
print(f"[{pt.misc.now()}] 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(f"[{pt.misc.now()}] compute optimal pc expansion")
surrogate = pt.chaos.PolynomialChaos(params, index_set, x_train, w_train, y_train)

# test PC approximation
N = 1000
print(f"[{pt.misc.now()}] 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"[{pt.misc.now()}]     test error L2 auto: {error_L2:4.2e}")
print(f"[{pt.misc.now()}]     test error L2 ref:  {error_L2_ref:4.2e}")

# print Sobol coefficients
print(f"[{pt.misc.now()}] 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} ",
          )
```

2.3 pythia

2.3.1 pythia package

pythia.basis module

Assemble sparse univariate and multivariate basis polynomials.

Build univariate or multivariate normalized basis polynomials depending on the domain and distribution (and other degrees of freedom) of the parameter(s).

Currently supported are the following distribution types:

- uniform
- normal
- Gamma
- Beta

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

Assemble multivariate polynomial basis.

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

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*.

Return type
List[Callable]

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

Normalize orthogonal polynomials.

Normalize a polynomial of an orthogonal system with respect to the scalar product

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

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.

Return type
List[Callable]

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.

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*.

Return type
List[Callable]

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.

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*.

Return type
List[Callable]

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.

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*.

Return type
List[Callable]

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.

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*.

Return type
List[Callable]

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.

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*.

Return type
List[List[Callable]]

pythia.chaos module

Sample-based computation of polynomial chaos expansion.

This module provides a class to compute the polynomial chaos approximation of an unknown function given via input/output training data pairs via linear least-squares regression.

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.

Parameters

- **x** (*array_like*) – Parameter realizations in which the approximation is evaluated.
- **partial** (*list of int*) – List that specifies the number of derivatives in each component. Length is the number of parameters.

Returns
Evaluation of polynomial expansion in x values.

Return type
ndarray

property mean: ndarray

Mean of the PC expansion.

Return type
ndarray

property std: ndarray

Standard deviation of the PC expansion.

Return type
ndarray

property var: ndarray

Variance of the PC expansion.

Return type
ndarray

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

Compute automatic choice of multiindices.

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

Return type
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.

Compute the optimal multiindices for a polynomial chaos expansion based on an estimate of the Sobol coefficient values.

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. Sobol indices with smaller magnitude are ignored.

Returns

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

Return type
Tuple[ndarray, ndarray]

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.

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
n – Batchsize for assembling of information matrix.

Return type
int

pythia.index module

Create, manipulate and store information about multiindices.

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

Bases: object

Generate index set object for sparse PC expansion.

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

get_index_number(*indices*)

Get enumeration number of indices.

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

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

Returns
Array containing the enumeration numbers of the indices.

Return type
ndarray

get_sobol_tuple_number(*sobol_tuples*)

Get enumeration indices of Sobol tuples.

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

Returns
Array containing the enumeration number of the Sobol tuples.

Return type
ndarray

index_to_sobol_tuple(*indices*)

Map array of indices to their respective Sobol tuples.

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

Returns
List of Sobol tuples.

Return type
List[Tuple]

sobol_tuple_to_indices(*sobol_tuples*)

Map Sobol tuples to their respective indices.

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

Returns
List of index arrays for each given Sobol tuple.

Return type
List[ndarray]

pythia.index.**add_indices**(*index_list*)

Add multiple arrays of multiindices.

Concatenate multiple arrays of multiindices, remove duplicates and sort them by sum of multiindices.

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

Returns
Array with all multiindices.

Return type
ndarray

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

Create a simplex index set.

A simplex index set consists of all multiindices with sum less then or equal the maximum given.

Parameters

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

Returns
Array with all possible multiindices in simplex set.

Examples

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

Return type
ndarray

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

Sort multiindices and remove duplicates.

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

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

Returns
Sorted index array.

Return type
ndarray

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

Set difference of two index arrays.

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.

Return type
ndarray

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

Create a tensor index set.

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.

Examples

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

It is also possible to use the tensor functions to create n-variate subsets for the overall tensor set.

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

Return type
ndarray

pythia.misc module

Miscellaneous functions to support PyThia core functionality.

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

Split iterable into different batches of batchsize n.

Parameters

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

Yields
Iterable for different batches.

Return type
Iterator

pythia.misc.**cartProd**(*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 $v_1, \ldots, v_n$.

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

Returns
Cartesian product array.

