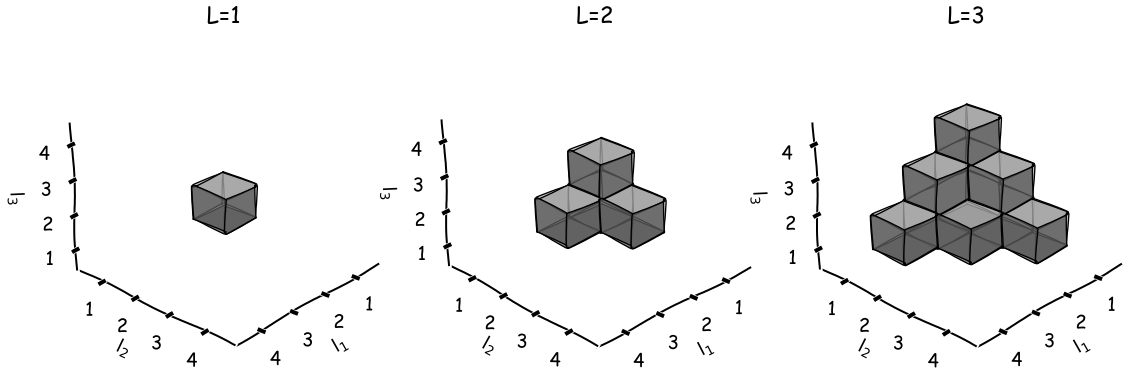
# Adaptive sparse-grid tutorial

Wouter Edeling
Centrum Wiskunde & Informatica, Scientific Computing group

March 18, 2022




Version 0.11

# Contents



# 1 Introduction

The general problem we consider is that of **forward uncertainty propagation** and building **surrogate models** of expensive computer codes. Here, let $f^{(d)}(\mathbf{x}) = f^{(d)}(x_1, \cdots x_d)$ be the output of some (expensive) computer model, as a function of $d$ input parameters $x_i$. In many practical cases, the knowledge we have about these inputs is imperfect, i.e. there is uncertainty in the input values for the $x_i$. In this case it makes sense to gauge the effect of this uncertainty on the output of the model. We will model the input uncertainty by prescribing independent probability density functions (pdfs), such that the total joint pdf

can be written as;

$$p(\mathbf{x}) = \prod_{i=1}^{d} p(x_i). \tag{1}$$

In other words, we replace the deterministic inputs variables with random variables, distributed as $x_i \sim p(x_i)$. The assumption of input independence is very common, although mainly made for convenience as it will simplify the analysis. In this tutorial we will stick to input pdfs of the form (1) for the sake of simplicity, although we note that methods do exist which are able to handle dependent inputs, see e.g. [15]. It should further be noted that the choice of inputs pdf (e.g. Gaussian, uniform, lognormal etc) has to be specified by the user. This can be difficult in its own right, and should be seen as a modelling task. From here on we assume that $p(\mathbf{x})$ is given (perhaps based on expert knowledge[1]), and investigate the corresponding uncertainty in the output $f^{(d)}(\mathbf{x})$. To do so we must propagate samples from $p(\mathbf{x})$ forward through the model in order to approximate the mean, variance or the entire pdf of the output $f^{(d)}(\mathbf{x})$. Monte Carlo (MC) sampling is a well-known method here. The advantage of MC sampling is its independence with respect to $d$, i.e. we can always apply this method regardless of the number of input parameters in play. A downside is the slow convergence rate, the error decreases with the square root of the number of code evaluations. Since many modern computer codes are very compute intensive (and are likely to require access to supercomputers), alternative methods have been developed. We will focus on Stochastic Collocation (SC) and Polynomial Chaos expansions [23]. These methods have the potential of exponential convergence, under certain constraints, therefore requiring significantly less samples from (expensive) computer codes. An additional advantage is that, besides estimating output uncertainty, these methods also build a cheap (polynomial) approximation of the computer code (i.e. a surrogate model) in the process.

One of the constraints of SC and (non-intrusive) PCE methods is the so-called curse of dimensionality. As discussed later, this essentially means an exponential increase in the number of required code evaluations with increasing input size $d$. This limits the use of 'standard' SC and PCE methods to low $d$ (typically no more than $d \approx 5$). However, this tutorial is on so-called sparse-grid methods [18], through which the SC (or PCE) method can be applied to higher-dimensional input spaces. Note that this certainly is not the first sparse-grid tutorial, see for instance [8]. However, our focus is on dimension-adaptive sparse-grid methods, which attempt to iteratively identify important (combinations of) inputs such that only these may be refined with additional samples. The aim here was to write in a clear manner, building the intuition behind the mathematics involved, and giving implementation details were useful. The tutorial starts with $d = 1$, after which the case for $d \gg 1$ is described. We also discuss post-processing of the results, in particular obtaining statistical moments and sensitivity indices. We close with practical examples using the EasyVVUQ toolkit [17] for uncertainty propagation.

[1]Instead, if finding the best input distribution based on available data is the goal, (Bayesian) inverse methods can be used.

# 2 Interpolation in 1D

Let $f(x)$ be a function with a single scalar input $x$. We aim to create an interpolation of $f$, denoted by

$$I^{(l)}f(x) = \sum_{i=1}^{m_l} f(x_i^{(l)})a_i^{(l)}(x). \tag{2}$$

Here, $l$ is the so-called **level** of the interpolant. It is an index which links to a **one-dimensional set of collocation points** $\{x_i^l\}$, for $i = 1, \cdots, m_l$. The higher the level, the more points $x_i^{(l)}$ are used in the construction of (2). These 1D collocation points are the central building blocks of Stochastic Collocation (SC) methods. Typically these points are the abscissas of some chosen quadrature rule. For instance, a Clenshaw-Curtis (CC) rule can generate different collocation points in $[0, 1]$ such that

- Level 1: $x_i^{(1)} \in \{0.5\}$,
- Level 2: $x_i^{(2)} \in \{0.0, 0.5, 1.0\}$,
- Level 3: $x_i^{(3)} \in \{0.0, 0.146, 0.5, 0.854, 1.0\}$.

Note that here, the higher level sets include all points from previous levels. When this is the case, we say that the quadrature rule is **nested**. This leads to efficient sampling plans in higher dimensions, but nestedness is not a strict requirement for sparse grid interpolation or integration. This does lead to 1D quadrature rules which increase exponentially (with some exceptions, e.g. Leja quadrature [13]), such that the number of points is given by

$$m_l = \begin{cases} 2^{l-1} + 1 & l > 1 \\ 1 & l = 1 \end{cases} \tag{3}$$

Finally, the $a_i^{(l)}(x)$ are the basis functions used to interpolate the code outputs $f(x_i^{(l)})$ to upsampled input locations $x$. Note that we will only interpolate to points $x$ that are within the support of the input pdf $\mathrm{p}(x)$, i.e. we will not extrapolate in the input space. In the case of the SC method, the basis consists often (although not necessarily) of the Lagrange interpolation polynomials, given by

$$a_i^{(l)}(x) = \prod_{\substack{1 \le j \le m_l \\ j \neq i}} \frac{x - x_j}{x_i - x_j} \tag{4}$$

Note that the product loops over the $m_l$ collocation points generated by the chosen quadrature rule. A property of the Lagrange polynomial associated with the i-th collocation point is that $a_i^{(l)}(x_i) = 1$ at this point, and $a_i^{(l)}(x_j) = 0$ at all other collocation points $x_j$. When examining (2) we see that the interpolation $I^{(l)}f^{(1)}$ will therefore exactly reproduce the code outputs $f^{(1)}(x_i)$ at the collocation points $x_i$.

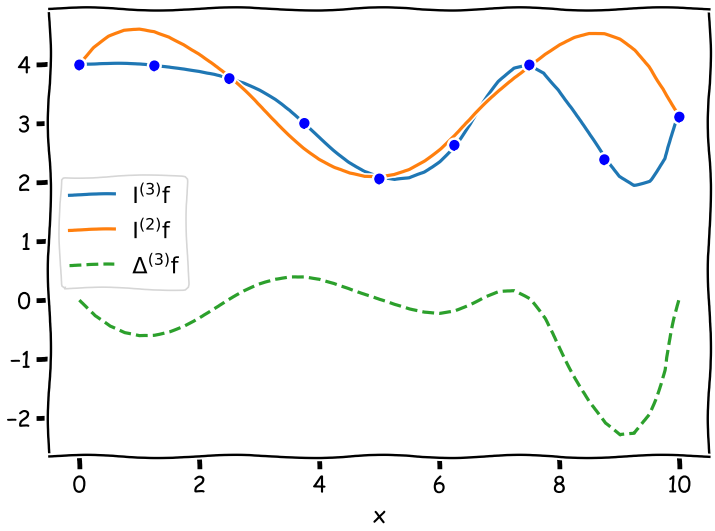



Figure 1: A level 3 one-dimensional interpolant, a level 2 interpolant and the corresponding difference formula.

Let us define the following difference formulas for interpolation in 1D:

$$\Delta^{(l)}f := I^{(l)}f - I^{(l-1)}f \quad \text{where} \quad I^{(0)}f^{(1)} := 0. \tag{5}$$

That is, $\Delta^{(l)}f$ is just the difference between the interpolations at successive levels, see Figure 1. These difference formulas are often used in sparse-grid interpolation (and quadrature). When interpolating a level $l-1$ interpolant using a level $l$ interpolating we retrieve the former, i.e.

$$I^{(l)}\left(I^{(l-1)}f\right) = I^{(l-1)}f. \tag{6}$$

Thus, a lower level interpolant can be exactly interpolated by a higher level interpolant (which is true for both nested and non-nested collocation points), see Figure 2 for an illustration. We can therefore write the difference formula as

$$\Delta^{(l)}f = I^{(l)}f - I^{(l)}\left(I^{(l-1)}f\right) = \sum_{i=1}^{m_l} f(x_i^{(l)})a_i^{(l)} - \sum_{i=1}^{m_l} I^{(l-1)}f^{(1)}(x_i^{(l)})a_i^{(l)} =$$
$$\sum_{i=1}^{m_l}\left[f(x_i^{(l)}) - I^{(l-1)}f(x_i^{(l)})\right]a_i^{(l)} = \sum_{i=1}^{m_l} s_i^{(l)}a_i^{(l)}. \tag{7}$$

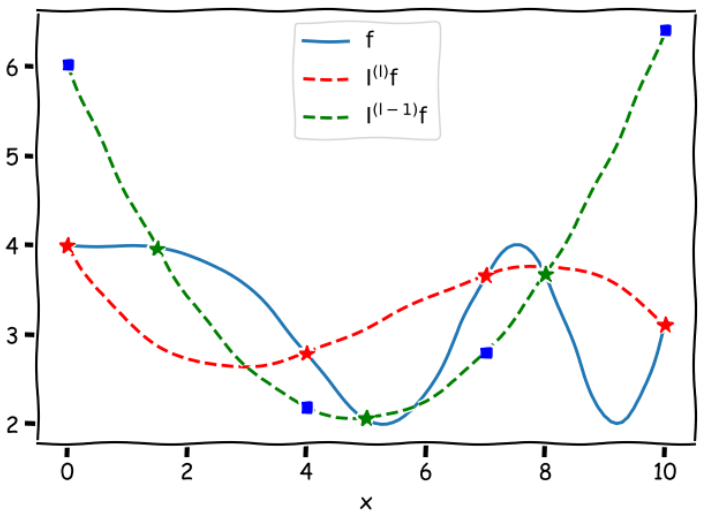



Figure 2: The blue solid line is the exact 1D function $f$. The red striped line is a (non-nested) level $l$ interpolant. The green striped line is a level $l-1$ counterpart, constructed from function evaluations at the collocation points $x_j^{(l-1)}$ denoted by the green star symbols.. If we evaluate $I^{(l-1)}f$ at $x_j^{(l)}$, we obtain the blue squares. Clearly, if these function values are used to construct a level $l$ interpolant of $I^{(l-1)}f$, which would be $I^{(l)}\left(I^{(l-1)}f\right)$, we just retrieve $I^{(l-1)}f$.

Here, $s_i^{(l)} := f(x_i^{(l)}) - I^{(l-1)}f^{(1)}(x_i^{(l)})$ is the **hierarchical surplus**, defined as the difference between the code output at a collocation point $x_i^{(l)}$ at level $l$, minus the level $l-1$ polynomial approximation of the code output at the same location. This can be thought of as a local measure of the accuracy of the interpolation. Furthermore, we can also write the interpolation (2) in terms of the difference formulas, in which case the $a_i^{(l)}$ basis function will form a hierarchical basis over different levels;

$$I^{(L)}f = \sum_{l=1}^{L} \Delta^{(l)}f = \sum_{l=1}^{L}\sum_{i=1}^{m_l} s_i^{(l)} a_i^{(l)} \tag{8}$$

Hence, to obtain a level $L$ interpolant, we can just create a telescopic sum of the difference formulas, which is the first equality of (8). The second equality is obtained by simply plugging in (7), and it shows that the $a_i^{(l)}$ form a hierarchical basis due to summation over $l$, which increases the number collocation point $m_l$ at every new level. This is also sketched in Figure 3, which assumes linear basis functions for simplicity.

