The Sparse Grids Matlab Kit user manual
v. 23-5 Robert

Chiara Piazzola[1,2] and Lorenzo Tamellini[1]

[1]Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes" - Consiglio Nazionale delle Ricerche, Via Ferrata, 5/A, 27100, Pavia, Italy
[2]Department of Mathematics - Technical University of Munich, Boltzmannstraße, 3, 85748, Garching bei München, Germany

October 9, 2023

Abstract

The Sparse Grids Matlab Kit provides a Matlab implementation of sparse grids, and can be used for approximating high-dimensional functions and, in particular, for surrogate-model-based uncertainty quantification. It is lightweight, high-level and easy to use, good for quick prototyping and teaching; however, it is equipped with some features that allow its use also in realistic applications. The goal of this paper is to provide an overview of the data structure and of the mathematical aspects forming the basis of the software, as well as comparing the current release of our package to similar available software.

Contents

1 Introduction

The Sparse Grids Matlab Kit (SGMK) implements the combination technique form of sparse grids for interpolation and quadrature of high-dimensional functions; it is geared towards Uncertainty Quantification (UQ) problems but general enough for other purposes. It is lightweight, high-level and (hopefully) easy to use, good for quick prototyping and teaching. However, some functionalities (evaluation recycling, buffering strategy for adaptive sparse grid generation, interface with Matlab parallelization and with other software by the *UM-Bridge* protocol [1]) make it convenient also for realistic applications.

The SGMK is freely available under the BSD2 license at `https://sites.google.com/view/sparse-grids-kit` (releases, documentation, release-related material) as well as on Github at `https://github.com/lorenzo-tamellini/sparse-grids-matlab-kit` (source code): the first version was released in 2014 (14-4 "Ritchie"), and the current version was released in 2023 (23-5 "Robert"). It is written in Matlab, and its compatibility with Octave has been tested. No installation is required, i..e, the SGMK folder just needs to be added to the working directory with the usual Matlab **`addpath`** command. The package is organized as illustrated in the following tree diagram:

```
Sparse Grids Matlab Kit
├── main
├── tools
├── src
└── docs-examples
```

Roughly speaking, the folder **`main`** collects all the "operative" functions: those for generating sparse grids, that will be discussed in Sect. 3.4, as well as the main functionalities (quadrature, interpolation, evaluation, plotting, computing derivatives, conversion to polynomial chaos expansions), that will be discussed in Sect. 4. The folder **`tools`** collects all the "building block" functions that are required to create a sparse grid (knots generation, level-to-knots computation, multi-index generation, see Sects. 3.1, 3.2, 3.3 respectively), as well as utility functions for e.g. evaluating polynomials, converting between data structures, and interfaces to Matlab parallel functions. In general, functions in this folder are functions that are ancillary to the functions in **`main`** but that the standard user of the SGMK will have to use. Conversely, the folder **`src`** contains source files with low-level functions that typically are not of interest of the standard users, but only for SGMK developers. Finally, the folder **`docs-examples`** collects complementary materials, such as several tutorials, a few test examples and a testing unit. The documentation provided in this user manual is further complemented by the additional documentation provided in the **`help`** of each function. Additionally, a thorough description of the mathematical background of functions and algorithms is provided in the main paper [2].

This manual is organized as follows: Sect. 2 introduces the minimal mathematical background necessary to understand the entities implemented in the SGMK. Then, Sect. 3 covers how sparse grids are generated in the SGMK, how they are stored in memory (data structure) and what options are available for their generation. Sect. 4 discusses the main functionalities made available by the SGMK, as well as how to interface the SGMK with external software, either by file exchange or by using the *UM-Bridge* protocol. Finally, Sect. 5 illustrates the usage of the SGMK over a simple (yet quite paradigmatic) problem of forward and inverse UQ for partial differential equations (PDEs). This final example should hopefully convey that it is quite straightforward to use the SGMK to perform a wide array of possible common tasks in UQ, with a relatively small number of lines of code. All the code snippets discussed in the paper, as well as the complete code for the final PDE example are available on the SGMK webpage.

2 A short overview on the basics of sparse grids

The aim of this section is to briefly recall the mathematical foundations of sparse grids, which guide the software implementation. We consider the two problems of a) approximating and b) computing weighted integrals of (the components of) a function $f : \mathbb{R}^N \to \mathbb{R}^V$ given some samples of f whose location we are free to choose. More specifically, we assume that f depends on N random variables $\mathbf{y} = (y_1, \dots, y_N) \in \Gamma$, with $\Gamma = \Gamma_1 \times \dots \times \Gamma_N \subset \mathbb{R}^N$ being the set of all possible values of $\mathbf{y}$. We denote by $\rho_n : \Gamma_n \to \mathbb{R}^+$ the probability density function (pdf) of each variable y_n, $n = 1, \dots, N$ and assume independence of $y_1, \dots, y_N$, such that the joint pdf of $\mathbf{y}$ is $\rho(\mathbf{y}) = \prod_{n=1}^{N} \rho_n(y_n), \forall \mathbf{y} \in \Gamma$.

The first step to build a sparse grid is to define a set of collocation knots for each variable y_n. We thus introduce the univariate discretization level $i_n \in \mathbb{N}_+$ and a so-called "level-to-knots function" that specifies the number of collocation knots to be used for each random variable, i.e.,

$$m : \mathbb{N}_+ \to \mathbb{N}_+, \; i_n \mapsto m(i_n). \tag{1}$$

Then, we denote by $\mathcal{T}_{n,i_n}$ the set of $m(i_n)$ discretization knots along y_n, i.e.,

$$\mathcal{T}_{n,i_n} = \left\{ y_{n,m(i_n)}^{(j_n)} : j_n = 1, \dots, m(i_n) \right\} \quad \text{for } n = 1, \dots, N. \tag{2}$$

Such sequence is usually chosen according to the probability distribution of the random variables ρ_n for efficiency reasons. Furthermore, for efficiency purposes it is beneficial if the sequences of knots are *nested*, i.e., if $\mathcal{T}_{n,i_n} \subset \mathcal{T}_{n,j_n}$ when $j_n \geq i_n$. These aspects will be further discussed in Sect. 3.1.

