Chaospy: An Open Source Tool for Designing Methods of Uncertainty Quantification

Jonathan Feinberg[1,2,3] (jonathan@feinberg.no)

Hans Petter Langtangen[1,4] (hpl@simula.no)

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Mathematics, University of Oslo
[3]Expert Analytics
[4]Department of Informatics, University of Oslo

August 30, 2015

Abstract

The paper describes the philosophy, design, functionality, and usage of the Python software toolbox Chaospy for performing uncertainty quantification via polynomial chaos expansions and Monte Carlo simulation. The paper compares Chaospy to similar packages and demonstrates a stronger focus on defining reusable software building blocks that can easily be assembled to construct new, tailored algorithms for uncertainty quantification. For example, a Chaospy user can in a few lines of high-level computer code define custom distributions, polynomials, integration rules, sampling schemes, and statistical metrics for uncertainty analysis. In addition, the software introduces some novel methodological advances, like a framework for computing Rosenblatt transformations and a new approach for creating polynomial chaos expansions with dependent stochastic variables.

Note: This paper is accepted in Journal of Computational Science and will be published as an Open Access paper.

Contents

1 Introduction

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.

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 [48], while for the latter, we refer to the excellent book by Xiu [58]. 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 [58], but are less general and less widely applicable than the two addressed in this paper.

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 a few viable options for the latter class of methods: *The Dakota Project* (referred to as Dakota) [15], the *Opus Open Turns library* (referred to as Turns) [2], *Uncertenty Quantification Toolkit* [11], and *MIT Uncertenty Quantification Library* [10]. In this paper we will focus on the former two: Dakota and Turns. 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.

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 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 requires the forward model f to be wrapped in a stand-alone callable executable. The common approach is then to link this executable 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.

Alternative to this direct approach is to interact with an application programming interface (API). 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 deeper level of integration between the user's model and the UQ tools. Also, exposing the software's internal components through an 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. This approach is used by the Turns software (using the languages Python or R) and is supported in Dakota through a library mode (using C++).

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.

This paper describes a new, third alternative open source software package called Chaospy [18]. 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 of 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 [40] 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.

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.

The paper is structured as follows. We start in Section 2 with a quick demonstration of how the software can be used to perform uncertainty quantification in a simple physical problem. Section 3 addresses probability distributions and the theory relevant to perform Monte Carlo simulation. Section 4 concerns non-intrusive polynomial chaos expansions, while conclusions and topics for further work appear in Section 5.

2 A Glimpse of Chaospy in Action

To demonstrate how Chaospy is used to solve an uncertainty quantification problem, we consider a simple physical example of (scaled) exponential decay with an uncertain, piecewise constant coefficient:

$$u'(x) = -c(x)u(x), \quad u(0) = u_0,\ c(x) = \begin{cases} c_0, & x < 0.5 \\ c_1, & 0.5 \leq x < 0.7 \\ c_2, & x \geq 0.7 \end{cases} \tag{2}$$

Such a model arises in many contexts, but we may here think of $u(x)$ as the porosity at depth x in geological layers and c_i as a (scaled) compaction constant in layer number i. For simplicity, we consider only three layers with three uncertain constants c_0, c_1, and c_2.

The model can easily be evaluated by solving the differential equation problem, here by a 2nd-order Runge-Kutta method on a mesh `x`, coded in Python as:

```
def model(x, u0, c0, c1, c2):
    def c(x):
        if   x < 0.5:           return c0
        elif 0.5 <= x < 0.7:    return c1
        else:                   return c2

    N = len(x)
    u = np.zeros(N)

    u[0] = u0
    for n in xrange(N-1):
        dx = x[n+1] - x[n]
        K1 = -dx*u[n]*c(x[n])
        K2 = -dx*u[n] + K1/2*c(x[n]+dx/2)
        u[n+1] = u[n] + K1 + K2
    return u
```

Alternatively, the model can be implemented in some external software in another programming language. This software can either be run as a stand-alone application, where the Python function `model` runs the application and communicates with it through input and output files, or the `model` function can communicate with the external software through function calls if a Python wrapper has been made for the software (there are numerous technologies available for creating Python wrappers for C, C++, and Fortran software).

