Using Python for scientific computing: Efficient and flexible evaluation of the statistical characteristics of functions with multivariate random inputs

R. Chudoba \textsuperscript{a,\ast}, V. Sádílek \textsuperscript{b}, R. Rypl \textsuperscript{a}, M. Vořechovský \textsuperscript{b}

\textsuperscript{a} RWTH Aachen University, Germany
\textsuperscript{b} Brno University of Technology, Czech Republic

\begin{abstract}
This paper examines the feasibility of high-level Python based utilities for numerically intensive applications via an example of a multidimensional integration for the evaluation of the statistical characteristics of a random variable. We discuss the approaches to the implementation of mathematically formulated incremental expressions using high-level scripting code and low-level compiled code. Due to the dynamic typing of the Python language, components of the algorithm can be easily coded in a generic way as algorithmic templates. Using the Enthought Development Suite they can be effectively assembled into a flexible computational framework that can be configured to execute the code for arbitrary combinations of integration schemes and versions of instantiated code. The paper describes the development cycle using a simple running example involving averaging of a random two-parametric function that includes discontinuity. This example is also used to compare the performance of the available algorithmic and executional features. The implemented package including further examples and the results of performance studies have been made available via the free repository \cite{1} and CPCPl library.
\end{abstract}

\begin{keyword}
Python
Enthought traits
NumPy
SciPy
C
Loopless programming
Multidimensional integration
Estimation of statistical moments
\end{keyword}

\section{Program summary}

\textbf{Program title:} spirrid

\textbf{Catalogue identifier:} AENL_v1_0

\textbf{Program summary URL:} \url{http://cpc.cs.qub.ac.uk/summaries/AENL_v1_0.html}

\textbf{Program obtainable from:} CPC Program Library, Queen's University, Belfast, N. Ireland

\textbf{Licensing provisions:} Special licence provided by the author

\textbf{No. of lines in distributed program, including test data, etc.:} 10722

\textbf{No. of bytes in distributed program, including test data, etc.:} 157099

\textbf{Distribution format:} tar.gz

\textbf{Programming language:} Python and C.

\textbf{Computer:} PC.

\textbf{Operating system:} LINUX, UNIX, Windows.

\textbf{Classification:} 4.13, 6.2.

\textbf{External routines:} NumPy (\url{http://numpy.scipy.org/}), SciPy (\url{http://www.scipy.com})

\textbf{Nature of problem:}
Evaluation of the statistical moments of a function of random variables.

\textbf{Solution method:}
Direct multidimensional integration.
1. Introduction

High-level languages for scientific computing offer application programmers a convenient and efficient tool for the formulation and implementation of mathematical models. Examples of widely used high-level software for the rapid prototyping of scientific applications are Maple, Matlab, Octave, R and S+. These tools provide rich documentation, visualization utilities, symbolic operators and a large number of numerical methods. The suitability of the Matlab toolkit for the prototyping of numerical applications and for teaching courses has been discussed e.g. in [2]. Compared to the low-level programming languages like FORTRAN and C or C++ these high-level tools make development more productive, especially at the early stages of application development. On the other hand, such high-level tools can seldom compete with the performance of applications written in compiled languages. The trade-off between flexibility and efficiency is a daily issue for application programmers in the area of scientific computing. An ideal development environment should provide both high productivity at the early stages of development and, at the same time, an easy way to accelerate code execution once the application has reached a mature state.

A particularly appealing development environment has emerged during the last decade in the area of open source software. The scripting language Python has established a platform for developing and integrating two rich numerical libraries: NumPy and SciPy. These libraries embody algorithms and methods developed over the past decades in the compiled languages FORTRAN and C++ (http://www.netlib.org). The flexibility of scripting in Python applied to scientific applications has been thoroughly presented in the textbook [3]. As further documented in [4,5], the high flexibility of the scripting language is not necessarily accompanied by a lower efficiency with respect to the compiled code. Indeed, when implementing mathematical expressions in vectorized form using compact array representation [6], the trade-off between flexibility and efficiency is reduced to an acceptable level [7].

Besides the comparison with the compiled low-level languages, the productivity and efficiency of the Python development environment has been compared with Matlab and Octave using several benchmark examples [8,9]. These studies came to the conclusion that the functionality of the Python based environment is comparable with that of commercial tools. However, in the authors’ opinion, the Python environment has a decisive advantage in one crucial area: it has an open and extensible object model behind its language so that it can evolve further. The possibility of extending the class definition in the language using metaclasses allows the incorporation of design patterns into the language that further increase the productivity of developers. In particular, the Enthought Tool Suite (http://www.enthought.com) has introduced a refined definition of a class using predefined traits [10]. A class trait corresponds to the usual attribute in an object-oriented language but extends it with information on initialization, validation, notification of changes and visualization. Such an extended attribute specification has been used to incorporate the Model-View-Controller design pattern into the language and provide automatic object visualization and dynamic control of state changes with minimum programming effort. This semantically rich language support allows developers to benefit from automatic generation of the user interface directly from the class view specification, persistence of objects, declaration of state dependencies and effective support for data visualization.

This paper reports on the feasibility of the Python-based environment for the development of scientific applications. The concepts, observations and conclusions made here are relevant for readers interested in three disciplines: estimation of statistical moments, programming of scientific computing applications and simulation of fiber bundles and/or of brittle matrix composites. Throughout this paper, the applied implementation concepts are demonstrated on the estimation of statistical moments of the stress–strain response of a set of parallel fibers. This task corresponds to the estimation of the statistical moments of an “elementary” function with random parameters. An example of such a function describing the stress–strain response of a fiber loaded in tension with two independent, identically distributed random parameters for stiffness and strength is shown in Fig. 1, left. Random realizations of the single fiber’s response are displayed in Fig. 1, right. The goal is to efficiently estimate the mean response of a fiber within the bundle, which is plotted as the solid black line in Fig. 1, right.

The implementation of this specific example has been generalized so that it is applicable for the statistical characterization of an arbitrary function with independent random parameters. The implemented framework falls into the domain of multivariate analysis packages, as does pycha [11]. We shall demonstrate that when using the aforementioned implementation environment, the abstract nature of the mathematical formulation can be reflected in the code without any additional performance penalty. The particular motivation of the paper is to

- show the rapid prototyping of an application starting with a simple Python script,
- show how to generalize and scale up the script to a configurable algorithmic object that can incorporate several variants of random sampling and options for the speeding up of the code,
- provide a framework for the verification of results and for studies of execution efficiency for all available features of the implementation.

