

Chaospy:

A modular implementation of Polynomial Chaos expansions and Monte Carlo methods

Simen Tennøe

Supervisors:

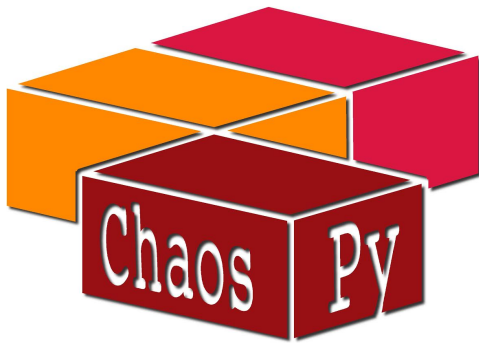
Jonathan Feinberg

Hans Petter Langtangen

Gaute Einevoll

Geir Halmes

University of Oslo, CINPLA



Chaospy is a Python toolbox for forward model UQ

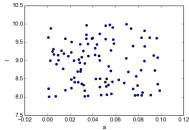


Properties of Chaospy

Chaospy is a Python toolbox for forward model UQ



Properties of Chaospy

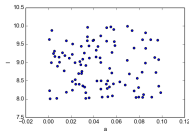


Monte Carlo methods

Chaospy is a Python toolbox for forward model UQ



Properties of Chaospy



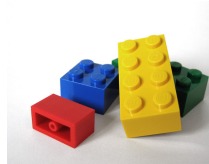
Monte Carlo methods

$$\sum_{n=0}^N c_n(x) P_n(q)$$

Polynomial Chaos

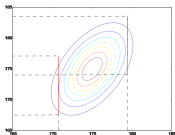
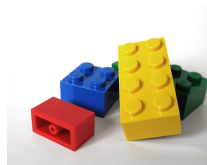
What is new in Chaospy

**Chaospy is modular and
therefore very flexible**



What is new in Chaospy

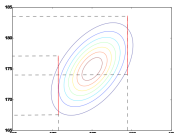
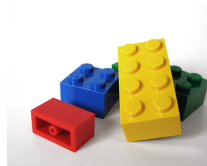
Chaospy is modular and therefore
very flexible



**Chaospy has support for
dependent variables**

What is new in Chaospy

Chaospy is modular and therefore very flexible

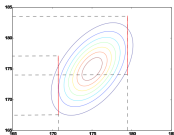
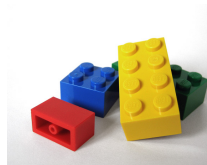


Chaospy has support for dependent variables

Chaospy has a large collection of methods and distributions

What is new in Chaospy

Chaospy is modular and therefore very flexible



Chaospy has support for dependent variables

Chaospy has a large collection of methods and distributions

It is easy to compare different methods on given a problem

Comparing Chaospy with Turns and Dakota

Feature	Dakota	Turns	Chaospy
Distributions	11	26	64
Copulas	1	7	6
Sampling schemes	4	7.5	7
Orthogonal polynomial schemes	4	3	5
Numerical integration strategies	7	0	7
Regression methods	5	4	8
Analytical metrics	6	6	7

Comparing Chaospy with Turns and Dakota

Feature	Dakota	Turns	Chaospy
Distributions	11	26	64
Copulas	1	7	6
Sampling schemes	4	7.5	7
Orthogonal polynomial schemes	4	3	5
Numerical integration strategies	7	0	7
Regression methods	5	4	8
Analytical metrics	6	6	7

Comparing Chaospy with Turns and Dakota

Feature	Dakota	Turns	Chaospy
Distributions	11	26	64
Copulas	1	7	6
Sampling schemes	4	7.5	7
Orthogonal polynomial schemes	4	3	5
Numerical integration strategies	7	0	7
Regression methods	5	4	8
Analytical metrics	6	6	7

Chaospy has support for many different methods

- ▶ Monte Carlo with variance reduction techniques

Chaospy has support for many different methods

- ▶ Monte Carlo with variance reduction techniques
- ▶ Intrusive and non-intrusive polynomial chaos

Chaospy has support for many different methods

- ▶ Monte Carlo with variance reduction techniques
- ▶ Intrusive and non-intrusive polynomial chaos
 - ▶ Pseudo-spectral method