The Chaospy package may be loaded by

```
import chaospy as cp
```

Each of the uncertain parameters must be assigned a probability density, and we assume that c_0, c_1, and c_2 are stochastically independent:

```
c0 = cp.Normal(0.5, 0.15)
c1 = cp.Uniform(0.5, 2.5)
c2 = cp.Uniform(0.03, 0.07)
# Joint probability distribution
distribution = cp.J(c0, c1, c2)
```

The sample points (c_0, c_1, c_2) in probability space, where the model is to be evaluated, can be chosen in many ways. Here we specify a third-order Gaussian Quadrature scheme tailored to the joint distribution:

```
nodes, weights = cp.generate_quadrature(
    order=3, domain=distribution, rule="Gaussian")
```

The next step is to evaluate the computational model at these sample points (object `nodes`):

```
x = np.linspace(0, 1, 101)
samples = [model(x, u0, node[0], node[1], node[2])
           for node in nodes.T]
```

Now, `samples` contains a list of arrays, each array containing u values at the 101 `x` values for one combination (c_0, c_1, c_2) of the input parameters.

To create a polynomial chaos expansion, we must generate orthogonal polynomials corresponding to the joint distribution. We choose polynomials of the same order as specified in the quadrature rule, computed by the widely used three-term recurrence relation (`ttr`):

```
polynomials = cp.orth_ttr(order=3, dist=distribution)
```

To create an approximate solver (or surrogate model), we join the polynomial chaos expansion, the quadrature nodes and weights, and the model samples:

```
model_approx = cp.fit_quadrature(
               polynomials, nodes, weights, samples)
```

The `model_approx` object can now cheaply evaluate the model at a point (c_0, c_1, c_2) in probability space for all x points in the `x` array. Built-in tools can be used to derive statistical information about the model response:

```
mean = cp.E(model_approx, distribution)
deviation = cp.Std(model_approx, distribution)
```

The `mean` and `deviation` objects are arrays containing the mean value and standard deviation at each point in `x`. A graphical illustration is shown in Figure 1.

The accuracy of the estimation is comparable to what Dakota and Turns can provide. Figure 2 shows that the estimation error in the three software toolboxes are almost indistinguishable. The error is calculated as the absolute difference between the true value and the estimated value integrated over the depth x:

$$\varepsilon_{\mathbb{E}} = \int_0^1 |\mathbb{E}(u) - \mathbb{E}(u_{\text{approx}})| \mathrm{d}x, \quad \varepsilon_{\mathbb{V}} = \int_0^1 |\mathbb{V}(u) - \mathbb{V}(u_{\text{approx}})| \mathrm{d}x$$

Both the point collocation method and the pseudo-spectral projection method are included. The former is calculated using two times the random collocation nodes as the number of polynomials, and the latter using Gaussian quadrature integration with quadrature order equal to polynomial order. Note that Turns does not support pseudo-spectral projection, and is therefore only compared using point collocation.

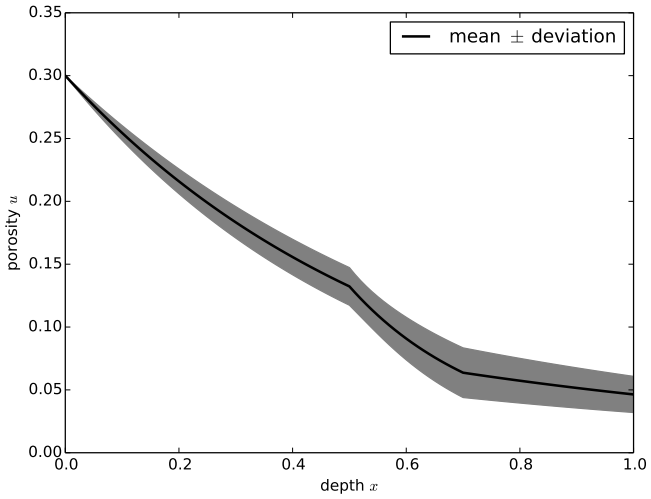

Figure 1: Solution of a simple stochastic differential equation with uncertain coefficients.