Next, we introduce the N-dimensional tensor grids, that are obtained by taking the Cartesian product of the N univariate sets of knots just introduced. For this purpose we collect the discretization levels i_n in a multi-index $\mathbf{i} = [i_1, \dots, i_N] \in \mathbb{N}_+^N$ and denote the corresponding tensor grid by $\mathcal{T}_\mathbf{i} = \bigotimes_{n=1}^N \mathcal{T}_{n,i_n}$. Using standard multi-index notation, we can then write

$$\mathcal{T}_\mathbf{i} = \left\{ \mathbf{y}_{m(\mathbf{i})}^{(\mathbf{j})} \right\}_{\mathbf{j} \leq m(\mathbf{i})}, \quad \text{with} \quad \mathbf{y}_{m(\mathbf{i})}^{(\mathbf{j})} = \left[y_{1,m(i_1)}^{(j_1)}, \dots, y_{N,m(i_N)}^{(j_N)} \right] \text{ and } \mathbf{j} \in \mathbb{N}_+^N,$$

where $m(\mathbf{i}) = [m(i_1), m(i_2), \dots, m(i_N)]$ and $\mathbf{j} \leq m(\mathbf{i})$ means that $j_n \leq m(i_n)$ for every $n = 1, \dots, N$.

A *tensor grid approximation* of $f(\mathbf{y})$ based on global Lagrangian interpolants collocated at these grid knots can then be written in the following form

$$\mathcal{U}_\mathbf{i}(\mathbf{y}) := \sum_{\mathbf{j} \leq m(\mathbf{i})} f\left(\mathbf{y}_{m(\mathbf{i})}^{(\mathbf{j})} \right) \mathcal{L}_{m(\mathbf{i})}^{(\mathbf{j})}(\mathbf{y}), \tag{3}$$

where $\left\{ \mathcal{L}_{m(\mathbf{i})}^{(\mathbf{j})}(\mathbf{y}) \right\}_{\mathbf{j} \leq m(\mathbf{i})}$ are the N-variate Lagrange basis polynomials, defined as tensor products of univariate Lagrange polynomials, i.e.

$$\mathcal{L}_{m(\mathbf{i})}^{(\mathbf{j})}(\mathbf{y}) = \prod_{n=1}^N l_{n,m(i_n)}^{(j_n)}(y_n) \quad \text{with} \quad l_{n,m(i_n)}^{(j_n)}(y_n) = \prod_{k=1, k \neq j_n}^{m(i_n)} \frac{y_n - y_{n,m(i_n)}^{(k)}}{y_{n,m(i_n)}^{(j_n)} - y_{n,m(i_n)}^{(k)}}.$$

Similarly, the *tensor grid quadrature* of $f(\mathbf{y})$, i.e. the approximation of the weighted integral of its components, can be computed by taking the integral of the Lagrangian interpolant in Eq. (2):

$$\begin{aligned} \mathcal{Q}_\mathbf{i} := \int_\Gamma \mathcal{U}_\mathbf{i}(\mathbf{y}) \rho(\mathbf{y}) \, \mathrm{d}\mathbf{y} &= \sum_{\mathbf{j} \leq m(\mathbf{i})} f\left(\mathbf{y}_{m(\mathbf{i})}^{(\mathbf{j})} \right) \left(\prod_{n=1}^N \int_{\Gamma_n} l_{n,m(i_n)}^{(j_n)}(y_n) \rho(y_n) \, \mathrm{d}y_n \right) \\ &= \sum_{\mathbf{j} \leq m(\mathbf{i})} f\left(\mathbf{y}_{m(\mathbf{i})}^{(\mathbf{j})} \right) \left(\prod_{i=1}^N \omega_{n,m(i_n)}^{(j_n)} \right) = \sum_{\mathbf{j} \leq m(\mathbf{i})} f\left(\mathbf{y}_{m(\mathbf{i})}^{(\mathbf{j})} \right) \omega_{m(\mathbf{i})}^{(\mathbf{j})}, \end{aligned} \tag{4}$$

where $\omega_{n,m(i_n)}^{(j_n)}$ are the standard quadrature weights obtained by computing the integrals of the associated univariate Lagrange polynomials, and $\omega_{m(\mathbf{i})}^{(\mathbf{j})}$ are their multi-variate counterparts.

Finally, building upon the concepts just introduced, the *sparse grid approximation* of $f(\mathbf{y})$ and of its weighted integral (cf. tasks a) and b) above), can be written as (see e.g. [2, Section 2] for the full derivation of this formula):

$$f(\mathbf{y}) \approx \mathcal{U}_\mathcal{I}(\mathbf{y}) = \sum_{\mathbf{i} \in \mathcal{I}} c_\mathbf{i} \, \mathcal{U}_\mathbf{i}, \quad c_\mathbf{i} := \sum_{\substack{\mathbf{j} \in \{0,1\}^N \\ \mathbf{i} + \mathbf{j} \in \mathcal{I}}} (-1)^{\|\mathbf{j}\|_1} \tag{5}$$

$$\int_\Gamma f(\mathbf{y}) \rho(\mathbf{y}) \, \mathrm{d}\mathbf{y} \approx \mathcal{Q}_\mathcal{I}(\mathbf{y}) = \sum_{\mathbf{i} \in \mathcal{I}} c_\mathbf{i} \, \mathcal{Q}_\mathbf{i}, \tag{6}$$

with $\mathcal{I} \subset \mathbb{N}_+^N$ denoting the multi-index set collecting the multi-indices in the sum. The *sparse grid* is then defined as

$$\mathcal{T}_\mathcal{I} = \bigcup_{\substack{\mathbf{i} \in \mathcal{I} \\ c_\mathbf{i} \neq 0}} \mathcal{T}_\mathbf{i}. \tag{7}$$

The equalities in Eqs. (5) and (6) are known as "combination technique" form of the sparse grids approximation and quadrature, and are valid only if $\mathcal{I}$ is chosen as downward closed, i.e.

$$\forall \mathbf{k} \in \mathcal{I}, \quad \mathbf{k} - \mathbf{e}_n \in \mathcal{I} \text{ for every } n = 1, \dots, N \text{ such that } k_n > 1. \tag{8}$$

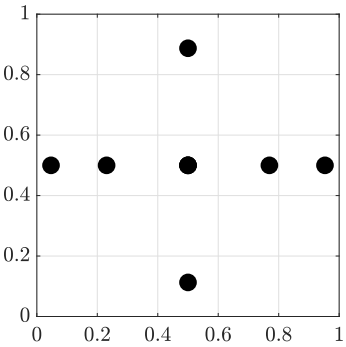


Figure 1: Sparse grid generated by Listing 1.

