Chaospy:
A modular implementation of Polynomial Chaos expansions and Monte Carlo methods

Simen Tennøe

Supervisors:
Jonathan Feinberg
Hans Petter Langtangen
Gaute Einevoll
Geir Halnes

University of Oslo, CINPLA
Chaospy is a Python toolbox for forward model UQ

Properties of Chaospy

Monte Carlo methods

Polynomial Chaos

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Dakota</th>
<th>Turns</th>
<th>Chaospy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributions</td>
<td>11</td>
<td>26</td>
<td>64</td>
</tr>
<tr>
<td>Copulas</td>
<td>1</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>Sampling schemes</td>
<td>4</td>
<td>7.5</td>
<td>7</td>
</tr>
<tr>
<td>Orthogonal polynomial schemes</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Numerical integration strategies</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Regression methods</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Analytical metrics</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
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

```python
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

\[
\frac{du(x)}{dx} = -au(x), \quad u(0) = l.
\]

\(u\) The quantity of interest.
\(x\) Spatial location.
\(a, l\) Parameters containing uncertainties.

\[
a \sim \text{Uniform}(0, 0.1) \quad l \sim \text{Uniform}(8, 10)
\]

We want to compute \(E(u)\) and \(\text{Var}(u)\).
Monte Carlo integration can be used for any model.
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

\[ \varepsilon_E = \int |E(u) - E(\hat{u})| \, dx \quad \varepsilon_{\text{Var}} = \int |\text{Var}(u) - \text{Var}(\hat{u})| \, dx \]
Chaospy has several variance reduction techniques for sampling a distribution

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

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

Latin Hypercube sampling:
```
nodes = dist.sample(100, "L")
```

Sobol sampling
```
nodes = dist.sample(100, "S")
```
The different sampling schemes available in Chaospy compared to Turns and Dakota

<table>
<thead>
<tr>
<th>Quasi-Monte Carlo scheme</th>
<th>Dakota</th>
<th>Turns</th>
<th>Chaospy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faure sequence</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Halton sequence</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Hammersley sequence</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Haselgrove sequence</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Korobov lattice</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Niederreiter sequence</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Sobol sequence</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other methods</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Antithetic variables</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Importance sampling</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Latin Hypercube sampling</td>
<td>Yes</td>
<td>Limited</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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

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

\( \hat{u}_M(x; q) \) 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

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

print P
[1.0, q0, q0^2-1.0, q0^3-3.0q0]
```
## Methods for generating expansions of orthogonal polynomials

<table>
<thead>
<tr>
<th>Orthogonalization Method</th>
<th>Dakota</th>
<th>Turns</th>
<th>Chaospy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Askey–Wilson scheme</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Bertran recursion</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cholesky decomposition</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Discretized Stieltjes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Modified Chebyshev</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Modified Gram–Schmidt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The pseudo-spectral method, used to calculate $c_n$, needs numerical integration, which demands generating quadrature nodes and weights.

```python
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

<table>
<thead>
<tr>
<th>Node and weight generators</th>
<th>Dakota</th>
<th>Turns</th>
<th>Chaospy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clenshaw-Curtis quadrature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cubature rules</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Gauss-Legendre quadrature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Gauss-Patterson quadrature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Genz-Keister quadrature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Leja quadrature</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Monte Carlo integration</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal Gaussian quadrature</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
One slide is enough for the full implementation with the pseudo-spectral method in Chaospy

```python
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
- polynomials
- integration schemes
- sampling schemes
- statistical analysis of the result

Custom polynomials:

```python
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:


Questions?