Chaospy has support for many different methods

- ▶ Monte Carlo with variance reduction techniques
- ▶ Intrusive and non-intrusive polynomial chaos
 - ▶ Pseudo-spectral method
 - ▶ Point collocation/regression

All Chaospy needs is a Python wrapper around the forward model

```
def solver(*node):  
    # node: tuple of the uncertain stochastic parameters  
  
    model.set_parameters(node)
```



Chaos Py

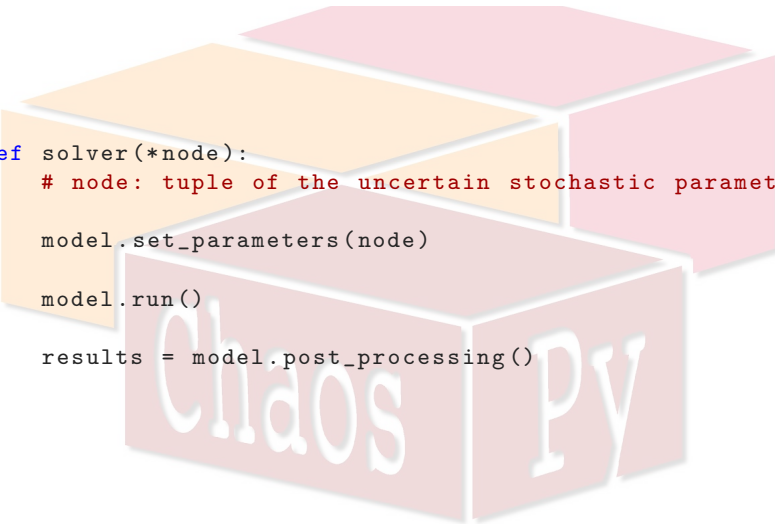
All Chaospy needs is a Python wrapper around the forward model

```
def solver(*node):  
    # node: tuple of the uncertain stochastic parameters  
  
    model.set_parameters(node)  
  
    model.run()
```

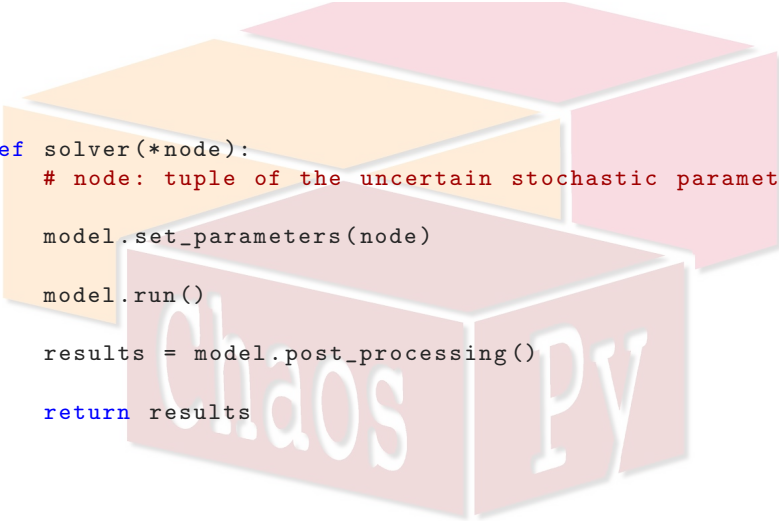
Chaos Py

All Chaospy needs is a Python wrapper around the forward model

```
def solver(*node):  
    # node: tuple of the uncertain stochastic parameters  
  
    model.set_parameters(node)  
  
    model.run()  
  
    results = model.post_processing()
```



All Chaospy needs is a Python wrapper around the forward model



```
def solver(*node):  
    # node: tuple of the uncertain stochastic parameters  
  
    model.set_parameters(node)  
  
    model.run()  
  
    results = model.post_processing()  
  
    return results
```

Chaospy is a completely generic software; for simplicity we use a very simple example problem

$$\frac{du(x)}{dx} = -au(x), \quad u(0) = I.$$

u The quantity of interest.

x Spatial location.

a, I Parameters containing uncertainties.