3 Generating sparse grids

Equations (5), (6), and (7) are the formulas that constitute the backbone of the implementation of the SGMK. We illustrate the package in two steps: in this section we discuss the creation of the sparse grid data structure, whereas in the next section we describe the main operations that can be done (interpolation, quadrature, etc).

The basic creation of a sparse grid is as easy as the following code snippet (Listing 1). We report it and then spend the rest of the section dissecting these lines and discussing the alternative options available for the same operations.

```
1 knots = @(n) knots_uniform(n,0,1);    % Gauss-Legendre knots on [0,1]
2 lev2knots = @lev2knots_doubling;
3 I = [1 1;
4      1 2;
5      2 1;
6      3 1];
7 S = create_sparse_grid_multiidx_set(I,knots,lev2knots);
8 Sr = reduce_sparse_grids(S);
9 plot_sparse_grid(S,[],'ok','LineWidth',3,'MarkerFaceColor','k','MarkerSize',12)
```

Listing 1: Basic creation of a sparse grid.

The first six lines declare the "ingredients" of the sparse grid, while the actual construction of the grid is delegated to Lines 7 and 8. The command **plot_sparse_grids** at Line 9 generates the plot of the sparse grid reported in Fig. 1. Most of the lines have a one-to-one correspondence with the mathematical description of sparse grids presented in the previous section:

- **knots** is the function to generate the univariate set of collocation knots $\mathcal{T}_{n,i_n}$ in Eq. (2);
- **lev2knots** is the level-to-knots function m in Eq. (1);
- **I** is the multi-index set $\mathcal{I}$ in Eqs. (5), (6), (7);
- **S** contains the sparse grid understood as collection of tensor grids $\mathcal{T}_{\mathbf{i}}$, $\mathbf{i} \in \mathcal{I}$, $c_{\mathbf{i}} \neq 0$;
- **Sr** contains the sparse grid understood as the union of tensor grids without repetitions of knots in common, see Eq. (7). We discuss more the generation of **Sr** in Sec. 3.4.1.

Next, we present and discuss the alternatives available in the SGMK for each of the operations in the listing above: assignment of the function **knots** is discussed in Sect. 3.1, assignment of the function **lev2knots** is discussed in Sect. 3.2, construction of the multi-index set **I** is discussed in Sect. 3.3, and finally construction of **S** and **Sr**, and the details of the respective data structure are discussed in Sect. 3.4. Note that the alternatives for the first three operations are provided in the **tools** folder

```
tools
├── knots_functions
├── lev2knots_functions
└── idxset_functions
```

whereas the functions for the sparse-grids generation are available in the **main** folder.

	domain	parameters	pdf
uniform	$\Gamma = [a, b]$	$a, b \in \mathbb{R}$	$\rho(y) = \frac{1}{b-a}$
normal	$\Gamma = \mathbb{R}$	$\mu, \sigma \in \mathbb{R}$	$\rho(y) = \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-\frac{(y-\mu)^2}{2\sigma^2}}$
exponential	$\Gamma = [0, \infty)$	$\lambda \in \mathbb{R},\ \lambda > 0$	$\rho(y) = \lambda \mathrm{e}^{-\lambda y}$
gamma	$\Gamma = [0, \infty)$	$\alpha, \beta \in \mathbb{R},\ \alpha > -1, \beta > 0$	$\rho(y) = \frac{\beta^{\alpha+1}}{\Gamma(\alpha+1)} x^{\alpha} \mathrm{e}^{-\beta y}$
beta	$\Gamma = [a, b]$	$a, b, \alpha, \beta \in \mathbb{R},\ \alpha, \beta > -1$	$\rho(y) = \frac{\Gamma(\alpha+\beta+2)}{\Gamma(\alpha+1)\Gamma(\beta+1)(b-a)^{\alpha+\beta+1}} (y-a)^{\alpha}(b-y)^{\beta}$
triangular	$\Gamma = [a, b]$	$a, b \in \mathbb{R}$	$\rho(y) = \frac{2}{(b-a)^2}(b-y)$

Table 1: Random variables and corresponding pdfs.

	knots	function	nestedness
uniform	Gauss–Legendre	`knots_uniform`	No
	Clenshaw–Curtis	`knots_CC`	Yes for suitable m
	Leja (standard, symmetric, P-disk)	`knots_leja`	Yes
	Equispaced	`knots_trap`	Yes for suitable m
	Midpoint	`knots_midpoint`	Yes for suitable m
normal	Gauss–Hermite	`knots_normal`	No
	Genz–Keister	`knots_GK`	Yes
	weighted Leja (standard, symmetric)	`knots_normal_leja`	Yes
exponential	Gauss–Laguerre	`knots_exponential`	No
	weighted Leja	`knots_exponential_leja`	Yes
gamma	Gauss–generalized Laguerre	`knots_gamma`	No
	weighted Leja	`knots_gamma_leja`	Yes
beta	Gauss–Jacobi	`knots_beta`	No
	weighted Leja (standard, symmetric)	`knots_beta_leja`	Yes
triangular	weighted Leja	`knots_triangular_leja`	Yes

Table 2: Different type of collocation knots, corresponding function of the SGMK and a flag on the nestedness property.

