Theory Guide to Chaospy: Non-intrusive Polynomial Chaos Expansion in Python

Jonathan Feinberg, University of Oslo
Hans Petter Langtangen, University of Oslo

March 4, 2015

Contents

1 Introduction **3**
- 1.1 Problem Definition . . . 3
- 1.2 Compared with other Software . . . 5
- 1.3 Tutorial . . . 7

2 Distributions **11**
- 2.1 Variable Construction . . . 13
- 2.2 Statistical Moments . . . 17
- 2.3 Variance Reduction Techniques . . . 19

3 Polynomial Chaos Expansions **21**
- 3.1 Orthogonal Polynomial Construction . . . 21
- 3.2 Manual Polynomial Construction . . . 23
- 3.3 Polynomial Expansions Constrution . . . 26
- 3.4 Three Terms Recursion . . . 28

4 Estimating Fourier Coefficients **29**
- 4.1 Numerical Integration . . . 34
- 4.2 Pseudo Spectral Projection . . . 36
- 4.3 Point Collocation Method . . . 37
- 4.4 Descriptive Tools . . . 39

1 Introduction

`chaospy` is a python module than implements probabilistic collocation method with polynomial chaos expansion. It contains a distribution class with focus on usage in generalized polynomial chaos. The toolbox is designed from the ground up for ease of use for the novice user. However at the same time the toolbox is design to be very flexible. It allows for construction and manipulation of random variables and polynomial expansion, customization of numerical integration schemes, and methods that brings user defined components into the mix. The toolbox also includes a large suite of tools which is design to make the customization simple.

1.1 Problem Definition

Defining the Problem We consider a computational science problem in space $\boldsymbol{x}$ and time t where the aim is to quantify the uncertainty in some response Y, computed by a forward model f, which depends on uncertain input parameters $\boldsymbol{Q}$:

$$Y = f(\boldsymbol{x}, t, \boldsymbol{Q}). \tag{1}$$

We treat $\boldsymbol{Q}$ as a vector of model parameters, and Y is normally computed as some grid function in space and time. The uncertainty in this problem stems from the parameters $\boldsymbol{Q}$, which are assumed to have a known joint probability density function $p_{\boldsymbol{Q}}$. The challenge is that we want to quantify the uncertainty in Y, but nothing is known about its density p_Y. The goal is then to either build the density p_Y or relevant descriptive properties of Y using the density $p_{\boldsymbol{Q}}$ and the forward model f. For all practical purposes this must be done by a numerical procedure.

Numerical Uncertainty Quantification In this paper, we focus on two approaches to numerically quantify uncertainty: Monte Carlo simulation and non-intrusive global polynomial chaos expansions. For a review of the former, there is a very useful book by Rubinstein, Reuven and Kroese [51], while for the latter, see the excellent book by Xiu [63]. Note that other methods for performing uncertainty quantification also exist, such as perturbation methods, moment equations, and operator based methods. These methods are all discussed in [63], but are less general and less widely applicable than the two treated in this paper.

Software Options Available The number of toolboxes available to perform Monte Carlo simulation is vastly larger than the number of toolboxes for non-intrusive polynomial chaos expansion. As far as the authors know, there are only two viable options for the latter class of methods: *The Dakota Project* (referred to as Dakota) [15] and *Opus Open Turns library* (referred to as Turns) [2]. Both packages consist of libraries with extensive sets of tools, where Monte Carlo simulation and non-intrusive polynomial chaos expansion are just two tools available among several others.

Note about Developer Perspective It is worth noting that both Dakota and Turns can be used from two perspectives: as a user and as a developer. Both packages are open source projects with comprehensive developer manuals. As such, they both allow anyone to extend the software with any functionality one sees fit. However, these extension features are not targeting the common user and require a deeper understanding of both coding practice and the underlying design of the library. In our opinion, the threshold for a common user to extend the library is so high that it is normally out of reach. Consequently, we are in this paper only considering Dakota and Turns from the point of view of the common user.

Dakota and Turns' Philosophy Dakota requires the forward model f to be wrapped in a stand-alone callable executable, which then is linked to the analysis software through a configuration file. The technical steps are somewhat cumbersome, but has their advantage in that already built and installed simulation software can be used without writing a line of code.

The Turns software adopts another philosophy. Instead of a standalone application, Turns offers an interface to its tools only through the programming languages Python and R. This approach requires the user to know how to program in the supported languages, but this also has clear benefits as an interface through a programming language allows for a more level of integration between the user's model and the tools used to analyse them. Also, exposing the software's internal components through an application programming interface (API) allows a higher detailed control over the tools and how they can be combined in statistical algorithms. This feature is attractive to scientists who would like the possibility to experiment with new or non-standard methods in ways not thought of before. For example, consider bootstrapping [13], a popular method for measuring the stability of any parameter estimation. Neither Dakota nor Turns support bootstrapping directly. However, since Turns exposes some of the inner components to the user, a programmer can combine these to implement a custom bootstrapping technique.

Introducing Chaospy This paper describes a new, third alternative software package called Chaospy. Like Dakota and Turns, it is a toolbox for analysing uncertainty using advanced Monte Carlo simulation and non-intrusive polynomial chaos expansions. However, unlike the others, it aims to assist scientists in constructing tailored statistical methods by combining a lot of fundamental and advanced building blocks. Chaospy builds upon the same philosophy as Turns in that it offers flexibility to the user, but takes it significantly further. In Chaospy, it is possible to gain detailed control and add user defined functionality to all the following: random variable generation, polynomial construction, sampling schemes, numerical integration rules, response evaluation, and point collocation. The software is designed from the ground up in Python to be modular and easy to experiment with. The number of lines of code to achieve a full uncertainty analysis is amazingly low. It is also very easy to compare a range of methods in a given problem. Standard statistical methods are easily accessible through a few lines of R or Pandas code, and one may think of Chaospy as a tool similar to R or Pandas, just tailored to polynomial chaos expansion and

Monte Carlo simulation.

Comparative Toolboxes Although Chaospy is designed with a large focus on modularity, flexibility, and customization, the toolbox comes with a wide range of pre-defined statistical methods. Within the scope of Monte Carlo sampling and non-intrusive polynomial chaos expansion, Chaospy has a competitive collection of methods, comparable to both Dakota and Turns. It also offers some novel features regarding statistical methods, first and foremostly a flexible framework for defining and handling input distributions, including dependent stochastic variables. Detailed comparisons of features in the three packages appear throughout the paper.

[**hpl:** *Should highlight here the theoretical foundations/methods that make Chaospy novel in the UQ landscape. Much is said about software flexibility, but early on (here!) it must be evident that there is a lot of new stuff in Chaospy.*]

Section Overview The paper is structured as follows. We start in Section **??** with a quick demonstration of how the software can be used to perform uncertainty quantification in a simple physical problem. Section 2 addresses probability distributions and the theory relevant to perform Monte Carlo simulation. Section 3 concerns non-intrusive polynomial chaos expansions, while conclusions and topics for further work appear in Section **??**.

1.2 Compared with other Software

Within the realm of polynomial chaos expansion there are not many alternatives as far as the author knowns. However there are a couple of exceptions: the Dakota project (`dakota`) [15] and the Opus Open Turns library (`turns`) [2]. A full compare of the three alternatives is beyond the scope of the paper, but a list of some of the features is as follows:

- In `dakota` only one transformation is supported: the Nataf transformation[44]. `turns` support more through a class of copulas[45]. `chaospy` supports copulas, but can easily be used to create any type of dependency that can be formulated as a Rosenblatt transformation[50].

- Neither `dakota` nor `turns` support easy to construct user defined distributions. Some flexibility exist through combining distributions and copulas in `turns`, but that is how far it extend. In `chaospy` this however have full flexibility, and is designed to make it very easy to construct both user defined variables, transformation and the possibility to join them together.

- Constructing a completely new distribution from scratch is done easily through a constructor. Only lower and upper limit, and a cumulative distribution function needs to be provided for each univariate distribution. Multivariate distributions are created by joining univariate distributions together with potentially any dependencies. Neither `dakota` nor `turns` supports this.

- Spite only needing a minimum of input to construct a distribution, it can still supports forward and inverse Rosenblatt transformations, raw

statistical moments, (Pseudo-)random number generator, and three terms recursion-relation coefficients generator. All of this is supported through approximation methods embedded in the software.

- All software packages can generate random samples for all it's distributions, however only `dakota` and `chaospy` supports variance reduction techniques [34]. The former supports 2 quasi-Monte Carlo methods and latin hypercube sampling, while the latter supports 4 methods, latin hypercube sampling and antithetic variates. And with the Rosenblatt transformation available, any user defined sample scheme on the unit hypercube can be mapped to the distribution space. On top of that, because of the modularity of the software, control variable is also easily available.

- For custom distribution outside the Askey-scheme polynomial expansions has to be estimated numerically. All software supports Gram-Schmidt orthogonalisation[61]. The more stable orthogonalisation through three terms recursion relation[20] is not supported by `turns`. The implementation in `dakota` and `chaospy` both support discretized Stieltjes. Modified Chebyshev is supported by `dakota` and `turns`, while `chaospy` is the only one that supports Cholesky decomposition and Bertran's recursive formula[5].

- All polynomial expansion generation above usually assumes stochastically independence between variables. In the case of `chaospy` with Gram-Schmidt, and Cholesky decomposition, this does not need to be the case. An orthogonal expansion can be generated directly given a multivariate distribution with dependent variables.

- If a custom polynomial is required, like Lagrange, Taylor etc., `chaospy` has a flexible polynomial class for allowing the user to define it's own polynomial expansion. It also comes provided with a collection of tools for performing advance manipulation of the expansions. This is not the case with `dakota` and `turns`.

- Estimating the Fourier coefficients using spectral projection is supported by `dakota` and `chaospy`, but not by `turns`. Both former has a large collection of schemes for implementation.

- Estimating the Fourier coefficients using point collocation method is supported by all software packages. Each supporting a semi-overlapping set of regression methods for over and under determined problems. Here `dakota` supports 7 methods, `turns` 4 and `chaospy` 6.

- The sample generation and numerical integration rule is disjoint in `chaospy`, implying that a user defined scheme can easily be used instead of the software predefined schemes. There are also tools for joining simple user defined schemes into more complicated multivariate schemes.

- `chaospy` does not come with a predetermined adaptive scheme, like `dakota` and `turns`. Instead since the distribution construction, sampling scheme, numerical integration scheme, polynomial construction, and Fourier coefficients estimation are all disjoint and flexible, a custom user-defined adaptive scheme can be constructed. So not only is it possible to mimic `dakota`'s and `turns`' adaptive schemes, but almost any custom scheme can be created with the `chaospy` framework.

The general take back from this list is that `dakota` has a large toolbox of features, but is lacking the handling of advanced variables, and `turns` has a better handle on advanced variables, but lack several of the tools the other softwares has. `chaospy` however has both the large collection and the ability to handle complex dependencies, not only from copulas, but dependent random variables in general. Also it is constructed to have a level of flexibility that allows for a high order of customization the other softwares can not do.

1.3 Tutorial

In this section, the `chaospy` package will be introduced. It will show how the package can be used to performing uncertainty quantification with a high level of flexibility. Starting with notation, `chaospy` is used together with the standard library `numpy`. To distinguish between the two, the former is abbreviated `pc` and latter is imported into the name space:

```
import chaospy as pc
from numpy import *
```

As an example to follow, consider the following generic problem description. We have a model `u(x,q)` that can be considered a wrapper for some larger numerical solver, where `x` are spatio-temporal coordinates, and `q` are model parameters defining the shape of the solution. It is assumed that the computational cost of evaluating the numerical solver is high, e.g. it can take hours or days to perform a single simulation. And furthermore it is assumed that `q` is unknown and can only be described as a random quantity with a known probability density function.

As an example consider a one dimensional diffusion equation with permeability piecewise constant. In this case we look at a problem of four layers where each layer has the permeability `0.001`, `1.`, `0.001` and `1.` respectively. The boundaries defining the discontinuous shift in permeability is here defined as uncertain with triangle distributions. We defined them as

```
q0 = pc.Triangle(0/6., 1/6., 2/6.)
q1 = pc.Triangle(2/6., 3/6., 4/6.)
q2 = pc.Triangle(4/6., 5/6., 6/6.)
Q = pc.J(q0,q1,q2)
```

Here `pc.J` is a joint operator making `Q` a multivariate random variable, and `q0`, `q1` and `q2` are marginals.