Chaospy is a completely generic software; for simplicity we use a very simple example problem

$$\frac{du(x)}{dx} = -au(x), \quad u(0) = I.$$

u The quantity of interest.

x Spatial location.

a, I Parameters containing uncertainties.

$$a \sim \text{Uniform}(0, 0.1)$$

$$I \sim \text{Uniform}(8, 10)$$

Chaospy is a completely generic software; for simplicity we use a very simple example problem

$$\frac{du(x)}{dx} = -au(x), \quad u(0) = I.$$

u The quantity of interest.

x Spatial location.

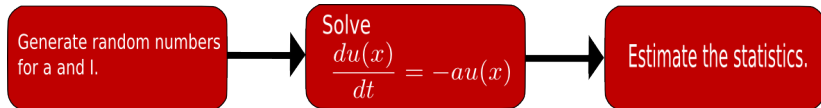
a, I Parameters containing uncertainties.

$$a \sim \text{Uniform}(0, 0.1)$$

$$I \sim \text{Uniform}(8, 10)$$

We want to compute $E(u)$ and $\text{Var}(u)$.

Monte Carlo integration can be used for any model



Monte Carlo with Chaospy

```
import chaospy as cp  
import numpy as np
```



Chaos Py

Monte Carlo with Chaospy

```
import chaospy as cp
import numpy as np

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
```



Chaos Py

Monte Carlo with Chaospy

```
import chaospy as cp
import numpy as np

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)

# Joint distribution
dist = cp.J(dist_a, dist_I)
```



Chaos Py

Monte Carlo with Chaospy

```
import chaospy as cp
import numpy as np

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)

# Joint distribution
dist = cp.J(dist_a, dist_I)

samples = dist.sample(size=1000)
```

Chaos Py

Monte Carlo with Chaospy

```
import chaospy as cp
import numpy as np

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)

# Joint distribution
dist = cp.J(dist_a, dist_I)

samples = dist.sample(size=1000)

# solver returns u(x), where x is fixed
# samples_u: list of all u(x) for each set of a and I
samples_u = [solver(a, I) for a, I in samples]
```

Monte Carlo with Chaospy

```
import chaospy as cp
import numpy as np

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)

# Joint distribution
dist = cp.J(dist_a, dist_I)

samples = dist.sample(size=1000)

# solver returns u(x), where x is fixed
# samples_u: list of all u(x) for each set of a and I
samples_u = [solver(a, I) for a, I in samples]

E = np.mean(samples_u, 0)
Var = np.var(samples_u, 0)
```

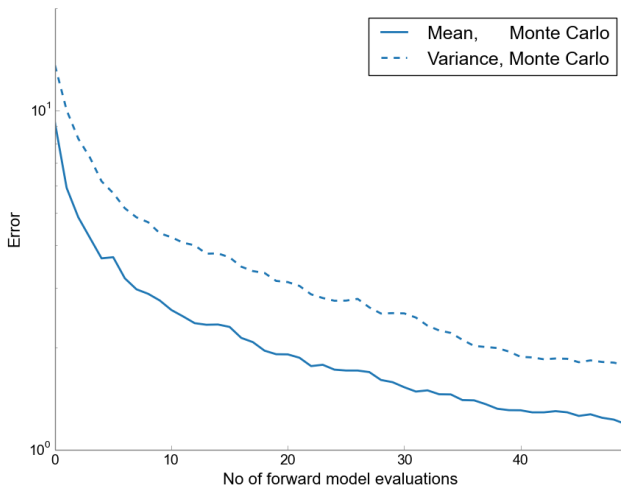
Convergence of Monte Carlo is slow

$$\varepsilon_E = \int |E(u) - E(\hat{u})| dx \qquad \varepsilon_{Var} = \int |\text{Var}(u) - \text{Var}(\hat{u})| dx$$

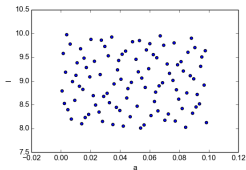
Convergence of Monte Carlo is slow

$$\varepsilon_E = \int |E(u) - E(\hat{u})| dx$$

$$\varepsilon_{Var} = \int |\text{Var}(u) - \text{Var}(\hat{u})| dx$$

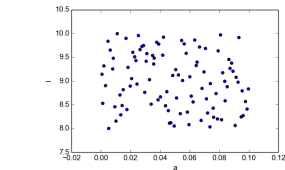
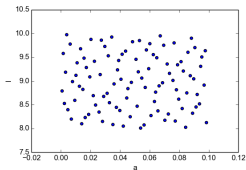