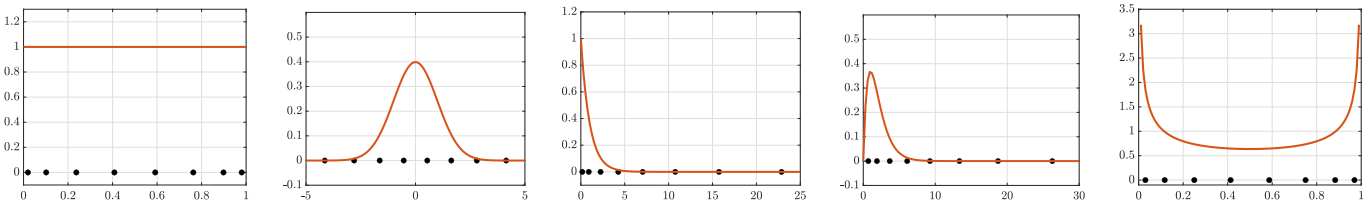
3.1 Knots functions

We begin by addressing the different types of collocation knots that are implemented in the SGMK, i.e., the possible options to replace the assignment in Line 1 of Listing 1, `knots = @(n) knots_uniform(n,0,1)`. As already hinted, the knots should be chosen according to the pdf of the corresponding random variables y_n; in Tab. 1 we report the random variables that are considered in the SGMK, along with the expression of the respective pdf, as well as the parameters on which the pdfs depend.

The SGMK provides two families of knots (Gauss-type knots and Leja-type knots) that can be used to construct collocation knots for almost all the random variables listed in Tab. 1, and some additional choices that are available only for certain random variables: specifically, Clenshaw–Curtis, equispaced knots, and midpoint knots for uniform random variables, and Genz–Keister knots for normal random variables. We give details on all these families of knots in the next paragraphs, but we mention already that one big distinction is that some of these families of knots are nested whereas others are not, and one should in general favor knots of the former class, as stated already in Sect. 2. More specifically, Gauss-type knots are not nested, whereas all the other choices are nested. The full list of knots available in the SGMK and the corresponding commands to generate them are reported in Tab. 2, together with a flag to indicate whether a specific choice is nested or not.

Gauss-type knots Given a pdf, Gauss-type collocation knots are defined as the zeros of the corresponding orthogonal polynomials:

- Legendre polynomials for uniform random variables;
- Hermite polynomials for normal (gaussian) random variables;
- Laguerre polynomials for exponential random variables;
- generalized Laguerre polynomials for gamma random variables;



(a) Gauss–Legendre knots for uniform variables on $[0, 1]$

(b) Gauss–Hermite knots for standard normal variables

(c) Gauss–Laguerre knots for exponential variables with $\lambda = 1$

(d) Gauss–generalized Laguerre knots for gamma variables with $\alpha = 2$ and $\beta = 1$

(e) Gauss–Jacobi knots for beta variables on $[0, 1]$ with $\alpha = \beta = 0.5$

Figure 2: Gauss-type knots with the corresponding pdfs.

- Jacobi polynomials for beta random variables.

The weights can be formally computed by taking the integral of the corresponding Lagrange polynomials. However, in practice, knots and weights are more conveniently derived from the eigenvalue decomposition of a suitable matrix, see e.g. [3] for further details. Note that Gauss-type quadrature rules employing k knots have degree of exactness $2k - 1$. In Fig. 2 we display the first 8 Gauss-type knots for the different random variables listed above.

Leja knots Leja knots were first introduced for unweighted interpolation on univariate and bivariate domains, see e.g. [4], and are therefore a suitable choice when y_n are uniform random variables on the interval $[a, b]$. They are built recursively as

$$t^{(1)} = b,\ t^{(2)} = a,\ t^{(3)} = \frac{a+b}{2},\ \text{and } t^{(j)} = \text{argmax}_{t\in[a\,b]} \prod_{k=1}^{j-1} |t - t^{(k)}|, \tag{9}$$

and the corresponding quadrature weights are obtained by evaluating the integrals of the Lagrangian polynomials in Eq. (4) by Gauss-type quadrature rules with appropriate degree of exactness. Observe that by construction Leja knots are nested.

A limitation of Leja knots is that they are not symmetric with respect to the mid-point $\frac{a+b}{2}$ of the interval. However, sequences of symmetric knots can be constructed generating only the even elements of the sequence with the standard formula in Eq. (9) and then symmetrizing them to obtain the odd elements, i.e.

$$\begin{aligned} &t^{(1)} = b,\ t^{(2)} = a,\ t^{(3)} = \frac{a+b}{2}, \\ &t^{(2j)} = \text{argmax}_{t\in[a\,b]} \prod_{k=1}^{2j-1} |t - t^{(k)}|, \\ &t^{(2j+1)} = \frac{a+b}{2} - \left(t^{(2j)} - \frac{a+b}{2}\right). \end{aligned} \tag{10}$$

Another class of Leja-type knots, called P-disk knots, can be obtained projecting on the real axis the Leja knots generated on the complex unit ball. P-disk knots are then naturally defined on $[-1, 1]$, and they amount to:

$$t^{(j)} = \cos\phi_j\,, \qquad \text{where } \begin{cases} \phi_1 = 0,\ \ \phi_2 = \pi,\ \ \phi_3 = \frac{\pi}{2}, \\ \phi_{2j+2} = \frac{\phi_{j+2}}{2}, \\ \phi_{2j+3} = \phi_{2j+2} + \pi. \end{cases} \tag{11}$$

Of course, a linear transformation needs to be applied for random variables on the general interval $[a, b]$. For more details on this family of knots we refer to [5]. The first 9 knots of the three classes just described are plotted in Fig. 3a.