Note that (8) is the same as (2), as both are 1D interpolation formulas, only written differently. In (2), the level $l$ is fixed, and the coefficients of the interpolant are just the code evaluations $f(x_i^{(l)})$. On the other hand, (8) builds the interpolant *from the ground up*, starting at $l = 1$. In this case, the coefficients become the hierarchical surplus coefficients $s_i^{(l)}$,

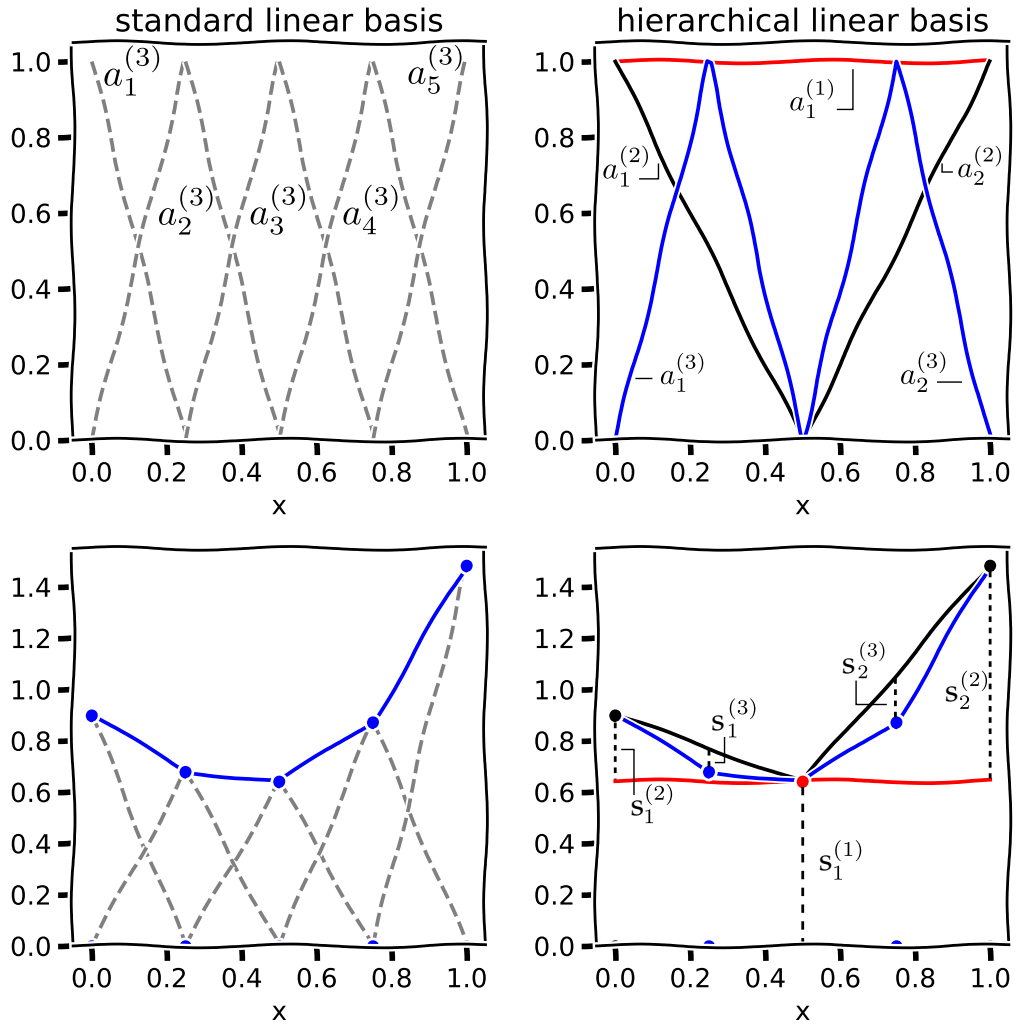



Figure 3: Top left is a standard linear (finite-element) interpolation basis, with below it a corresponding linear interpolant. Top right displays a linear hierarchical basis. Below it we show a series of 3 hierarchical interpolants ($L = 1, 2$ and 3) and the hierarchical surplus coefficients $s_i^{(l)}$ used in their construction. Figure recreated from [14].

dependent upon as the code output at level $l$, and the corresponding interpolant at the previous level. One advantage of adopting the notation of (8) is that the concept of **refinement**, i.e. adding more points to an existing sampling plan, is build into the summation over increasing levels. Also, note that if we select a nested quadrature rule, adding another level means we only have to evaluate the code at a relatively small number of 'new' points, the other points are quaranteed to be present in one or more of the previous levels, see the CC example at the beginning of this section. A non nested rule will in general generate a larger number of unique new points, especially in the case of more than one input parameter.

# 3 Extension to higher dimensions

This section deals with the extension of the preceding analysis to higher input dimensions. First, we will give a broad algorithmic description of sparse-grid sampling, followed by a section on the mathematics behind sparse-grid interpolation for high-dimensional input problems.

## 3.1 Algorithm and sparse-grid classification

If we have $d > 1$ input variables, the 1D interpolation and quadrature rules can be extended to $d$ dimensions via the use of tensor products. To do so we make use of $d$-dimensional **multi indices** $\mathbf{l} := (l_1, l_2, \cdots, l_d)$. Here, the $l_i \geq 1$ entries are integers which represent the level of a 1D quadrature rule, assigned to the i-th variable. For instance, in 2 dimensions, $\mathbf{l} = (l_1, l_2) = (2, 2)$, assigns a level 2 rule to the both inputs. If the level 1 rule generates the point $x_i^{(1)} \in \{0.5\}$, and the level 2 rule gives $x_i^{(2)} \in \{0.0, 0.5, 1.0\}$, the multi index $\mathbf{l} = (2, 2)$ is associated with the tensor product $\{0.0, 0.5, 1.0\} \otimes \{0.0, 0.5, 1.0\}$, which yields sampling plan shown on the left of Figure 4. A sampling plan constructed from a single (uniform) multi index is typical for the *standard* SC method. Here the user will have to specify the order that is used for every input. If the user selects a 1D rule consisting of $m$ points, the total number of points in $d$ dimensions will be $m^d$ (provided that the same rule is used for all inputs). Since we must evaluate our (potentially expensive) model at each point, this exponential increase in the cost (with $d$) is known as **the curse of dimensionality**. Typically, the standard SC method is therefore not used much beyond $d = 5$.

However, we do not have to restrict ourselves to a single multi-index. Instead, consider the set $\Lambda = \{(1, 1), (1, 2), (2, 1)\}$. Keeping the same 1D points as before, the sampling plan created from the combination of all 3 $\mathbf{l} \in \Lambda$ is shown in the middle of Figure 4. This simply means combining the points from $\{0.5\} \otimes \{0.5\}$, $\{0.5\} \otimes \{0.0, 0.5, 1.0\}$ and $\{0.0, 0.5, 1.0\} \otimes \{0.5\}$. Note that even though we are combining multiple tensor products, the result is a sampling plan which is more **sparse** (5 instead of 9 points) compared to the standard SC case with

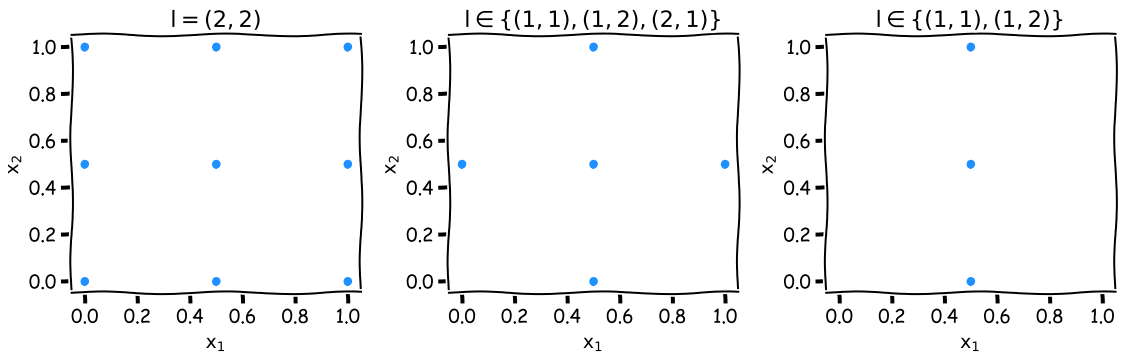



Figure 4: Various 2D sampling plans created from a tensor product of 1D quadrature points.

$\Lambda = \{(2,2)\}$. Because the 1D rule is nested, some 2D points from the different tensor products will overlap, which helps to create a more sparse sampling plan. This is especially true for higher input dimensions.

Even though $\Lambda = \{(1,1),(1,2),(2,1)\}$ creates a 'sparse grid', it is still an **isotropic** multi index set. This means that each input is treated the same, which results in a sampling plan that is still symmetric (see again the middle plot of Figure 4). More precisely, isotropic grids of a given level $L$ are described by the set of all multi indices which satisfy

$$|\mathbf{l}|_1 - d + 1 \leq L, \tag{9}$$

where $|\mathbf{l}|_1 = l_1 + l_2 + \cdots + l_d$. It is instructive to visualize this set in 2D, see Figure 5. Note that condition (9) selects a 'triangle' of multi indices in the $(l_1, l_2)$ domain. Generalizing this to higher dimensions implies that (9) selects a 'simplex' set of multi indices, see Figure 6 for a 3D example.

Even though isotropic sparse grids generate less points than standard SC grids, in high dimensions the number of points can still become very large. For instance at $L = 5$, an isotropic sparse grid based on CC nodes consists of 801 points for $d = 5$. Doubling the number of inputs to $d = 10$ yields a sparse grid with 8801 points, roughly a ten-fold increase. Hence, depending on the cost of a single code evaluation and the available computational budget, isotropic sparse grids much beyond $d = 10$ can become a computational bottleneck, see [3] for a counting method for the number of points in sparse grids.

A potential remedy for this is to use **adaptive, anisotropic** sparse-grid methods, which originate from [9, 10]. If for instance input $x_1$ does not have a significant impact on the output of the code, it makes sense to not refine the grid in that direction. An example of this is shown on the right side of Figure 4, where $\mathbf{l} = (2,1)$ is excluded from $\Lambda$. Of course,

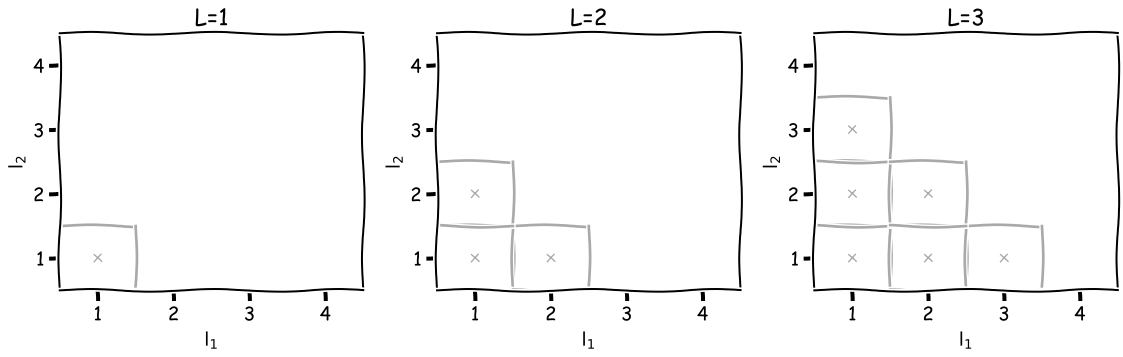



Figure 5: Isotropic multi-index sets of level 1, 2 and 3 for $d = 2$.

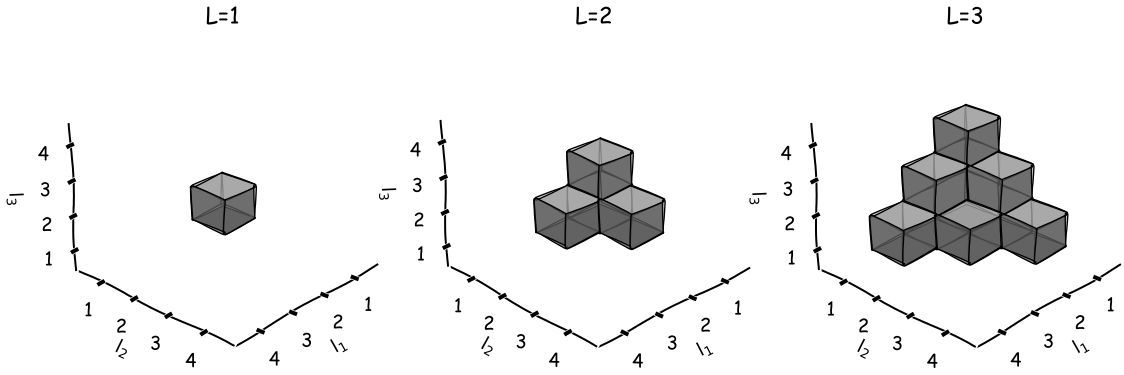



Figure 6: Isotropic multi-index sets of level 1, 2 and 3 for $d = 3$.

we cannot assume that we always know which inputs are influential *a priori*, but we can try to find out on-the-fly in an adaptive manner. Note however that it is easy to construct a function in which each input is equally important (e.g. $f(\mathbf{x}) = \sum_i x_i^2$), in which case adaptive sparse-grid methods hold no advantage over isotropic sparse grids. However, in practise we often observe that the output variance of a simulation code is dominated by a relatively small subset of the input parameters. In this case we say that the problem has a low-dimensional **effective dimension** that we wish to exploit through adaptive sampling.