The paper is organized as follows. First, the symbolic specification of the implemented algorithms is presented in Section 2. After that, simple implementations of the algorithm in pure Python on the one hand and using the NumPy and SciPy packages on the other hand are constructed in Section 3. Next, the algorithm implementation is generalized to incorporate other types of sampling and languages (Section 4). The introduced variants of the algorithm are then merged in the general interactive framework for multidimensional integration in Section 5 using Enthought traits as a glue. The configurable package allows for a thorough comparison of the efficiency of several versions of the sampling types described in
Section 8. The execution efficiency is compared for three types of code generated and compiled on the fly for the current response function and its randomization in Section 9. The code segments provided throughout the paper constitute an executable script included in the spirrid package that is available for downloading from the online CPCPLibrary and in the github.com repository [1].

2. Estimation of statistical moments of a function with independent random variables

The goal is to estimate the statistical moments of a random problem \( Q \) given as a function of a control variable \( \xi \) and of random variables \( \theta_1, \ldots, \theta_m \), constituting the random domain \( \Theta \):

\[
Q = q(\xi; \theta_1, \ldots, \theta_m)
\]

with \( q(\xi; \theta) \) further referred to as a response function. The \( k \)-th raw statistical moment of such a random problem is given as

\[
\mu_k(\xi) = \int_{\Theta} \left[ q(\xi; \theta) \right]^k g(\theta) \, d\theta.
\]

Since only independent random variables are considered here, the joint probability density function \( g(\theta) \) (PDF) of the random vector \( \theta \) can be replaced with the product of univariate marginal densities

\[
\mu_k(\xi) = \int_{\Theta_1} \cdots \int_{\Theta_m} \left[ q(\xi; \theta_1 \cdots \theta_m) \right]^k \prod_{i=1}^{m} g_i(\theta_i) \, d\theta_1 \cdots d\theta_m.
\]

The integration is to be performed numerically as a summation of discrete values distributed over the discretized random domain

\[
\mu_k(\xi) = \sum_{\Theta_1} \cdots \sum_{\Theta_m} \left[ q(\xi; \theta_1 \cdots \theta_m) \right]^k \frac{\Delta G_1(\theta_1) \cdots \Delta G_m(\theta_m)}{\Delta G},
\]

where \( \Delta G(\theta) \) denotes the weighting factors that depend on the particular type of sampling as specified below. The distribution of the integration points \( \Theta_m \) within the random domain can be expressed as

\[
\Theta_i = [\theta_{ij}, j = 1 \cdots n_i], \quad i = 1 \cdots m,
\]

where \( n_i \) is the number of discretization points for the \( i \)-th variable and \( j \) counts the discretization point along a variable. There are many ways to cover the random domain by sampling points. In order to demonstrate the expressive power of the language and to discuss the computational efficiency of the possible implementation techniques we shall implement the integral (3) in four different ways:

I. **Equidistant grid of random variables (TGrid).** The \( i \)-th random variable is covered by regularly spaced values \( \theta_i \). The probability associated with an integration cell is given as

\[
\Delta G_i(\theta_i) = g_i(\theta_i) \Delta \theta_i
\]

with \( \Delta \theta_i \) denoting the spacing between discretization points.

II. **Non-equidistant grid of random variables.** Sampling points are generated through an equidistant grid of sampling probabilities \( \pi_{ij} \) (PGrid). The transformation of sampling probabilities into the random domain \( \Theta \) in Eq. (5) is performed for each random variable enumerated as \( i = 1 \cdots m \) using the inverse cumulative distributions (also referred to as percent point function):

\[
\theta_i = G_i^{-1}(\pi_{ij})
\]

where the values \( \pi_{ij} \) are obtained as

\[
\pi_{ij} = \frac{j - \frac{1}{2}}{n_i}, \quad j = 1 \cdots n_i.
\]

The integration cells rendered through this kind of sampling share the same probability (weight) for each variable:

\[
\Delta G(\theta) = \prod_{i=1}^{m} \frac{1}{n_i}.
\]

This type of sampling is introduced here as a combination of the grid-type of sampling with the concept of the Monte Carlo type of sampling with constant integration cell probability.

III. **Crude Monte Carlo Sampling (MCS).** Instead of using a structured grid of \( \pi_{ij} \) as given in Eq. (8) the sampling probabilities are selected randomly from \( \pi_{ij} \in (0, 1) \) and the corresponding sampling points are selected according to Eq. (7). The underlying data structure of \( \Theta \), arrays can be flattened as the number of sampling points \( n_i = n_{\text{sim}} \) is equal for all random variables and integration can be done in a single loop as

\[
\mu_k(\xi) = \sum_{j=1}^{n_{\text{sim}}} \left[ q(\xi; \theta_{ij}, \ldots, \theta_{mj}) \right]^k \frac{1}{n_{\text{sim}}},
\]

IV. **Latin Hypercube Sampling (LHS).** An enhanced version of Monte Carlo type sampling which uses stratification of the distribution functions of input random variables to ensure uniform coverage of the sampling probabilities [12–14]. An LHS sample tends to be more uniformly distributed through the unit hypercube of sampling probabilities \( \pi_{ij} \) than a Monte Carlo sample. As a result,
the variance of the statistics estimators such as the one in Eq. (11) is usually reduced in comparison with MCS. Indeed, Stein [15] has shown that LHS reduces the variance compared to simple random sampling (crude Monte Carlo). The amount of variance reduction increases with the degree of additivity and monotonicity in the random quantities on which the function q depends.

Similarly as in the case of MCS, the underlying data structure can be flattened and integration can be performed in a single loop as prescribed in Eq. (11). The sampling points in Eq. (7) are obtained through sampling probabilities calculated as:

\[
x_{ij} = \frac{r_{ij}}{n_{\text{sim}}} , \quad j = 1 \ldots n_{\text{sim}}
\]

with \( r_{ij} \) representing an \((m, n_{\text{sim}})\) matrix containing random permutations of a sequence 1, . . . , \( n_{\text{sim}} \) on each row. By using this scheme, the medians of each stratum are selected as sampling points.

Further improvements of the method can be achieved by selecting probabilistic means in each stratum [16]. We remark that the random ordering of sampling probabilities does not ensure an exact unit correlation matrix of simulated data especially for small sample sizes, see [17]. A possible remedy has been proposed in [16] by employing an algorithm diminishing the undesired correlation of the sample.