To implement a model into the automated framework of `chaospy`, the model must be a single input and single output function `U=u(q)`, where `q` is either a scalar or a vector and the output `U` is either a scalar, a vector or a tensor. To achieve this, it is often prudent to include a model wrapper. For the diffusion equation such a wrapper might have the form

```
def model_wrapper(q):
    layers = [0.001, 1., 0.001, 1.]
    bounds = [0, q[0], q[1], q[2], 1]
    U_0, U_L = 0, 1
    sl = SerialLayers(layers, bounds, U_0, U_L)
    return sl(z)
```

To visualize the variability in the model we generate 5 samples from `Q` and evaluate the model:

```
samples = Q.sample(5)
U = [model_wrapper(q) for q in samples.T]
```

In figure 1 the five model solution are plotted as functions of `x`. They illustrate

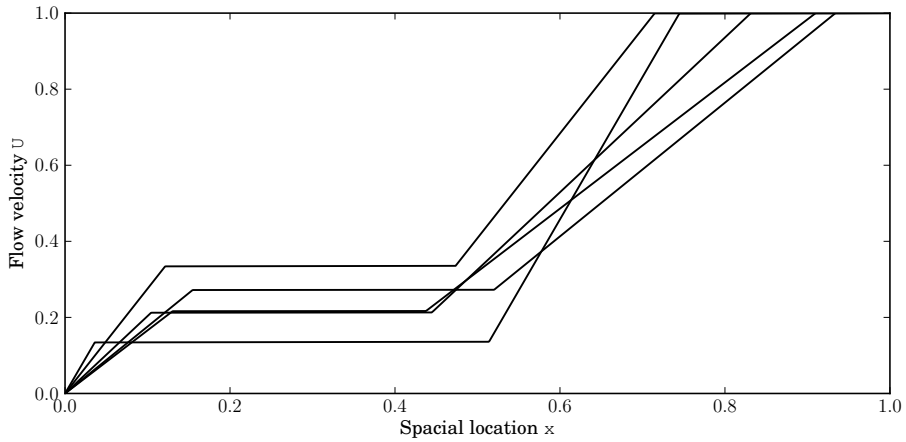

Figure 1: Example evaluations of the diffusion equation solver illustrating the variations in solutions when the boundaries varies.

that changes in the input parameter can in some case have large effect on the output.

When implementing uncertainty quantification on a large scale simulation tool, there might be a need to perform analysis on various approximation orders. This might include not being satisfied with the accuracy of the analysis, and wants to increase the sample size as a result. And since many methods are nested, this should be possible. To avoid having to run the simulation multiple

times using the same parameter setup, `chaospy` has the `lazy_eval` function wrapper. It is called before any evaluation is done as follows:

```
model_wrapper = pc.lazy_eval(model_wrapper, load="model_data.d")
```

The `load` argument tells the software to load saved data from disk if the file exists. At the end of any simulation, the result can be saved to disk by performing

```
model_wrapper.save("model_data.d")
```

Now any call to the `model_wrapper` that has been called before will taken from memory instead of being calcuated again.

The aim when performing uncertainty quantification on forward models, is to find the uncertainty on the output `U` given uncertainty in the input `q`. In practice it is sufficient to find descriptive statistics like the mean and variance. And because of the assumption of high computational cost in the solver, it will be assumed that an analytical approach is unfeasible and the number of evaluation should be kept as low as possible.

With the model wrapper and the distribution at hand it is possible to use `chaospy` to create an polynomial approximation using polynomial chaos expansion with various non-intrusive implementation techniques. For example a 2. order pseudo-spectral method using full tensor product Gaussian quadrature rule [63], can be implemented as follows:

```
approx = pc.pcm(model_wrapper, 2, Q)
```

which uses 27 evaluations. To approximation tries to approximate the model solver as a function of the uncertain parameters using polynomials. To illustrate the difference, both the model solver and the approximation is evaluated for a random sample from `Q`:

```
sample = Q.sample()
U1 = model_wrapper(sample)
U2 = approx(*sample)
```

In figure 2 the two are plotted in the same figure. It does well as an approximation between the layers, but at the boundaries it has some discrepancies.

The advantage of using an approximation is that it can be evaluated at a much cheaper cost than the full model solver, and since it is defined using polynomials, most analysis can be done analytically. For example the mean, and 5th and 95th percentile can be calculated as follows:

```
E = pc.E(approx, Q)
q05, q95 = pc.Perc(approx, [5,95], Q)
```

As a point of reference, it is also possible to do the same using Monte Carlo simulation. For example 10^3 samples quasi-Monte Carlo using the Halton sequence[26]:

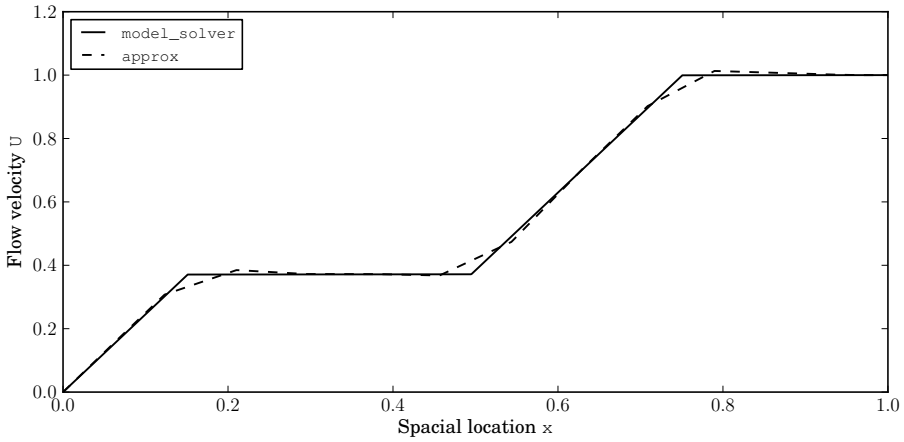

Figure 2: Both the true values from the model solver plotted together with the approximations using polynomial chaos expansion.

```
samples = Q.sample(10**3, "H")
U = [model_wrapper(q) for q in samples.T]
E = mean(U, 0)
q05, q95 = percentile(U, [5, 95], 0)
```

All three statistics for both analysis can be observed in figure 3, where the two

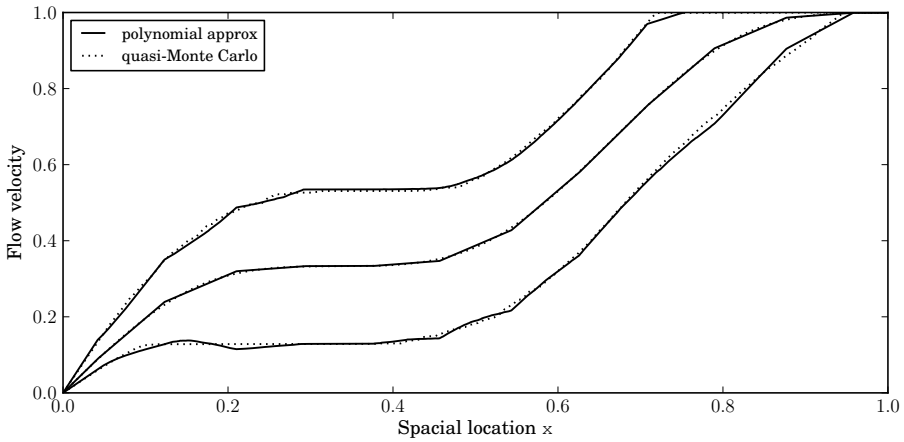

Figure 3: Mean flow plotted inside an $\alpha = 0.9$ confidence interval.

methods give about the same results. The mean represent the expected result

under uncertainty and the percentile represent lower and upper thresholds for the values.

2 Distributions

Dependent variables are complicated

Numerical methods for uncertainty quantification need to generate pseudo-random realizations

$$\{\boldsymbol{Q}_k\}_{k\in I_K} \qquad I_K = \{1,\dots,K\},$$

from the density $p_{\boldsymbol{Q}}$. Each $\boldsymbol{Q} \in \{\boldsymbol{Q}_k\}_{k\in I_K}$ is multivariate with the number of dimensions $D > 1$. Generating realizations from a given density $p_{\boldsymbol{Q}}$ is often non-trivial, at least when D is large. A very common assumption made in uncertainty quantification is that each dimension in $\boldsymbol{Q}$ consists of stochastically independent components. Stochastic independence allows for a joint sampling scheme to be reduced to a series of univariate samplings, drastically reducing the complexity of generating a sample $\boldsymbol{Q}$.

Unfortunately, the assumption of independence does not always hold in practice. We have examples from many research fields where stochastic dependence must be assumed, including modelling of climate [35], iron-ore minerals [7], finance [12], and ion channel densities in detailed neuroscience [1]. There also exists examples where introducing dependent random variables is beneficial for the modelling process, even though the original input was stochastically independent [17]. In any cases, modelling of stochastically dependent variables are required to perform uncertainty quantification adequately. A strong feature of Chaospy is its support for stochastic dependence.

Introducing the Rosenblatt Transformation All random samples are in Chaospy generated using Rosenblatt transformations $T_{\boldsymbol{Q}}$ [50]. It allows for a random variable $\boldsymbol{U}$, generated uniformly on a unit hypercube $[0,1]^D$, to be transformed into $\boldsymbol{Q} = T_{\boldsymbol{Q}}^{-1}(\boldsymbol{U})$, which behaves as if it were drawn from the density $p_{\boldsymbol{Q}}$. It is easy to generate pseudo-random samples from a uniform distribution, and the Rosenblatt transformation can then be used as a method for generating samples from arbitrary densities.

Definition of a Rosenblatt Transformation The Rosenblatt transformation can be derived as follows. Consider a probability decomposition, for example for a bivariate random variable $\boldsymbol{Q} = (Q_0, Q_1)$:

$$p_{Q_0,Q_1}(q_0,q_1) = p_{Q_0}(q_0)p_{Q_1|Q_0}(q_1 \mid q_0), \tag{2}$$

were p_{Q_0} is an marginal density function, and $p_{Q_1|Q_0}$ is a conditional density. For the multivariate case, the density decomposition will have the form

$$p_{\boldsymbol{Q}}(\boldsymbol{q}) = \prod_{d=0}^{D-1} p_{Q'_d}(q'_d), \tag{3}$$

where

$$Q'_d = Q_d \mid Q_0, \ldots, Q_{d-1} \qquad q'_d = q_d \mid q_0, \ldots, q_{d-1} \tag{4}$$

denotes that Q_d and q_d are dependent on all components with lower indices. A forward Rosenblatt transformation can then be defined as

$$T_{\boldsymbol{Q}}(\boldsymbol{q}) = (F_{Q'_0}(q'_0), \ldots, F_{Q'_{D-1}}(q'_{D-1})), \tag{5}$$

where $F_{Q'_d}$ is the cumulative distribution function:

$$F_{Q'_d}(q'_d) = \int_{-\infty}^{q_d} p_{Q'_d}(r \mid q_0, \ldots, q_{d-1}) \, \mathrm{d}r. \tag{6}$$

This transformation is bijective, so it is always possible to define the inverse Rosenblatt transformation $T_{\boldsymbol{Q}}^{-1}$ in a similar fashion.

Numerical Estimation of Inverse Rosenblatt Transformations

To implement the Rosenblatt transformation in practice, we need to identify the inverse transform $T_{\boldsymbol{Q}}^{-1}$. Unfortunately, $T_{\boldsymbol{Q}}$ is non-linear which often is without a closed-form solution, making analytical calculations of the transformation's inverse difficult. In the scenario where we do not have a representation of the inverse transformation, a numerical scheme has to be employed. To the authors' knowledge, there are no standards for defining such a numerical scheme. The following paragraphs therefore describe our proposed method for calculating the inverse transformation numerically.

Approx Method Overview The problem of calculating the inverse transformation $T_{\boldsymbol{Q}}^{-1}$ can, by decomposing the definition of the forward Rosenblatt transformation in (5), be reformulated as

$$F_{Q'_d}^{-1}(u \mid q_0, \ldots, q_{d-1}) = \left\{ r : F_{Q'_d}(r \mid q_0, \ldots, q_{d-1}) = u \right\} \qquad d = 0, \ldots, D-1.$$

In other words, the challenge of calculating the inverse can be reformulated as a series of one dimensional root-finding problems. In Chaospy, these roots are found by employing a Newton-Raphson scheme. However, to also ensure convergence, the scheme is coupled with a bisection method. The bisection method is applicable here since the problem is one-dimensional and the functions of interest are by definition monotone. When the Newton-Raphson method fails to converge at an increment, a bisection step gives the Newton-Raphson a new start location away from the previous location. This algorithm ensures fast and reliable convergence towards the root.