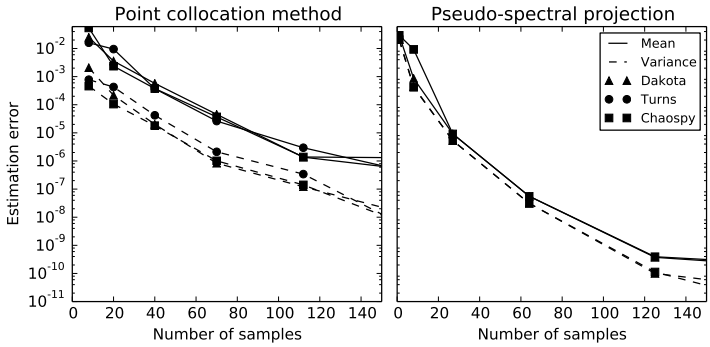

Figure 2: The error in estimates of the mean and variance, computed by Dakota, Turns, and Chaospy using point collocation and pseudo-spectral projection, is almost identical.

3 Modelling Random Variables

3.1 Rosenblatt Transformation

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 [6], finance [12], and ion channel densities in detailed neuroscience models [1]. There also exists examples where introducing dependent random variables is beneficial for the modelling process, even though the original input was stochastically independent [19]. 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.

All random samples are in Chaospy generated using Rosenblatt transformations $T_{\boldsymbol{Q}}$ [47]. 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.

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

$$p_{Q_0,Q_1}(q_0,q_1) = p_{Q_0}(q_0)p_{Q_1|Q_0}(q_1 \mid q_0), \tag{3}$$

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

$$p_{\boldsymbol{Q}}(\boldsymbol{q}) = \prod_{d=0}^{D-1} p_{Q'_d}(q'_d), \tag{4}$$

where

$$Q'_d = Q_d \mid Q_0,\dots,Q_{d-1} \qquad q'_d = q_d \mid q_0,\dots,q_{d-1} \tag{5}$$

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

$$T_{\boldsymbol{Q}}(\boldsymbol{q}) = (F_{Q'_0}(q'_0), \ldots, F_{Q'_{D-1}}(q'_{D-1})), \tag{6}$$

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

$$F_{Q'_d}(q'_d) = \int_{-\infty}^{q_d} p_{Q'_d}(r \mid q_0, \ldots, q_{d-1})\mathrm{d}r. \tag{7}$$

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

3.2 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 often non-linear without a closed-form formula, making analytical calculations of the transformation's inverse difficult. In the scenario where we do not have a symbolic 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.

The problem of calculating the inverse transformation $T_{\boldsymbol{Q}}^{-1}$ can, by decomposing the definition of the forward Rosenblatt transformation in (6), 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 transformation 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 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.

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 $r_0 = (up_0 - lo_0)u + lo_0$:

$$r_{k+1} = r_k - \frac{F_{Q'_d}(r_k \mid q_0, \ldots, q_{d-1}) - u}{p_{Q'_d}(r_k \mid q_0, \ldots, q_{d-1})}, \tag{8}$$

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:

$$r_{k+1} = \frac{up_k + lo_k}{2}. \tag{9}$$

In either case, the bounds are updated according to

$$(lo_{k+1}, up_{k+1}) = \begin{cases} (lo_k, r_{k+1}) & F_{Q'_d}(r_{k+1} \mid q_0, \ldots, q_{d-1}) > u \\ (r_{k+1}, up_k) & F_{Q'_d}(r_{k+1} \mid q_0, \ldots, q_{d-1}) < u \end{cases} \tag{10}$$

The algorithm repeats the steps in (8), (9) and (10), until the residual $|F_{Q'_d}(r_k \mid q_0, \ldots, q_{d-1}) - u|$ 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.

3.3 Constructing Distributions

The backbone of distributions in Chaospy is the Rosenblatt transformation $T_{\boldsymbol{Q}}$. The method, as described in the previous 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.

Following the definition in (6), 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 defines a normal variable with a normally distributed mean:

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

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

```
>>> joint = cp.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 (4) 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 `cp.J` function.

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.

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. 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 = cp.J(cp.Normal(0,1), cp.Normal(0,1), cp.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.

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 below. The titles 'D', 'T' and 'C' represents Dakota, Turns and Chaospy respectively. The elements 'y' and 'n' represent the answers 'yes' and 'no' indicating if the distribution is supported or not. The list shows that Dakota's support is limited to 11 distributions, and Turns has a collection of 26 distributions.

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

The Chaospy software supports in addition custom distributions through the function `cp.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 = cp.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 `cp.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 3.2.

3.4 Copulas

Dakota and Turns do not support the Rosenblatt transformation applied to multivariate distributions with dependencies. Instead, the two packages model dependencies using copulas [42]. 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 has 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. The implementation is based of the software toolbox RoseDist [16]. In practice, this approach allow copulas to be defined in a Rosenblatt transformation setting. For example, to construct a bivariate normal distribution with a Clayton copula in Chaospy, we do the following:

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

A list of supported copulas is provided below. 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

3.5 Variance Reduction Techniques

As noted in the beginning of Section 3, 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 very high, so if f is assumed to be computationally expensive, the number of samples needed frequently make Monte Carlo simulation infeasible for practical applications. 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 the tables below, and they show that Dakota, Turns and Chaospy support 4, 7, and 7 variance reduction techniques, respectively.

Quasi-Monte Carlo Scheme	Dakota	Turns	Chaospy
Faure sequence [20]	no	yes	no
Halton sequence [27]	yes	yes	yes
Hammersley sequence [28]	yes	yes	yes
Haselgrove sequence [29]	no	yes	no
Korobov latice [34]	no	no	yes
Niederreiter sequence [43]	no	yes	no
Sobol sequence [52]	no	yes	yes

Other Methods	Dakota	Turns	Chaospy
Antithetic variables [48]	no	no	yes
Importance sampling [48]	yes	yes	yes
Latin Hypercube sampling [39]	yes	limited	yes

One of the more popular variance reduction technique is the *quasi-Monte Carlo scheme* [48]. 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 standard Monte Carlo requires 10^6 samples for a given accuracy,

quasi-Monte Carlo can often get away with only 10^3. Note that this would break some of the statistical properties of the samples [55].

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.

4 Polynomial Chaos Expansions

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

$$\hat{f}(\boldsymbol{x}, t, \boldsymbol{Q}) = \sum_{n \in I_N} c_n(\boldsymbol{x}, t) \boldsymbol{\Phi}_n(\boldsymbol{Q}) \qquad I_N = \{0, \dots, N\}, \tag{11}$$

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 (11), where the polynomials $\{\boldsymbol{\Phi}_n\}_{n \in I_N}$ are orthogonal on a custom weighted function space L_Q:

$$\langle \boldsymbol{\Phi}_n, \boldsymbol{\Phi}_m \rangle = \mathbb{E}[\boldsymbol{\Phi}_n(\boldsymbol{Q}) \boldsymbol{\Phi}_m(\boldsymbol{Q})] = \int \cdots \int \boldsymbol{\Phi}_n(\boldsymbol{q}) \boldsymbol{\Phi}_m(\boldsymbol{q}) p_{\boldsymbol{Q}}(\boldsymbol{q}) \mathrm{d}\boldsymbol{q} = 0, \quad n \neq m. \tag{12}$$

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 [59]. This method is supported by Dakota and Chaospy, but not Turns.

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 (12) is satisfied. This will be the topic of Section 4.1 In Section 4.2 we show how to estimate the

coefficients $\{c_n\}_{n \in I_N}$. Last, in Section 4.7, tools used to quantify uncertainty in polynomial chaos expansions will be discussed.

4.1 Orthogonal Polynomials Construction

From (12) 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 three mentioned software toolboxes support these expansions.

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 arithmetics are ill-posed, making a numerical approach quite a challenge [21]. This section explores the various approaches for constructing polynomial expansions. A full list of methods is found in the table below. 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 [19]	no	no	yes
Discretized Stieltjes [22]	yes	no	yes
Modified Chebyshev [22]	yes	yes	no
Modified Gram-Schmidt [22]	yes	yes	yes

Looking beyond an analytical approach, the most popular method for constructing orthogonal polynomials is the discretized Stieltjes procedure [53]. 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 [60]. As described in detail in Section 3.1, 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 [60].

Dakota, Turns, and Chaospy all support generalized polynomial chaos expansions for independent stochastic variables and the Normal/Nataf copula listed in the table in Section 3.4. Since Chaospy has the Rosenblatt transformation underlying 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 [19]. In the table in Section 4.1, only Chaospy's implementation of Bertran's recursion method [5], Cholesky decomposition [17] and modified Gram-Schmidt orthogonalization [22] support construction of orthogonal polynomials for multivariate dependent densities directly.

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]:

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

The relevant Chaospy code for creating this polynomial expansion looks like

```
>>> q0, q1 = cp.variable(2)
>>> phi = cp.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 the table below.