In order to make the explanation of the implementation illustrative let us introduce a simple two-parametric response function depicted in Fig. 1 (left) as

\[
q(\varepsilon; \lambda, \xi) = \lambda \cdot \varepsilon \cdot H(\xi - \varepsilon).
\]

This function defines the stress for a given positive control strain \( \varepsilon \) of a linear elastic, brittle fiber with the stiffness parameter \( \lambda \) and breaking strain \( \xi \). The symbol \( H(x) \) represents the Heaviside function yielding zero for \( x < 0 \) and one for \( x \geq 0 \). We deliberately chose a function containing discontinuity in order to study the ability of the integration algorithm to reproduce the smoothness of the average response of an ensemble with a constituent response governed by a non-smooth function. If many fibers act in parallel, as is the case in a crack bridge in a composite, they exhibit imperfections. This means that the material parameters \( \lambda \) and \( \xi \) must be considered random. The mean stress–strain behavior of a fiber within a bundle can be obtained using Eq. (4) as

\[
\mu_q(\varepsilon) = \sum_{\theta_\lambda} \sum_{\theta_\xi} q(\varepsilon; \lambda, \xi) \cdot g_\lambda g_\xi \frac{\Delta \theta_\lambda \Delta \theta_\xi}{\Delta \varepsilon}.
\]

The result of this expression is plotted in Fig. 1 (right, black curve). It demonstrates that the average response of the filament is nonlinear. The described integral exhibits the structure of a strain–based fiber bundle model describing the behavior of yarns and composite materials [18]. The authors have used the procedure for modeling the tensile tests of multi-filament yarns [19].

### 3. Scripting implementation of an equidistant integration scheme

In order to demonstrate the possible approaches to the implementation of the numerical model, let us construct a short script delivering the result of Eq. (14). In the first step we define the probability distributions of the parameters \( \lambda \) and \( \xi \) as normal and generate an equidistant discretization of the random domains \( \theta_\lambda \) and \( \theta_\xi \) with 20 values together with the values of marginal densities \( g_\lambda \) and \( g_\xi \).

```python
from scipy.stats.distributions import norm
import numpy as np

# set the mean and standard deviation of la and xi
m_la, std_la = 10.0, 1.0
m_xi, std_xi = 1.0, 0.1

# construct objects representing normal distributions
pdistrib_la = norm(loc=m_la, scale=std_la)
pdistrib_xi = norm(loc=m_xi, scale=std_xi)

# get operators for probability density functions
g_la = pdistrib_la.pdf
g_xi = pdistrib_xi.pdf

# number of integration points set equal for both variables
n_i = 10

# generate midpoints of n_i intervals in the range (-1,1)
theta_arr = np.linspace(-1.0 - 1.0 / n_i, 1.0 - 1.0 / n_i, n_i)
theta_lam = m_la + 4 * std_la * theta_arr
theta_xi = m_xi + 4 * std_xi * theta_arr

# get the size of the integration cells
d_la = (8 * std_la) / n_i
d_xi = (8 * std_xi) / n_i

d_G = pdistrib_la.pdf

def q_eq13(eps, lam, xi):
    return q(eps, lam, xi)

def q_eq14(eps, lam, xi):
    return 

def mu_q_eq14_loops(eps_arr):
    mu_q_arr = np.zeros_like(eps_arr)
    for i, eps in enumerate(eps_arr):
        mu_q = 0.0
        for la in theta_lam:
            for xi in theta_xi:
                dG = g_la(la) * g_xi(xi) * d_xi
                mu_q += q_eq13(eps, la, xi) * dG
        mu_q_arr[i] = mu_q
    return mu_q_arr

# construct an array of control strains
eps_arr = np.linspace(0.1, 1.2, 80)
mu_q_arr = mu_q_eq14_loops(eps_arr)
```

1 Code blocks in Python are introduced by indentation levels.
2 Another option would be to use masked arrays skipping the array values associated with the True value of the logical expression defining the mask. However, as we shall comment in Section 9 this option is connected with significant losses of performance.
This implementation is naive and extremely slow and is introduced here solely for illustrative purposes. The above script is about 1000 times slower than the same code written in the C language. It is obvious that in this form Python code cannot compete with FORTRAN or C compiled code.

Efficient and more competitive Python code can be produced using the numerical package numpy. The basic concept behind speeding up the code is to avoid interpreted loops in the implementation by using the array-based operators of numpy. Consistent with Eq. (14), the terms $Q$ and $ΔG$ shall be calculated separately for all combinations of the parameters within the discretized random domain $Θ = Θ_λ × Θ_ξ$ as two-dimensional arrays and, subsequently, the array-based product and summation operators of numpy are to be applied. The array containing the weight factors $ΔG$ can be conveniently evaluated using the vectorized product operator over two numpy arrays:

$$dG_la = g_la(θ_la) * d_la$$
$$dG_xi = g_xi(θ_xi) * d_xi$$
$$dG_grid = dG_la[:, np.newaxis] * dG_xi[ np.newaxis,:,:]$$

In the first two lines, 45–46, the PDF functions $g_{la}$ and $g_{xi}$ previously constructed at lines 10–11 are reused. However, now the random variables are represented as the arrays $θ_{la}$ and $θ_{xi}$. The result of the call $g_{la}(θ_{la})$ is an array of PDF values for all values in $θ_{la}$. Such a vectorized evaluation is executed completely in the compiled code of the numerical library numpy. The loop over the array elements is not explicitly given in the code. Therefore, this kind of code is often somewhat impractically referred to as loopless.

The implicit loops are also used during the computation of the $ΔG_{grid}$ at line 47 delivering the two-dimensional array of joint integration weights: $ΔG_{i,j}(θ_{λ}, θ_{ξ}) = ΔG_{i}(θ_{λ}) · ΔG_{j}(θ_{ξ}); ∀θ_{λ} ∈ Θ_λ, ∀θ_{ξ} ∈ Θ_ξ$. Line 47 demonstrates the use of index operators in numpy to construct an outer product of two arrays in order to avoid slow Python loops. The machinery behind the implicit loops uses two concepts:

- A new dimension (np.newaxis) with the length of 1 was added to the one-dimensional arrays $dG_{la}$ and $dG_{xi}$ so that they became two-dimensional column (dG_{la}[: , np.newaxis]) and row (dG_{xi}[np.newaxis,:]) matrices, respectively. Note that the index operator [:] stands for a slice, i.e. all values along that dimension. Thus, the construct dG_{xi}[:, np.newaxis] reshapes an array and makes it open for operations with operands possessing the second dimension.

- The orthogonal shape of the arrays $dG_{la}[:, np.newaxis]$ and $dG_{xi}[np.newaxis,:]$ is used for “broadcasting” the data across dimensions, a concept illustrated in Fig. 2: If a column matrix $A$ of the shape $(m, 1)$ is multiplied with a row matrix $B$ of the shape $(1, n)$ the resulting matrix will have the shape $(m, n)$. The nonexistent columns of $A$ and nonexistent rows of $B$ are augmented with the values from the first column and row, respectively, so that $a_{i,j-1} = a_{i,j}$ and $b_{i-1,j} = b_{i,j}$. The described technique is applicable in multiple dimensions:

$$dG_grid = dG_{t1}[: , np.newaxis,np.newaxis] * \ 
            dG_t2[ np.newaxis, ; , np.newaxis] * \ 
            dG_t3[ np.newaxis, np.newaxis,:]$$

It significantly increases the speed of the code compared to pure Python due to the vectorized evaluation of the algorithm and is more than sufficient for the prototyping phase of application development. Further speedup is possible as shall be discussed later in Section 9.