Algorithm Description The Newton-Raphson-bisection hybrid method is implemented as follows. The initial values are the lower and upper bounds $[lo_0, up_0]$. If $p_{Q'_d}$ is unbound, the interval is selected such that it approximately covers the density. For example for a standard normal random variable, which is unbound,

the interval $[-7.5, 7.5]$ will approximately cover the whole density with an error about 10^{-14}. The algorithm starts with a Newton-Raphson increment, using the initial value $r_0 = (up_0 - lo_0)u + lo_0$:

$$r_{k+1} = r_k - \frac{F_{Q'_d}(r_k \mid q_0, \ldots, q_{d-1}) - u}{p_{Q'_d}(r_k \mid q_0, \ldots, q_{d-1})}, \tag{7}$$

where the density $p_{Q'_d}$ can be approximated using finite differences. If the new value does not fall in the interval $[lo_k, up_k]$, this proposed value is rejected, and is instead replaced with a bisection increment:

$$r_{k+1} = \frac{up_k + lo_k}{2}. \tag{8}$$

In either case, the bounds are updated according to

$$(lo_{k+1}, up_{k+1}) = \begin{cases} (lo_k, r_{k+1}) & F_{Q'_d}(r_{k+1} \mid q_0, \ldots, q_{d-1}) > u \\ (r_{k+1}, up_k) & F_{Q'_d}(r_{k+1} \mid q_0, \ldots, q_{d-1}) < u \end{cases}. \tag{9}$$

The algorithm repeats the steps in (7), (8) and (9), until the error $|F_{Q'_d}(r_k \mid q_0, \ldots, q_{d-1}) - u|$ is sufficiently small.

The described algorithm overcomes one of the challenges of implementing Rosenblatt transformations in practice: how to calculate the inverse transformation. Another challenge is how to construct a transformation in the first place. This is the topic of the next section.

2.1 Variable Construction

Overview The backbone of distributions in Chaospy is the Rosenblatt transformation $T_{\boldsymbol{Q}}$. The method, as described in the last section, assumes that $p_{\boldsymbol{Q}}$ is known to be able to perform the transformation and its inverse. In practice, however, we first need to construct $p_{\boldsymbol{Q}}$, before the transformation can be used. This can be a challenging task, but in Chaospy a lot of effort has been put into constructing novel tools for making the process as flexible and painless as possible. In essence, users can create their own custom multivariate distributions using a new methodology as described next.

The Joining of Conditionals Following the definition in (5), each Rosenblatt transformation consists of a collection of conditional distributions. We express all conditionality through distribution parameters. For example, the location parameter of a normal distribution can be set to be uniformly distributed, say on $[-1, 1]$. The following interactive Python code does the task:

```
>>> uniform = Uniform(lo=-1, up=1)
>>> normal = Normal(mu=uniform, sigma=0.1)
```

We now have two stochastic variables, `uniform` and `normal`, whose joint bivariate distribution can be constructed through the `J` function:

```
>>> joint = J(uniform, normal)
```

The software will, from this minimal formulation, try to sort out the dependency ordering and construct the full Rosenblatt transformation. The only requirement is that a decomposition as in (3) is in fact possible. The result is a fully functioning forward and inverse Rosenblatt transformation. The following code evaluates the forward transformation (the density) at $(1, 0.9)$, the inverse transformation at $(0.4, 0.6)$, and draws a random sample from the joint distribution:

```
>>> print joint.fwd([1, 0.9])
[ 1.          0.15865525]
>>> print joint.inv([0.4, 0.6])
[-0.2        -0.17466529]
>>> print joint.sample()
[-0.05992158 -0.07456064]
```

Distributions in higher dimensions are trivially obtained by including more arguments to the `J` function.

Arithmetic Operators As an alternative to the explicit formulation of dependency through distribution parameters, it is also possible to construct dependencies implicitly through arithmetic operators. For example, it is possible to recreate the example above using addition of stochastic variables instead of letting a distribution parameter be stochastic. More precisely, we have a uniform variable on $[-1, 1]$ and a normally distributed variable with location at $x = 0$. Adding the uniform variable to the normal variable creates a new normal variable with stochastic location:

```
>>> uniform = Uniform(lo=-1, up=1)
>>> normal0 = Normal(mu=0, scale=0.1)
>>> normal = normal0 + uniform
>>> joint = J(uniform, normal)
```

As before, the software automatically sorts the dependency ordering from the context. Here, since the uniform variable is present as first argument, the software recognises the second argument as a normal distribution, conditioned on the uniform distribution, and not the other way around.

Multivariate Operators Another favorable feature in Chaospy is that multiple transformations can be stacked on top of each other. For example, consider the example of a multivariate log-normal random variable $\boldsymbol{Q}$ with three dependent components. (Let us ignore for a moment the fact that Chaospy already offers such a distribution.) Trying to decompose this distribution is a very cumbersome task if performed manually. However, this process can be drastically simplified through variable transformations, for which Chaospy has strong support. Many transformations in probability theory are well known, and a log-normal distribution, for example, can be expressed as

$$\boldsymbol{Q} = e^{\boldsymbol{Z}L + \boldsymbol{b}},$$

where $\boldsymbol{Z}$ are standard normal variables, and L and $\boldsymbol{b}$ are predefined matrix and vector, respectively. To implement this particular transformation, we only have to write

```
>>> Z = J(Normal(0,1), Normal(0,1), Normal(0,1))
>>> Q = e**(Z*L + b)
```

The resulting distribution is fully functional multivariate log-normal, assuming L and $\boldsymbol{b}$ are properly defined.

Collection of Univariate Distribution One obvious prerequisite for using univariate distributions to create conditionals and multivariate distributions, is the availability of univariate distributions. Since the univariate distribution is the fundamental building block, Chaospy offers a large collection of 64 univariate distributions. They are all listed in Table 1. The list also shows that Dakota's support is limited to 11 distributions, and Turns has a collection of 26 distributions.

User Defined Constructor The Chaospy software supports in addition custom distributions through the function `constructor`. To illustrate its use, consider the simple example of a uniform random variable on the interval $[lo, up]$. The minimal input to create such a distribution is

```
>>> Uniform = constructor(
...     cdf=lambda self,x,lo,up: (x-lo)/(up-lo),
...     bnd=lambda self,x,lo,up: (lo,up) )
>>> uniform = Uniform(lo=-1, up=1)
```

Here, the two provided arguments are a cumulative distribution function (`cdf`), and a boundary interval function (`bnd`), respectively. The `constructor` function also takes several optional arguments to provide extra functionality. For example, the inverse of the cumulative distribution function – the point percentile function – can be provided through the `ppf` keyword. If this function is not provided, the software will automatically approximate it using the method described in Section 2.

Copulas

Dakota and Turns does not support the Rosenblatt transformation applied to multivariate distributions with dependencies. Instead, the two packages model dependencies using copulas [45]. A copula consists of stochastically independent multivariate distributions made dependent using a parameterized function g. Since the Rosenblatt transformation is general purpose, it is possible to construct any copula directly. However, this can quickly become a very cumbersome task since each copula must be decomposed individually for each combination of independent distributions and parameterization of g. To simplify the user's efforts, Chaospy have dedicated constructors that can reformulate a copula coupling into a Rosenblatt transformation. This is done following the work of Lee [36] and approximated using finite differences. In practice, this allow Copulas

Distribution	D	T	C	**Distribution**	D	T	C
Alpha	n	n	y	Anglit	n	n	y
Arcsinus	n	n	y	Beta	y	y	y
Brandford	n	n	y	Burr	n	y	y
Cauchy	n	n	y	Chi	n	y	y
Chi-Square	n	y	y	Double Gamma	n	n	y
Double Weibull	n	n	y	Epanechnikov	n	y	y
Erlang	n	n	y	Exponential	y	y	y
Exponential Power	n	n	y	Exponential Weibull	n	n	y
Birnbaum-Sanders	n	n	y	Fisher-Snedecor	n	y	y
Fisk/Log-Logistic	n	n	y	Folded Cauchy	n	n	y
Folded Normal	n	n	y	Frechet	y	n	y
Gamma	y	y	y	Gen. Exponential	n	n	y
Gen. Extreme Value	n	n	y	Gen. Gamma	n	n	y
Gen. Half-Logistic	n	n	y	Gilbrat	n	n	y
Truncated Gumbel	n	n	y	Gumbel	y	y	y
Hypergeometric Secant	n	n	y	Inverse-Normal	n	y	n
Kumaraswamy	n	n	y	Laplace	n	y	y
Levy	n	n	y	Log-Gamma	n	n	y
Log-Laplace	n	n	y	Log-Normal	y	y	y
Log-Uniform	y	y	y	Logistic	n	y	y
Lomax	n	n	y	Maxwell	n	n	y
Mielke's Beta-Kappa	n	n	y	Nakagami	n	n	y
Non-Central Chi-Squared	n	y	y	Non-Central Student-T	n	y	y
Non-central F	n	n	y	Normal	y	y	y
Pareto (First kind)	n	n	y	Power Log-Normal	n	n	y
Power Normal	n	n	y	Raised Cosine	n	n	y
Rayleigh	n	y	y	Reciprocal	n	n	y
Rice	n	y	n	Right-skewed Gumbel	n	n	y
Student-T	n	y	y	Trapezoidal	n	y	n
Triangle	y	y	y	Truncated Exponential	n	n	y
Truncated Normal	n	y	y	Tukey-Lamdba	n	n	y
Uniform	y	y	y	Wald	n	n	y
Weibull	y	y	y	Wigner	n	n	y
Wrapped Cauchy	n	n	y	Zipf-Mandelbrot	n	y	n

Table 1: List of supported continuous distributions in software. The titles 'D', 'T' and 'C' represents Dakota, Turns and Chaospy respectively. The elements 'y' and 'n' represents the answers 'yes' and 'no' indicated if the distribution is supported or not.

to be define in a Rosenblatt transformation setting. For example, to construct a bivariate normal distribution with a Clayton copula in Chaospy, we do the following:

```
>>> joint = J(Normal(0,1), Normal(0,1))
>>> clayton = Clayton(joint, theta=2)
```

A list of supported copulas are listed in Table 2. It shows that Turns supports 7 methods, Chaospy 6, while Dakota offers 1 method.

Supported Copulas	Dakota	Turns	Chaospy
Ali-Mikhail-Haq	no	yes	yes
Clayton	no	yes	yes
Farlie-Gumbel-Morgenstein	no	yes	no
Frank	no	yes	yes
Gumbel	no	yes	yes
Joe	no	no	yes
Minimum	no	yes	no
Normal/Nataf	yes	yes	yes

Table 2: The list of supported copulas in the various software packages.

2.2 Statistical Moments

In section 2 the different method required to create a user defined distribution through the `pc.construct` function. One of the optional function that can be provided is `mom`. It represents raw statistical moment where the first argument represents the exponent of the variable. For example, the first three moments of a standard uniform random variable would be:

```
dist = pc.Uniform(0,1)
print dist.mom([0,1,2])
#  [ 1.          0.5         0.33333333]
```

The moment function that creates these numbers have the following form:

```
def mom(self, k):
    return 1./(1+k)
dist.addattr(mom=mom)
```

For simple arithmetical operators as addition and multiplication, moment function is transitive, meaning that the moment is calculated analytically for these operators as long as the underlying variables are also calculated analytically. For other transformations, like exponent, division, etc. the possibility of performing analytical moments becomes more difficult. If that is the case, the moments will be estimated numerically by recognising the moments as

$$\mu_{\boldsymbol{k}} = \mathbb{E}\left[\prod_{n=0}^{N-1} q_n^{k_n}\right] = \int \cdots \int p_{\boldsymbol{q}}(\boldsymbol{q}) \prod_{n=0}^{N-1} q_n^{k_n} \, \mathrm{d}q_0 \cdots \mathrm{d}q_{N-1}.$$

If the process of calculating the raw moments analytically fails, the software automatically switches to quadrature generation. In many cases, letting the software choosing the integration rule is just fine. However, there are also those cases where an custom rule might make more sense. In those cases, it possible to pass extra keyword arguments to `mom` which will be passed to `quadgen` internally. To illustrate this, consider the following custom distribution:

```
def pdf(self, x):
    out = 9/16.*(x+1)*(x<1/3.)
    out += 3/4.*(x>=1/3.)
    return out

def cdf(self, x):
    out = 9/32.*(x+1)**2*(x<1/3.)
    out += (3*x+1)/4.*(x>=1/3.)
    return out

def bnd(self):
    return -1,1

MyDist = pc.construct(
    pdf=pdf, cdf=cdf, bnd=bnd)
```

It is a trapezoid distribution with corner parameters $(-1, 1/3, 1, 1)$. This gives the density a discontinuity in the derivative at `x=1/3`. This is a problem when using quadrature rules for estimating the moments. For example using order 100 to estimate the mean would give us:

```
dist = MyDist()
print dist.mom(1, order=100)
#  0.277778240534
```

However if we simple set a composite rule with a midpoint in `1/3` we get much higher accuracy:

```
print dist.mom(1, order=5, composite=1/3.)
#  0.277777777778
```

The analytical mean is $0.2\overline{7}$, making the latter much more accurate.