Chaospy has several variance reduction techniques for sampling a distribution



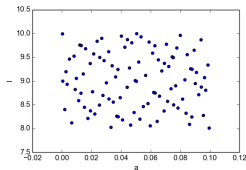
Hammersley sampling:

```
nodes = dist.sample(100, "M")
```



Latin Hypercube sampling:

```
nodes = dist.sample(100, "L")
```



Halton sampling

```
nodes = dist.sample(100, "H")
```

Sobol sampling

```
nodes = dist.sample(100, "S")
```


The different sampling schemes available in Chaospy compared to Turns and Dakota

	Dakota	Turns	Chaospy
Quasi-Monte Carlo scheme			
Faure sequence	No	Yes	No
Halton sequence	Yes	Yes	Yes
Hammersley sequence	Yes	Yes	Yes
Haselgrove sequence	No	Yes	No
Korobov lattice	No	No	Yes
Niederreiter sequence	No	Yes	No
Sobol sequence	No	Yes	Yes
Other methods			
Antithetic variables	No	No	Yes
Importance sampling	Yes	Yes	Yes
Latin Hypercube sampling	Yes	Limited	Yes

Quasi-Monte Carlo with Latin Hypercube sampling

```
import chaospy as cp
import numpy as np

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

samples = dist.sample(size=1000)

samples_u = [solver(a, I) for a, I in samples]

E = np.mean(samples_u, 0)
Var = np.var(samples_u, 0)
```

Quasi-Monte Carlo with Latin Hypercube sampling

```
import chaospy as cp
import numpy as np

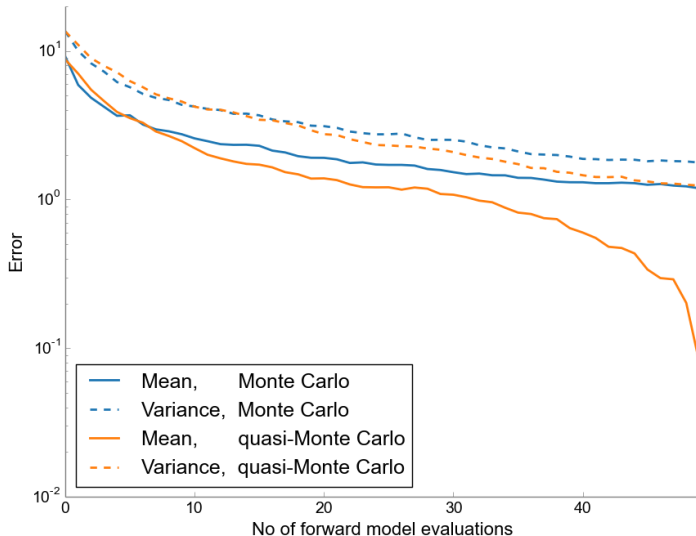
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

samples = dist.sample(size=1000, rule="L")

samples_u = [solver(a, I) for a, I in samples]

E = np.mean(samples_u, 0)
Var = np.var(samples_u, 0)
```

Convergence of quasi-Monte Carlo is better than Monte Carlo, but still slow



Mapping in probability space; the idea behind Polynomial Chaos (PC) theory is to approximate our forward model with a polynomial

$$u(x; q) \approx \hat{u}_M(x; q) = \sum_{n=0}^N c_n(x) P_n(q)$$

Coefficient Polynomial

$\hat{u}_M(x; q)$ is the mapping from the uncertain variables q to the response variable u , x is a fixed variable.

Mapping in probability space; the idea behind Polynomial Chaos (PC) theory is to approximate our forward model with a polynomial

$$u(x; q) \approx \hat{u}_M(x; q) = \sum_{n=0}^N c_n(x) P_n(q)$$

Coefficient Polynomial

$\hat{u}_M(x; q)$ is the mapping from the uncertain variables q to the response variable u , x is a fixed variable.

Mean and variance are calculated from $\hat{u}_M(x; q)$.

Mapping in probability space; the idea behind Polynomial Chaos (PC) theory is to approximate our forward model with a polynomial

$$u(x; q) \approx \hat{u}_M(x; q) = \sum_{n=0}^N \underbrace{c_n(x)}_{\text{Coefficient}} \underbrace{P_n(q)}_{\text{Polynomial}}$$

$\hat{u}_M(x; q)$ is the mapping from the uncertain variables q to the response variable u , x is a fixed variable.