Return type
ndarray

pythia.misc.**distributionDict**()

Set aliases for distribution descriptions.

Deprecated since version 2.0.0: *distributionDict* will be removed in PyThia 3.0.0.

Returns
Dictionary with aliases for distribution descriptions.

Return type
`Dict`

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

Dörfler marking for arbitrary values.

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)`.

Return type
`Tuple[ndarray, ndarray, int]`

pythia.misc.**formatTime**(*dt*)

Converts time (seconds) to time format string.

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

Returns
Formatted time string.

Return type
`str`

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.

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.

Return type
`ndarray`

pythia.misc.**get_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.

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.

Return type
ndarray

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.

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.

Return type
bool

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

Print a line of 80 characters by repeating indicator.

An additional message can be given.

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.

Return type
str

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

Alias for numpy.load().

Return type
ndarray

pythia.misc.**now**()

Get string of current machine date and time.

Returns
Formatted date and time string.

Return type
str

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

Wrapper for numpy save.

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

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.

Return type
None

pythia.misc.**shiftCoord**(*x*, *T*, *I*)

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

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

Parameters

- **x** (*array_like*) – Points in interval T.
- **T** (*array_like*) – Original interval.
- **I** (*array_like*) – Target interval.

Returns
Shifted values for *x*.

Return type
ndarray

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)`.

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*.

Return type
Sequence

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

Compute the weighted Least-Squares sampling bound.

The number of samples n is chosen such that

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

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.

Return type
int

pythia.parameter module

PyThia classes containing Parameter information.

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

Bases: object

Class used for stochasic parameters.

Parameters

- **index** (*int*) – Enumeration index of the parameter.
- **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: Union[List, Tuple, ndarray]

index: int

mean: Optional[float] = None

name: str

var: Optional[float] = None

pythia.sampler module

Sampler classes for generating in random samples and PDF evaluations.

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.

Return type
float

property dimension

Dimension of the parameters.

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

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.

Return type
float

pdf(*x*)

Evaluate PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

property std: float

Standard deviation of the distribution.

Return type
float

property var: float

Variance of the distribution.

Return type
float

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.

Return type
float

property dimension: float

Dimension of the parameters.

Return type
float

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Note: Not yet implemented.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

property mass: float

Mass of the PDF.

Return type
float

property maximum: float

Maximum value of the PDF.

The maximum of the Gamma distribution is given by

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

Return type
float

property mean: float

Mean value of the distribution.

Return type
float

pdf(*x*)

Evaluate PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

property std: float

Standard deviation of the distribution.

Return type
float

property var: float

Variance of the distribution.

Return type
float

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.

Return type
float

property dimension: float

Dimension of the parameters.

Return type
float

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

`log_pdf`(*x*)

Evaluate log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

`property mass: float`

Mass of the PDF.

Return type
float

`property maximum: float`

Maximum value of the PDF.

Return type
float

`property mean: float`

Mean value of the distribution.

Return type
float

`pdf`(*x*)

Evaluate PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

`sample`(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

`property std: float`

Standard deviation.

Return type
float

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.

Return type
ndarray

property dimension: int

Dimension of the parameters.

Return type
int

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate log-PDF.

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

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

property mass: float

Mass of the PDF.

Return type
float

property maximum: float

Maximum value of the PDF.

Return type
float

property mean: ndarray

Mean of the PDF.

Return type
ndarray

pdf(*x*)

Evaluate PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

weight(*x*)

Weights of the parameter product PDF.

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

Returns
Array of uniform weights for samples.

Return type
ndarray

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.

Return type
ndarray

property dimension: int

Dimension of the parameters.

Return type
int

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate log-PDF.

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

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

property mass: float

Mass of the PDF.

Return type
float

property maximum: float

Maximum value of the PDF.

Return type
float

property mean: ndarray

Mean of the PDF.

Return type
ndarray

pdf(*x*)

Evaluate PDF.

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

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

weight(*x*)

Weights of the product PDF.

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

Returns
Array of uniform weights for samples.

Return type
ndarray

class pythia.sampler.**Sampler**

Bases: ABC

Base class for all continuous samplers.

abstract property cov: Union[float, ndarray]

(Co)Variance of the pdf.

Return type
Union[float, ndarray]

abstract property dimension: int

Dimension of the ambient space.