Having the ability to pass arguments to `quadgen` through `mom` is useful, but it requires the user to remember the arguments needed for the problem at hand. As alternative it is also possible to save the moment estimation using the `momgen` function. It is a generator that creates and buffers quadrature nodes and weights. When called it uses this nodes and weights to estimate the moments. It is design to fit as the `mom` function provided to `pc.construct`. For our example above `momgen` could be implemented as follows:

```
mom = pc.momgen(
    order=100,
    domain=dist,
    composite=1/3.)
```

```
dist.addattr(mom=mom)
print dist.mom([1,2,3])
#   [ 0.27777778  0.2962963   0.15925926]
```

2.3 Variance Reduction Techniques

Why Use Variance Reduction Techniques? As noted in the beginning of Section 2, by generating samples $\{\boldsymbol{Q}_k\}_{k\in I_K}$ and evaluating the response function f, it is possible to draw inference upon Y without knowledge about p_Y, through Monte Carlo simulation. Unfortunately, the number of samples K to achieve reasonable accuracy can often be high. If f is assumed to be computationally expensive, the number of samples needed, might make Monte Carlo simulation infeasible to apply in practice. As a way to mitigate this problem, it is possible to modify $\{\boldsymbol{Q}_k\}_{k\in I_K}$ from traditional pseudo-random samples, so that the accuracy increases. Schemes that select non-traditional samples for $\{\boldsymbol{Q}_k\}_{k\in I_K}$ to increase accuracy are known as *variance reduction techniques*. A list of such techniques are presented in Table 3, and it shows that Dakota, Turns and Chaospy support 4, 7, and 7 variance reduction techniques, respectively.

Introducing Quasi-Monte Carlo Simulation One of the more popular variance reduction technique is the *quasi-Monte Carlo scheme* [51]. The method consists of selecting the samples $\{\boldsymbol{Q}_k\}_{k\in I_K}$ to be a low-discrepancy sequence instead of pseudo-random samples. The idea is that samples placed with a given distance from each other increase the coverage over the sample space, requiring fewer samples to reach a given accuracy. For example, if the latter requires 10^6 samples for a given accuracy, quasi-Monte Carlo can often get away with only 10^3. Note that this would break som of the statistical properties of the samples [57].

Where Quasi-Monte Carlo can be used Most of the theory on quasi-Monte Carlo methods focuses on generating samples on the unit hypercube $[0,1]^N$. The option to generate samples directly on to other distributions exists, but is often very limited. To the authors' knowledge, the only viable method for including most quasi-Monte Carlo methods into the vast majority of non-standard probability distributions, is through the Rosenblatt transformation. Since Chaospy is built around the Rosenblatt transformation, it has the novel feature of supporting quasi-Monte Carlo methods for all probability distributions. Turns and Dakota only support Rosenblatt transformations for independent variables and the Normal copula.

Sometimes the quasi-Monte Carlo method is infeasible because the forward model is too computationally costly. The next section describes polynomial chaos expansions, which often require far fewer samples than the quasi-Monte Carlo method for the same amount of accuracy.

Generating random samples in the software can be done through using the `sample` instance method. Many of the variance reduction techniques are automated and can be evoked through a simple string as second argument. For example to generate 5 samples from Halton sequence for a bivariate independent normal distribution:

Quasi-Monte Carlo Scheme	Dakota	Turns	Chaospy
Faure sequence [19]	no	yes	no
Halton sequence [26]	yes	yes	yes
Hammersley sequence [27]	yes	yes	yes
Haselgrove sequence [28]	no	yes	no
Korobov latice [33]	no	no	yes
Niederreiter sequence [46]	no	yes	no
Sobol sequence [54]	no	yes	yes
Other Methods	Dakota	Turns	Chaospy
Antithetic variables [51]	no	no	yes
Importance sampling [51]	yes	yes	yes
Latin Hypercube sampling [41]	yes	limited	yes

Table 3: The different sampling schemes available.

```
dist = pc.Iid(pc.Normal(), 2)
print dist.sample(5, "H")
#  [[-1.15034938  0.31863936 -0.31863936  1.15034938 -1.53412054]
#   [-0.1397103   0.76470967 -0.76470967  0.1397103   1.22064035]]
```

All the variance reduction techniques can be evoked through the second argument with one exception: Antithetic variate. The variate is a method for inducing negative correlation between the samples on the unit hypercube by also including $1 - U$ for every U. The idea is that the negative correlation is maintained through the Rosenblatt transformation. Unfortunately this does not necessarily hold true; The correlation might be positive after a transformation. And if that is the case, the variance will be increased instead of reduced. When creating an antithetic variate there is usually more than one axis to mirror. However, which axes to mirror, if there exists, has to be investigated. To evoke antithetic variate, `sample` has an `antithetic` keyword argument. It takes an array of boolean values, where a true value represents an axes to mirror. For example our bivariate normal distribution:

```
dist = pc.Iid(pc.Uniform(0,1), 2)
X = dist.sample(5, antithetic=[True, False])
print X
#  [[ 0.65358959  0.11500694  0.95028286  0.34641041  0.88499306]
#   [ 0.5178086   0.12752546  0.78766732  0.5178086   0.12752546]]
print 1-X
#  [[ 0.34641041  0.88499306  0.04971714  0.65358959  0.11500694]
#   [ 0.4821914   0.87247454  0.21233268  0.4821914   0.87247454]]
```

The antithetic variate is here along the first axis. It can be observed in that the first two values in the first axis of `X` can be observed as the last two values in `1-X`. Likewise, since the second axis is not mirrored, the two first and two last values are identical.

3 Polynomial Chaos Expansions

Introducing Polynomial Chaos Expansions Polynomial chaos expansions represent a collection of methods that can be considered a subset of polynomial approximation methods, but designed in particular for uncertainty quantification. A general polynomial approximation can be defined as

$$\hat{f}(\boldsymbol{x}, t, \boldsymbol{Q}) = \sum_{n \in I_N} c_n(\boldsymbol{x}, t) \boldsymbol{\Phi}_n(\boldsymbol{Q}) \qquad I_N = \{0, \ldots, N\}, \tag{10}$$

where $\{c_n\}_{n \in I_N}$ are coefficients (often known as Fourier coefficients) and $\{\boldsymbol{\Phi}_n\}_{n \in I_N}$ are polynomials. If $\hat{f}$ is a good approximation of f, it is possible to either infer statistical properties of $\hat{f}$ analytically or through cheap numerical computations where $\hat{f}$ is used as a surrogate for f.

A polynomial chaos expansion is defined as a polynomial approximation, as in (10), where the polynomials $\{\boldsymbol{\Phi}_n\}_{n \in I_N}$ are orthogonal on a custom weighted function space L_Q:

$$\langle \boldsymbol{\Phi}_n, \boldsymbol{\Phi}_m \rangle = \mathbb{E}[\boldsymbol{\Phi}_n(\boldsymbol{Q}) \boldsymbol{\Phi}_m(\boldsymbol{Q})] = \int \cdots \int \boldsymbol{\Phi}_n(\boldsymbol{q}) \boldsymbol{\Phi}_m(\boldsymbol{q}) p_{\boldsymbol{Q}}(\boldsymbol{q}) \, \mathrm{d}\boldsymbol{q} = 0 \quad n \neq m. \tag{11}$$

As a side note, it is worth noting that in parallel with polynomial chaos expansions, there also exists an alternative collocation method based on multivariate Lagrange polynomials [64]. This method is supported by Dakota and Chaospy, but not Turns.

Section Overview To generate a polynomial chaos expansion, we must first calculate the polynomials $\{\boldsymbol{\Phi}_n\}_{n \in I_N}$ such that the orthogonality property in (11) is satisfied. This will be the topic of Section 3.1 In Section 4 we show how to estimate the Fourier coefficients $\{c_n\}_{n \in I_N}$. Last, in Section 4.4, tools used to quantify uncertainty in polynomial chaos expansions will be discussed.

3.1 Orthogonal Polynomial Construction

For Each distribution, an Expansion From (11) it follows that the orthogonality property is not in general transferable between distributions, since a new set of polynomials has to be constructed for each $p_{\boldsymbol{Q}}$. The easiest approach to construct orthogonal polynomials is to identify the probability density $p_{\boldsymbol{Q}}$ in the so-called Askey-Wilson scheme [3]. The polynomials can then be picked from a list, or be built from analytical components. The continuous distributions supported in the scheme include the standard normal, gamma, beta, and uniform distributions respectively through the Hermite, Laguerre, Jacobi, and Legendre polynomial expansion. All the various software toolboxes support these expansions. However, it is worth noting that Chaospy in addition supports log-normal distribution through the Stieltjes-Wigert polynomials [59]. The log-normal distribution is a popular distribution with applications in engineering as well as in medicine [38, 58] and economy [6, 10, 40].

Expansion Building is Unstable Moving beyond the standard collection of the Askey-Wilson scheme, it is possible to create custom orthogonal polynomials, both analytically and numerically. Unfortunately, most methods involving finite precision floating-point numbers, are ill-posed, making a numerical approach quite a challenge [20]. This section explores the various approaches for being able to construct polynomial expansions. A full list of methods are listed in Table 4. It shows that Dakota, Turns and Chaospy support 4, 3 and 5 orthogonalisation methods, respectively.

Orthogonalization Method	Dakota	Turns	Chaospy
Askey-Wilson Scheme [3]	yes	yes	yes
Bertran recursion [5]	no	no	yes
Cholesky Decomposition [17]	no	no	yes
Discretized Stieltjes [21]	yes	no	yes
Modified Chebyshev [21]	yes	yes	no
Modified Gram-Schmidt [21]	yes	yes	yes

Table 4: Methods for generating expansions of orthogonal polynomials.

Independent Components Simplify Everything Looking beyond an analytical approach, the most popular method for constructing orthogonal polynomials is the discretized Stieltjes procedure [55]. As far as the authors know, it is the only truly numerically stable method for orthogonal polynomial construction. It is based upon one-dimensional recursion coefficients that are estimated using numerical integration. Unfortunately, the method is only applicable in the multivariate case if the components of $p_{\boldsymbol{Q}}$ are stochastically independent.

Generalized Polynomial Chaos Expansions One approach to model densities with stochastically dependent components numerically, is to reformulate the uncertainty problem as a set of independent components through generalised polynomial chaos expansion [65]. As described in detail in Section 2, a Rosenblatt transformation allows for the mapping between any domain and the unit hypercube $[0,1]^D$. With a double transformation we can reformulate the response function f as

$$f(\boldsymbol{x},t,\boldsymbol{Q}) = f(\boldsymbol{x},t,T_{\boldsymbol{Q}}^{-1}(T_{\boldsymbol{R}}(\boldsymbol{R}))) \approx \hat{f}(\boldsymbol{x},t,\boldsymbol{R}) = \sum_{n\in I_N} c_n(\boldsymbol{x},t)\boldsymbol{\Phi}_n(\boldsymbol{R}),$$

where $\boldsymbol{R}$ is any random variable drawn from $p_{\boldsymbol{R}}$, which for simplicity is chosen to consists of independent components. Also, $\{\boldsymbol{\Phi}_n\}_{n\in I_N}$ is constructed to be orthogonal with respect to $L_{\boldsymbol{R}}$, not $L_{\boldsymbol{Q}}$. In any case, $\boldsymbol{R}$ is either selected from the Askey-Wilson scheme, or calculated using the discretized Stieltjes procedure. We remark that the accuracy of the approximation deteriorate if the transformation composition $T_{\boldsymbol{Q}}^{-1} \circ T_{\boldsymbol{R}}$ is not smooth [65]. Dakota, Turns, and Chaospy all support generalized polynomial chaos expansions for independent stochastic variables and the Normal/Nataf copula listed in Table 2. Since Chaospy has

the Rosenblatt transformation in the bottom of the computational framework, generalized polynomial chaos expansions are in fact available for all densities.