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

One thing worth noting is that polynomial chaos expansions suffers from the curse of dimensionality: The number of terms grows exponentially with the number of dimensions [57]. As a result, Chaospy does not support neither high dimensional nor infinite dimensional problems (random fields). One approach to address such problems with polynomial chaos expansion is to first reduce the number of dimension through techniques like Karhunen-Loeve expansions [49].

If software implementations of such methods can be provided, the user can easily extend Chaospy to high and infinite dimensional problems.

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 (12). To verify that two elements in `phi` are indeed orthogonal under the standard bivariate normal distribution, one writes

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

More details of operators used to perform uncertainty analysis are given in Section 4.7.

4.2 Calculating Coefficients

There are several methodologies for estimating the coefficients $\{c_n\}_{n\in I_N}$, typically categorized either as non-intrusive or intrusive, where non-intrusive means that the computational procedures only requires evaluation of f (i.e., software for f can be reused 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 [24]. 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.

Within the realm of non-intrusive methods, there are in principle two viable methodologies available: pseudo-spectral projection [26] and the point collocation method [31]. 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.

4.3 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,

$$c_n = \frac{\mathbb{E}[Y\mathbf{\Phi}_n]}{\mathbb{E}[\mathbf{\Phi}_n^2]} = \frac{1}{\mathbb{E}[\mathbf{\Phi}_n^2]} \int \cdots \int p_{\boldsymbol{Q}}(\boldsymbol{q}) f(\boldsymbol{x}, t, \boldsymbol{q}) \mathbf{\Phi}_n(\boldsymbol{q}) \mathrm{d}\boldsymbol{q} \tag{13}$$

$$\approx \frac{1}{\mathbb{E}[\mathbf{\Phi}_n^2]} \sum_{k \in I_K} w_k p_{\boldsymbol{Q}}(\boldsymbol{q}_k) f(\boldsymbol{x}, t, \boldsymbol{q}_k) \mathbf{\Phi}_n(\boldsymbol{q}_k) \qquad I_K = \{0, \ldots, K-1\},$$

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.

The table below shows the various quadrature schemes offered by Dakota and Chaospy (recall that Turns does not support pseudo-spectral projection).

Node and Weight Generators	Dakota	Turns	Chaospy
Clenshaw-Curtis quadrature [9]	yes	no	yes
Cubature rules [54]	yes	no	no
Gauss-Legendre quadrature [25]	yes	no	yes
Gauss-Patterson quadrature [44]	yes	no	yes
Genz-Keister quadrature [23]	yes	no	yes
Leja quadrature [41]	no	no	yes
Monte Carlo integration [48]	yes	no	yes
Optimal Gaussian quadrature [25]	yes	no	yes

All techniques for generating nodes and weights in Chaospy are available through the `cp.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, cp.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]
```

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 [51]. 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 [7]. In addition to the tensor product rules, there are a few native multivariate cubature rules that allow for low order multivariate integration [54]. Both Dakota and Chaospy also support the Smolyak sparse grid and anisotropy.

Chaospy has support for construction of custom integration rules defined by the user. The `cp.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 `cp.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 ]
```

Software for constructing and executing a general-purpose integration scheme is useful for several computational components in uncertainty quantification. For example, in Section 4.1 when constructing orthogonal polynomials using raw statistical moments, or calculating discretized Stieltjes' recurrence coefficients, numerical integration is relevant. Like the `ppf` function noted in Section 3.3, 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 `cp.generate_quadrature` function described above.

To compute the Fourier coefficients and the polynomial chaos expansion, we use the `cp.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 4.1, the nominator $\mathbb{E}\left[\mathbf{\Phi}_n^2\right]$ in (13) can be calculated more accurately using recurrence coefficients [22]. Special numerical features like this can be added by including optional arguments in `cp.fit_quadrature`.

4.4 Point Collocation

The other non-intrusive approach to estimate the 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:

$$\begin{bmatrix} \Phi_0(\boldsymbol{q}_0) & \cdots & \Phi_N(\boldsymbol{q}_0) \\ \vdots & & \vdots \\ \Phi_0(\boldsymbol{q}_{K-1}) & \cdots & \Phi_N(\boldsymbol{q}_{K-1}) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_N \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{q}_0) \\ \vdots \\ f(\boldsymbol{q}_{K-1}) \end{bmatrix}, \tag{14}$$

Unlike pseudo spectral projection, the locations of the collocation nodes are not required to follow any integration rule. Hosder [31] 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 3.5. 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.