Remark. *Leja sequences are not unique, since the function to be maximized at each iteration might have more than one maximum; this fact is well-known in literature, see e.g. [6, 7]. In our implementation, at each iteration j the maximization problem is solved by a "brute–force zooming" algorithm: we look for the maximum over a grid in $[a, b]$ with increasingly small step-size, over a window that narrows around the location of the current maximum; note that the initial step-size is chosen small enough that the maximum is always guaranteed to be the global maximum and not a local*

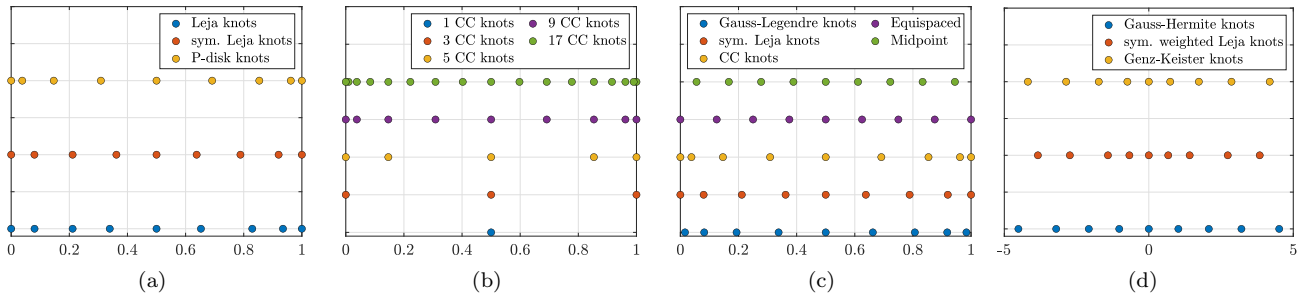


Figure 3: (a)-(c): Collocation knots for uniform random variables on $[0, 1]$: (a) Leja knots; (b) nested Clenshaw–Curtis (CC) knots; (c) comparison of the different types of knots. (d): Collocation knots for standard normal random variables.

one. This computation is only performed once and then collocation knots are saved on file, therefore its computational cost is not an issue. When multiple global maxima are present, the implemented algorithm selects the smallest one (with sign). The function that computes the Leja knots is not *released as a* **`function`** *per se in the SGMK but can be found at the beginning of the tutorial* **`docs-examples/test_compute_normal_leja_and_convergence_test.m`** *(with minor variations due to the fact that such tutorial is discussing* weighted Leja *knots – more about these knots in the following).*

Weighted Leja knots It is possible to extend the construction of Leja knots to the case when y_n are random variables with a probability distribution other than uniform, see [7]. These knots are the so-called weighted Leja knots and can be computed again recursively, by suitably introducing a weight in Eq. (9), i.e., solving

$$t^{(j)} = \text{argmax}_{t\in\Gamma} \sqrt{\rho(t)} \prod_{k=1}^{j-1} (t - t^{(k)}),$$

where ρ is the pdf of the random variable and Γ its domain. The same computational considerations discussed for the standard (non-weighted) Leja knots are valid for the weighted Leja knots as well. As for standard Leja sequences, weighted sequences can be symmetrized, extending analogously Eq. (10). In the SGMK, weighted Leja knots are available for all supported random variables, whereas symmetric weighted Leja knots are available only for random variables with symmetric pdfs, namely for normal and beta random variables (in the latter case, only when the pdf parameters α, β are identical, cf. Tab. 1).

Clenshaw–Curtis knots Besides the variants of Leja knots, the so-called Clenshaw–Curtis knots are another nested alternative to Gauss-type knots in case of uniform random variables. They are computed on $[-1, 1]$ as

$$t_K^{(j)} = \cos\left(\frac{(j-1)\pi}{K-1}\right), \quad 1 \le j \le K,$$

and then linearly transformed to any generic interval $[a, b]$; the corresponding weights can be efficiently computed by Fast Fourier Transform, see [8]. Quadrature formulas based on this family of knots result to be as accurate as the Gauss-type quadrature. Moreover, two sets of Clenshaw-Curtis knots with K_1 and K_2 knots are nested if $(K_2 - 1)/(K_1 - 1) = 2^l$ for some integer l. Such sequences are obtained e.g. for $K = 1, 3, 5, 9, 17$, see Fig. 3b.

Equispaced and midpoint knots These two sequences of knots are provided for uniform random variables only. They provide classical low-order quadrature formulas [9] and can therefore be useful for quadrature of functions with low regularity. In particular:

- Equispaced knots in $[a, b]$ are provided by the function **`knots_trap`** that implements the trapezoidal quadrature rule with knots $t_K^{(j)}$ and weights $\omega_K^{(j)}$

$$\begin{cases} t_K^{(j)} = a + h(j-1) & 1 \le j \le K \\ \omega_K^{(1)} = \omega_K^{(K)} = h/2 & \\ \omega_K^{(j)} = h & 2 \le j \le K-1 \end{cases} \quad \text{with } h = \frac{1}{K-1}.$$

	definition	function
linear	$m(i_n) = i_n$	`lev2knots_lin`
2-step	$m(i_n) = 2(i_n - 1) + 1$	`lev2knots_2step`
doubling	$m(1) = 1$, $m(i_n) = 2^{i_n - 1} + 1$	`lev2knots_doubling`
tripling	$m(i_n) = 3^{i_n - 1}$	`lev2knots_tripling`
Genz–Keister	$m(1) = 1, m(2) = 3, m(3) = 9$, $m(4) = 19$, $m(5) = 35$	`lev2knots_GK`

Table 3: Types of level-to-knots functions and corresponding function of the SGMK.

Similarly to Clenshaw–Curtis knots, two sets of equispaced knots with K_1 and K_2 knots are nested if $(K_2 - 1)/(K_1 - 1) = 2^l$ for some integer l.

- Midpoint knots are provided by the function `knots_midpoint`, that implements the midpoint quadrature rule:

$$\begin{cases} t_K^{(j)} = a + \dfrac{h}{2} + h(j-1) & 1 \le j \le K \\ \omega_K^{(j)} = h & 1 \le j \le K \end{cases} \quad \text{with } h = \frac{1}{K}.$$

  Two sets of midpoint knots with K_1 and K_2 knots are nested if $K_2/K_1 = 3^l$ for some integer l.