The Direct Multivariate Approach. Given that both the density $p_{\boldsymbol{Q}}$ has stochastically dependent components, and the transformation composition $T_{\boldsymbol{Q}}^{-1} \circ T_{\boldsymbol{R}}$ is not smooth, it is still possible to generate orthogonal polynomials numerically. As noted above, most methods are numerically unstable, and the accuracy in the orthogonality can deteriorate with polynomial order, but the methods can still be useful [17]. In Table 4, only Chaospy's implementation of Bertran's recursion method [5], Cholesky decomposition [16] and modified Gram-Schmidt orthogonalization [21] support construction of orthogonal polynomials for multivariate dependent densities directly.

Custom Polynomial Expansion

Custom Polynomial Expansions In the most extreme cases, an automated numerical method is insufficient. Instead, a polynomial expansion has to be constructed manually. User-defined expansions can be created conveniently, as demonstrated in the next example involving a second-order Hermite polynomial expansion, orthogonal with respect to the normal density [3]:

$$\{\mathbf{\Phi}_n\}_{n \in I_6} = \left\{1, Q_0, Q_1, Q_0^2 - 1, Q_0 Q_1, Q_1^2 - 1\right\}$$

The relevant Chaospy code for creating this polynomial expansion looks like

```
>>> q0, q1 = variable(2)
>>> phi = Poly([1, q0, q1, q0**2-1, q0*q1, q1**2-1])
>>> print phi
[1, q0, q1, q0^2-1, q0q1, -1+q1^2]
```

Chaospy contains a collection of tools to manipulate and create polynomials, see Table 5.

Descriptive Tools Chaospy includes operators such as the expectation operator $\mathbb{E}$. This is a helpful tool to ensure that the constructed polynomials are orthogonal, as defined in (11). To verify that two elements in `phi` are indeed orthogonal under the standard bivariate normal distribution, one writes

```
>>> dist = J(Normal(0,1), Normal(0,1))
>>> print E(phi[3]*phi[5], dist)
0.0
```

More details of operators used to perform uncertainty analysis will be given in Section 4.4.

3.2 Manual Polynomial Construction

As the name suggestion, one of the components in polynomial chaos expansion is the use of polynomials. This section will introduce how polynomials can be created and manipulated in `chaospy`.

Function	Description
`all`	Test all coefficients for non-zero
`any`	Test any coefficients for non-zero
`around`	Round to a given decimal
`asfloat`	Set coefficients type as float
`asint`	Set coefficient type as int
`basis`	Create monomial basis
`cumprod`	Cumulative product
`cumsum`	Cumulative sum
`cutoff`	Truncate polynomial order
`decompose`	Convert from series to sequence
`diag`	Construct or extract diagonal
`differential`	Differential operator
`dot`	Dot-product
`flatten`	Flatten an array
`gradient`	Gradient (or Jacobian) operator
`hessian`	Hessian operator
`inner`	Inner product
`mean`	Average
`order`	Extract polynomial order
`outer`	Outer product
`prod`	Product
`repeat`	Repeat polynomials
`reshape`	Reshape axes
`roll`	Roll polynomials
`rollaxis`	Roll axis
`rolldim`	Roll the dimension
`std`	Empirical standard deviation
`substitute`	Variable substitution
`sum`	Sum along an axis
`swapaxes`	Interchange two axes
`swapdim`	Swap the dimensions
`trace`	Sum along the diagonal
`transpose`	Transpose the coefficients
`tril`	Extract lower triangle of coefficients
`tricu`	Extract cross-diagonal upper triangle
`var`	Empirical variance
`variable`	Simple polynomial constructor

Table 5: List of tools for creating and manipulating polynomials.

To start at a simplest level, a simple polynomial can be created through `variable` constructor. For example to construct a simple bivariate polynomial:

```
x,y = pc.variable(2)
print x
# q0
```

A collection of polynomial can be manipulated using basic arithmetic operators and joined together into polynomial expansions:

```
polys = pc.Poly([1, x, x*y])
print polys
#  [1, q0, q0q1]
```

Note that constants and simple polynomials can be joined together into arrays without any problems.

In practice, having the ability to fine tune a polynomial exactly as one wants it can be useful, but it can also be cumbersome when dealing with larger arrays for application. To automate the construction of simple polynomials, there is the `basis` constructor. In its simplest forms it creates an array of simple monomials:

```
print pc.basis(4)
#  [1, q0, q0^2, q0^3, q0^4]
```

It can be expanded to include number of dimensions and a lower bound for the polynomial order:

```
print pc.basis(1, 2, dim=2)
#  [q0, q1, q0^2, q0q1, q1^2]
```

There is also the possibility to create anisotropic expansions:

```
print pc.basis(1, [1, 2])
#  [q0, q1, q0q1, q1^2, q0q1^2]
```

and the possibility to fine tune the sorting of the polynomials:

```
print pc.basis(1, 2, dim=2, sort="GRI")
#  [q0^2, q0q1, q1^2, q0, q1]
```

To manipulate a polynomial there is a collection of tools available. In table 5 they are listed with description. Much of the behavior is overlapping with the numerical library `numpy`. The functions that overlaps in name is backwards compatible, e.g. if anything else than a polynomial is inserted into `pc.sum`, the argument is passed to `numpy.sum`.

Any constructed polynomial is a callable. The argument can either be inserted positional or as keyword arguments `q0`, `q1`,...

```
poly = pc.Poly([1, x**2, x*y])
print poly(2, 3)
#  [1 2 6]
print poly(q1=3, q0=2)
#  [1 2 6]
```

The input can be a mix of scalars and arrays, as long as the shapes together can be joined to gether in a common compatible shape:

```
print poly(2, [1,2,3,4])
#  [[1 1 1 1]
#   [4 4 4 4]
#   [2 4 6 8]]
```

It is also possible to perform partial evaluation, i.e. evaluating some of the dimensions. To tell the polynomial that a dimension should not be evaluated either leave the argument empty or pass a masked value `numpy.ma.masked`. For example:

```
print poly(2)
#  [1, 4, 2q1]
print poly(ma.masked, 2)
#  [1, q0^2, 2q0]
print poly(q1=2)
#  [1, q0^2, 2q0]
```

The type of return value for the polynomial is `numpy.ndarray` if all dimensions are filled. If not, it returns a new polynomial.

In addition to input scalars, arrays and masked values, it is also possible to pass simple polynomials as argument. This allows for variable substitution. For example, to swap two dimensions, one could simply do the following:

```
print poly(y, x)
#  [1, q1^2, q0q1]
```

It is also possible to do all the over mentioned methods together in the same time. For example, partial evaluation and variable substitution:

```
print poly(q1=y**3-1)
#  [1, q0^2, q0q1^3-q0]
```

3.3 Polynomial Expansions Constrution

In equation (11) the inner product and norm are defined through three equivalent expressions: through inner product $\langle \cdot, \cdot \rangle$ and norm $\|\cdot\|$, through the expected value operator $\mathbb{E}[\cdot]$, and through the more explicit integral form $\int \cdot p_{\boldsymbol{Q}} \, \mathrm{d}\boldsymbol{q}$. In `chaospy` the inner product is formulated through the second one `E`. It is implemented as follows. It takes two positional arguments: a polynomial and a distribution respectively. For example use of the expected value:

```
q = pc.variable()
dist = pc.Normal()
print pc.E(q, dist)
#  0.0
```

The expected value operator, as a reformulation as the inner product in (11) is all that is required to verify that any two polynomials are orthogonal as in the equation. In addition there exists various methods for constructing orthogonal polynomial expansion. One of the simpler examples of this is the Gram-Schmidt orthogonalization procedure [61]. Here is an example implementation in the software:

```
dist = pc.Normal()
orths = []
for poly in pc.basis(0, 4, dim=1):

    for orth in orths:

        coef = pc.E(orth*poly, dist)/pc.E(orth**2, dist)
        poly = poly - orth*coef

    orths.append(poly)

orths = pc.Poly(orths)
print orths
#  [1.0, q0, q0^2-1.0, q0^3-3.0q0, q0^4-6.0q0^2+3.0]
```

The resulting polynomial expansion is the Hermite polynomials. To verify that a polynomial expansion is orthogonal, each polynomial in the expansion can be pairwise checked. However because of the tools available for polynomials in the software, it is possible to perform matrix operations and do such a check with much less effort. For example using the outer product rule:

```
orths2 = pc.outer(orths, orths)
print pc.E(orths2, dist)
#  [[  1.   0.   0.   0.   0.]
#   [  0.   1.   0.   0.   0.]
#   [  0.   0.   2.   0.   0.]
#   [  0.   0.   0.   6.   0.]
#   [  0.   0.   0.   0.  24.]]
```

Each element in the resulting matrix represents equation (11) for various n and m. The equation assumes all elements to be zero for all $n \neq m$, which is reflected in the off diagonal elements.

Analogous to polynomials that can be constructed automatically, there are also constructors for creating orthogonal polynomials. There are four available listed in table 4. For example using Bertran's recursive formula [5] is implemented as follows:

```
dist = pc.Gamma(2)
print pc.orth_bert(2, dist)
#  [1.0, q0-2.0, q0^2-6.0q0+6.0]
```

Note that all of the methods support the same sorting arguments as `basis` described in the beginning of the section and quadrature argument passed to `quadgen` as described in section 4.1, if moments are estimated. The latter also holds true for the expected value operator `E`.

3.4 Three Terms Recursion

Creating an orthogonal polynomial expansion can be numerical unstable when using raw statistical moments as input [20]. This can be a problem if constructing large expansions since the error blows up. Given that the distribution is univariate it is instead possible to create orthogonal polynomials stabilized using the three terms recursion relation:

$$\Phi_{n+1}(q) = \Phi_n(q)(q - A_n) - \Phi_{n-1}(q)B_n, \tag{12}$$

where

$$A_n = \frac{\langle q\mathbf{\Phi}_n, \mathbf{\Phi}_n\rangle}{\langle \mathbf{\Phi}_n, \mathbf{\Phi}_n\rangle} = \frac{\mathbb{E}\big[q\Phi_n^2\big]}{\mathbb{E}[\Phi_n^2]} \qquad B_n = \frac{\langle \mathbf{\Phi}_n, \mathbf{\Phi}_n\rangle}{\langle \mathbf{\Phi}_{n-1}, \mathbf{\Phi}_{n-1}\rangle} = \frac{\mathbb{E}\big[\Phi_n^2\big]}{\mathbb{E}\big[\Phi_{n-1}^2\big]} \tag{13}$$

A multivariate polynomial expansion can be created using tensor product rule of univariate polynomials expansions. This assumes that the distribution is stochastically independent.

In the `chaospy` toolbox three terms recursion coefficient can be generating by calling the `ttr` instance method:

```
dist = pc.Uniform(-1,1)
print dist.ttr([0,1,2,3])
#  [[ 0.          0.          0.          0.        ]
#   [-0.          0.33333333  0.26666667  0.25714286]]
```

Looking back to section 2 and the `pc.contruct` function, `ttr` can be added as a keyword argument. So tailored recursion coefficients can be added for user defined distributions.

If the `ttr` function is missing, which can often the case, the coefficients can be calculated using discretized Stieltjes method [20]. The method consists of alternating between calculating expression (12) and expression (13) given starting values $\Phi_{-1} = 0$ and $\Phi_0 = 1$. Since the expected value operator is used, this method can also be considered as a statistical moment based method, however the coefficients calculations in equation (13) can be estimated using numerical integration, and made stable. In `chaospy` if the `ttr` is missing, it is estimated using `quadgen` with Clenshaw-Curtis nodes and weights. The default is order 40, however, as with all the other instance methods so far, it is possible to set the wanted parameters using keyword argument. In this case the keyword argument `acc` can be used to change the default. In section 2.2 the `momgen` function was introduced. Analogous there is also a `ttrgen` function that does the same, but for the `ttr`. In other words, it is possible to fix the parameters in the estimation of `ttr` in any distribution. Note that the keyword `rule="G"` is

disabled since the Golub-Welsch algorithm also depends upon the three terms recursion coefficients for it's calculations [24].