The obvious way to solve the over-determined system in (14) is to use least squares minimization, which resembles the standard statistical linear regression approach of fitting a polynomial to a set of data points. 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 [46], 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 numerical stability is a potential problem. The table below lists the regression methods offered by Dakota, Turns, and Chaospy.

Regression Schemes	Dakota	Turns	Chaospy
Basis Pursuit [8]	yes	no	no
Bayesian Auto. Relevance Determination [56]	no	no	yes
Bayesian ridge [37]	no	no	yes
Elastic Net [61]	yes	no	yes
Forward Stagewise [30]	no	yes	no
Least Absolute Shrinkage and Selection [14]	yes	yes	yes
Least Angle and Shrinkage with AIC/BIC [62]	no	no	yes
Least Squares Minimization	yes	yes	yes
Orthogonal matching pursuit [38]	yes	no	yes
Singular Value Decomposition	no	yes	no
Tikhonov Regularization [46]	no	no	yes

Generating a polynomial chaos expansion using linear regression is done using Chaospy's `cp.fit_regression` function. It takes the same arguments as `cp.fit_quadrature`, except that quadrature weights are omitted, and optional arguments define the rule used to optimize (14).

4.5 Model Evaluations

Irrespectively of the method used to estimate the coefficients c_k, 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 like `mpi4py`. Both Dakota and Turns support parallel evaluation of f values, but the feature is embeded into the code, potentially limiting the customization options of the parallelization.

4.6 Extension of polynomial expansions

There is much literature that extends on the theory of polynomial chaos expansion [57]. For example, Isukapalli showed that the accuracy of a polynomial expansion could be increased by using partial derivatives of the model response [32]. This theory is only directly supported by Dakota. In Turns and Chaospy the support is 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 4.1 and 4.2, and the rich set of manipulations of polynomials is then available for $\hat{f}$ as well.

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 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 3.5, 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 [33] or with more specialized statistical libraries like Pandas [40], Scikit-learn [45], Scikit-statsmodel [50], 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 = cp.Normal(0,1).sample(10**5)
>>> samples = f_approx(*q_samples)
>>> 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.

4.7 Descriptive Tools

The last step in uncertainty quantification based on polynomial chaos expansions is to quantify the uncertainty. In polynomial chaos expansion this is done by using the uncertainty in the model approximation `f_approx` as a substiute for the uncertainty in the model 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, samples)
>>> 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 the table below.

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

5 Conclusion and Further Work

Until now there have only been a few real software alternatives for implementing non-intrusive polynomial chaos expansions. Two of the more popular implementations, Dakota and Turns, are both high-quality software that can be applied to a large array of problems. The present paper has introduced a new alternative: Chaospy. Its aim is to be an experimental foundry for scientists. Besides featuring a vast library of state-of-the-art tools, Chaospy allows for a high degree of customization in a user-friendly way. Within a few lines of high-level Python code, the user can play around with custom distributions, custom polynomials, custom integration schemes, custom sampling schemes, and custom statistical analysis of the result. Throughout the text we have compared the built-in functionality of the three packages, and Chaospy do very well in this comparison, which is summarized in the table below. But the primary advantage of the package is the strong emphasis on offering well-designed software building blocks, with a high abstraction level, that can easily be combined to create tailored uncertainty quantification algorithms for new problems.

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

Although the primary aim of the software is to construct polynomial chaos expansions, the software is also a state-of-the-art toolbox for performing Monte Carlo simulation, either directly on the forward model or in combination with polynomial chaos expansions. Variance reduction techniques are included to speed up the convergence, and because Chaospy is based on Rosenblatt transformations, efficient quasi-Monte Carlo sampling is available for any distribution. Another novel feature of Chaospy is the ability to handle *stochastically dependent* input variables through a new mathematical technique.

5.1 Acknowledgement

The work is supported by funding from Statoil ASA through the Simula School of Research and Innovation, and by a Center of Excellence grant from the Research Council of Norway through the Center for Biomedical Computing.

Thanks also go to Vinzenz Gregor Eck, Stuart Clark, Karoline Hagane, Samwell Tarly, and a random distribution of unnamed bug fixers for their contributions.