Having constructed the array with probabilistic weights $ΔG$ we may approach the array-based implementation of the response function $q(ε; λ, ξ)$ given in Eq. (13). We remark again that instead of using the control blocks if and else as in the listing (lines 27–29) to introduce discontinuity (i.e. to distinguish whether or not the fiber is broken) a simple implementation of the Heaviside function has been used (lines 23–25). In this form, the function parameters $λ$ and $ξ$ can have both scalar or array values so that a “vectorized” evaluation of the function is possible. The use of if and else would only permit scalar parameters and no array-based, loopless evaluation would be possible.

In analogy to the probabilistic weights $ΔG$, the values $Q$ of the response function can be obtained in a loopless manner using the broadcasting concept. The expression $q(ε, θ_{la}[:, np.newaxis], θ_{xi}[np.newaxis,:])$ evaluates the response function for all combinations of $λ$ and $ξ$ and stores them in a two-dimensional array. Thus, the expression in Eq. (14) previously implemented using loops at lines 34–39 can be reimplemented in a loopless form in the method

```python
def mu_q_eq14(eps):
    """Loopless calculation of mean value."""
    q_grid = q_eq13(eps, theta_1a[:, np.newaxis], theta_xi[ np.newaxis,:])
    q_dG_grid = q_grid * dG_grid
    return np.sum(q_dG_grid)
```

Note that q_grid and $dG_{grid}$ have the same shape $(n_i, n_i)$ so that the variable q_dG_grid calculated at line 53 contains element-by-element products of the response function values and associated probabilistic weights. At line 54, summation over all dimensions is performed to obtain the mean value.

By issuing for example the call mu_q_eq14(1.0) one obtains the mean response for the control variable 1.0, see the circle in Fig. 1, right. In order to get the stress for multiple values of the strain in a single call, the function mu_q_eq14 can be vectorized\(^3\) to permit an array as an input parameter:

```python
mu_q_eq14_vct = np.vectorize(mu_q_eq14)
mu_q_arr = mu_q_eq14_vct(eps_arr)
```

\(^3\) The np. vectorize method performs a repeated evaluation of the supplied function over an array in a Python loop. Due to the lower efficiency of the vectorized operator it should only be used in outer loops of an application. For inner loops, either implicitly vectorized operators of numpy combined with broadcasting, or acceleration packages like cython or weave discussed later should be used.
Finally, plotting can be done using the matplotlib package. We shall use the matplotlib-like interface for matplotlib called
pylab:

```python
import pylab as p
p.plot(eps_arr, mu_q_arr)
p.show()
```

This code produces the diagram shown in Fig. 1 (right) with the mean response shown as a black curve. The example demonstrates the
prototyping step in the development of a numerical application. The script is limited to a particular response function and predefined randomization
pattern.

4. Generalization for other sampling schemes

The abstract nature of the mathematical formulation (Eq. (4))
permits the use of an arbitrary type of the response function
$q(\epsilon; \theta)$ and probability distributions $g_i(\theta_i)$. It is desirable to
preserve this flexibility of the mathematical model also in its
implementation. The script implemented in the previous section uses the equidistant discretization given in Eq. (5). In order to
implement the other three discretizations of the random domain
mentioned in Section 2 we shall capture the discretization-

independent part of the code in a generic algorithmic template.
The following code exploits the fact that functions can be treated
as variables in Python and defines the integration procedure
without specifying the particular type of data structure for the
sampling/integration points:

```python
def get_mu_q_fn(q_eq13, 1.0 / n_sim, *theta):
    """Return a method evaluating the mean of q()."""
    def mu_q(q_eps):
        Q_dG = q(q_eps, *theta) * dG
        return np.sum(Q_dG)
    def mu_q_fn(eps):
        mu_q_arr = mu_q_fn(eps_arr)
    return np.vectorize(mu_q)

The function get_mu_q_fn instantiates the generic template for the
integral in Eq. (14) and/or Eq. (11) for an arbitrary response
function q and the list of variables *theta and dG. The code does not make any assumption about the particular type of input variables
and thus it can be used to instantiate the integration procedures for all four sampling schemes introduced in Section 2. In this form, the code captures the algorithmic
commonalities of the integration methods and remains open for
further specializations:

1. **TGrid sampling.** The previously constructed values of the $\Delta G$ and $\theta_i$ are reused to instantiate the template and calculate the results as

```python
# SAMPLING:
# ... reuse dG_grid and theta (lines 18, 19 and 45–47)
# INSTANTIATION:
mu_q_fn = get_mu_q_fn(q_eq13, dG_grid,
    theta['la'][:,:np.newaxis],
    theta['xi'][np.newaxis,:])
# CALCULATION:
mu_q_arr = mu_q_fn(eps_arr)
```

Note that in contrast with the first implementation (line 52), the
broadcasting is now performed outside of the generic integration
function at line 73 to keep the integration template open for a
flattened sampling data structure.

2. **PGrid sampling.** The integration algorithm based on a grid of
constant sampling probabilities can be instantiated using the code

```python
# SAMPLING:
# equidistant sampling probabilities (see Eq. 8)
j_arr = np.arange(1, n_i + 1)
pi_arr = (j_arr - 0.5) / n_i
# use ppf (percent point function) to get sampling points
# (pdistrib_la and pdistrib_xi was defined at lines 7, 8)
theta['la'] = pdistrib_la.ppf(pi_arr)
theta['xi'] = pdistrib_xi.ppf(pi_arr)
# get the total number of integration points
# for 2 random variables with equal n_i
n_sim = n_i ** 2

# INSTANTIATION:
mu_q_fn = get_mu_q_fn(q_eq13, 1.0 / n_sim,
    theta['la'][::-np.newaxis],
    theta['xi'][np.newaxis,:])

# CALCULATION:
mu_q_arr = mu_q_fn(eps_arr)
```

Here, broadcasting is used again at line 92 and instead of a grid of
weighted parameters (dG_grid) only a single value $\Delta G = 1/n_{sim}$
is used for instantiation.

III. **MCS sampling.** The integration template can also be used for
the non-regular discretization used in Monte Carlo sampling.
For a comparison between grid sampling and the Monte Carlo
types of sampling we shall consider an equal number of function
evaluations, so that $n_{sim} = \prod_{i=1}^m n_i$, where $n_i$ represents the
number of values along each of the $m$ dimensions in the grid
discretizations.