The basic idea is to start with a level 1 rule for all inputs, and to adaptively rank order (combinations of) inputs, keeping all ineffective inputs at a low level, while increasing the level of those that are effective. Thus, we start with $\Lambda := \{(1, \cdots, 1)\}$, placing just a single sample in the $d$-dimensional input domain. Now, let the **forward neighbours** of all $\mathbf{l} \in \Lambda$, defined by the set $\{\mathbf{l} + \mathbf{e}_i \mid 1 \leq i \leq d\}$, where $\mathbf{e}_i$ is the elementary basis vector in the i-th direction, e.g. $\mathbf{e}_2 = (0, 1, 0, \cdots, 0)$. The *new* forward neighbours of the set $\Lambda$ are then the forward neighbours for all $\mathbf{l} \in \Lambda$, which are not already in $\Lambda$. Similarly, the **backward neighbours** of $\mathbf{l}$ are given by $\{\mathbf{l} - \mathbf{e}_i \mid l_i > 1, 1 \leq i \leq d\}$. An index set $\Lambda$ is said to be **admissible** if all backward neighbours of $\Lambda$ are in $\Lambda$. Finally, at any given iteration of the adaptive sampling process, the **new admissible forward neighbours define potential directions in which to refine the grid**.

As there are quite a number of definitions here, let us look at a 2D example (Figure 7). At the first iteration $\Lambda = \{(1, 1)\}$, which has two new forward neighbours ($(1, 2)$ and $(2, 1)$), both of which are admissible. Each of these are admissible because all their backward neighbours (in this case only $(1, 1)$) are in $\Lambda$. The intuition behind this definition of admissibility lies in the hierarchical construction of the interpolant, i.e. in the use of difference formulas, which are $\Delta^{(l)} f := I^{(l)} f - I^{(l-1)} f$ for the 1D case (see (8)). As discussed later, these difference formulas are also applied in higher input dimensions. Clearly, to compute the 1D difference formula, one not only needs the level $l$ interpolant, but also its backward neighbour at $l - 1$. Generalizing this to $d > 1$ means that in order to construct a hierarchical interpolant based on difference formulas, all backward neighbours for every $\mathbf{l} \in \Lambda$ need to be available, which explains the stated definition of an admissible multi index set.

In order to create an anisotropic grid (not equal in all directions), we only accept one multi index from the admissible set into $\Lambda$ per iteration. In Figure 7 this was $(2, 1)$, which means refinement along the $x_1$ direction. Since we now have $\Lambda = \{(1, 1), (2, 1)\}$, a new forward neighbour ($(3, 1)$) has become admissible. Note that in iteration 2, $(2, 2)$ is also a forward neighbour of $\Lambda$. It is not admissible however, as one of it backward neighbours ($(1, 2)$) is not (yet) part of $\Lambda$. In iteration 3 we refine $x_1$ once more, and only in the 4th iteration do we refine $x_2$ for the first time. Once $(1, 2)$ is part of $\Lambda$, $(2, 2)$ does become admissible. If selected in some later iteration, this would entail *simultaneous* refinement of both $x_1$ and $x_2$. Hence, the algorithm is not limited to one-at-a-time refinement. We can refine along lines, planes, cubes and hypercubes, as long as these are subspaces of the $d$-dimensional input space.

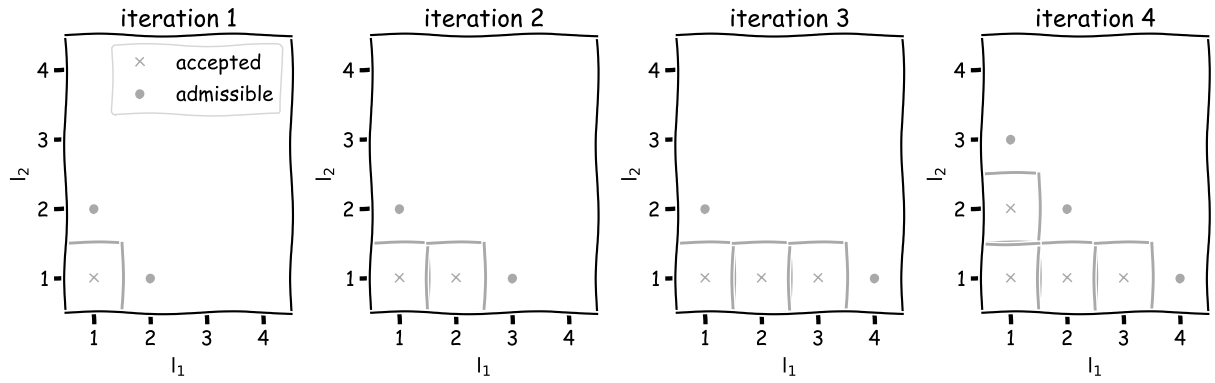



Figure 7: An admissible progression of the adaptive sampling algorithm.

Remember that we only wish to refine the 'important' parameters, which means we must rank order the admissible multi indices. This is done by computing an error measure for each admissible multi index, after which we only include the multi index with the largest error into the accepted set $\Lambda$. A common choice for the error measure is based on the hierarchical surplus, defined under (7) for the 1D case. In general, for a given (new, admissible) $\mathbf{l}$ it reads;

$$s\left(\mathbf{x}_\mathbf{j}^{(\mathbf{l})}\right) := f\left(\mathbf{x}_\mathbf{j}^{(\mathbf{l})}\right) - I^{(\Lambda)} f\left(\mathbf{x}_\mathbf{j}^{(\mathbf{l})}\right), \quad \mathbf{x}_\mathbf{j}^{(\mathbf{l})} \in X_\mathbf{l} \backslash X_\Lambda. \tag{10}$$

Here, $\mathbf{x}_\mathbf{j}^{(\mathbf{l})}$ is a new sampling point generated by $\mathbf{l}$, i.e. $\mathbf{x}_\mathbf{j}^{(\mathbf{l})} \in X_\mathbf{l} \backslash X_\Lambda$, where $X_\Lambda$ is the current accepted grid generated by $\Lambda$, and $X_\mathbf{l}$ is the updated grid generated by $\Lambda \cup \{\mathbf{l}\}$. Hence, the surplus is the difference between the code output $f$ and the interpolant $I^{(\Lambda)} f$ (constructed from the current index set $\Lambda$, as defined later), evaluated at new points that were not used in the construction of $I^{(\Lambda)} f$. An error measure can then be defined by simply taking the mean of all normed surplus values generated by $\mathbf{l}$, e.g. [13];

$$e^{(\mathbf{l})} := \frac{1}{\#(X_\mathbf{l} \backslash X_\Lambda)} \sum_{\mathbf{x}_\mathbf{j}^{(\mathbf{l})} \in X_\mathbf{l} \backslash X_\Lambda} \| s\left(\mathbf{x}_\mathbf{j}^{(\mathbf{l})}\right) \|_2. \tag{11}$$

Note that there is no restriction on the dimension of the code output, while $f$ (and $s$) could be scalar or a vector-valued output, (11) will always be a scalar error measure. It should be noted that there are other error measures which do not involve the surplus. For instance, we can use a quadrature-based error measure that selects the $\mathbf{l}$ which yields the largest change in the variance of $f$. Finally, note that error measures involve the code output $f$, which means that the obtained sampling plan is *conditional on the choice of the code output*. If we for instance let $f$ be a velocity profile from a turbulent flow simulation, the grid will get refined along input dimensions to which the velocity is sensitive. Other outputs, e.g. those

related to the pressure, could well be sensitive to other input parameters, making the grid less suitable for analysis on these outputs.

The overall steps of the adaptive sampling algorithm are shown in pseudocode below, and the next section describes the mathematics behind the algorithm in more detail.

**Algorithm 1** Adaptive sparse-grid sampling

```
Λ ← (1, 1, ··· , 1)
Evaluate f at X_Λ
n ← number of iterations
while i ≤ n do
    Compute new admissible l
    Evaluate f at new points X_l \ X_Λ
    For all admissible l, compute e^(l)
    Λ ← l*, where l* = max_l e^(l)
end while
```

## 3.2 Interpolation in multiple dimensions

As mentioned in the preceding section, tensor products of 1D quadrature rules are used in the case of multiple inputs. For the standard SC method, which has a multi index set $\Lambda$ with just a single multi index $\mathbf{l} = (l_1, l_2, \cdots, l_d)$, the $d$-dimensional equivalent of (2) is

$$I^{(\mathbf{l})} f = I^{(l_1)} \otimes \cdots \otimes I^{(l_2)} f := \sum_{j_1=1}^{m_{l_1}} \cdots \sum_{j_d=1}^{m_{l_d}} f\left(x_{j_1}^{(l)}, \cdots, x_{j_d}^{(l)}\right) a_{j_1}^{(l)}(x_1) \otimes \cdots \otimes a_{j_d}^{(l)}(x_d) \qquad (12)$$

A 2D example of the tensor-product interpolation polynomial $a_{j_1}^{(l)} \otimes a_{j_2}^{(l)}$ is shown in Figure 8. In particular, it shows the 1D Lagrange polynomials (4) corresponding to the 3rd and 4th quadrature node of a level 2 CC rule, and the corresponding 2D tensor product $a_3^{(2)}(x_1) \otimes a_4^{(2)}(x_2)$. In the 1D case $a_3^{(2)}(x) = 1$ at its 'own' quadrature point $x_3^{(l)}$ and zero at the others, and likewise for $a_4^{(2)}$. Note from Figure 8 that in 2D something similar occurs, as $a_3^{(2)}(x_1) \otimes a_4^{(2)}(x_2) = 1$ at $(x_1, x_2) = (x_3^{(l)}, x_4^{(l)})$, and zero at the other quadrature points $(x_{j_1}^{(l)}, x_{j_2}^{(l)})$. Hence, in general (12) is exactly interpolatory, i.e. it reduces to the code output $f\left(x_{j_1}^{(l)}, \cdots, x_{j_d}^{(l)}\right)$ at the collocation points.

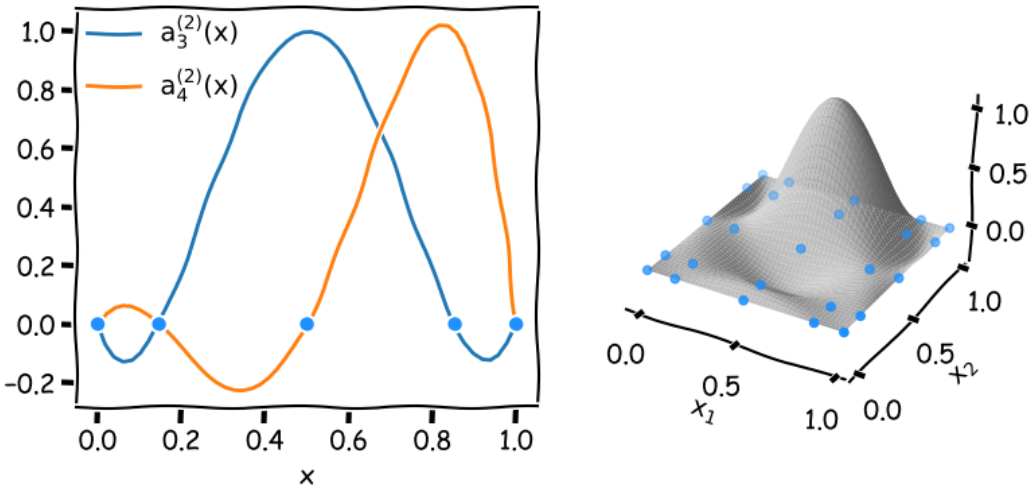



Figure 8: Left: 2 one-dimensional Lagrange interpolation polynomials, defined on 5 CC quadrature points in $[0, 1]$ (a level 2 grid in 1D). Right: the 2D extension of the 1D Lagrange polynomials: $a_3^{(2)}(x_1) \otimes a_4^{(2)}(x_2)$. The blue dots represent the collocation points.