Multivariate orthogonal polynomial expansions are created by multiplying univariate polynomials together:

$$\boldsymbol{\Phi}_n = \Phi_{1,n_1} \cdots \Phi_{N,n_N},$$

where Φ_{i,n_i} represents the n_i-th polynomial in the univariate expansion orthogonal with respect to the i-th component of $\boldsymbol{Q}$. For the orthogonality to hold, it must be assumed that $p_{\boldsymbol{Q}}$ is stochastically independent. This to assure the third equality in

$$\begin{aligned}\langle \boldsymbol{\Phi}_n, \boldsymbol{\Phi}_m \rangle &= \mathbb{E}[\boldsymbol{\Phi}_n \boldsymbol{\Phi}_m] = \mathbb{E}[\Phi_{1,n_1}\Phi_{i,m_1} \cdots \Phi_{N,n_N}\Phi_{N,m_N}] \\ &= \mathbb{E}[\Phi_{1,n_1}\Phi_{1,m_1}] \cdots \mathbb{E}[\Phi_{N,n_N}\Phi_{N,m_N}] \\ &= \langle \Phi_{1,n_1}, \Phi_{1,m_1} \rangle \cdots \langle \Phi_{N,n_N}, \Phi_{N,m_N} \rangle .\end{aligned}$$

Since each univariate polynomial expansion is orthogonal, this implies that the multivariate also is orthogonal.

In `chaospy` constructing orthogonal polynomial using the three term recursion scheme can be done through `orth_ttr`. For example:

```
dist = pc.Iid(pc.Gamma(1), 2)
orths = pc.orth_ttr(2, dist)
print orths
#  [1.0, q1-1.0, q0-1.0, q1^2-4.0q1+2.0, q0q1-q1-q0+1.0, q0^2-4.0q0+2.0]
```

The method will use the `ttr` function if available, and discretized Stieltjes otherwise.

4 Estimating Fourier Coefficients

A Few Words About Intrusive Galerkin There are several methodologies for estimating Fourier coefficients $\{c_n\}_{n \in I_N}$, typically categorized as non-intrusive and intrusive, where non-intrusive means that the computational procedures can evaluate f as a black box. Intrusive methods need to incorporate information about the underlying forward model in the computation of the coefficients. In case of forward models based on differential equations, one performs a Galerkin formulation for the coefficients in probability space, leading effectively to a D-dimensional differential equation problem in this space [23]. Back et al. [4] demonstrated that the computational cost of such an intrusive Galerkin method in some cases was higher than some non-intrusive methods. None of the three toolboxes discussed in this paper have support for intrusive methods.

Pseudo-spectral vs. Point Collocation Within the realm of non-intrusive methods, there are in principle two viable methodologies available: pseudo-spectral projection [25] and the point collocation method [30]. The former applies a numerical integration scheme to estimate Fourier coefficients, while the latter solves a linear system arising from a statistical regression formulation.

Dakota and Chaospy support both methodologies, while Turns only supports point collocation. We shall now discuss the practical, generic implementation of these two methods in Chaospy.

Integration Methods

The pseudo-spectral projection method is based on a standard least squares minimization in the weighted function space $L_{\boldsymbol{Q}}$. Since the polynomials are orthogonal in this space, the associated linear system is diagonal, which allows a closed-form expression for the Fourier coefficients. The expression involves high-dimensional integrals in $L_{\boldsymbol{Q}}$. Numerical integration is then required,

$$
\begin{aligned}
c_n &= \frac{\mathbb{E}[Y\boldsymbol{\Phi}_n]}{\mathbb{E}[\boldsymbol{\Phi}_n^2]} = \frac{1}{\mathbb{E}[\boldsymbol{\Phi}_n^2]} \int \cdots \int p_{\boldsymbol{Q}}(\boldsymbol{q}) f(\boldsymbol{x}, t, \boldsymbol{q}) \boldsymbol{\Phi}_n(\boldsymbol{q}) \, \mathrm{d}\boldsymbol{q} \qquad (14) \\
&\approx \frac{1}{\mathbb{E}[\boldsymbol{\Phi}_n^2]} \sum_{k \in I_K} w_k p_{\boldsymbol{Q}}(\boldsymbol{q}_k) f(\boldsymbol{x}, t, \boldsymbol{q}_k) \boldsymbol{\Phi}_n(\boldsymbol{q}_k) \qquad I_K = \{0, \ldots, K-1\},
\end{aligned}
$$

where w_k are weights and $\boldsymbol{q}_k$ nodes in a quadrature scheme. Note that f is only evaluated for the nodes $\boldsymbol{q}_k$, and these evaluations can be made once. Thereafter, one can experiment with the polynomial order since any c_n depends on the same evaluations of f.

Software Compares Table 6 shows the various quadrature schemes offered by Dakota and Chaospy (recall that Turns does not support pseudo-spectral projection). All techniques for generating nodes and weights in Chaospy are available through the `generate_quadrature` function. Suppose we want to generate optimal Gaussian quadrature nodes for the normal distribution. We then write

```
>>> nodes, weights = cp.generate_quadrature(3, Normal(0, 1), rule="Gaussian")
>>> print nodes
[[-2.33441422 -0.74196378  0.74196378  2.33441422]]
>>> print weights
[ 0.04587585  0.45412415  0.45412415  0.04587585]
```

Multivariate Integration Scheme Most quadrature schemes are designed for univariate problems. To extend a univariate scheme to the multivariate case, integration rules along each axis can be combined using a tensor product. Unfortunately, such a product suffers from the curse of dimensionality and becomes a very costly integration procedure for large D. In higher-dimensional problems one can replace the full tensor product by a Smolyak sparse grid [53]. The method works by taking multiple lower order tensor product rules and joining them together. If the rule is nested, i.e., the same samples found at a low order are also included at higher order, the number of evaluations can be further reduced. Another feature is to add anisotropy such that some dimensions are sampled more than others [8]. In addition to the tensor product rules, there are

Node and Weight Generators	Dakota	Turns	Chaospy
Clenshaw-Curtis quadrature [11]	yes	no	yes
Cubature rules [56]	yes	no	no
Gauss-Legendre quadrature [24]	yes	no	yes
Gauss-Patterson quadrature [47]	yes	no	yes
Genz-Keister quadrature [22]	yes	no	yes
Leja quadrature [43]	no	no	yes
Monte Carlo integration [51]	yes	no	yes
Optimal Gaussian quadrature [24]	yes	no	yes

Table 6: Various numerical integration strategies implemented in the three software toolboxes.

a few native multivariate cubature rules that allow for low order multivariate integration [56]. Both Dakota and Chaospy support the Smolyak sparse grid and anisotropy.

Custom Integration Rules Chaospy has support for construction of custom integration rules defined by the user. The `rule_generator` function can be used to join a list of univariate rules using tensor grid or Smolyak sparse grid. For example, consider the trapezoid rule:

```
>>> def trapezoid(n):
...     X = np.linspace(0, 1, n+1)
...     W = np.ones(n+1)/n
...     W[0] *= 0.5; W[-1] *= 0.5
...     return X, W
...
>>> nodes, weights = trapezoid(2)
>>> print nodes
[ 0.   0.5  1. ]
>>> print weights
[ 0.25  0.5   0.25]
```

The `rule_generator` function takes positional arguments, each representing a univariate rule. To generate a rule for the multivariate case, with the same one-dimensional rule along two axes, we do the following:

```
>>> mvtrapezoid = cp.rule_generator(trapezoid, trapezoid)
>>> nodes, weights = mvtrapezoid(2, sparse=True)
>>> print nodes
[[ 0.   0.5  1.   0.   0.   1. ]
 [ 0.   0.   0.   0.5  1.   1. ]]
>>> print weights
[ 0.     0.25   0.125  0.25   0.125  0.25 ]
```

Relevance back to Dist and Orth Software for constructing and executing a general-purpose integration scheme is useful for several computational components in uncertainty quantification. For example, in Section 3.1 when constructing orthogonal polynomial using raw statistical moments, or calculating

discretized Stieltjes' recurrence coefficients, numerical integration is relevant. Like the `ppf` function noted in Section 2.1, the moments and recurrence coefficients can be added directly into each distribution. However, when these are not available, Chaospy will automatically estimate missing information by quadrature rules, using the `generate_quadrature` function described above.

The fit_quadrature function To compute the Fourier coefficients and the polynomial chaos expansion, we use the `fit_quadrature` function. It takes four arguments: the set of orthogonal polynomials, quadrature nodes, quadrature weights, and the user's function for evaluating the forward model (to be executed at the quadrature nodes). Note that in the case of the discretized Stieltjes method discussed in Section 3.1, the nominator $\mathbb{E}\left[\mathbf{\Phi}_n^2\right]$ in (14) can be calculated more accurately using recurrence coefficients [21]. Special numerical features like this can be added by including optional arguments in `fit_quadrature`.

Point Collocation

The other non-intrusive approach to estimate the Fourier coefficients $\{c_k\}_{k\in I_K}$ is the point collocation method. One way of formulating the method is to require the polynomial expansion to equal the model evaluations at a set of collocation nodes $\{\boldsymbol{q}_k\}_{k\in I_K}$, resulting in an over-determined set of linear equations for the Fourier coefficients:

$$\begin{bmatrix} \Phi_0(\boldsymbol{q}_0) & \cdots & \Phi_N(\boldsymbol{q}_0) \\ \vdots & & \vdots \\ \Phi_0(\boldsymbol{q}_{K-1}) & \cdots & \Phi_N(\boldsymbol{q}_{K-1}) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{q}_0) \\ \vdots \\ f(\boldsymbol{q}_{K-1}) \end{bmatrix}, \tag{15}$$

Unlike pseudo spectral projection, the locations of the collocation nodes are not required to follow any integration rule. Hosder [30] showed that the solution using Hammersley samples from quasi-Monte Carlo samples resulted in more stable results than using conventional pseudo-random samples. In other words, well placed collocation nodes might increase the accuracy. In Chaospy these collocation nodes can be selected from integration rules or from pseudo-random samples from Monte Carlo simulation, as discussed in Section 2.3. In addition, the software accepts user defined strategies for choosing the sampling points. Turns also allows for user-defined points, while Dakota has its predefined strategies.

Linear Regression The obvious way to solve the over-determined system in (15) is to use least squares minimization, which resembles the standard statistical linear regression approach. However, from a numerical point of view, this might not be the best strategy. If the numerical stability of the solution is low, it might be prudent to use Tikhonov regularization [49], or if the problem is so large that the number of coefficients is very high, it might be useful to force some of the coefficients to be zero through least angle regression [14]. Being able to run and compare alternative methods is important in many problems to see if the numerical stability is a potential problem. Table 7 lists the regression methods offered by Dakota, Turns, and Chaospy.

Implementation of `fitter_lr` Generating a polynomial chaos expansion using linear regression is done using Chaospy's `fit_regression` function. It takes the same arguments as `fit_quadrature`, except that quadrature weights are omitted, and optional arguments define the rule used to optimize (15).

Regression Schemes	Dakota	Turns	Chaospy
Basis Pursuit [9]	yes	no	no
Bayesian Auto. Relevance Determination [60]	no	no	yes
Bayesian ridge [37]	no	no	yes
Elastic Net [66]	yes	no	yes
Forward Stagewise [29]	no	yes	no
Least Absolute Shrinkage & Selection [14]	yes	yes	yes
Least Angle & Shrinkage with AIC/BIC [67]	no	no	yes
Least Squares Minimization	yes	yes	yes
Orthogonal matching pursuit [39]	yes	no	yes
Singular Value Decomposition	no	yes	no
Tikhonov Regularization [49]	no	no	yes

Table 7: List of supported regression methods for estimating Fourier coefficients

Model Evaluations

Irrespectively of the method used to estimate the Fourier coefficients, the user is left with the job to evaluate the forward model (response function) f, which is normally by far the most computing-intensive part in uncertainty quantification. Chaospy does not impose any restriction on the simulation code used to compute the forward model. The only requirement is that the user can provide an array of values of f at the quadrature or collocation nodes. Chaospy users will usually wrap any complex simulation code for f in a Python function `f(q)`, where `q` is a node in probability space (i.e., `q` contains values of the uncertain parameters in the problem). For example, for pseudo-spectral projection, samples of f can be created as

```
samples = [f(node) for node in nodes.T]
```

or perhaps done in parallel if f is time consuming to evaluate:

```
import multiprocessing as mp
pool = mp.Pool(mp.cpu_count())
samples = pool.map(f, nodes.T)
```

The evaluation of all the f values can also be done in parallel with MPI in a distributed way on a cluster using the Python module `mpi4py`. Both Dakota and Turns include support parallel evaluation of f values. Also, in many applications, the computation of f at a single point `q` must be done in parallel, but

this is only a concern inside the software that computes f and does not affect Chaospy.

4.1 Numerical Integration

In the course of implementing PCE, numerical integration is often required at many steps to estimate quantities that are not available analytically. For example it is required when approximating an orthogonal polynomial expansion, to fit a function to the PCE, and to estimate the models statistical moments. Common for all the integrals is that they are on the form:

$$I = \int \cdots \int W(\boldsymbol{q}) H(\boldsymbol{q}) \, \mathrm{d}q_0 \cdots \mathrm{d}q_{N-1} \approx \hat{I} = \sum_{k=0}^{K} H(\boldsymbol{q}_k) w_k, \tag{16}$$

where W is an known weight function, H is a function with unknown shape, and q_k and w_k are abscissas and weights. Since the probability density function is a common denominator in many integrations, it is selected as this weight function. For integrals that doesn't include a probability, the weight is set to 1.