```python
# SAMPLING:
# generate n_sim random realizations
# using pdistrib objects (lines 7, 8)
theta['la_rvs'] = pdistrib['la_rvs'](n_sim)
theta['xi_rvs'] = pdistrib['xi_rvs'](n_sim)

# INSTANTIATION:
mu_q_fn = get_mu_q_fn(q_eq13, 1.0 / n_sim,
    theta['la_rvs'], theta['xi_rvs'])

# CALCULATION:
mu_q_arr = mu_q_fn(eps_arr)
```

The $\Delta G$ value is again constant. The integration code is now instan-
tiated with flat arrays of sampled values.

IV. **LHS sampling.** The Latin Hypercube Sampling is constructed
using a perturbation of $n_{sim}$ values of $\theta_i$ obtained using the percent
point function.

```python
# SAMPLING:
# sampling probabilities (see Eq. 12), n_sim as above
j_arr = np.arange(1, n_sim + 1)
pi_arr = (j_arr - 0.5) / n_sim
# get the ppf values (percent point function)
# using pdistrib objects defined at lines 7, 8
theta['la_ppf'] = pdistrib['la_ppf'](pi_arr)
theta['xi_ppf'] = pdistrib['xi_ppf'](pi_arr)
# make random permutations of both arrays to diminish
# correlation (not necessary for one of the random variables)
theta['la'] = np.random.permutation(theta['la_ppf'])
theta['xi'] = theta['xi_ppf']

# INSTANTIATION:
mu_q_fn = get_mu_q_fn(q_eq13, 1.0 / n_sim,
    theta['la'], theta['xi'])

# CALCULATION:
mu_q_arr = mu_q_fn(eps_arr)
```
Fig. 3 shows the results for the four instantiated versions of the algorithm. For the sampling schemes using structured grid (TGrid, PGrid) \( n_{int} = 10 \) integration points are used in each direction. For the Monte Carlo types of sampling (MCS, LHS) the same number of sampling points, \( n_{sim} = 10^2 \), is used.

Let us summarize that all four examples could be realized using the generic template `get_mu_q` implemented at lines 61–66. Due to the use of implicit loops over numpy arrays the template code can handle both grid-based and Monte Carlo types of sampling. This documents the fact that using loopless code the algorithmic structure could be captured in an abstract way without strong assumptions about the particular representation of the sampling data.

5. Interactive algorithmic object

While the previous section was focused on capturing the structural commonalities of the algorithmic procedure in the generic code of the `get_mu_q_fn` template (lines 61–66), in this section we intend to generalize the code for the specification of the random problem, generation of sampling data and include also loop-based code as an alternative to the vectorized code described so far.

In the context of our running example, we want to avoid the part of code that is entitled as SAMPLING in the four implementations given in the previous section. At the same time, we want to represent the random problem, the sampling methods and the execution code in independent code entities so that they can be combined interactively.

The specified functionality has been provided in the form of an interactive algorithmic object. The random problem can be defined and interactively configured using the SPIRRID class by supplying an arbitrary response function and by specifying its control and random parameters. The complete list of attributes defining a general multi-variant random problem includes the following items:

- \( q \): defines the response function. It is either a function or a "callable" object possessing the (\( \cdot \)) call operator. An example is a standard Python function
  
  ```python
  def q(name1, name2, name3, name4, ...)
  ```

  Parameter names are used for further specification of control and random variables included either in the `eps_vars` or `theta_vars` attributes.

- `eps_vars`: specifies the control variables as a dictionary of (name, value) pairs

  ```python
  eps_vars = {‘name1’ : value_arr1,
              ‘name2’ : value_arr2}
  ```

  where `value_arr` defines the evaluation range of the control variable.

- `theta_vars`: specifies random variables as a dictionary of (name, value) pairs

  ```python
  theta_vars = {‘name3’ : RV(distr3, loc3, scale3, n_i3),
                ‘name4’ : RV(distr4, loc4, scale4, n_i4)}
  ```

  where `value_objects` constructed as `RV(distr_type, loc, scale, n_i)` define the random variables by specifying the type of probability distribution (`distr_type`) and its parameters `loc`, `scale` and the number of integration points `n_i` in the respective dimension of the random domain.

- `sampling_type`: within the random domain is specified using the option `sampling_type = [‘TGrid’, ‘PGrid’, ‘MCS’, ‘LHS’]`. By choosing the type of sampling, the corresponding arrays of \( \Theta \) values and the associated statistical weights \( \Delta G \) are generated for the integration algorithm.

- `codegen_type`: is selected using option `codegen_type = [‘numpy’, ‘weave’, ‘cython’]`. Based on this selection the integration code is generated on the fly for the currently configured random problem and sampling either using the vectorized numpy package or compiled loop-based C code produced using the weave or cython packages.

As an application example, let us reproduce the random problem defined previously at lines 108–123:

```python
from stats.spirrid import SPIRRID , RV
# DEFINITION: random problem, sampling type, code type
s = SPIRRID(q=eq13,
            eps_vars={‘eps’ : np.linspace(0, 1.2, 80)},
            theta_vars={‘la’ : RV(distr=’norm’, loc=10,
                                scale=1.0, n_i=10),
                        ‘xi’ : RV(distr=’norm’, loc=1,
                                 scale=0.1, n_i=10)}
```
6. Implementation of the SPIRRID class using traits

After a brief description of the user view of the SPIRRID class, let us shortly explain the approach to its implementation. The structure of the code is depicted in Fig. 4 in the form of a UML class diagram [20]. The diagram indicates the decomposition of the problem into three parts:

randomization: representing the specification of the random problem in terms of the response function and classification of its parameters into control and random.

sampling: delivering the sampling data \( \theta \) and \( dG \) for the given randomization.

code generation: providing the instantiated execution method either as a vectorized numpy code or as a compiled C code generated with the help of weave or cython packages.

Our Python implementation of the class diagram uses the traits package provided within the Enthought Tool Suite (http://www.enthought.com). This package introduces an extended attribute definition into Python classes by including specification of the attribute’s type and of its dynamic behavior. In our experience, traits can significantly contribute to a compact implementation of an object-oriented application since the implemented static class structure can be easily enhanced with the specification of state transitions involved in the life-cycle of an interactive object.

Let us demonstrate the implementation approach on a part of the code showing the definition of the classes RV, Function Randomization and SPIRRID depicted in Fig. 4:

```python
from traits.api import Import, Callable, Dict, 
from code_gen import 
from stats.pdistrib import PDistrib

class FunctionRandomization(HasTraits):
    # ... ...
    distr = Property(depends_on='loc , scale , shape')
    n_i = Int(40 , desc='number of integration points')
    loc = Float(0.0 , desc='location parameter')
    scale = Float(0.0 , desc='scale parameter')
    shape = Float(1.0 , desc='shape parameter')
    # probability distribution
    distr = Property(depends_on='loc , scale , shape')
    @cached_property
    def _get_distr(self):
        # Distr class wraps the scipy.stats.distributions
        return PDistrib(type = self.distr_type, loc = self.loc ,
                        scale = self.scale , shape = self.shape)