Observe that these two rules depart a bit from the general framework of the package, in that they do not derive quadrature weights as integral of global Lagrange polynomials, but rather as integrals of *piecewise* Lagrange polynomials (linear and constant, respectively). However, we remark that the package does *not* provide piecewise interpolation, but only global interpolation. Therefore, if one were to use these knots to build a sparse grid *interpolant* of a function, Runge phenomena can be expected, leading to poor approximations. These knots are thus to be intended essentially as quadrature knots only. To summarize, in Fig. 3c we give a comparison of the types of collocation knots available for uniform random variables.

Genz–Keister knots The sequence of Genz–Keister knots is obtained by modifying the sequence of Gauss–Hermite knots to enforce nestedness, see [10]. The knots and the corresponding weights are tabulated and available for $K = 1, 3, 9, 19, 35$. In Fig. 3d we display the first 9 Genz–Keister knots, together with Gauss-type and symmetric weighted Leja knots for normal random variables.

3.2 Level-to-knots functions

The above list of knots can be used in conjunction with different types of level-to-knots functions (see Eq. (1)) that associate the discretization levels collected in the multi-index $\mathbf{i}$ to the number of collocation knots to be used; of course, some choices are smarter than others, such as those that guarantee nestedness. For example, in the case of Clenshaw–Curtis knots, any number of knots can be used, but a specific relation must hold between the cardinality of the sequences of knots such that they are nested (see discussion above). In the case of symmetric Leja knots instead, it is natural to use a level-to-knots function that adds two knots (or, more generally, an even number of knots) at each level, to make sure that sequences that are constructed by symmetrization are actually used as such.

In Tab. 3 we list the level-to-knots functions that are implemented in the SGMK, which are the possible options to replace the assignment in Line 2 of Listing 1, `lev2knots = @lev2knots_doubling`. The function *linear* can be used for Gauss-type and Leja knots: however, as already pointed out, in case of symmetric Leja knots it is convenient to resort to the functions of type *2-step* or *doubling*. The function of type *doubling* is essential for the generation of sequences of nested Clenshaw–Curtis and equispaced knots; the function *tripling* must be used to obtain nested midpoint knots, and the tabulated function *Genz–Keister* is specific for the case of Genz–Keister knots. Table 4 gives an overview on the possible combination of knots and level-to-knots functions that generate sequences of nested collocation knots.

3.3 Multi-index sets

As discussed in [2, Section 2], the quality of the sparse grid approximations (5) and (6) depends on the choice of the multi-index set $\mathcal{I}$. One approach to this end is to design the set based on some knowledge on the problem at hand, such as the smoothness of the function to be approximated; an alternative "a-posteriori"/adaptive approach will be discussed later on.

	knots	level-to-knots functions for nestedness
uniform	Leja - standard	linear, 2-step, doubling
	Leja - symmetric	linear, 2-step, doubling
	Leja - P-disk	linear, 2-step, doubling
	Clenshaw–Curtis	doubling
	equispaced	doubling
	midpoint	tripling
normal	weighted Leja - standard	linear, 2-step, doubling
	weighted Leja - symmetric	linear, 2-step, doubling
	Genz–Keister	Genz–Keister
exponential	weighted Leja	linear, 2-step, doubling
gamma	weighted Leja	linear, 2-step, doubling
beta	weighted Leja - standard	linear, 2-step, doubling
	weighted Leja - symmetric	linear, 2-step, doubling
triangular	weighted Leja - standard	linear, 2-step, doubling

Table 4: Families of knots with the corresponding level-to-knots functions to ensure nestedness.

Note that multi-indices are always stored in the SGMK as *row vectors* with N components, and multi-index sets are correspondingly stored as matrices, each row being a multi-index; moreover, it is required that such matrices are in *lexicographic order* by row. This check is of course *not* delegated to the user (SGMK functions operating on multi-index sets will check ordering and throw an error if a set is not sorted).

Another requirement on multi-index sets is the already mentioned downward-closedness property, see Eq. (8): to this end, the SGMK provides the function **`check_set_admissibility(I)`**, which in case of non-downward-closedness of the set **`I`** also returns the smallest downward-closed set that includes **`I`**. For visualization purposes, the SGMK also provides the function **`plot_multiidx_set`** (which supports only the cases $N = 2, N = 3$).

Small multi-index sets can be typed in "manually", such as in Lines 3-6 of Listing 1, but of course dedicated functions are available. In particular **`multiidx_box_set(jj)`**, where **`jj`** is a multi-index, generates the hyper-rectangular (a.k.a. "box") set $\mathcal{I} = \{\mathbf{i} \in \mathbb{N}_+^N : \mathbf{i} \leq \mathbf{j}\}$. Furthermore, most multi-index sets that can be written as $\mathcal{I} = \{\mathbf{i} \in \mathbb{N}_+^N : r(\mathbf{i}) \leq w\}$ for $r : \mathbb{N}_+^N \to \mathbb{R}_+$ and $w \in \mathbb{N}_+$ can be created invoking the function **`multiidx_gen`** as follows

```
1 N = 2;
2 rule = @(ii) sum(ii-1);
3 w = 4;
4 base = 1;
5 I = multiidx_gen(N,rule,w,base);
```

Listing 2: Creation of a multi-index set.

where **`rule`** is a **`@`**-function such that **`rule(ii)`** evaluates $r(\mathbf{i})$, and the integer value w defines the size of the multi-index set, and hence controls the quality of the associated sparse grid approximation. In the following, we refer to r as *rule of the approximation* and to w as *level of the approximation*. The implementation of **`multiidx_gen`** is recursive over n, $n = 1, \ldots, N$. This is a flexible solution that can accommodate for a wide range of rules r (although admittedly not the most efficient speed-wise): the only requests are that r is non-decreasing in each argument, and that $r(\mathbf{j}) > w \Rightarrow r([\mathbf{j}, 1]) > w, \forall \mathbf{j} \in \mathbb{N}_+^n, n < N$.