The core of the numerical integration functionality in `chaospy` can be found in the `quadgen` function. For example, to create a simple second order Clenshaw-Curtis (CC) nodes and weights, on the interval $[0, 1]$, we have

```
nodes, weights = pc.quadgen(2, [0,1])
print nodes
#  [[ 0.   0.5  1. ]]
print weights
#  [ 0.16666667  0.66666667  0.16666667]
```

This is an example of an integration rule where the weight function W is set to 1, since no distribution was provided.

To include a weight function, the only change required is to replace the interval $[0, 1]$ to a random variable. For example take a Beta distribution on the same interval with parameters `a=2` and `b=2`. A third order implementation of CC quadrature would be as follows:

```
dist = pc.Beta(2,2)
nodes, weights = pc.quadgen(3, dist)
print nodes
#  [[ 0.14644661  0.5         0.85355339]]
print weights
#  [ 0.2  0.6  0.2]
```

Two things happens automatically when using CC with a distribution. Firstly, all node where the probability is zero are removed. No need to evaluate a function at locations which isn't included in any calculations. The Beta distribution has both endpoints are identified as zero, such that the third order samples are reduced to only three nodes. Note that by changing one of the parameters to 1, this would not be the case. The zero is found by removing elements such

that the sum of weights only changes with at most 10^{-15}. Note also that CC without endpoints is known as Féjer quadrature [18].

The other difference with the second implementation is the incorporation of the probability distribution. The weights are multiplied with the corresponding probability, and then normalized to 1. The former ensures that the distribution is accounted for consistently through all rules, and the latter ensures that a zeroth order polynomial is integrated correctly as described in the discretized Stieltje's procedure[55]. In practice these two features can be observed in that the weights doesn't include the endpoint, which CC usually does, and that it sums up to one.

In addition to CC, there are other quadrature rules available: optimal Gaussian quadrature and Gauss-Legendre quadrature. The former is calculated using Golub-Welsch algorithm[24] using the probability density function as weights. The latter is the optimal Gaussian quadrature for the uniform distribution, known as Gauss-Legendre quadrature. Given the use of a uniform distribution, these two would be the same, but for other distributions, it is modified to account for density, unnecessary nodes removed, and normalized to one, like CC.

To change the quadrature rule in the software, use the `rule` keyword argument. The three methods can be selected this argument to `"C"`, `"G"` and `"E"` respectively. To create the optimal Gaussian quadrature nodes and weights for the Beta distribution, we have the following:

```
nodes, weights = pc.quadgen(2, dist, rule="G")
print nodes
#  [[ 0.17267316  0.5         0.82732684]]
print weights
#  [ 0.23333333  0.53333333  0.23333333]
```

Irrespectively of which quadrature rule being used, they all work best on smooth multiple differentiable functions. If this is not the case, the integration accuracy might be affected. However if the location of where the discontinuities are known it is possible to introduce composite rule through `composite` keyword. The keyword can either be an integer, representing the number of evenly space intervals used. For example with a first order Gauss-Legendre quadrature:

```
nodes, weights = pc.quadgen(1, [0,1], rule="E", composite=2)
print nodes
#  [[ 0.10566243  0.39433757  0.60566243  0.89433757]]
print weights
#  [ 0.25  0.25  0.25  0.25]
```

If however a float or an array of floats is provided, it represents the location of the composite breaks. This allows for more detailed control over the composite growth:

```
nodes, weights = pc.quadgen(1, [0,1], rule="E", composite=[0.2])
print nodes
#  [[ 0.04226497  0.15773503  0.36905989  0.83094011]]
```

```
print weights
#  [ 0.1  0.1  0.4  0.4]
```

Here the dividing point is 0.2 making the left interval much smaller than the right one. This is reflected in the change of size in the weights.

4.2 Pseudo Spectral Projection

In practice the following four components are needed to perform psedo spectral projection:

- A distribution for the unknown function parameters as described in section 2:

```
dist = pc.Iid(pc.Normal(), 2)
```

- Integration nodes and weights:

```
nodes, weights = pc.quadgen(2, dist, rule="G")
print nodes
#  [[-1.73205081 -1.73205081 -1.73205081  0.          0.
#     0.          1.73205081  1.73205081  1.73205081]
#   [-1.73205081  0.          1.73205081 -1.73205081  0.
#     1.73205081 -1.73205081  0.          1.73205081]]
print weights
#  [ 0.02777778  0.11111111  0.02777778  0.11111111  0.44444444
#    0.11111111  0.02777778  0.11111111  0.02777778]
```

- An orthogonal polynomial expansion as described in section 3 where the weight function is the distribution in the first step:

```
orth = pc.orth_ttr(2, dist)
print orth
#  [1.0, q1, q0, q1^2-1.0, q0q1, q0^2-1.0]
```

- A function evaluated using the nodes generated in the second step. For example:

```
def model_solver(q):
    return [q[0]*q[1], q[0]*e**-q[1]+1]
solves = [model_solver(q) for q in nodes.T]
```

To bring it together, expansion, nodes, weights and solves are used as arguments to create approximation:

```
approx = pc.fitter_quad(orth, nodes, weights, solves)
print pc.around(approx, 14)
#  [q0q1, -1.58058656357q0q1+1.63819248006q0+1.0]
```

Note that in this case the function output is bivariate. The software is designed to create an approximation of any discretized model as long as it is compatible with `numpy` shapes.

As mentioned in section 3.4, moment based construction of polynomials can be unstable. This might also be the case for the denominator $\mathbb{E}\left[\mathbf{\Phi}_n^2\right]$. So when using three terms recursion, it is common to use the recurrence coefficients to estimated the denominator. They are calculated as follows:

$$\begin{aligned}\mathbb{E}\left[\mathbf{\Phi}_n^2\right] &= \mathbb{E}\left[\Phi_{1,n_1}^2 \cdots \Phi_{N,n_N}^2\right] = \mathbb{E}\left[\Phi_{1,n_1}^2\right] \cdots \mathbb{E}\left[\Phi_{N,n_N}^2\right] \\ &= \mathbb{E}[\Phi_{1,0}] \prod_{i_1=1}^{n_1} \frac{\mathbb{E}\left[\Phi_{1,i_1}^2\right]}{\mathbb{E}\left[\Phi_{1,i_1-1}^2\right]} \cdots \mathbb{E}[\Phi_{N,0}] \prod_{i_N=1}^{n_1} \frac{\mathbb{E}\left[\Phi_{N,i_N}^2\right]}{\mathbb{E}\left[\Phi_{N,i_N-1}^2\right]} \\ &= \prod_{i_1=1}^{n_1} B_{1,i_1} \cdots \prod_{i_N=1}^{n_N} B_{N,i_N},\end{aligned}$$

where $B_{n,i}$ is i-th iteration of the second recurrence coefficient as defined in (13), but defined for dimension n. Alternative is to numerically estimate this term using numerical integration from section 4.1.

To include these stable norms in the calculations can be done with almost no extra effort:

```
orth, norms = pc.orth_ttr(2, dist, retall=True)
approx2 = pc.fitter_quad(orth, nodes, weights, solves, norms=norms)
```

Note that at low polynomial order, the error is very small. For example the largest coefficient between the two approximation:

```
print np.max(abs(approx-approx2).coeffs(), -1)
#  [  2.44249065e-15   3.77475828e-15]
```

The `coeffs` function returns all the polynomial coefficients.

4.3 Point Collocation Method

To perform point collocaiton method in practice, we need the following four components:

- A distribution for the unknown function parameters as described in section 2:

```
dist = pc.Iid(pc.Normal(), 2)
```

- Collocation nodes as described in section 2:

```
nodes = dist.sample(12, "M")
print nodes
#  [[ 1.3074948   1.1461811   0.48907479 -0.99729764 -2.04207273  0.3737412
#    -0.17126603  0.40045745  0.25582169 -1.50683751 -0.47799293  1.21890579]
#   [ 0.61939522  1.72367491 -0.55533514 -0.00905152 -0.87071076 -0.04532524
#    -0.95908033 -1.5433918  -1.10189542  1.19303123 -0.85594892 -0.97358421]]
```

- An orthogonal polynomial expansion as described in 3.2 where the weight function is the distribution in the first step:

```
orth = pc.orth_ttr(2, dist)
print orth
#  [1.0, q1, q0, q1^2-1.0, q0q1, q0^2-1.0]
```

- A function evaluated using the nodes generated in the second step:

```
def model_solver(q):
    return [q[0]*q[1], q[0]*e**-q[1]+1]
solves = [model_solver(q) for q in nodes.T]
```

To bring it together, expansion, nodes and solves are used as arguments to create approximation:

```
approx = pc.fitter_lr(orth, nodes, solves)
print pc.around(approx, 14)
#  [q0q1, 0.161037230451q1^2-1.23004736008q0q1+0.152745901925q0^2+
#           0.0439971157359q1+1.21399892993q0+0.86841679007]
```

In the example described above, the number of collocation points K is selected to be twice the number of unknown coefficients $N + 1$. This follows the default outlined in [30]. Changing this is obviously possible. When the number of parameter is equal the number of unknown, the, the polynomial approximation becomes an interpolation method and overlap with Lagrange polynomials. If the number of samples are fewer than the number of unknown, classical least squares can not be used. Instead it possible to use methods for doing estimation with too few samples. In table 7 a list of regression methods are listed which includes such problems. For example there is orthogonal matching pursuit [39]. It forces the result to have at most one non-zero coefficient. To implement it use the keyword `rule="OMP"`, and to force the number of coefficients to be for example 1: `n_nonzero_coefs=1`. In practice:

```
approx = pc.fitter_lr(orth, nodes, solves,
        rule="OMP", n_nonzero_coefs=1)
print approx
#  [q0q1, 1.52536467971q0]
```

Except for least squares, Tikhonov regularization with and without cross validation, all the method listed is taken from `sklearn` software. All optional arguments for various methods is covered in both `sklearn.linear_model` and in `pc.fitter_lr`.

4.4 Descriptive Tools

Analytical Metrics Exists The last step in uncertainty quantification using polynomial chaos expansion is to quantify the uncertainty in the approximation $\hat{f}$. For the most popular statistical metrics, like mean, variance, correlation, a polynomial chaos expansion allows for analytical analysis, which is easy to calculate and has high accuracy. This property is reflected in all the three toolboxes. To calculate the expected value, variance and correlation of a simple (here univariate) polynomial approximation `f_approx`, with a normally distributed ξ_0 variable, we can with Chaospy write

```
>>> f_approx = fit_quadrature(orth, nodes, weights, solves)
>>> print f_approx
[q0, q0^2, q0^3]
>>> dist = Normal(0,1)
>>> print E(f_approx, dist)
[ 0.  1.  0.]
>>> print Var(f_approx, dist)
[  1.   2.  15.]
>>> print Corr(f_approx, dist)
[[ 1.          0.          0.77459667]
 [ 0.          1.          0.        ]
 [ 0.77459667  0.          1.        ]]
```

A list of supported analytical metrics is listed in Table 8.

Method	Dakota	Turns	Chaospy
Covariance/Correlation	yes	yes	yes
Expected value	yes	yes	yes
Conditional expectation	no	no	yes
Kurtosis	yes	yes	yes
Sensitivity index	yes	yes	yes
Skewness	yes	yes	yes
Variance	yes	yes	yes

Table 8: List of common statistical operators that can be used to analytical evaluation of polynomials.

Extension of polynomial expansions

There is much literature that extends on the theory of polynomial chaos expansion [62]. For example, Isukapalli showed that the accuracy of a polynomial expansion could be increased by using partial derivatives of the model response [31]. This theory is only directly supported by Dakota. In Turns and Chaospy the support is only indirect by allowing the user to add the feature manually.

To be able to incorporate partial derivatives of the response, the partial derivative of the polynomial expansion must be available as well. In both Turns

and Chaospy, the derivative of a polynomial can be generated easily. This derivative can then be added to the expansion, allowing us to incorporate Isukapalli's theory in practice. This is just an example on how manipulation of the polynomial expansions and model approximations can overcome the lack of support for a particular feature from the literature.