    # operators delegated to scipy.stats via PDistr
    pdf = DelegateTo('distr')
    cdf = DelegateTo('distr')
    ppdf = DelegateTo('distr')

    class FunctionRandomization(HasTraits):
        q = Callable(rp_spec=True)
        eps_vars = Dict(Str, Array, rp_spec=True)
        theta_vars = Dict(Str, RV, rp_spec=True)
        @on_trait_change('+ rp_spec')
        def _validate_random_problem(self):
            # Prepare the input data structure.
            # ... skipped here ...

    class SPIRRID(FunctionRandomization):
        sampling_type = Trait(TGrid, PGrid)
        randomization = Trait('TGrid', 'PGrid', 'MCS', 'LHS')
        sp_spec = True
        sampling_type = Trait('TGrid', 'PGrid', 'MCS', 'LHS')
        sp_spec = True
        sampling_type = Trait('TGrid', 'PGrid', 'MCS', 'LHS')
        sp_spec = True
        sampling_type = Trait('TGrid', 'PGrid', 'MCS', 'LHS')
        sp_spec = True
        sampling_type = Trait('TGrid', 'PGrid', 'MCS', 'LHS')
        sp_spec = True
```

This loop produces a diagram showing the response obtained using the four sampling schemes and prints the execution times needed for the included types of sampling. In an indicated way, data on convergence and execution efficiency characterizing particular configurations of the algorithmic object can be gathered, processed and plotted. This is performed later in Sections 8 and 9 providing a systematic performance comparison of the discussed sampling methods on the one hand and of the implementation type, either vectorized, loopless or compiled, loop-based on the other hand.
Remark 2 (Representation of Random Variables). Random variables are introduced using the class RV shown at lines 159–178. The class consists of traits specifying the type of probability distribution distr_type, its parameters loc, scale, shape and the number of integration points n_i. Except for these input traits, RV class includes a property trait for the probability distribution distr. The distr property is defined as a @cached_property which means that there is a hidden attribute of a type PDistrib that gets automatically constructed upon first access to distr. If any of the traits listed in the depends_on specifier of the property has changed, the hidden PDistrib instance gets reconstructed upon the next access to the distr attribute by invoking the associated “getter” method _get_distr(). The last three traits (pdf, cdf, ppf) of the RV class defined as DelegatesTo traits are proxies accessing methods of the distr trait. These methods are used during the generation of the sampling data in the subclasses of Sampling class.

Remark 3 (Specification of Multi-Variate Random Problem). Specification of the random problem including the response function, control variables and random variables is defined and managed by the FunctionRandomization class (lines 180–189). Based on the given control variables eps_vars and random variables theta_vars the response function q is inspected for the matching names of parameters. Note that these traits are grouped together by specifying the rp_spec = True metadata in the trait constructor. The +rp_spec specifier can then be used for compact specification of state dependencies. One way how to introduce a dynamic dependency is by using the decorator on_trait_change('+rp_spec'). This is exemplified at line 186 in order to trigger the validation method _validate_random_problem() whenever any of the q, eps_vars and theta_vars traits has been modified. Another example of dependency specification is provided for a lazy update of the list of control parameters in the cached property eps_1st (line 192) providing the input arrays in an order that corresponds with the specification given in the response function definition.

Remark 4 (Integration of Algorithmic Features). The SPIRRID class defined at lines 198–232 inherits from the FunctionRandomization and integrates the algorithmic features for sampling types (sampling_type) and code types (codegen_type) as trait attributes containing the lists of available sampling methods and code generators, respectively. Depending on the current value of sampling_type and codegen_type the instances of the corresponding class given in Fig. 4 are provided through the sampling and codegen properties, respectively. In the associated “getter” methods, an instance of the sampling type (line 211) or code generator (line 222) is constructed and provided with a backward reference to the SPIRRID object (self). Using this reference, the algorithmic feature can extract information on the ran-
Remark 5 (Instantiation of the Algorithm and of the Results). The property trait mu_q_arr for the array of mean values (line 224) depends on attributes categorized as +rp_spec, +sp_spec and +cg_spec. In its getter method, the control variables for the parameters eps_lst are first arranged in an orthogonal array using the make_ogrid method at line 229. As a consequence, for more than one control variables, the broadcasting rules discussed previously in Section 4 in the context of the random domain apply. This means that for problems with more than one control variables (eps_vars) the mean values are automatically calculated for all combinations of their values. The particular calculation method mu_q_fn is instantiated for the current random problem and sampling method at line 230 upon the access to the equally named property of the current code generator codegen. The obtained vectorized function mu_q_fn accepts a list of control variables and runs the calculation.

In order to explain what is meant with the code generator let us briefly describe the definition of CodeGenNumpy class:

```python
from code_gen import CodeGen
class CodeGenNumpyCodeGen():
    # Code generator for vectorized numpy code.
    mu_q_fn = Property(depends_on='spirrid,rp_spec,sp_spec')
    @cached_property
    def _get_mu_q_fn(self):
        # Return a method evaluating the mean of q().
        s = self.spirrid
        # construct dictionary of random variables
        theta_args = dict(zip(s.theta_var_names, s.sampling.theta))
        # merge theta and eps dictionaries
        args = dict(theta_args.items(), eps_args.items())
        # calculate the product
        Q_dG = q(**args) * spirrid.sampling.dG
        return np.sum(Q_dG)

    def mu_q(*eps):
        # construct dictionary of control variables
        eps_args = dict(zip(s.etc.tuple_names, eps)
        # merge theta and eps dictionaries
        eps_args = dict(eps_args.items(), theta_args.items())
        # calculate the product
        Q_dG = q(**args) * spirrid.sampling.dG
        return np.vectorize(mu_q)
```

The implementation of the getter method _get_mu_q_fn mimics the get_mu_q_fn template defined previously in Section 3 at lines 61–66. One significant difference, however, is that the getter method _get_mu_q_fn furnishes the generated mu_q_fn method with a sequence *eps of vectorized control parameters and, thus, supports broadcasting. Another difference is that the sampling data theta and dG are not passed as an argument of the instantiated mu_q_fn function but are accessed via the sampling attribute of the containing SPIRRID instance. As a consequence, a change of the sampling method is automatically reflected by the re-instantiation of the calculation method. Note that the generated method is stored as a cached hidden value and is reused if there was no change in its dependencies.

In our opinion the shown implementation of the CodeGenNumpy class provides a very compact and short code including a high flexibility with respect to the specification of the random problem and to the choice of the particular sampling method. The ingredients contributing to this kind of implementation are the dynamic typing of the Python language, notification mechanism introduced by the traits package and the broadcasting technique for vectorized implementation provided by the numpy package. Later in Section 9 we shall also comment on the implementation of other generator classes (CodeGenCython and CodeGenWeave) producing compiled loop-based C code.

7. User interface for an algorithmic object

As already mentioned, the SPIRRID class is prepared for interactive use either through a command line or through a user interface. For a class defined using traits, the user interface can be constructed by specifying views to the object using the traits.ui package included in the Enthought suite. Views are defined declaratively by specifying lists of traits to be included in the user interface. For example, a view to the RV class (lines 159–178) can be defined and rendered into the UI window shown in Fig. 5 as follows:

```python
from traits.ui import View, Item

v = View(Item('distr_type', label='Distribution type'),
         Item('loc', label='Location'),
         Item('scale', label='Scale'),
         Item('shape', label='Shape'),
         title='Random variable',
         buttons=['OK', 'Cancel'])

e = RV(distr_type='norm', scale=10, shape=1)
e.configure_traits(view=v)
```

A more complex user interface is exemplified in Fig. 6 for an interactive SPIRRID instance. The example reflects the previously introduced computational components: (1) the response function, (2) specification of random parameters, (3) statistical distribution, (4) execution control, (5) configuration and (6) plotting of the calculated mean curves as movable tabs within the user interface window.

Besides views, traited classes also have an attached Controller class responsible for handling interaction events. By default, there is an automatically constructed view and controller object for each traited class instance. This means that even at the initial stages of development a simple user interface is automatically available. In the later stages, the views and controllers can be easily refined into a form suitable for a particular type of application.

The declarative mapping of class traits to a view makes it possible to define a user interface to an application without sticking to a particular UI library. Indeed, a switch between different types of user interface toolkits is done by switching between the UI backends. Currently, the back-ends for the wx and Qt libraries are available. Further details of the UI mapping machinery would go beyond the scope of the present paper and can be found in [21].
8. Computational efficiency of the implemented sampling schemes

The implemented algorithmic object shall now be used for performance studies comparing the efficiency of the available algorithmic features. The studies have been implemented as scripts interacting with a single SPIRRID instance.

The convergence of the integration algorithm for the available sampling schemes shall be evaluated using an analytical solution available for our running example. The exact mean response of the two-parametric response function given in Eq. (13) with both parameters considered random and normally distributed can be obtained as:

$$\mu_{q}^{\text{exact}}(\varepsilon) = \frac{\mu_{\lambda} \varepsilon}{2} \left(1 - \text{erf}\left(\frac{\varepsilon - \mu_\xi}{\sqrt{2} \sigma_\xi}\right)\right),$$

where \(\text{erf}(x)\) denotes the Gauss error function defined as \(\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt\). The convergence of the sampling schemes used for the numerical estimation of the mean has been studied depending on the number of sampling points and on executional time. The error has been defined both locally and globally. Since the estimate of the statistical characteristics of the peak value is of special significance, the local error measure has been defined as the relative maximum deviation along the control variable \(\varepsilon\) discretized using \(n_\varepsilon\) points:

$$e_{\text{max}} = \max_i\left|\mu_q(\varepsilon_i) - \mu_{q}^{\text{exact}}(\varepsilon_i)\right| / \max_i\left[\mu_{q}^{\text{exact}}(\varepsilon_i)\right], \quad i = 1 \cdots n_\varepsilon.$$  (16)

The global error measure has been introduced as the relative root mean square error within the range of the control variable \(\varepsilon\):

$$e_{\text{rms}} = \left[\sum_{i=1}^{n_\varepsilon} \left(\mu_q(\varepsilon_i) - \mu_{q}^{\text{exact}}(\varepsilon_i)\right)^2 / \max_i\left[\mu_{q}^{\text{exact}}(\varepsilon_i)\right] - \min_i\left[\mu_{q}^{\text{exact}}(\varepsilon_i)\right]\right]^{1/2}, \quad i = 1 \cdots n_\varepsilon.$$  (17)

Fig. 7 shows the convergence behavior in double logarithmic scale for both types of errors for an increasing number of sampling points. Both diagrams document the fact that \(LHS\) covers the random domain in a significantly more efficient way than all the other implemented methods. The convergence behavior of the other three sampling methods is comparable.

A more relevant comparison of efficiency is given in Fig. 8 which shows the measured CPU time for the studied sampling methods instead of \(n_{\text{sim}}\). Also in this view, the \(LHS\) method is superior to the other methods. \(P\)\(Gr\)\(id\) sampling is revealed to be slightly more efficient than Monte Carlo and \(T\)\(Gr\)\(id\). The higher efficiency is due to the fact that during the response function evaluation, some interim results can be reused in the broadcasted dimension and, consequently, a significant number of operations can be saved. Such a caching of interim values within a random subspace is not possible in the Monte Carlo type of sampling due to the unstructured coverage of the \(m\)-dimensional space. The positive effect of vectorized evaluation increases as the number of random parameters grows. However, it also depends on the type of the response function and on the current choice of the random parameters. It must be also considered that the vectorized evaluation of the response function \(q(\varepsilon)\) in the \(m\)-dimensional space is connected with the exponential growth of memory consumption. Further studies of convergence for varied sampling types and response functions, also including all permutations of
randomized parameters, have been provided in the documentation of the spirrid package [1].

9. Execution efficiency of vectorized and compiled code

Previous studies shown in Fig. 8 reveal that the CPU time required to achieve sufficiently accurate results ($e_{\text{rms}} \leq 10^{-3}$) for a function with two random variables using the numpy code is less than 1 s. For more complex functions with more random variables further speedup might be desirable. There are several ways to improve the execution efficiency of Python code. An overview of speedup possibilities for numerical Python code has been provided in [22].

In order to accelerate the present application two options have been used: cython [23] and weave [24]. They have been incorporated into the spirrid package analogically to the numpy code generator as subclasses of the CodeGen class (see Fig. 4) providing the callable method `mu_q_fn` as a cached property. In comparison to the CodeGenNumpy described in Section 6 the generation of the code in CodeGenWeave and CodeGenCython is more complex and lengthy as it involves the string based assembly of the code, its subsequent compilation and integration into the running application.

The generation for loop-based cython code is sketched in Fig. 9. The codegen object in the middle box assembles the loops over the control and random domains, inserts the access operator to the particular values in the theta and dG arrays, inserts the call to the response function q for current theta value and, finally, inserts the expression for the weighted sum of the obtained response value. Let us remark that the code of the response function q must be provided in a form integrable into the code generator, i.e. both for cython in the form of Python code extended with type declarators and for weave code as a string in the C language.

Execution times have been measured for the two-parametric function in Eq. (13) of our running example with two types of sampling: LHS and PGrid. The size of the sample for both sampling schemes was chosen in order that a comparable level of accuracy was reached based on the studies of sampling efficiency shown earlier in Fig. 8. The accuracy required for the studies was $e_{\text{rms}} \leq 5 \cdot 10^{-5}$, which corresponds to $n_{\text{sim}} = 440^2$ for LHS and $n_{\text{sim}} = 5000^2$ for PGrid. CPU time was measured for three phases of the computation:

- sampling time needed for preparing the arrays $\Theta_i$ (Eq. (5)) and $\Delta G_i$ within the sampling object,
- time needed to prepare and compile the method `mu_q_method` within the codegen object, and
- time needed to execute the integration procedure on the prepared data arrays.