A list of multi-index sets $\mathcal{I}$ commonly found in literature and the implementation of the corresponding **`rule`** in Matlab is available in Tab. 5. The vectors $\mathbf{g} \in \mathbb{R}_+^N$ appearing in the definitions in Tab. 5[1] are the so-called *anisotropy weights*, whose purpose is to tune the shape of the multi-index set in such a way that more collocation knots are placed in the random variables deemed to be more important; in particular, the larger the weight, the more penalized the corresponding variable, i.e. the lower the number of collocation knots placed on that random variable. Whenever $g_1 = \ldots = g_N$, the resulting grid is named *isotropic*, and *anisotropic* otherwise. Finally, we clarify the role of the line **`base=1`** in Listing 2: its purpose is to specify that the smallest entry of each multi-index is 1 (another option would be **`base=0`**; multi-index sets like these will be needed later on, in Sect. 4.4). We further illustrate the use of **`multiidx_gen`** with the help of the following example.

Example 1 (Multi-index sets)**.** *The two-dimensional isotropic multi-index set $\mathcal{I}_{sum}(4)$ can be generated as in Listing 2. For the case of anisotropic sets we have to further specify the vector of weights* $\mathbf{g}$*:*

[1]The peculiar writing **`g(1:length(ii))`** is due to fact that **`multiidx_gen`** works recursively on the directions $n = 1$ to N, and therefore we need to be able to call the **`@`**-function also when the length of **`g`** and **`ii`** do not match.

definition	function
$\mathcal{I}_{\max}(w) = \left\{\mathbf{i} \in \mathbb{N}_+^N : \max_n g_n(i_n - 1) \leq w\right\}$	`rule = @(ii) max(g(1:length(ii)).*(ii-1))`
$\mathcal{I}_{\text{sum}}(w) = \left\{\mathbf{i} \in \mathbb{N}_+^N : \sum_{n=1}^N g_n(i_n - 1) \leq w\right\}$	`rule = @(ii) sum(g(1:length(ii)).*(ii-1))`
$\mathcal{I}_{\text{prod}}(w) = \left\{\mathbf{i} \in \mathbb{N}_+^N : \prod_{n=1}^N i_n^{g_n} \leq w\right\}$	`rule = @(ii) prod((ii).^g(1:length(ii)))`

Table 5: Multi-index sets and corresponding definitions in Matlab.

name	set generated by the rule	level-to-knots functions
`'TP'`	$\mathcal{I}_{\max}$	`linear`
`'TD'`	$\mathcal{I}_{\text{sum}}$	`linear`
`'HC'`	$\mathcal{I}_{\text{prod}}$	`linear`
`'SM'`	$\mathcal{I}_{\text{sum}}$	`doubling`

Table 6: Pre-sets available in **`define_function_for_rule`**. Rules are defined in Tab. 5, level-to-knots functions in Tab. 3.

```
1 N = 2;
2 g = [1,2];
3 rule = @(ii) sum(g(1:length(ii)).*(ii-1));
4 w = 4;
5 base = 1;
6 I_aniso = multiidx_gen(N,rule,w,base);
7 plot_multiidx_set(I_aniso,'sk','LineWidth',2,'MarkerSize',22,'MarkerFaceColor','b')
```

Figure 4 (top-row) shows the isotropic and anisotropic version of the index sets, together with the corresponding sparse grids, generated using symmetric Leja knots and level-to-knots function of type 2-step. The code to generate such grids is almost identical to the one in Listing 1 and will be discussed in details later on.

The following rows of the figure show multi-index sets and grids for the other choices of sets introduced in Tab. 5, i.e., $\mathcal{I}_{max}$ (mid-row of the figure) and $\mathcal{I}_{prod}$ (bottom-row of the figure), again in their isotropic and anisotropic versions (left and right part of the figure, respectively). The code to generate these multi-index sets is identical to the listing above (other than changing Line 3 with the appropriate definition from Tab. 5).Note that $\mathcal{I}_{max}$ sets can be equivalently generated with the command ***`multiidx_box_set`****. Finally, we also mention that for faster generation of $\mathcal{I}_{sum}$, the function* ***`fast_TD_set`*** *is available - the reason for this naming will be clearer later on.*

Typing rules of Tab. 5 might be inconvenient, especially if anisotropic rules are needed. To this end, the SGMK provides the convenience function **`define_functions_for_rule`** that takes as input a string ("the name of the sparse grid") and returns the associated rule and level-to-knots function for a number of common cases (see Tab. 6):

```
1 [lev2knots,rule] = define_functions_for_rule(<string-name>)
```

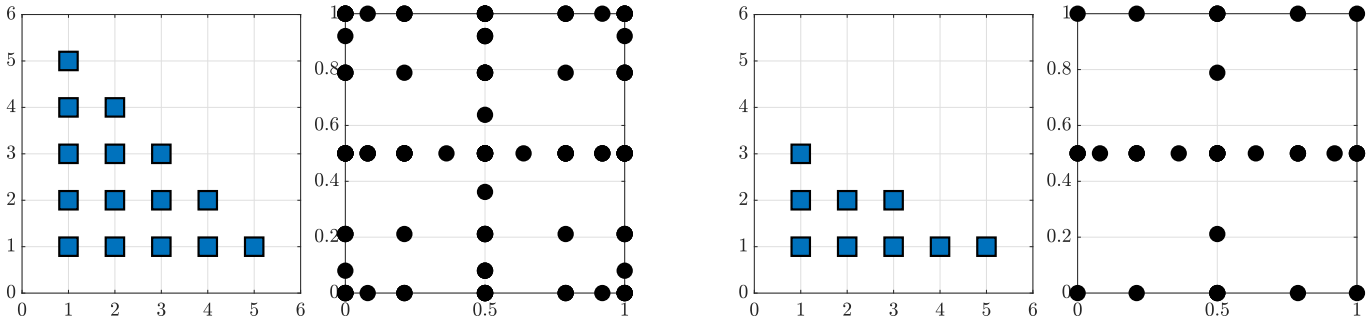
The first three names (`'TP'`,`'TD'`,`'HC'`) are borrowed from the literature on polynomial approximation spaces, and refer to the Tensor Product, Total Degree, and Hyberbolic Cross spaces, respectively, see [2, Section 2]. The fourth name `'SM'` refers to one of the most used type of sparse grid, the so-called *Smolyak* grid, which is obtained combining $\mathcal{I}_{\text{sum}}$ with the level-to-knots function of type *doubling* (the complete example will be given in Listing 3).