Mean and variance are calculated from $\hat{u}_M(x; q)$.

Mapping in probability space; the idea behind Polynomial Chaos (PC) theory is to approximate our forward model with a polynomial

$$u(x; q) \approx \hat{u}_M(x; q) = \sum_{n=0}^N c_n(x) P_n(q)$$

Coefficient Polynomial

$\hat{u}_M(x; q)$ is the mapping from the uncertain variables q to the response variable u , x is a fixed variable.

Mean and variance are calculated from $\hat{u}_M(x; q)$.

P_n are orthogonal polynomials and are generally calculated through the three-term discretized Stiltjes recursion

```
dist = cp.Normal()
```

```
P = cp.orth_ttr(3, dist)
```

Chaos Py

P_n are orthogonal polynomials and are generally calculated through the three-term discretized Stiltjes recursion

```
dist = cp.Normal()
P = cp.orth_ttr(3, dist)
print P
[1.0, q0, q0^2-1.0, q0^3-3.0q0]
```

Chaos Py

Methods for generating expansions of orthogonal polynomials

Orthogonalization Method	Dakota	Turns	Chaospy
Askey–Wilson scheme	Yes	Yes	Yes
Bertran recursion	No	No	Yes
Cholesky decomposition	No	No	Yes
Discretized Stieltjes	Yes	No	Yes
Modified Chebyshev	Yes	Yes	No
Modified Gram–Schmidt	Yes	Yes	Yes

The pseudo-spectral method, used to calculate c_n , needs numerical integration, which demands generating quadrature nodes and weights

```
dist = cp.Normal()
```

```
nodes, weights = cp.generate_quadrature(2, dist, rule="G")
```

Chaos

Py

The pseudo-spectral method, used to calculate c_n , needs numerical integration, which demands generating quadrature nodes and weights

```
dist = cp.Normal()

nodes, weights = cp.generate_quadrature(2, dist, rule="G")

print nodes
[[-1.73205081  0.          1.73205081]]
print weights
[ 0.16666667  0.66666667  0.16666667]
```

Numerical integration strategies implemented in the three software toolboxes

Node and weight generators	Dakota	Turns	Chaospy
Clenshaw-Curtis quadrature	Yes	No	Yes
Cubature rules	Yes	No	No
Gauss-Legendre quadrature	Yes	No	Yes
Gauss-Patterson quadrature	Yes	No	Yes
Genz-Keister quadrature	Yes	No	Yes
Leja quadrature	No	No	Yes
Monte Carlo integration	Yes	No	Yes
Optimal Gaussian quadrature	Yes	No	Yes

One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)
```

The logo for Chaospy is composed of three 3D rectangular blocks. Two blocks are light orange and are positioned behind a larger, light pink block. The pink block is in the foreground and has the words "Chaos" and "Py" written on its front face in a white, serif font. The "Chaos" is on the left and "Py" is on the right, separated by a vertical line.

Chaos Py

One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

P = cp.orth_ttr(2, dist)
```

The logo for Chaospy, consisting of the words "Chaos" and "Py" in a white, serif font, positioned on the front face of a large, 3D pink cube. The cube is part of a larger arrangement of three cubes: one pink cube in the background, one orange cube to the left, and this pink cube in the foreground. The cubes are slightly offset from each other, creating a sense of depth.

Chaos Py

One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

P = cp.orth_ttr(2, dist)

nodes, weights = cp.generate_quadrature(3, dist)
```

Chaos Py

One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

P = cp.orth_ttr(2, dist)

nodes, weights = cp.generate_quadrature(3, dist)

samples_u = [solver(*node) for node in nodes.T]
```