Each version of the code was executed twice in order to show the difference between CPU time with and without the setup and compilation of the generated C code. CPU time obtained for LHS shown in Fig. 10 (left) reveal that a significant amount of time was spent on the permutation of $\Theta$ arrays. For all three types of code a standard permutation algorithm available in the numpy package was used so that the sampling time remained constant ($\approx 0.19$ s) in all runs.
The shortest execution time was achieved in the second run of the compiled weave code. The CPU time achieved by the cython version of the code was only insignificantly slower. This is not surprising since both compiled versions lead to an optimized C code with a similar structure. As already stated in [22] the efficiency of weave and cython can be regarded as equivalent. The overall CPU time of the scripting numpy version was about 2.5 times longer than for both compiled versions. Regarding the time required by the pure integration procedure, the compiled code is 12 times faster than the scripting code. This relation is in agreement with the studies published in [22].

Even though the grid-based sampling schemes are significantly slower than LHS it is interesting to examine the effect of compilation on the speedup for the PGrid sampling shown in the right diagram of Fig. 10. Regarding the CPU time required by the pure integration (numpy: 18.45 s, weave: 3.78 s) we can see that the speedup factor (≈5) is much smaller than in the case of LHS (≈12). As already discussed at the end of Section 8, the vectorized evaluation of the response function can increase the efficiency of numpy compared to the nested loop implementation in weave and cython due to the caching of interim results during broadcasting from a smaller subspace to the m-dimensional domain. Even though the increase in efficiency is not sufficient to compensate for the slower convergence of the sampling, it is helpful to keep this issue in mind when implementing scripts for the grid-based accumulation of values.

In the simple example used for this study, the setup and compilation time of the first run was longer than the time spent on computation. For larger problems with a more complex response function, more random variables and more sampling points, the proportion of time spent on setup and compilation would certainly diminish. A more detailed analysis of performance considering further factors affecting efficiency would go beyond the scope of this paper. Studies of more complex functions with different combinations of random variables have been included in the spirrid package. For the sake of completeness, let us remark that further speedup of the code would be possible using code parallelization and/or more advanced treatment of discontinuities in the response function. Regarding parallelization, an approach suggested in [25] exploiting the dynamic and generic nature of the Python language is a possible choice.

The effect of discontinuity representation deserves more detailed comment: The generic implementation of grid based sampling schemes (PGrid and TGrid) for compiled code uses nested loops. The code of the response function is simply inserted into the innermost loop over the random domain. This means that if the function body contains the control statements if, else in the innermost loop over the random domain. This means that if the function body contains the control statements if, else test does not necessarily occur directly at the intermediate level of iteration over its values. Performing the test in the innermost loop causes unnecessary passes through sampling points that would be skipped if the test occurred outside of the inner loop. For special cases it might be desirable to manually factor out the test to some higher level of iteration avoiding unnecessary passes through the inner loops. This case is not treated in the present implementation focused primarily on the flexibility of the algorithmic structure with interactively selectable integration variables. An expression analysis of the function would be necessary to provide an optimum placement of the control statements within the embedded iteration loops to achieve a general solution to this problem.

A similar problem occurs for the numpy code using the Heaviside function. Sampling points with zero Heaviside function values could be immediately skipped to avoid further calculation with zeros. A possible remedy might be the use of masked arrays provided by the numpy package. However, in our case the extra cost connected with access to values in a masked array turned out to be higher than the savings gained by skipping some operations in the masked sampling points.

10. Conclusions

In the present paper we have reported on the feasibility of the Python based environment for the development of scientific computing applications. We have gone through the whole development life-cycle including mathematical formulation, prototyping
using simple scripts, generalization of the algorithmic structure, designing an algorithmic object reflecting the state dependencies between the editable components and speeding up of the code to achieve the efficiency of low-level compiled code. The resulting algorithmic object for statistical integration can be interactively edited by modifying the response function, declaring its parameters as control or random variables, choosing and configuring probability distributions of the random variables, selecting from four types of sampling schemes and configuring the execution code for the integration.

During the implementation of the sampling schemes, an interesting possibility emerged covering the random domain using a regular grid with constant probabilities (PGr.i.d.). To the knowledge of the authors, this type of sampling has not yet been mentioned elsewhere. This type of sampling becomes interesting in connection with the vectorized evaluation as its efficiency grows with the increasing dimension \( m \) of the integration domain.

The vectorized evaluation of functions using implicit loops over \( m \)-dimensional arrays is sometimes referred to as loopless programming. It makes the scripting code in the early stages of prototyping very short and compact, which contributes to its scalability in the later stages of development. At the same time, the performance of the loopless code for calculations of moderately sized problems is within an acceptable range. For larger problems, including complex response functions with several random variables, it can even reach the level of optimized loop-based inline C code. Further improvements of the loopless code are possible using techniques that accelerate algebraic operations with large vectors and arrays [26]. Further possibilities for speedup are provided by high-level libraries for vectorized programming using graphical processors [27] or library for optimization and evaluation of mathematical expressions involving multidimensional arrays [28].

The use of general numerical libraries in connection with the trained class provided by Enthought traits greatly simplifies the later stages of application development by alleviating the formulation of the interfaces both for the scripting interaction and for the construction of the user interface. The significant advantage of the Enthought environment is its multi-platform usage capability. It is very easy to extend the application with visualization features (matplotlib, mayavi), a graphics interface and persistence management of the data. It is a powerful environment for scientific computations based on the Python high-level programming language. The script constructed throughout the paper and the trained implementation of the spirrid package based on the Enthought traits library has been made available for downloading and testing through the Computer Physics Communications Program Library.

Acknowledgments

The work of the first and third author was supported by the Deutsche Forschungsgemeinschaft (DFG project no. CH/276/2-1 and SFB532/T05). The work of the second and fourth author was supported by the Czech Science Foundation (GAČR grant no. GAP105/11/1551). The work of the last three authors was additionally supported by project no. FAST-S-12-1804. This support is gratefully acknowledged.

References