Whether we are using isotropic or adaptive anisotropic grids, we can use a common expression for the interpolation in $d$ dimensions. In particular, the $d$-dimensional equivalent of (8) is given by

$$I^{(\Lambda)} f = \sum_{\mathbf{l} \in \Lambda} \Delta^{(l_1)} \otimes \cdots \otimes \Delta^{(l_d)} f. \tag{13}$$

As noted before, the only difference between isotropic and adaptive sparse grids is the way in which multi indices are added to $\Lambda$, for the discussion here the distinction is not important. The definition of the difference formulas in (13) remains unaltered, except they now appear in a tensor product construction. So in 2D, any given term can be expanded as

$$\begin{aligned} \Delta^{(l_1)} \otimes \Delta^{(l_2)} f &= \left(I^{(l_1)} - I^{(l_1-1)}\right) \otimes \left(I^{(l_2)} - I^{(l_2-1)}\right) f \\ I^{(l_1)} \otimes I^{(l_2)} f - I^{(l_1)} \otimes I^{(l_2-1)} f - I^{(l_1-1)} \otimes I^{(l_2)} f + I^{(l_1-1)} \otimes I^{(l_2-1)} f. \end{aligned} \tag{14}$$

Here, each final term is a separate (standard) SC tensor product similar to (12), e.g.;

$$I^{(l_1)} \otimes I^{(l_2-1)} f^{(2)} = \sum_{j_1=1}^{m_{l_1}} \sum_{j_2=1}^{m_{l_2-1}} f\left(x_{j_1}^{(l_1)}, x_{j_2}^{(l_2-1)}\right) a_{j_1}^{(l_1)}(x_1) \otimes a_{j_2}^{(l_2-1)}(x_2). \tag{15}$$

Again, the admissibility condition ensures that all terms like this can be computed. Through expanding all difference formulas in (13) we can build up a picture similar to that of Figure 3, which shows the hierarchical construction of the sparse-grid interpolation. For instance,

if $\Lambda = \{(1,1),(1,2),(2,1)\}$, we get,

$$\begin{aligned} I^{(\Lambda)} f^{(2)} &= \Delta^{(1)} \otimes \Delta^{(1)} f^{(2)} + \Delta^{(1)} \otimes \Delta^{(2)} f^{(2)} + \Delta^{(2)} \otimes \Delta^{(1)} f^{(2)} \\ &= I^{(1)} \otimes I^{(2)} f^{(2)} + I^{(2)} \otimes I^{(1)} f^{(2)} - I^{(1)} \otimes I^{(1)} f^{(2)} \end{aligned} \tag{16}$$

With an expansion as (16) we can exactly represent a quadratic function of the form $f = c_1 x_1^2 + c_2 x_1 + c_3 x_2^2 + c_4 x_2 + c_5$, where the $c_i$ are scalar coefficients. This is true if we again use a 1D quadrature rule that generates 1 point at level 1, and 3 points at level 2. The contribution of each $I^{(i)} \otimes I^{(j)} f^{(2)}$ tensor-product term to the hierarchical interpolation (16) is shown in Figure 9 for an example quadratic function of the stated form. Note that $I^{(2)} \otimes I^{(1)} f^{(2)}$ varies quadratically along $x_1$ and is constant in the $x_2$ direction. The converse is true for $I^{(1)} \otimes I^{(2)} f^{(2)}$. And since the constant effect is present in both tensor products, $I^{(1)} \otimes I^{(1)} f^{(2)}$ must be subtracted.

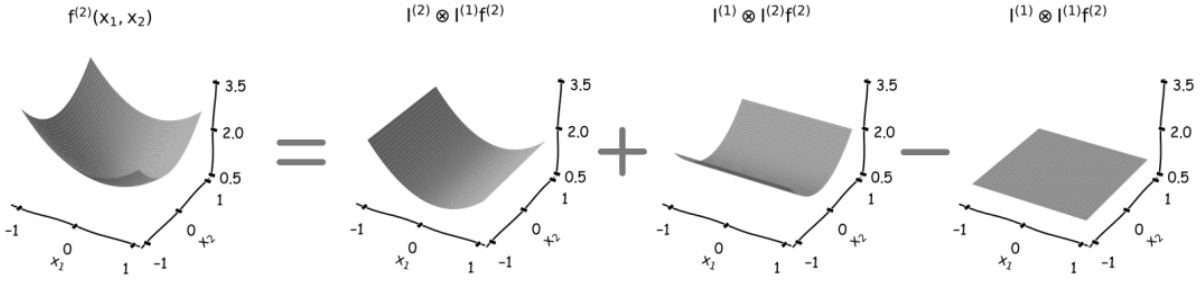



Figure 9: Hierarchical construction of an exact interpolation of $f = (x_1 - 0.2)^2 + x_2^2 + 1$, using multi indices $\Lambda = \{(1,1),(1,2),(2,1)\}$. Both inputs are uniformly distributed as $x_i \sim \mathcal{U}[-1,1]$.

The index set $\Lambda$ that is used in this example will generate a 'cross-like' sampling plan as shown in the middle plot of Figure 4. So if we vary $x_1$, $x_2$ will be constant and vice versa. In such a grid it is not possible to represent so-called *interaction effects*, i.e. the effect on the output $f$ due to *simultaneously* varying $x_1$ and $x_2$ together. It also means that a quadratic function $f$ that contains a term involving $x_1 x_2$ cannot be exactly represented by $I^{(\Lambda)} f$ if $\Lambda = \{(1,1),(1,2),(2,1)\}$, as each tensor product shown in Figure 9 only varies in at most 1 direction. If our quadratic polynomial $f$ does include a cross term $x_1 x_2$, $\Lambda$ must also include $(2,2)$ in order to obtain an exact $I^{(\Lambda)} f$. While in practise most models will not be polynomial, it is useful to think of the expressive power of $I^{(\Lambda)} f$ in this way. The index $\mathbf{l} = (1,4,1,\cdots,1)$ will add the ability to represent $x_2^n$ up to a certain power $n$ (depending on the number of points $m_4$ contained in a level 4 rule), whereas $\mathbf{l} = (1,1,2,3,1,\cdots,1)$ adds the ability to represent a cross term $x_3^n x_4^m$, and so on.

### 3.2.1 Efficient computation in high dimensions

In the example given in (16), note that a lot of terms cancel out (and also $I^{(0)} := 0$), such that we are left with only 3 (out of 12) tensor products. These remaining tensor products correspond to the multi indices in $\Lambda = \{(1,1),(1,2),(2,1)\}$ with coefficients -1, 1 and 1 respectively. However, note that if we have 20 inputs, each $\Delta^{(l_1)} \otimes \cdots \otimes \Delta^{(l_d)} f$ term contains $2^{20}$ tensor products of the form (12). In high dimensions $d \gg 1$, expanding all terms in each $\Delta^{(l_1)} \otimes \cdots \otimes \Delta^{(l_d)} f$ in a brute force manner is therefore inefficient. Luckily, we can employ the so-called **combination coefficient** approach. This is a way to compute the coefficients multiplying each unique tensor product (i.e. the -1, 1 and 1 from before), without expanding the difference formulas and manually figuring out which terms cancel. The combination coefficients are given by [9];

$$\boxed{c_{\mathbf{l}} = \sum_{z_1=0}^{1} \cdots \sum_{z_d=0}^{1} (-1)^{\|\mathbf{z}\|_1} \, \chi(\mathbf{l}+\mathbf{z})}, \quad \text{where} \quad \chi(\mathbf{l}) = \begin{cases} 1 & \mathbf{l} \in \Lambda \\ 0 & \text{otherwise} \end{cases}, \quad \text{and} \quad \|\mathbf{z}\|_1 = \sum_{i=1}^{d} |z_i|. \tag{17}$$

With the $c_{\mathbf{l}}$ known, we can rewrite (13) as

$$\begin{aligned} I^{(\Lambda)} f &= \sum_{\mathbf{l} \in \Lambda} \Delta^{(l_1)} \otimes \cdots \otimes \Delta^{(l_d)} f \\ &= \boxed{\sum_{\mathbf{l} \in \Lambda} c_{\mathbf{l}} \sum_{j_1=1}^{m_{l_1}} \cdots \sum_{j_d=1}^{m_{l_d}} f\left(x_{j_1}^{(l)}, \cdots, x_{j_d}^{(l)}\right) a_{j_1}^{(l)}(x_1) \otimes \cdots \otimes a_{j_d}^{(l)}(x_d)} \end{aligned} \tag{18}$$

From this formulation it is most clear that sparse-grid interpolation involves taking a *linear combination of tensor products*, with coefficients determined by (17).

By considering the multi indices $\mathbf{l}+\mathbf{z}$ as done in (17), we are reaching all multi indices that can potentially affect the contribution of $\mathbf{l}$ to the overall interpolant. To see this and further develop an intuition on the logic behind the $c_{\mathbf{l}}$ expression, consider again a 2D example. Assume the 4 multi indices shown in Figure 10 are all accepted into $\Lambda$. Hence there are also 4 $\Delta^{(l_i)} \otimes \Delta^{(l_j)} f^{(2)}$ terms, one for each $\mathbf{l}$ depicted. For each of these terms we ask whether it adds or subtracts $I^{(l_1)} \otimes I^{(l_2)} f^{(2)}$, which is indicated by the + and − signs. Why these locations add or subtract can be seen from Table 1, in which we write out each $\Delta^{(l_i)} \otimes \Delta^{(l_j)} f^{(2)}$ term. Here we clearly see that the forward neighbours of $(l_1, l_2)$ accessed via $\mathbf{z} = (0,1)$ and $\mathbf{z} = (1,0)$ will subtract (exclude) $I^{(l_1)} \otimes I^{(l_2)} f^{(2)}$, whereas $\mathbf{z} = (0,0)$ and $\mathbf{z} = (1,1)$ will add it instead. Hence, $(-1)^{\|\mathbf{z}\|_1}$, predicts the correct sign. If for instance $(l_1+1, l_2+1)$ is not in $\Lambda$, we must not consider the contribution of $\Delta^{(l_1+1)} \otimes \Delta^{(l_2+1)} f^{(2)}$. This is ensured by multiplying $(-1)^{\|\mathbf{z}\|_1}$ with the $\chi(\mathbf{l}+\mathbf{z})$ function, see again (17).

In the general case of $d$ input dimensions, any tensor product $\Delta^{(l_1+z_1)} \otimes \cdots \otimes \Delta^{(l_d+z_d)} f$ will also contain a single positive or negative $I^{(l_1)} \otimes \cdots \otimes I^{(l_d)} f$ term. If $\mathbf{z} = (0, \cdots, 0)$, it will be

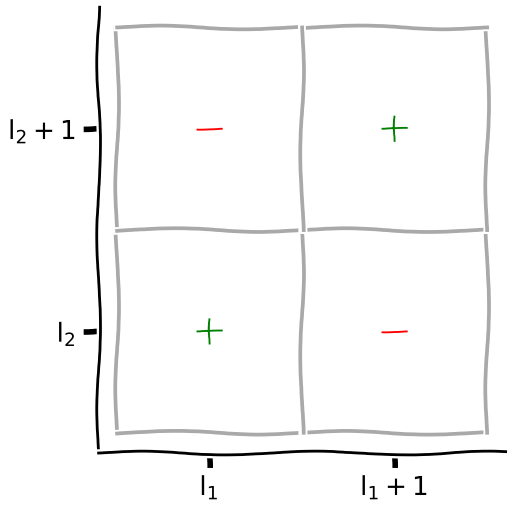



Figure 10: Four accepted 2D multi indices. A '+' means that $\Delta^{(l_i)} \otimes \Delta^{(l_j)} f$ adds the $I^{(l_1)} \otimes I^{(l_2)} f$ term, and a '−' means that $\Delta^{(l_i)} \otimes \Delta^{(l_j)} f$ subtracts $I^{(l_1)} \otimes I^{(l_2)} f$.

| $\mathbf{l} + \mathbf{z}$ | $\Delta^{(l_i)} \otimes \Delta^{(l_j)} f$ | in/exclude $I^{(l_1)} \otimes I^{(l_2)} f$ | $\|\mathbf{z}\|_1$ |
|---|---|---|---|
| $(l_1 + 0, l_2 + 0)$ | $\left(I^{(l_1)} - I^{(l_1-1)}\right) \otimes \left(I^{(l_2)} - I^{(l_2-1)}\right) f$ | include | 0 |
| $(l_1 + 1, l_2 + 0)$ | $\left(I^{(l_1+1)} - I^{(l_1)}\right) \otimes \left(I^{(l_2)} - I^{(l_2-1)}\right) f$ | exclude | 1 |
| $(l_1 + 0, l_2 + 1)$ | $\left(I^{(l_1)} - I^{(l_1-1)}\right) \otimes \left(I^{(l_2+1)} - I^{(l_2)}\right) f$ | exclude | 1 |
| $(l_1 + 1, l_2 + 1)$ | $\left(I^{(l_1+1)} - I^{(l_1)}\right) \otimes \left(I^{(l_2+1)} - I^{(l_2)}\right) f$ | include | 2 |

Table 1: Reasoning behind (17) for $d = 2$.

positive just as in Table 1. For the other $\mathbf{z}$, any $z_i = 1$ entry will cause the i-th component of $I^{(l_1)} \otimes \cdots \otimes I^{(l_d)} f$ to appear as negative, i.e. $I^{(l_i+1)} - I^{(l_i)}$. Hence, any $\mathbf{z}$ with an even number of $z_i = 1$ entries will generate a positive $I^{(l_1)} \otimes \cdots \otimes I^{(l_d)} f$, and vice versa for a $\mathbf{z}$ with an odd number of $z_i = 1$ entries. We can therefore conclude that also in the general case $(-1)^{\|\mathbf{z}\|_1}$ gives the correct sign. We repeat this procedure for every $\mathbf{l} \in \Lambda$ to find all $c_\mathbf{l}$.