Chaos Py

One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

P = cp.orth_ttr(2, dist)

nodes, weights = cp.generate_quadrature(3, dist)

samples_u = [solver(*node) for node in nodes.T]

u_hat = cp.fit_quadrature(P, nodes, weights, samples_u,
                          rule="Gaussian")
```

One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

P = cp.orth_ttr(2, dist)

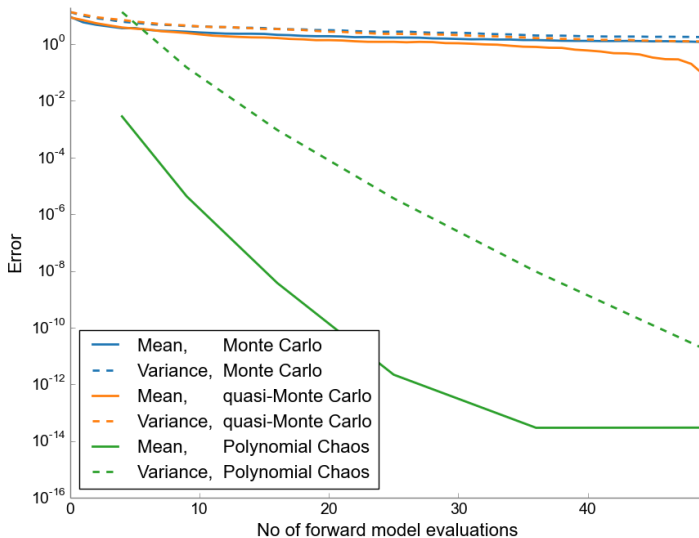
nodes, weights = cp.generate_quadrature(3, dist)

samples_u = [solver(*node) for node in nodes.T]

u_hat = cp.fit_quadrature(P, nodes, weights, samples_u,
                           rule="Gaussian")

mean = cp.E(u_hat, dist)
var = cp.Var(u_hat, dist)
```

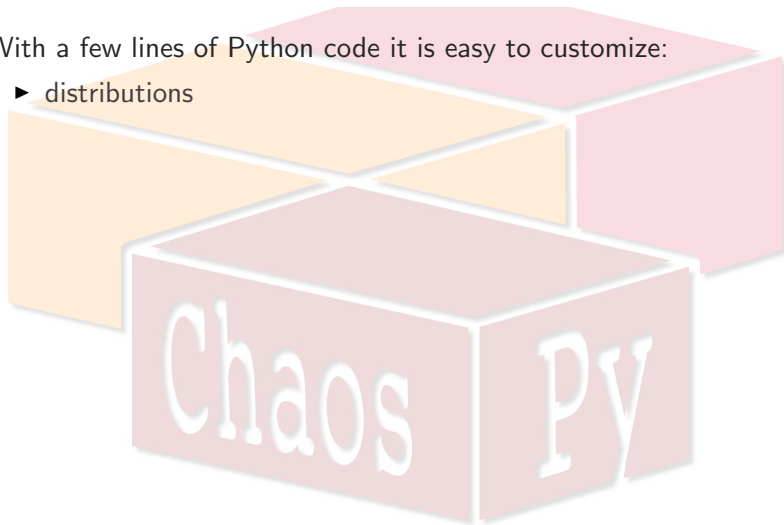
Convergence of polynomial chaos is much faster than the Monte Carlo methods



Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

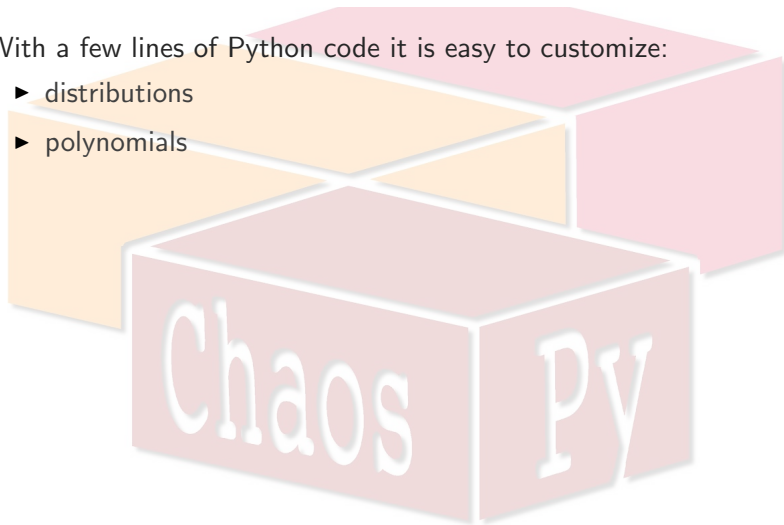
- distributions



Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials



Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes



Chaos Py

Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes
- ▶ sampling schemes



Chaos

Py

Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes
- ▶ sampling schemes
- ▶ statistical analysis of the result