To be able to support many current and possible future extensions of polynomial chaos, a large collection of tools for manipulating polynomials must be available. In Dakota, no such tools exist from a user perspective. In Turns, there is support for some arithmetic operators in addition to the derivative. In Chaospy, however, the polynomial generated for the model response is of the same type as the polynomials generated in Sections 3.1 and 4, and the rich set of manipulations of polynomials is then available for $\hat{f}$ as well.

Introducing Secondary Monte Carlo Simulation Beyond the analytical tools for statistical analysis of $\hat{f}$, either from the toolbox or custom ones by the user, there are many statistical metrics that cannot easily be expressed as simple closed-form formulas. Such metrics often include confidence intervals, sensitivity indices, p-values in hypothesis testing, to mention a few. In those scenarios, it makes sense to perform a secondary uncertainty analysis through Monte Carlo simulation. Evaluating the approximation $\hat{f}$ is normally computationally much cheaper than evaluating the full forward model f, thus allowing a large number of Monte Carlo samples within a cheap computational budget. This type of secondary simulations are done automatically in the background in Dakota and Turns, while Chaospy does not feature automated tools for secondary Monte Carlo simulation. Instead, Chaospy allows for simple and computationally cheap generation of pseudo-random samples, as described in Section 2.3, such that the user can easily put together a tailored Monte Carlo simulation to meet the needs at hand. Within a few lines of Python code, the samples can be analyzed with the standard Numpy and the Scipy libraries [32] or with more specialized statistical libraries like Pandas [42], Scikit-learn [48] Scikit-statsmodel [52], and Python's interface to the rich R environment for statistical computing. For example, for the specific $\hat{f}$ function illustrated above, the following code computes a 90 percent confidence interval, based on 10^5 pseudo-random samples and Numpy's functionality for finding percentiles in discrete data:

```
>>> q_samples = Normal(0,1).sample(10**5)
>>> samples = f_approx(*q_samples)
>>> import numpy as np
>>> p05, p95 = np.percentile(samples, [5, 95], axis=-1)
>>> print p05[:3]
[ 1.         1.0000004  1.0000016]
>>> print p95[:3]
[ 1.          1.00038886  1.00155544]
```

Since the type of statistical analysis of $\hat{f}$ often strongly depends on the physical problem at hand, we believe that the ability to quickly compose custom solutions by putting together basic building blocks is very useful in uncertainty quantification. This is yet another example of the need for a package with a strong focus on easy customization.

References

[1] P. Achard and E. De Schutter. Complex parameter landscape for a complex neuron model. *PLoS Computational Biology*, 2(7), July 2006.

[2] G. Andrianov, S. Burriel, S. Cambier, A. Dutfoy, I. Dutka-Malen, E. De Rocquigny, B. Sudret, P. Benjamin, R. Lebrun, and F. Mangeant. Open TURNS, an open source initiative to Treat Uncertainties, Risks N Statistics in a structured industrial approach. In *Proc. ESREL2007 safety and reliability conference. Stavenger, Norway*, 2007.

[3] R. Askey and J. A Wilson. *Some basic hypergeometric orthogonal polynomials that generalize Jacobi polynomials*. Amer Mathematical Society, 1985.

[4] J. Back, F. Nobile, L. Tamellini, and R. Tempone. Stochastic spectral Galerkin and collocation methods for PDEs with random coefficients: a numerical comparison. In *Spectral and High Order Methods for Partial Differential Equations*, pages 43–62. Springer, 2011.

[5] M. Bertran. Note on Orthogonal Polynomials in v-Variables. *SIAM Journal on Mathematical Analysis*, 6(2):250–257, 1975.

[6] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[7] R. C. Boardman and J. E. Vanna. A review of the application of copulas to improve modelling of non-bigaussian bivariate relationships (with an example using geological data). In *International Congress on Modelling and Simulation*, 2011.

[8] J. Burkardt. The combining coefficient for anisotropic sparse grids. Technical report, Technical report, Virginia Tech. 125, 2009.

[9] S. S. Chen, D. L. Donoho, and M. A. Saunders. Atomic decomposition by basis pursuit. *SIAM journal on scientific computing*, 20(1):33–61, 1998.

[10] F. Clementi and M. Gallegati. Paretos law of income distribution: evidence for Germany, the United Kingdom, and the United States. In *Econophysics of Wealth Distributions*, pages 3–14. Springer, 2005.

[11] C. W. Clenshaw and A. R. Curtis. A method for numerical integration on an automatic computer. *Numerische Mathematik*, 2(1):197–205, 1960.

[12] J. Dobric and F. Schmid. A goodness of fit test for copulas based on Rosenblatt's transformation. *Computational Statistics & Data Analysis*, 51(9):4633–4642, 2007.

[13] B. Efron. Bootstrap methods: another look at the jackknife. *The annals of Statistics*, pages 1–26, 1979.

[14] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[15] M. S. Eldred, A. A. Giunta, B. G. van Bloemen Waanders, S. F. Wojtkiewicz, W. E. Hart, and M. P. Alleva. *DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 4.1 reference manual*. Sandia National Laboratories Albuquerque, NM, 2007.

[16] J. Feinberg and H. P. Langtangen. Uncertainty Quantification of Diffusion in Layered Media by a New Method Based on Polynomial Chaos Expansion. Norwegian University of Science and Technology, May 2013. Akademika Publishing.

[17] J. Feinberg and H. P. Langtangen. Multivariate Polynomial Chaos with Dependent Variables. Unbublished journal article, 2015.

[18] Leopold Fjer. On the infinite sequences arising in the theories of harmonic analysis, of interpolation, and of mechanical quadratures. *Bulletin of the American Mathematical Society*, 39(8):521–534, 1933.

[19] S. Galanti and A. Jung. Low-discrepancy sequences: Monte Carlo simulation of option prices. *The Journal of Derivatives*, 5(1):63–83, 1997.

[20] W. Gautschi. Construction of Gauss-Christoffel quadrature formulas. *Mathematics of Computation*, 22(102):251, April 1968.

[21] W. Gautschi. *Orthogonal Polynomials: Computation and Approximation*. Oxford University Press, USA, June 2004.

[22] A. Genz and B. Keister. Fully Symmetric Interpolatory Rules for Multiple Integrals over Infinite Regions. *Comp. Appl. Math*, 72, 1996.

[23] R. Ghanem and P. D. Spanos. *Stochastic finite elements: a spectral approach*. Courier Dover Publications, August 2003.

[24] G. H. Golub and J. H. Welsch. *Calculation of Gauss quadrature rules*, volume 23. Mathematics of Computation, 1967.

[25] D. Gottlieb and S. A. Orszag. *Numerical Analysis of Spectral Methods: Theory and Applications*. Society for Industrial and Applied Mathematics, 1977.

[26] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2(1):84–90, 1960.

[27] J. M. Hammersley. Monte Carlo methods for solving multivariable problems. *Annals of the New York Academy of Sciences*, 86(3):844–874, 1960.

[28] C. B. Haselgrove. A method for numerical integration. *Mathematics of computation*, pages 323–337, 1961.

[29] T. Hastie, J. Taylor, R. Tibshirani, and G. Walther. Forward stagewise regression and the monotone lasso. *Electronic Journal of Statistics*, 1:1–29, 2007.

[30] S. Hosder, R. W. Walters, and M. Balch. Efficient sampling for non-intrusive polynomial chaos applications with multiple uncertain input variables. In *Proceedings of the 48th Structures, Structural Dynamics, and Materials Conference*, volume 125, Honolulu, HI, 2007.

[31] S. S. Isukapalli. *Uncertainty analysis of transport-transformation models*. PhD thesis, Rutgers, The State University of New Jersey, 1999.

[32] E. Jones, T. Oliphant, and P. Peterson. SciPy: Open source scientific tools for Python. *http://www. scipy. org/*, 2001. URL: http://scipy.org/.

[33] N. M. Korobov. The approximate calculation of multiple integrals using number theoretic methods. *Acad. Nauk SSSR*, 115:1062–1065, 1957.

[34] D. P. Kroese, T. Taimre, and Zdravko I. Botev. *Handbook of Monte Carlo Methods*, volume 706. John Wiley & Sons, 2011.

[35] P. Laux, G. Jackel, R. M. Tingem, and H. Kunstmann. Impact of climate change on agricultural productivity under rainfed conditions in CameroonA method to improve attainable crop yields by planting date adaptations. *Agricultural and Forest Meteorology*, 150(9):1258–1271, 2010.

[36] A. J. Lee. Generating random binary deviates having fixed marginal distributions and specified degrees of association. *The American Statistician*, 47(3):209–215, 1993.

[37] D. J. C. MacKay. Bayesian interpolation. *Neural computation*, 4(3):415–447, 1992.

[38] R. W. Makuch, D. H. Freeman Jr, and M. F. Johnson. Justification for the lognormal distribution as a model for blood pressure. *Journal of chronic diseases*, 32(3):245–250, 1979.

[39] S. G. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *Signal Processing, IEEE Transactions on*, 41(12):3397–3415, 1993.

[40] B. Mandelbrot, R. L. Hudson, and E. Grunwald. The (mis) behaviour of markets. *The Mathematical Intelligencer*, 27(3):77–79, 2005.

[41] M. D. McKay, R. J. Beckman, and W. J. Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[42] W. McKinney. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 2012.

[43] A. Naraya and J. Jakeman. Adaptive Leja sparse grid construction for stochastic collocation and high-dimensional approximation. *arXiv e-print*, 2014. https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/1404.5663v1.pdf.

[44] A. Nataf. Dtermination des distributions de probabilites dont les marges sont donnes. *Comptes rendus de lacademie des sciences*, 225:42–43, 1962.

[45] R. B. Nelsen. *An introduction to copulas*. Springer, 1999.

[46] H. Niederreiter. Point sets and sequences with small discrepancy. *Monatshefte fur Mathematik*, 104(4):273–337, 1987.

[47] T. Patterson. The optimum addition of points to quadrature formulae. *Mathematics of Computation*, 22(104):847–856, 1968.

[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, and V. Dubourg. Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[49] R. M. Rifkin and R. A. Lippert. Notes on regularized least squares. *Journal of Linear Algebra*, 18:281–288, June 2009.

[50] M. Rosenblatt. Remarks on a Multivariate Transformation. *The Annals of Mathematical Statistics*, 23(3):470–472, 1952.

[51] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo Method*. Wiley-Interscience, 2 edition, December 2007.

[52] S. Seabold and J. Perktold. Statsmodels: econometric and statistical modeling with Python. In *Proceedings of the 9th Python in Science Conference*, pages 57–61, 2010.

[53] S. A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. In *Dokl. Akad. Nauk SSSR*, volume 4, page 123, 1963.

[54] I. M. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[55] T. J. Stieltjes. Quelques recherches sur la thorie des quadratures dites mcaniques. *Ann. Sci. cole Norm. Sup. (3)*, 1:409–426, 1884.

[56] A. H. Stroud. *Approximate calculation of multiple integrals*, volume 431. Prentice-Hall Englewood Cliffs, NJ, 1971.

[57] S. Tezuka. *Uniform random numbers: Theory and practice*. Kluwer Academic Publishers Boston, 1995.

[58] W. Wang, Z. Wu, C. Wang, and R. Hu. Modelling the spreading rate of controlled communicable epidemics through an entropy-based thermodynamic model. *Science China Physics, Mechanics and Astronomy*, 56(11):2143–2150, 2013.

[59] M. Wilck. A general approximation method for solving integrals containing a lognormal weighting function. *Journal of aerosol science*, 32(9):1111–1116, 2001.

[60] D. P. Wipf and S. S. Nagarajan. A new view of automatic relevance determination. In *Advances in Neural Information Processing Systems*, pages 1625–1632, 2007.

[61] J. Witteveen and H. Bijl. Modeling arbitrary uncertainties using Gram-Schmidt polynomial chaos. In *44th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, Nevada, 2006.

[62] D. Xiu. Fast numerical methods for stochastic computations: a review. *Communications in computational physics*, 5(2-4):242–272, 2009.

[63] D. Xiu. *Numerical Methods for Stochastic Computations: A Spectral Method Approach*. Princeton University Press, July 2010.

[64] D. Xiu and J. S. Hesthaven. High-order collocation methods for differential equations with random inputs. *SIAM Journal on Scientific Computing*, 27:1118, 2005.

[65] D. Xiu, D. Lucor, C. H Su, and G. E Karniadakis. Stochastic modeling of flow-structure interactions using generalized polynomial chaos. *Journal of Fluids Engineering*, 124:51, 2002.

[66] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

[67] H. Zou, T. Hastie, and R. Tibshirani. On the degrees of freedom of the lasso. *The Annals of Statistics*, 35(5):2173–2192, 2007.