References

[1] P. Achard and E. De Schutter. Complex parameter landscape for a complex neuron model. *PLoS Computational Biology*, 2(7), 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 and statistics in a structured industrial approach. In *Procedings ESREL 2007 safety and reliability conference.*, 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] 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.

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

[8] 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.

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

[10] Patrick R. Conrad and Youssef M. Marzouk. Adaptive Smolyak pseudospectral approximations. *SIAM Journal on Scientific Computing*, 35(6):A2643–A2670, 2013.

[11] B. J. Debusschere, H. N. Najm, P. P. Pebay, O. M. Knio, R. G. Ghanem, and O. P. Le Maitre. Numerical challenges in the use of polynomial chaos representations for stochastic processes. *SIAM Journal on Scientific Computing*, 26(2):698–719, 2004.

[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. *Annals of Statistics*, pages 1–26, 1979.

[14] B. Efron, T. Hastie, I. Johnstone, and R. Tibshirani. Least angle regression. *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 S. Clark. RoseDist: Generalized Tool for Simulating with Non-Standard Probability Distributions. In J. Piantadosi, R. S. Anderssen, and J. Boland, editors, *International Congress on Modelling and Simulation*, pages 367–372, Adelaide, Australia, 2013. http://www.mssanz.org.au/modsim2013/A7/feinberg.pdf.

[17] J. Feinberg and H. P. Langtangen. Uncertainty Quantification of Diffusion in Layered Media by a New Method Based on Polynomial Chaos Expansion. In H. I. Andersson and B. Skallerud, editors, *Seventh National Conference on Computational Mechanics MekIT'13*, Norwegian University of Science and Technology, 2013. Akademika Publishing.

[18] J. Feinberg and H. P. Langtangen. Chaospy Software Package for Uncertainty Quantification, 2014. \emph{https://github.com/hplgit/chaospy}.

[19] J. Feinberg and H. P. Langtangen. Multivariate Polynomial Chaos with Dependent Variables. *SIAM Journal on Scientific Computing*, 2015. Submitted. URL: *http://bit.ly/1Bkp72S*.

[20] S. Galanti and A. Jung. Low-discrepancy sequences: Monte {C}arlo simulation of option prices. *Journal of Derivatives*, 5(1):63–83, 1997.

[21] W. Gautschi. Construction of {G}auss-{C}hristoffel quadrature formulas. *Mathematics of Computation*, 22(102):251, 1968.

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

[23] A. Genz and B. Keister. Fully Symmetric Interpolatory Rules for Multiple Integrals over Infinite Regions. *Journal of Computational and Applied Mathematics*, 72, 1996.

[24] R. Ghanem and P. D. Spanos. *Stochastic Finite Elements: a Spectral Approach*. Courier Dover Publications, 2003.

[25] G. H. Golub and J. H. Welsch. *Calculation of Gauss Quadrature Rules*, volume 23. Mathematics of Computation, 1967.

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

[27] 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.

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

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

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

[31] 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.

[32] S. S. Isukapalli. Uncertainty analysis of transport-transformation models, 1999.

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

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

[35] P. Laux, G. Jackel, R. M. Tingem, and H. Kunstmann. Impact of climate change on agricultural productivity under rainfed conditions in Cameroon - a 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] S. G. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.

[39] 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.

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

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

[42] R. B. Nelsen. *An Introduction to Copulas*. Springer, 1999.

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

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

[45] 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 {P}ython. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

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

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

[49] S. Sakamoto and R. Ghanem. Polynomial chaos decomposition for the simulation of non-Gaussian nonstationary stochastic processes. *Journal of Engineering Mechanics*, 128(2):190–201, 2002.

[50] 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.

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

[52] 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.

[53] T. J. Stieltjes. Quelques recherches sur la theorie des quadratures dites mecaniques. *Ann. Sci. Ecole Norm. Sup. (3)*, 1:409–426, 1884.

[54] A. H. Stroud. *Approximate Calculation of Multiple Integrals*, volume 431. Prentice-Hall Englewood Cliffs, NJ, 1971.

[55] S. Tezuka. *Uniform Random Numbers: Theory and Practice*. Kluwer Academic Publishers, 1995.

[56] 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.

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

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

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

[60] 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.

[61] 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.

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