Chaos Py

Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes
- ▶ sampling schemes
- ▶ statistical analysis of the result



Chaos

Py

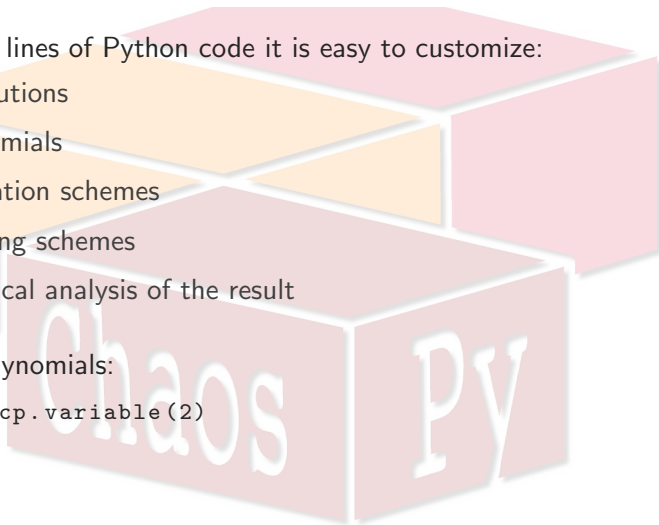
Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes
- ▶ sampling schemes
- ▶ statistical analysis of the result

Custom polynomials:

```
q0, q1 = cp.variable(2)
```



Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes
- ▶ sampling schemes
- ▶ statistical analysis of the result

Custom polynomials:

```
q0, q1 = cp.variable(2)
phi = cp.Poly([1, q0, q1, q0**2 - 1, q0*q1])
```

Chaospy is an ideal tool for research in UQ for the statistics expert

With a few lines of Python code it is easy to customize:

- ▶ distributions
- ▶ polynomials
- ▶ integration schemes
- ▶ sampling schemes
- ▶ statistical analysis of the result

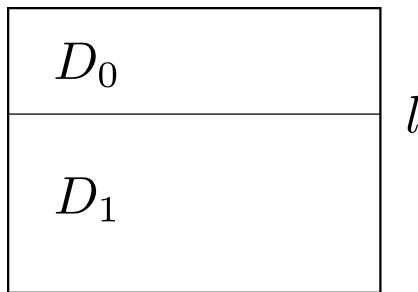
Custom polynomials:

```
q0, q1 = cp.variable(2)
phi = cp.Poly([1, q0, q1, q0**2 - 1, q0*q1])

print phi
[1, q0, q1, q0^2-1, q0q1]
```

Chaospy handles Polynomial Chaos expansions with stochastically dependent variables

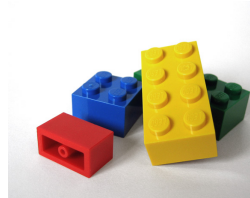
Diffusion in layered media with uncertain boundary, l , and uncertain diffusion constants, D_0 , D_1 .



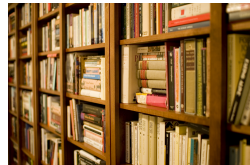
Uncertain l slows down convergence, but introduction of auxiliary *dependent* variables restores convergence.

Summary: Chaospy is a Python toolbox for forward model UQ with advanced Monte Carlo methods and Polynomial Chaos expansions

Chaospy is modular, flexible,
with syntax that resembles
the mathematics



A vast collection of methods,
ideal for method comparisons



Summary: Chaospy is a Python toolbox for forward model UQ with advanced Monte Carlo methods and Polynomial Chaos expansions

Installation instructions:

<https://github.com/hplgit/chaospy>

Reference:

Feinberg, J., & Langtangen, H. P. (2015). Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal Of Computational Science*, 11, 46-57

<http://hplgit.github.io/chaospy/doc/pub/chaospy-4screen.pdf>



Summary: Chaospy is a Python toolbox for forward model UQ with advanced Monte Carlo methods and Polynomial Chaos expansions

Installation instructions:

<https://github.com/hplgit/chaospy>

Reference:

Feinberg, J., & Langtangen, H. P. (2015). Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal Of Computational Science*, 11, 46-57

<http://hplgit.github.io/chaospy/doc/pub/chaospy-4screen.pdf>

Questions?