While (17) prevents us from executing a brute-force expansion of the $\Delta^{(l_1)} \otimes \cdots \otimes \Delta^{(l_d)} f$ terms, its implementation must still be done with care. This is because a brute-force computation of (17) is again inefficient in high dimensions, as there are $2^d$ possible $\mathbf{z}$ vectors. So, if we have $d = 20$ inputs (which can certainly happen), we would need to check whether or not 1048576 different $\mathbf{l} + \mathbf{z}$ vectors are in $\Lambda$. Moreover, most of these vectors will not be in high dimensions, i.e. yield $\chi(\mathbf{l} + \mathbf{z}) = 0$, especially in anisotropic grids. A better algorithm will turn this around, and instead loop over the (relatively few) $\mathbf{l}$ that are in $\Lambda$, and check for each $\mathbf{k} \in \Lambda$ if it generates a valid $\mathbf{z}$. We now have $\mathbf{l} + \mathbf{z} = \mathbf{k} \in \Lambda$, and $\mathbf{z}$ is valid (i.e. $\chi(\mathbf{l} + \mathbf{z}) = 1$), when $\mathbf{z} = \mathbf{k} - \mathbf{l}$ has entries consisting of only 0's and 1's. Only for these $\mathbf{k}$ do we have to compute $(-1)^{\|\mathbf{z}\|_1} \chi(\mathbf{l} + \mathbf{z})$. This implies a 'double for-loop implementation' over the entries of $\Lambda$, which is computationally a lot cheaper compared to the brute-force approach in high dimensions. An Python3 implementation is given below.

```
import numpy as np

def compute_comb_coef(Lambda):
    comb_coef = {}
    for l in Lambda:
        coef = 0.0
        # subtract l from all multi indices
        for k in Lambda:
            z = k - l
            # check if z contains only 0's and 1's
            if np.array_equal(z, z.astype(bool)):
                coef += (-1)**(np.sum(z))
        comb_coef[tuple(l)] = coef
    return comb_coef

# example multi indices
Lambda = np.array([[1,1], [1, 2], [2, 1]])
print(compute_comb_coef(Lambda))
```

This will print `{(1, 1): -1.0, (1, 2): 1.0, (2, 1): 1.0}` to screen.

## 3.3 Pitfalls

A well-known pitfall for SC and PCE methods is the curse of dimensionality, as previously mentioned. We have discussed ways to postpone the curse using the dimension-adaptive sampling algorithm. Here, we will briefly discuss another pitfall, namely that of the un-

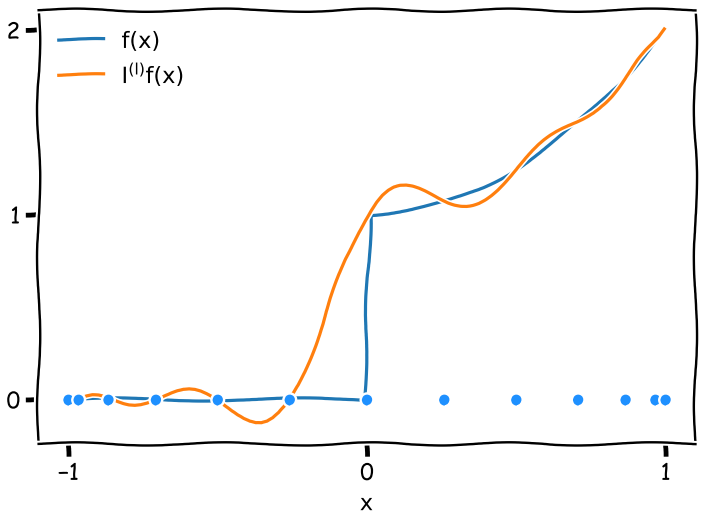



Figure 11: A discontinuous function $f(x)$ (blue) and its (standard SC) interpolant $I^{(l)}f(x)$ (orange) constructed using global Lagrange polynomials.

derlying regularity assumption. Consider again the (standard) SC expansion (12). The use of *global* (Lagrange) interpolation polynomials implicitly assumes that the function $f(\mathbf{x})$ is regular in the input space $\mathbf{x}$, e.g. that it does not display very large gradients or discontinuities. It is well-known from numerical analysis that interpolating such functions with a global polynomial basis can lead to an interpolant $I^{(\Lambda)}f$ that shows unrealistic oscillations, see Figure 11 for a 1D example. Clearly, $I^{(l)}f(x)$ is not a good representation of $f(x)$ here, as it oscillates near the discontinuity.

As the problem is the use of smooth, global polynomials as a basis for a function with sharp local features, a potential solution involves a change to a *local basis*. Consider again the 1D example of Figure 11. If one is able to detect the discontinuity in the stochastic input space at $x = 0$, we could represent this particular function exactly by a constant basis for $x < 0$ and a quadratic basis for $x \geq 0$. Methods that are able to do this operate in a similar fashion to the dimension-adaptive algorithm described previously, in the sense that they use an iterative sampling strategy in combination with an error measure. Rather than detecting important inputs, these error measures detect local irregularities, such that i) new samples can be clustered near the irregularity, and ii) the local basis can be adapted. Examples of such methods can be found here [14, 22, 6].

# 4 Analysis on sparse grids

Typical analyses of forward uncertainty-propagation problems include mean and variance estimation, and the computation of Sobol sensitivity indices. Both the moments and the Sobol indices take on a particularly simple (analytic) form in the case of Polynomial Chaos Expansions (PCEs) of $f$. To facilitate easy post-processing of the results we therefore first transform the SC expansion (18) into an equivalent PCE expansion.

## 4.1 PCE reformulation

The PCE equivalent of the *standard* SC expansion (12) reads (for a given $\mathbf{l} \in \Lambda$);

$$I^{(\mathbf{l})} f(\mathbf{x}) = \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}} \eta_{\mathbf{k}}^{(\mathbf{l})} \, \phi_{\mathbf{k}}(\mathbf{x}). \tag{19}$$

Here, $\mathbf{x} := (x_1, \cdots, x_d)$ and $\phi_{\mathbf{k}} := \phi_{k_1}(x_1) \otimes \cdots \otimes \phi_{k_d}(x_d)$. PCE expansions differ in various ways from SC expansions. First, the basis polynomials $\phi_{\mathbf{k}}$ are constructed to be **orthonormal** with respect to the input density, i.e.

$$\mathbb{E}\left[\phi_{\mathbf{j}} \phi_{\mathbf{k}}\right] = \int \phi_{\mathbf{j}} \phi_{\mathbf{k}} \, p\left(\mathbf{x}\right) \mathrm{d}\mathbf{x} = \begin{cases} 1 & \mathbf{j} = \mathbf{k} \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

It is also common to use orthogonal basis polynomials, in which case the constant $\mathbb{E}\left[\phi_{\mathbf{j}}^2\right]$ will be different from 1. Here we always use an orthonormal basis, which will yield slightly simpler expressions for the moments and Sobol indices.

The second difference concerns the response coefficients $\eta_{\mathbf{k}}$. These are not simply the code outputs as in the SC expansion, and instead must be computed in a separate procedure. Briefly, if we momentarily equate the code output to the PCE expansion;

$$f = \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}} \eta_{\mathbf{k}}^{(\mathbf{l})} \, \phi_{\mathbf{k}} \Leftrightarrow \mathbb{E}\left[f \phi_{\mathbf{j}}\right] = \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}} \eta_{\mathbf{k}}^{(\mathbf{l})} \, \mathbb{E}\left[\phi_{\mathbf{k}} \phi_{\mathbf{j}}\right] = \eta_{\mathbf{j}}^{(\mathbf{l})}, \tag{21}$$

such the $\eta_{\mathbf{j}}^{(\mathbf{l})}$ can be interpreted as orthogonal projection coefficients, i.e. $\eta_{\mathbf{j}}^{(\mathbf{l})} = \mathbb{E}\left[f \phi_{\mathbf{j}}\right]$. This expectation can be computed using numerical quadrature, see for instance [23]. We use a slightly different method, described in [4, 11] and outlined below. First however, note that unlike the SC expansion (12), summation in (19) does not take place over the collocation points $\mathbf{x}_{\mathbf{j}}^{(l)} := (x_{j_1}^{(l_1)}, \cdots, x_{j_d}^{(l_d)})$. Instead, it takes place over multi indices $\mathbf{k} = (k_1, \cdots, k_d) \in \Lambda_{\mathbf{l}}$, where the set $\Lambda_{\mathbf{l}}$ is somehow truncated by $\mathbf{l}$ (and is a subset of $\Lambda$ in our case). Here we will use the following truncation;

$$\Lambda_{\mathbf{l}} := \{\mathbf{k} \mid \mathbf{k} \leq \mathbf{l}, \, \mathbf{l} \in \Lambda\}. \tag{22}$$

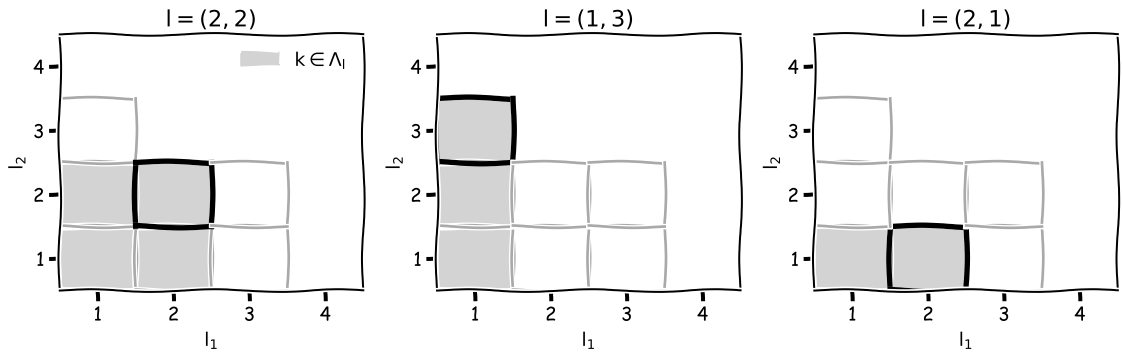



Figure 12: Three examples of $\Lambda_{\mathbf{l}} \subseteq \Lambda$, where $\Lambda = \{(1,1),(1,2),(1,3),(2,1),(2,2),(3,1),(3,2)\}$.

The inequality $\leq$ applies element wise. Examples of $\Lambda_{\mathbf{l}}$ for $d = 2$ are shown in Figure 12.

For each given $\mathbf{l} \in \Lambda$, let us equate the SC expansion with the corresponding PCE expansion:

$$\sum_{j_1=1}^{m_1} \cdots \sum_{j_d=1}^{m_1} f^{(d)}\left(\mathbf{x}_{\mathbf{j}}^{(\mathbf{l})}\right) a_{\mathbf{j}}^{(\mathbf{l})}(\mathbf{x}) = \sum_{\mathbf{k}\in\Lambda_{\mathbf{l}}} \eta_{\mathbf{k}}^{(\mathbf{l})}\, \phi_{\mathbf{k}}(\mathbf{x}). \tag{23}$$

Here we introduced the shorthand $\mathbf{x}_{\mathbf{j}}^{(\mathbf{l})} := (x_{j_1}^{(l_1)}, \cdots, x_{j_d}^{(l_d)})$ and $a_{\mathbf{j}}^{(\mathbf{l})}(\mathbf{x}) := a_{j_1}^{(l_1)}(x_1) \otimes \cdots \otimes a_{j_d}^{(l_d)}(x_d)$. As in (21), we use the orthogonal nature of the PCE basis polynomials to isolate each PCE coefficient [4, 11];

$$\begin{aligned}
\eta_{\mathbf{k}}^{(\mathbf{l})} &= \sum_{j_1=1}^{m_1} \cdots \sum_{j_d=1}^{m_d} f\left(\mathbf{x}_{\mathbf{j}}^{(\mathbf{l})}\right) \mathbb{E}\left[a_{\mathbf{j}}^{(\mathbf{l})}(\mathbf{x})\phi_k(\mathbf{x})\right] \\
&= \sum_{j_1=1}^{m_1} \cdots \sum_{j_d=1}^{m_d} f\left(\mathbf{x}_{\mathbf{j}}^{(\mathbf{l})}\right) \int a_{j_1}^{(l_1)}(x_1)\phi_{k_1}(x_1)p(x_1)\mathrm{d}x_1 \otimes \cdots \otimes \int a_{j_d}^{(l_d)}(x_d)\phi_{k_1}(x_d)p(x_d)\mathrm{d}x_d \\
&= \sum_{j_1=1}^{m_1} \cdots \sum_{j_d=1}^{m_d} f\left(\mathbf{x}_{\mathbf{j}}^{(\mathbf{l})}\right) v_{k_1}^{(l_1,j_1)} \otimes \cdots \otimes v_{k_d}^{(l_d,j_d)}.
\end{aligned} \tag{24}$$

Note that we used the the assumption of independent inputs in the 2nd equality. Hence, (24) shows that each $\eta_{\mathbf{k}}^{(\mathbf{l})}$, with $\mathbf{k} \in \Lambda_{\mathbf{l}}$ can be written as a SC-type expansion with coefficients given by ;

$$\nu_{k_i}^{(l_i,j_i)} = \int a_{j_i}^{(l_i)}(x_i)\phi_{k_i}(x_i)\; p(x_i)\mathrm{d}x_i \quad i = 1\cdots, d. \tag{25}$$

This can be integrated exactly over the support of $p(x_i)$ using (Gaussian) quadrature, since both $a_{j_i}^{(l_i)}$ and $\phi_{k_i}$ are polynomials of $x_i$. The $a_{j_i}^{(l_i)}$ are given by (4), and $\phi_{k_i}$ needs to be orthonormal to $p(x_i)$, see again (20). Many well-known input distributions have corresponding

orthogonal polynomials, e.g. Hermite polynomials for Gaussian inputs, or Legendre polynomials in the case of uniform distributions. A practical solution to compute (25), is to use a software package that can automatically generate both the input distributions and the orthonormal $\phi_{k_i}$, such as the Chaospy Python3 package [7], which also provides libraries for Gaussian quadrature.