Return type
int

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*.

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).

Return type
ndarray

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*.

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).

Return type
ndarray

abstract log_pdf(*x*)

Log-density of the samplers distribution.

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

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.

Return type
ndarray

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.

Return type
float

abstract property mean: Union[float, ndarray]

Mean value of the pdf.

Return type
Union[float, ndarray]

abstract pdf(*x*)

Density of the samplers distribution.

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

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

Returns
Density values at the points.

Return type
ndarray

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.

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.

Return type
ndarray

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.

Return type
float

property dimension: int

Parameter dimension.

Return type
int

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of uniform log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of uniform log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate uniform log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

property mass: float

Mass of the PDF.

Return type
float

property maximum: float

Maximum value of the PDF.

Return type
float

property mean: float

Mean value of the distribution.

Return type
float

pdf(*x*)

Evaluate uniform PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from uniform distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

property std: float

Standard deviation of the distribution.

Return type
float

property var: float

Variance of the distribution.

Return type
float

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

Bases: *Sampler*

Weighted Least-Squares sampler as in Cohen & Migliorati 2017.

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.

property cov: ndarray

Covariance of the PDF.

Return type
ndarray

property dimension: int

Dimension of the parameters.

Return type
int

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate log-PDF.

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

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

property mass

Mass of the PDF.

property maximum: float

Maximum value of the PDF.

Return type
float

property mean: ndarray

Mean of the PDF.

Return type
ndarray

pdf(*x*)

Evaluate PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

weight(*x*)

Weights for the PDF.

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

Returns
weights of evaluation points *x*.

Return type
ndarray

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

Bases: *Sampler*

WLS sampler for tensor multivariate basis.

Sampler for weighted Least-Squares sampling as described by Cohen and Migliorati. Only full tensor space can be sampled. This allows for univariate weighted Least-Squares sampling in each component.

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.

property cov: ndarray

Covariance of the PDF.

Return type
ndarray

property dimension: int

Dimension of the parameters.

Return type
int

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

`log_pdf`(*x*)

Evaluate log-PDF.

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

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

`property mass: float`

Mass of the PDF.

Return type
float

`property maximum: float`

Maximum value of the PDF.

Return type
float

`property mean: ndarray`

Mean of the PDF.

Return type
ndarray

`pdf`(*x*)

Evaluate PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

`sample`(*shape*)

Draw samples from distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

weight(*x*)

Weights for the PDF.

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

Returns
Weights of evaluation points *x*.

Return type
ndarray

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

Bases: *Sampler*

Sampler for univariate optimally distributed samples on given domain.

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

property cov: float

(Co)Variance of the distribution.

Return type
float

property dimension: int

Parameter dimension.

Return type
int

domain: ndarray

grad_x_log_pdf(*x*)

Evaluate gradient of uniform log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

hess_x_log_pdf(*x*)

Evaluate Hessian of uniform log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

log_pdf(*x*)

Evaluate uniform log-PDF.

Parameters
x (*array_like*) – Evaluation points.

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

Return type
ndarray

property mass: float

Mass of the PDF.

Return type
float

property maximum: float

Maximum value of the PDF.

Return type
float

property mean: float

Mean value of the distribution.

Return type
float

pdf(*x*)

Evaluate uniform PDF.

Parameters
x (*array_like*) – Evaluation points.

Returns
Values of PDF evaluated in *x*.

Return type
ndarray

sample(*shape*)

Draw samples from weighted least-squares parameter distribution.

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

Returns
Random samples of specified shape.

Return type
ndarray

property std: float

Standard deviation of the distribution.

Return type
float

property var: float

Variance of the distribution.

Return type
float

weight(*x*)

Weights for the pdf.

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

Returns
w – Weights of evaluation points *x*.

Return type
array_like

Return type
ndarray

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

Assign a univariate sampler to the given parameter.

Parameters
param (pythia.parameter.Parameter) –

Returns
Univariate sampler.

Return type
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.

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.

Return type
ndarray

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

Compute essential maximum of function by point evaluations.

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*.

Return type
float

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

Draw samples from pdf by rejection sampling.

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.

Return type
ndarray

Module contents

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

H

I

L

M

N

P

R

S

T

U

V

W