Once we have computed all $\eta_{\mathbf{k}}^{(\mathbf{l})}$, $\mathbf{k} \in \Lambda_{\mathbf{l}}$, for every $\mathbf{l} \in \Lambda$, we can rewrite our sparse-grid interpolant (18) in PCE form, i.e.:

$$I^{(\Lambda)} f = \sum_{\mathbf{l} \in \Lambda} c_{\mathbf{l}} \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}} \eta_{\mathbf{k}}^{(\mathbf{l})} \phi_{\mathbf{k}}. \tag{26}$$

Before discussing the moment estimation and sensitivity analysis, we will simplify (26) further. In particular, we will write this (potentially anisotropic) sparse-grid expression into *standard* (non-sparse) PCE form. To do so we must bring $c_{\mathbf{l}}$ into the second summation. This can be done by expanding (26), and grouping all terms of like $\mathbf{k}$. For instance if $\Lambda = \{(1,1),(1,2),(2,1),(2,2)\}$, (26) becomes;

$$\begin{aligned} I^{(\Lambda)} f = {} & c_{11}\left(\eta_{11}^{11}\phi_{11}\right) + c_{12}\left(\eta_{11}^{12}\phi_{11} + \eta_{12}^{12}\phi_{12}\right) + c_{21}\left(\eta_{11}^{21}\phi_{11} + \eta_{21}^{21}\phi_{21}\right) + \\ & c_{22}\left(\eta_{11}^{22}\phi_{11} + \eta_{12}^{22}\phi_{12} + \eta_{21}^{22}\phi_{21} + \eta_{22}^{22}\phi_{22}\right) \\ \\ = {} & \left(c_{11}\eta_{11}^{11} + c_{12}\eta_{11}^{12} + c_{21}\eta_{11}^{21} + c_{22}\eta_{11}^{22}\right)\phi_{11} + \left(c_{12}\eta_{12}^{12} + c_{22}\eta_{12}^{22}\right)\phi_{12} + \\ & \left(c_{21}\eta_{21}^{21} + c_{22}\eta_{21}^{22}\right)\phi_{21} + \left(c_{22}\eta_{22}^{22}\right)\phi_{22} \end{aligned} \tag{27}$$

In the first equality, the terms between the brackets $(\cdot)$ are indexed by $\Lambda_{\mathbf{l}}$, see the 2nd column of Table 2. In the 2nd equality we grouped all terms with the same $\mathbf{k}$, with $\mathbf{k}$ as indicated in (26). Note that we can write this final expression as

$$I^{(\Lambda)} f = \sum_{\mathbf{l} \in \Lambda} \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}^{-1}} c_{\mathbf{k}} \eta_{\mathbf{l}}^{(\mathbf{k})} \phi_{\mathbf{l}}. \tag{28}$$

This brought the combination coefficients into the second summation as we wished, and we now the sum over multi indices in $\Lambda_{\mathbf{l}}^{-1}$. The $\mathbf{k} \in \Lambda_{\mathbf{l}}^{-1}$ for the example above are shown in the 3rd column of Table 2. Note that these can be interpreted as the 'inverse' of $\Lambda_{\mathbf{l}}$, defined as;

$$\Lambda_{\mathbf{l}}^{-1} := \{\mathbf{k} \mid \mathbf{k} \geq \mathbf{l},\ \mathbf{l} \in \Lambda\}. \tag{29}$$

From (28) we can make a definition for new 'generalized' PCE coefficients;

$$\boxed{\psi^{(\mathbf{l})} := \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}^{-1}} c_{\mathbf{k}} \eta_{\mathbf{l}}^{(\mathbf{k})}}, \tag{30}$$

| $\mathbf{l}$ | $\Lambda_{\mathbf{l}} := \{\mathbf{k} \mid \mathbf{k} \leq \mathbf{l},\ \mathbf{l} \in \Lambda\}$ | $\Lambda_{\mathbf{l}}^{-1} := \{\mathbf{k} \mid \mathbf{k} \geq \mathbf{l},\ \mathbf{l} \in \Lambda\}$ |
|---|---|---|
| $(1,1)$ | $(1,1)$ | $(1,1),(1,2),(2,1),(2,2)$ |
| $(1,2)$ | $(1,1),(1,2)$ | $(1,2),(2,2)$ |
| $(2,1)$ | $(1,1),(2,1)$ | $(2,1),\ (2,2)$ |
| $(2,2)$ | $(1,1),(1,2),(2,1),(2,2)$ | $(2,2)$ |

Table 2: Multi indices $\mathbf{k} \in \Lambda_{\mathbf{l}}$ and $\mathbf{k} \in \Lambda_{\mathbf{l}}^{-1}$ when $\Lambda = \{(1,1),(1,2),(2,1),(2,2)\}$.

which allows us to finally write (28) in standard PCE form:

$$I^{(\Lambda)} f^{(d)} = \sum_{\mathbf{l} \in \Lambda} \psi^{(\mathbf{l})} \phi_{\mathbf{l}}(\mathbf{x}). \tag{31}$$

To recap;

$$\begin{aligned} I^{(\Lambda)} f &= \sum_{\mathbf{l} \in \Lambda} \Delta^{(l_1)} \otimes \cdots \otimes \Delta^{(l_d)} f \\ &= \sum_{\mathbf{l} \in \Lambda} c_{\mathbf{l}} \sum_{j_1=1}^{m_1} \cdots \sum_{j_d=1}^{m_1} f^{(d)} \left( \mathbf{x}_{\mathbf{j}}^{(\mathbf{l})} \right) a_{\mathbf{j}}^{(\mathbf{l})}(\mathbf{x}) \\ &= \sum_{\mathbf{l} \in \Lambda} c_{\mathbf{l}} \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}} \eta_{\mathbf{k}}^{(\mathbf{l})} \phi_{\mathbf{k}}(\mathbf{x}) = \sum_{\mathbf{l} \in \Lambda} \sum_{\mathbf{k} \in \Lambda_{\mathbf{l}}^{-1}} c_{\mathbf{k}} \eta_{\mathbf{l}}^{(\mathbf{k})} \phi_{\mathbf{l}}(\mathbf{x}) \\ &= \sum_{\mathbf{l} \in \Lambda} \psi^{(\mathbf{l})} \phi_{\mathbf{l}}(\mathbf{x}), \end{aligned} \tag{32}$$

which shows that an (anisotropic) sparse SC expansion can be written in standard PCE form.

## 4.2 Moment estimation

Computing the mean and variance from a standard PCE expansion is straightforward. Using the fact that $\phi_{\mathbf{l}}(\mathbf{x}) = 1$ for $\mathbf{l} = (1, \cdots, 1)$, we get the following expression for the mean

$$\mathbb{E}\left[ I^{(\Lambda)} f \right] = \sum_{\mathbf{l} \in \Lambda} \psi^{(\mathbf{l})} \mathbb{E}\left[ \phi_{\mathbf{l}} \cdot 1 \right] = \psi^{(\mathbf{1})}. \tag{33}$$

That is, the mean just equals the first (generalized) PCE coefficient $\psi^{(\mathbf{l})}$ for $\mathbf{l} = (1, \cdots, 1) =: \mathbf{1}$. The variance is also a simple expression involving only the PCE coefficients:

$$\mathbb{V}\mathrm{ar}\left[I^{(\Lambda)} f\right] = \mathbb{E}\left[\left(\sum_{\mathbf{l}\in\Lambda} \psi^{(\mathbf{l})}\phi_{\mathbf{l}} - \psi^{(\mathbf{1})}\right)^2\right] = \mathbb{E}\left[\left(\sum_{\substack{\mathbf{l}\in\Lambda \\ \mathbf{l}\neq\mathbf{1}}} \psi^{(\mathbf{l})}\phi_{\mathbf{l}}\right)^2\right] = \sum_{\substack{\mathbf{l}\in\Lambda \\ \mathbf{l}\neq\mathbf{1}}} \left(\psi^{(\mathbf{l})}\right)^2. \tag{34}$$

The first equality again uses $\phi_{\mathbf{1}} = 1$, and the last equality follows form the orthonormal nature of the PCE basis, such that all cross terms $\phi_{\mathbf{l}}\phi_{\mathbf{k}}$ have a zero expectation if $\mathbf{l} \neq \mathbf{k}$, and $\mathbb{E}\left[\phi_{\mathbf{l}}^2\right] = 1$ when $\mathbf{l} = \mathbf{k}$. Equations (33) and (34) are standard PCE results, which in light of (32), can also be applied in the case of sparse SC expansions.

## 4.3 Sparse-grid sensitivity analysis

Sobol indices are global variance-based sensitivity measures of a function $f(\mathbf{x})$ with respect to its inputs $\mathbf{x} \in \mathbb{R}^d$ [19]. The term 'global' means that the sensitivity is not assessed at a single $\mathbf{x}$ location in the input space, which would make it a local method. Instead, the sensitivity is computed over the entire input space, defined by the support of the input pdf $p(\mathbf{x})$. In particular, let $\mathbb{V}\mathrm{ar}[f_{\mathbf{u}}]$ be a so-called *partial variance*, where the multi-index $\mathbf{u}$ can be any subset of $\mathcal{U} := \{1, 2, \cdots, d\}$. Each partial variance measures the fraction of the total variance in the output $f$ that can be attributed to the input parameter combination indexed by $\mathbf{u}$. The Sobol indices are defined as the normalised partial variances, i.e.

$$S_{\mathbf{u}} := \frac{\mathbb{V}\mathrm{ar}\left[f_{\mathbf{u}}\right]}{\mathbb{V}\mathrm{ar}\left[f\right]}, \tag{35}$$

where $\mathbb{V}\mathrm{ar}[f] = \sum_{\mathbf{u}\subseteq\mathcal{U}} \mathbb{V}\mathrm{ar}[f_{\mathbf{u}}]$ is the total variance of $f$. So if $\mathbf{u} = \{1\}$, then $S_{\mathbf{u}}$ is a *first-order* index that measure the sensitivity of $f$ to changes in $x_1$ alone. A second-order index such as $\mathbf{u} = \{2, 4\}$ measures the sensitivity due to simultaneous changes in $x_2$ and $x_4$ together. Since all partial variances are positive, the sum of all possible $S_{\mathbf{u}}$ equals 1.

The PCE formulation is particularly suited for sensitivity analysis, since similar to the statistical moments, the Sobol indices can also be calculated from the PCE coefficients $\psi^{(\mathbf{l})}$ in a simple post-processing procedure [20]. The partial variances can be computed with

$$\mathbb{V}\mathrm{ar}\left[I^{(\Lambda)} f_{\mathbf{u}}\right] = \sum_{\mathbf{k}\in\mathcal{K}_{\mathbf{u}}} \left(\psi^{(\mathbf{k})}\right)^2 \quad \text{where} \quad \mathcal{K}_{\mathbf{u}} = \{\mathbf{k} \mid k_i > 1 \text{ when } k_i \in \mathbf{u}, \;\; k_j = 1 \text{ when } k_j \notin \mathbf{u}\}. \tag{36}$$

Note that this is a similar expression as the full variance (34). The only difference is multi-index set $\mathcal{K}_{\mathbf{u}}$, which can be interpreted as the set of all multi indices corresponding to varying only the inputs indexed by $\mathbf{u}$. That is, if for instance $\mathbf{u} = \{1\}$, $\mathcal{K}_{\mathbf{u}}$ is the subset of $\Lambda$, with all

indices $\mathbf{k}$ where $k_1 > 1$ and all other $k_j = 1$. The reasoning behind this is that all inputs with $k_j = 1$ are at level 1 (with just a single code evaluation), and therefore cause no variation in the output for that $\mathbf{k}$. Only including each multi index $\mathbf{k} \in \Lambda$, for which $k_1 > 1$ therefore isolates the effect of $x_1$ on the variance of the output. Likewise, building $\mathcal{K}_\mathbf{u}$ such that it includes all $\mathbf{k} \in \Lambda$ where only $k_2 > 1$ and $k_4 > 1$ provides an estimate of the corresponding second-order partial variance. Following the definition of the Sobol indices (35), we just divide (36) by the total variance (34) to obtain the PCE estimate of $S_\mathbf{u}$.

# 5 Examples

In this section we will demonstrate the adaptive sampling algorithm on a number of test functions. In particular, we will use EasyVVUQ [17], a Python3 forward uncertainty propagation toolkit. The full code for each example below is available as a Jupyter notebook, see [5].

## 5.1 Polynomial function with 10 inputs

We start with a simple polynomial function[2]. In particular, we will take the following function with $d = 10$ from [12, 21]

$$f(\mathbf{x}) = 6x_1 + 4x_2 + 5.5x_3 + 3x_1x_2 + 2.2x_1x_2 + 1.4x_2x_3 + x_4 + 0.5x_5 + 0.2x_6 + 0.1x_7. \quad (37)$$

From this functional form we can already tell that the first 3 inputs are influential, and have interaction effects between them. Inputs $x_4$ through $x_7$ are less important, and inputs $x_8$ through $x_{10}$ have no effect at all. This function serves as a sanity check, since all 10 terms of (37) can be represented exactly with a level 2 refinement. Hence, if the algorithm functions properly, it will pick out exactly the right refinement $\mathbf{l}$, such that after 10 refinements $I^{(\Lambda)}f$ is exact. In this case the hierarchical surplus error (11) of the 11th iteration must be zero.

We start by importing our uncertainty-quantification libraries;

```
import easyvvuq as uq
import chaospy as cp
```

EasyVVUQ can be installed via the command line (see [1] for more options):

```
pip install easyvvuq
```

This will also install Chaospy [7], which we will use for the specification of the input distributions. We then start by defining some general properties of the parameter space:

[2]For a database of useful test functions, see the Virtual Library of Simulation Experiments at [21]

```
# number of inputs
d = 10

# params dict
params = {}
for i in range(d):
    params['x%d' % (i + 1)] = {'type': 'float', 'default': 0.5}
```

All we do here is define the type (float) and nominal value of each input. Next we will set up the encoder, which will create the input files for the model using an input template. In this case the input file is just a comma-separated file of 10 values. To create an EasyVVUQ input template we just replace each value with a flag:

```
$x1,$x2,$x3,$x4,$x5,$x6,$x7,$x8,$x9,$x10
```

The encoder will swap out the flags for values drawn from the corresponding input distribution;

```
# input file encoder
encoder = uq.encoders.GenericEncoder(template_fname='loeppky.template',
   delimiter='$', target_filename='input.csv')
```

Here, `'loeppky.template'` is the aforementioned template file, `delimiter` is used to identify the flags, and `'input.csv'` is the name given to each input file.

The model writes a CSV file containing the prediction $f(\mathbf{x})$. The decoder will read this file and store its contents within the EasyVVUQ database.

```
# Quantity of Interest, also the column name of the output CSV file
QOI = 'f'
# CSV output file decoder
decoder = uq.decoders.SimpleCSV(target_filename='output.csv',
   output_columns=[QOI])
```

The model here is cheap enough such that ensembles can be executed locally. However, in many cases of practical interest the model will be too expensive for local execution. In this case the tools such as QCG-PilotJob [2] or FabSim3 [17] can be used in combination with EasyVVUQ to submit and/or execute the ensemble on a remote supercomputer. Local exeution is defined via;

```
import os
# local execution of loeppky.py
execute = ExecuteLocal('{}/loeppky.py'.format(os.getcwd()))
```

Here, `loeppky.py` is the Python3 script containing the implementation of (37).

All steps we want to execute are combined into an 'Actions' object:

```
# location where the run directories are stored
WORK_DIR = '/tmp'
# actions to be undertaken
actions = Actions(CreateRunDirectory(root=WORK_DIR, flatten=True),
 Encode(encoder), execute, Decode(decoder))
```

The steps undertaken here (in order) are i) create directories for each individual run, ii) encode the input files, iii) execute the model runs, and iv) decode the output files. Next, the central EasyVVUQ object (a so-called 'campaign'), is created:

```
campaign = uq.Campaign(name='loeppky', params=params, actions=actions,
   work_dir=WORK_DIR)
```

We now determine which inputs which we wish to make random. In this case we will make all $d = 10$ inputs uniformly distributed;

```
vary = {}
for i in range(d):
    vary['x%d' % (i + 1)] = cp.Uniform(0, 1)
```

The `vary` dictionary holds all random inputs. If it only contains a subset of all inputs, the remaining inputs will automatically be set to their default value defined in `params`.

We select the dimension-adaptive SC sampler via;

```
# dimension-adaptive SC sampler
sampler = uq.sampling.SCSampler(vary=vary, polynomial_order=1, sparse=True
   , quadrature_rule='C', growth=True, dimension_adaptive=True)
# add sampler to campaign
campaign.set_sampler(sampler)
```

Here,

- `sparse=True` and `dimension_adaptive=True` selects the anisotropic, dimension-adaptive SC sampler.
- `polynomial_order=1` should be interpreted in the sparse context as starting the sampling plan with a level 1 quadrature rule for all inputs[3].
- `quadrature_rule='C'` selects the Clenshaw-Curtis quadrature rule.
- `growth=True` selects an exponential growth rule (i.e. (3)), which makes the Clenshaw Curtis rule nested.

The following command executes all steps in the Actions object:

[3]This is not a good variable name, and might be changed in future versions.

```
campaign.execute().collate(progress_bar=True)
```

To analyse the results (and execute the dimension adaptivity), we need an `SCAnalysis` object:

```
analysis = uq.analysis.SCAnalysis(sampler=sampler, qoi_cols=[QOI])
# perform analysis (moments, Sobol indices, and update internal state of
    analysis)
campaign.apply_analysis(analysis)
```

However, as all inputs are still at level 1, we have performed only a single code evaluation at this point. Below we refine the grid several times in an anisotropic fashion;

```
number_of_adaptations = 11
for i in range(number_of_adaptations):
    # compute candidate refinements
    sampler.look_ahead(analysis.l_norm)
    # run ensemble (at new locations only)
    campaign.execute().collate(progress_bar=True)
    # get data frame
    data_frame = campaign.get_collation_result()
    # adapt the sampling plan
    analysis.adapt_dimension(QOI, data_frame)
    # we must apply the analysis to update its internal state
    campaign.apply_analysis(analysis)
```

Here;

- `look_ahead`: determines the new admissible candidate refinements, i.e. it computes the new forward neighbours that are not yet accepted into $\Lambda$. Furthermore, `analysis.l_norm` equals $\Lambda$.
- `campaign.get_collation_result()`: get a data frame with all code samples.
- `adapt_dimension`: compute the hierarchical surplus error (11) at all candidate refinements $\mathbf{l}^*$, and accept the one with the highest surplus into $\Lambda$.

This command retrieves the result after all refinements:

```
results = campaign.get_last_analysis()
```

One aspect we can analyse is the progression of the adaptive sampling algorithm. If we execute

```
analysis.adaptation_table()
```

we obtain Figure 13. This is essentially just a visualization of the multi indices $\mathbf{l} \in \Lambda$, displayed in the order they were added. As expected, at iteration 0 all inputs are at level 1

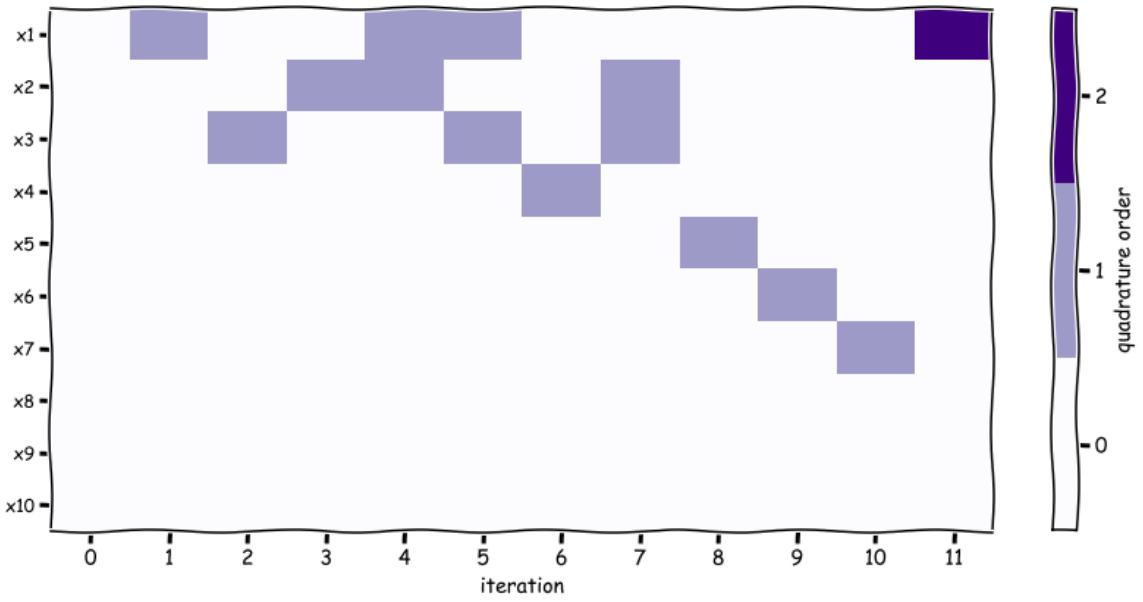



Figure 13: Refinement history of the polynomial model (37) from Loeppky [12] after 11 iterations. The colors display the quadrature order of the Clenshaw-Curtis rule. Order zero is level one, order one is level two etc. Hence, each column displays the $\mathbf{l}$ that was accepted into $\Lambda$ at that iteration.

(quadrature order 0). In the next 3 iterations $x_1$, $x_3$ and $x_2$ are refined, individually, to level 2 (as expected). Note that this represents refinement along a line in a 10 dimensional input space. Next, the interaction effects between $x_1, x_2$ and $x_1, x_3$ are found, meaning that the corresponding planes are refined. The algorithm continuous refining (combinations) of input parameters to level 2 until the 10th iteration, and in iteration 11 $x_1$ is refined once more to level 3. According to the discussion at the beginning of this section, this last refinement should not have changed $I^{(\Lambda)} f^{(10)}$, as we can see from Figure 13 that all relevant $\mathbf{l}$ have been identified. To double check, we can print the refinement errors to screen via:

```
print(analysis.get_adaptation_errors())

[4.3, 3.65, 3.1, 0.75, 0.55, 0.5, 0.35, 0.25, 0.1, 0.05, 7.54e-15]
```

This indeed shows that the accuracy gain of the 11th iteration is virtually zero.

The `results` dictionary also contains the mean and variance (or standard deviation), computed via (33) and (34). In Figure 14 we plot these statistics versus the mean and standard deviation computed via Monte Carlo, versus the number of model evaluations. Since MC statistics are random variables, we repeated the MC analysis 100 times for every fixed value of model evaluations, in order to compute 90% confidence intervals. As expected the confidence intervals are large initially, and converge somewhere between $10^3$ and $10^4$ model evaluations. The full SC surrogate is (in this case) pointwise exact with a little over 100 model runs,

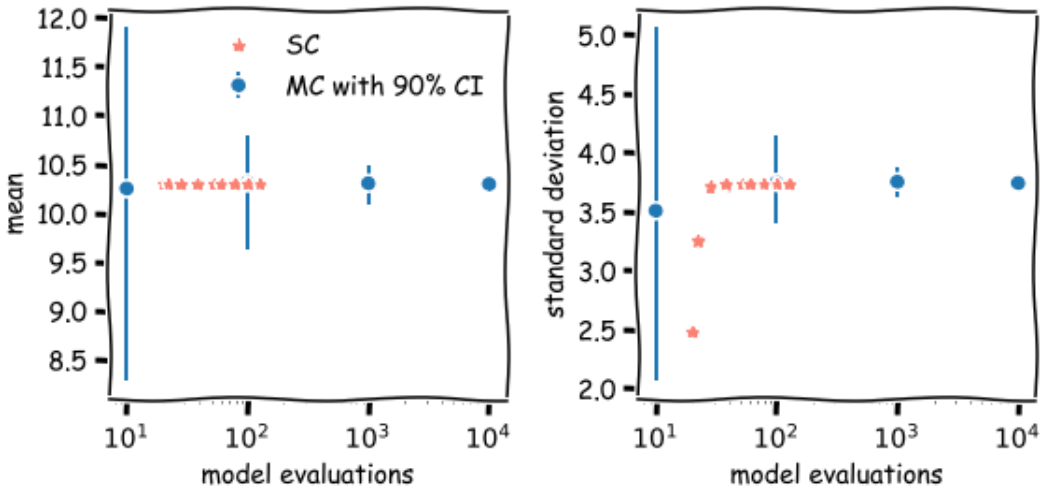



Figure 14: The MC and SC mean (left) and standard deviation (right) for the polynomial model.

including the unnecessary 11th refinement. That said, from Figure 14 we can see that mean and variance are already accurate at an even lower number of model evaluations.

It is further worthwhile to contrast the performance against the standard SC and the isotropic sparse grids methods. Regarding the former, using a level 2 rule for all 10 inputs would have generated $3^{10} = 59049$ model runs. A isotropic sparse grid with truncation given by (9) and $L = 3$ would also have been exact, but as a cost of 221 model runs [3]. This is larger than 100, although still reasonable, at least for $d = 10$.

## 5.2 Wing weight function

Our second example also has 10 uncertain inputs, but involves a non-polynomial function, which we can therefore only approximate. In particular, we consider a function which models the weight of a (light) aircraft wing:

$$f(\mathbf{x}) = 0.036 S_w^{0.758} W_{fw}^{0.0035} \left(\frac{A}{cos^2(\Lambda)}\right)^{0.6} q^{0.006} \lambda^{0.04} \left(\frac{100 t_c}{cos(\Lambda)}\right)^{-0.3} (N_z W_{dg})^{0.49} + S_w W_p \quad (38)$$

The inputs distributions (taken from `https://www.sfu.ca/~ssurjano/wingweight.html`), are displayed in Table 3.

As there are no significant differences in the EasyVVUQ setup compared to the previous example, we will move straight on to the results. The visualization of the $\mathbf{l} \in \Lambda$ is shown in Figure 15, for 20 refinement steps. This is less intuitive than the polynomial case, and we

| Parameter | Name | Distribution |
|---|---|---|
| $x_1 = S_w$ | wing area (ft2) | $\mathcal{U}[150, 200]$ |
| $x_2 = W_{fw}$ | weight of fuel in the wing (lb) | $\mathcal{U}[220, 300]$ |
| $x_3 = A$ | aspect ratio | $\mathcal{U}[6, 10]$ |
| $x_4 = \Lambda$ | quarter-chord sweep (degrees) | $\mathcal{U}[-10, 10]$ |
| $x_5 = q$ | dynamic pressure at cruise (lb/ft2) | $\mathcal{U}[16, 45]$ |
| $x_6 = \lambda$ | taper ratio | $\mathcal{U}[0.5, 1.0]$ |
| $x_7 = t_c$ | aerofoil thickness to chord ratio | $\mathcal{U}[0.08, 0.18]$ |
| $x_8 = N_z$ | ultimate load factor | $\mathcal{U}[2.5, 6]$ |
| $x_9 = W_{dg}$ | flight design gross weight (lb) | $\mathcal{U}[1700, 2500]$ |
| $x_{10} = W_p$ | paint weight (lb/ft2) | $\mathcal{U}[0.025, 0.08]$ |

Table 3: Wing-weight input distributions.

can see that most inputs are refined in some manner, and only 2 are ignored. The refinement errors are:

```
print(analysis.get_adaptation_errors())

[53.37, 39., 32.28, 29.33, 24.23, 8.05, 6.67, 5.79, 5.01, 4.87, 4.81,
   4.23, 3.66, 3.59, 3.57, 3.5, 3.03, 2.63, 1.01, 0.92]
```

Judging from the (in this case) monotonic and fairly rapid drop in the error, the sampling algorithm is still successful. The first two moments, again versus the MC estimates, are shown in Figure 16. In this case we sampled the model 229 times at the end, although the statistics already converge around 100 model runs.

Sobol indices (35) give a better picture on the importance of each input than the visualization of Figure 15, and are shown in Figure 17. The 1st, 3rd, 7th, 8th and 9th input show up as important. These results are in line with the sensitivity analysis performed in [16] (Table 4.4). Finally, the sum total of all first-order Sobol indices is displayed, which is very close to one, such that we know interaction effects do not play an important role here.

# Acknowledgments

This research was funded by the European Union Horizon 2020 research and innovation programme under grant agreement #800925 (VECMA project). We thank the EPSRC for funding the Software Environment for Actionable and VVUQ-evaluated Exascale Applications (SEAVEA) grant (EP/W007711/1). I would also like to thank André van Veen for proofreading this document.

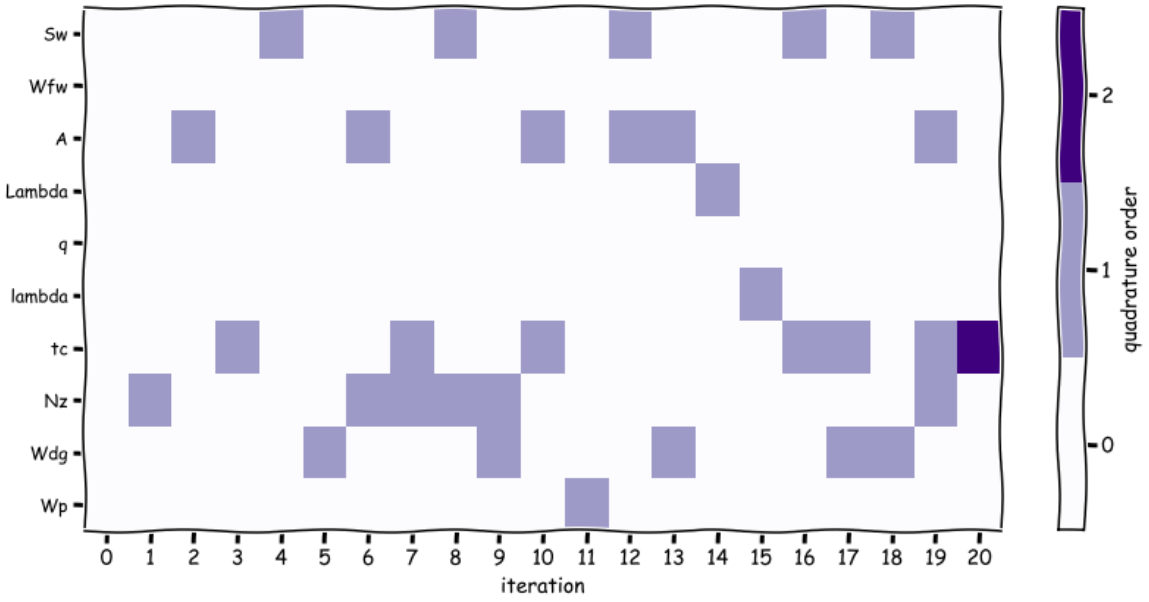



Figure 15: Refinement history of the wing weight model (38) after 20 iterations. The colors display the quadrature order of the Clenshaw-Curtis rule. Order zero is level one, order one is level two etc. Hence, each column displays the $\mathbf{l}$ that was accepted into $\Lambda$ at that iteration.

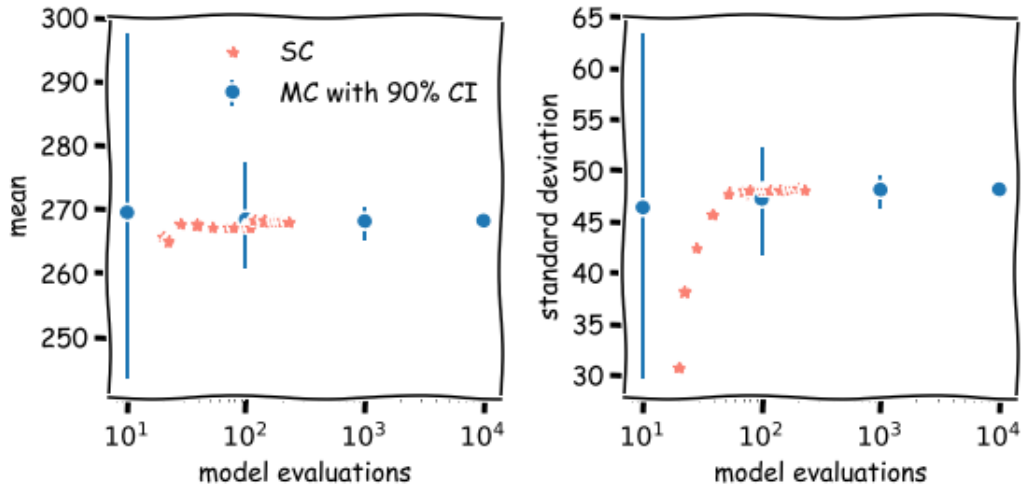



Figure 16: The MC and SC mean (left) and standard deviation (right) for the pwing weight model.

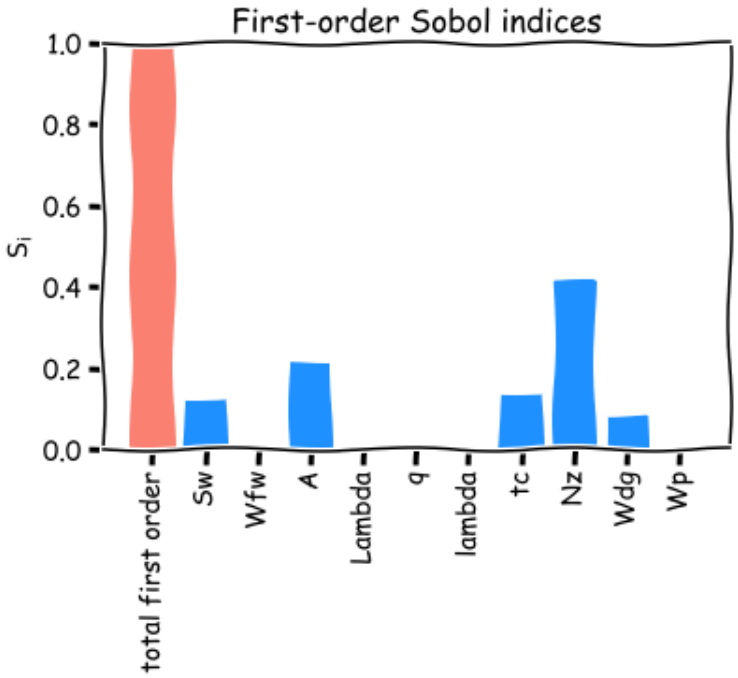



Figure 17: The first-order Sobol indices for the wing-weight model (blue bars). The total first-order contribution is also depicted (red bar).

# References


[1] EasyVVUQ documentation. `https://easyvvuq.readthedocs.io/`.

[2] B. Bosak, T. Piontek, P. Karlshoefer, E. Raffin, J. Lakhlili, and P. Kopta. Verification, validation and uncertainty quantification of large-scale applications with qcg-pilotjob. In *International Conference on Computational Science*, pages 495–501. Springer, 2021.

[3] J. Burkardt. Counting abscissas in sparse grids. 2014.

[4] G.T. Buzzard. Global sensitivity analysis using sparse grid interpolation and polynomial chaos. *Reliability Engineering & System Safety*, 107:82–89, 2012.

[5] W.N. Edeling. Sparse-grid tutorial EasyVVUQ notebooks. `https://github.com/wedeling/sparse_grid_tutorial`.

[6] W.N. Edeling, R.P. Dwight, and P. Cinnella. Simplex-stochastic collocation method with improved scalability. *Journal of Computational Physics*, 310:301–328, 2016.

[7] J. Feinberg and H.P. Langtangen. Chaospy: An open source tool for designing methods of uncertainty quantification. *Journal of Computational Science*, 11:46–57, 2015.

[8] Jochen Garcke et al. Sparse grid tutorial. *Mathematical Sciences Institute, Australian National University, Canberra Australia*, page 7, 2006.

[9] T. Gerstner and M. Griebel. Numerical integration using sparse grids. *Numerical Algorithms*, 3(18):209–232, 1998.

[10] T. Gerstner and M. Griebel. Dimension–adaptive tensor–product quadrature. *Computing*, 71(1):65–87, 2003.

[11] J.D. Jakeman, M.S. Eldred, G. Geraci, and A. Gorodetsky. Adaptive multi-index collocation for uncertainty quantification and sensitivity analysis. *International Journal for Numerical Methods in Engineering*, 121(6):1314–1343, 2020.

[12] J.L. Loeppky, B.J. Williams, and L.M. Moore. Global sensitivity analysis for mixture experiments. *Technometrics*, 55(1):68–78, 2013.

[13] D. Loukrezis, U. Römer, and H. De Gersem. Assessing the performance of leja and clenshaw-curtis collocation for computational electromagnetics with random input data. *International Journal for Uncertainty Quantification*, 9(1), 2019.

[14] X. Ma and N. Zabaras. An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations. *Journal of Computational Physics*, 228(8):3084–3113, 2009.

[15] T.A. Mara, S. Tarantola, and P. Annoni. Non-parametric methods for global sensitivity analysis of model output with dependent inputs. *Environmental modelling & software*, 72:173–183, 2015.

[16] H. Moon. *Design and analysis of computer experiments for screening input variables*. PhD thesis, The Ohio State University, 2010.

[17] R.A. Richardson, D.W. Wright, W. Edeling, V. Jancauskas, J. Lakhlili, and P.V. Coveney. Easyvvuq: a library for verification, validation and uncertainty quantification in high performance computing. *Journal of open research software*, 8(1), 2020.

[18] S.A. Smolyak. Quadrature and interpolation formulas for tensor products of certain classes of functions. In *Doklady Akademii Nauk*, volume 148, pages 1042–1045. Russian Academy of Sciences, 1963.

[19] I.M. Sobol. On sensitivity estimation for nonlinear mathematical models. *Matematicheskoe Modelirovanie*, 2(1):112–118, 1990.

[20] B. Sudret. Global sensitivity analysis using polynomial chaos expansions. *Reliability Engineering & System Safety*, 93(7):964–979, 2008.

[21] S. Surjanovic and D. Bingham. Virtual library of simulation experiments: Test functions and datasets. Retrieved March 16, 2022, from `http://www.sfu.ca/~ssurjano`.

[22] J.A.S. Witteveen and G. Iaccarino. Simplex stochastic collocation with ENO-type stencil selection for robust uncertainty quantification. *Journal of Computational Physics*, 239:1–21, 2013.

[23] D. Xiu. *Numerical methods for stochastic computations*. Princeton university press, 2010.