3.4 Sparse grid generation and data structure

We are finally ready to discuss Lines 7 and 8 of Listing 1, that are responsible of generating the sparse grid. The approach reported in the snippet can be called *a-priori*, since it requires to specify the multi-index set $\mathcal{I}$ *before* sampling the function f. In this section we detail this approach first, and then discuss also the "dual" approach, i.e. the *adaptive* algorithm to generate sparse grids, in which the multi-index set $\mathcal{I}$ is computed *simultaneously* to the sampling of the function f (Sect. 3.4.2). Note that regardless of the way in which sparse grids are generated, they are eventually stored with the same data structure, that we describe below while discussing the *a-priori* approach.

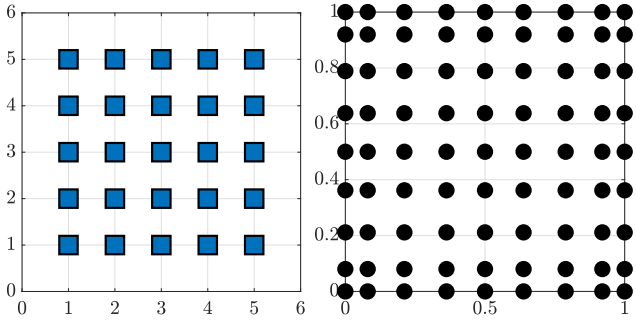
3.4.1 A-priori sparse grids

Listing 1 shows a first way to generate a sparse grid with the multi-index set determined a-priori: directly typing in the set, (or using the functions **`multiidx_box_set`**, **`multiidx_gen`**, or **`define_functions_for_rule`**), and then using

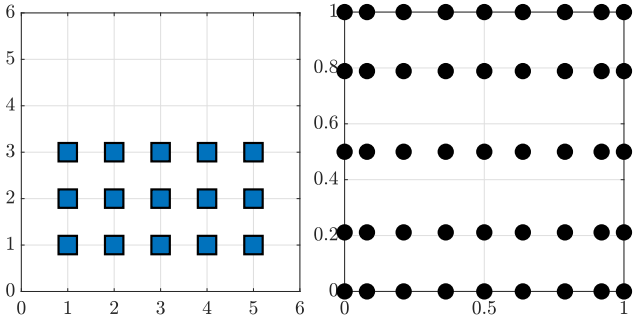


(a) Isotropic $\mathcal{I}_{\text{sum}}(4)$ (left) and corresponding grid (right)

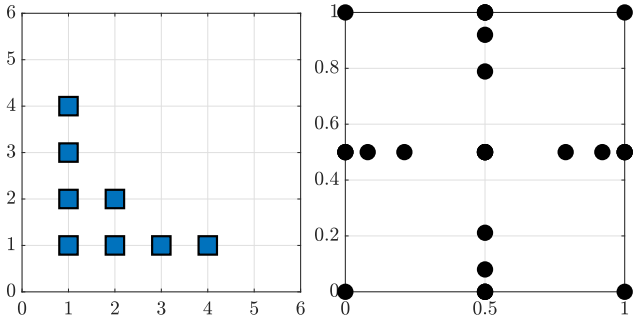
(b) Anisotropic $\mathcal{I}_{\text{sum}}(4)$ (left) and corresponding grid (right)



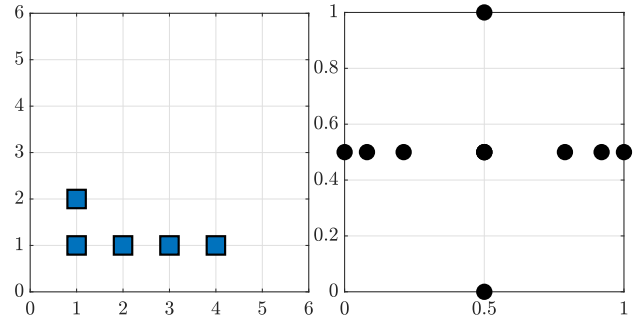
(c) Isotropic $\mathcal{I}_{\text{max}}(4)$ (left) and corresponding grid (right)



(d) Anisotropic $\mathcal{I}_{\text{max}}(4)$ (left) and corresponding grid (right)



(e) Isotropic $\mathcal{I}_{\text{prod}}(4)$ (left) and corresponding grid (right)



(f) Anisotropic $\mathcal{I}_{\text{prod}}(4)$ (left) and corresponding grid (right)

Figure 4: Isotropic and anisotropic multi-index sets and corresponding sparse grids of symmetric Leja knots on $\Gamma = [0,1]^2$ generated using level-to-knots function of type 2-step.

the function **create_sparse_grid_multiidx_set**. Another possibility is to call the command **create_sparse_grid**, that takes care of generating both the multi-index set given the rule and the associated sparse grid in one call, as shown in the following snippet.

```
N = 2;
w = 3;
knots = @(n) knots_CC(n,0,1);
lev2knots = @lev2knots_doubling;
rule = @(ii) sum(ii-1);
S = create_sparse_grid(N,w,knots,lev2knots,rule);
```

Listing 3: Basic creation of a sparse grid given a multi-index set defined by a rule.

The output of **create_sparse_grid** (and of **create_sparse_grid_multiidx_set**) is a sparse grid in the so-called *extended format*, that stores separately the information of each tensor grid composing the sparse grid (more precisely, of those grids whose coefficient in the combination technique formula is non-zero, see Eqs. (5), (6), and (7)). The data structure chosen to this end is a structure array, where each structure identifies one of the tensor grids:

```
>> S
S =

1x7 struct array with fields:

knots
weights
size
knots_per_dim
m
coeff
idx
```

Each tensor grid structure contains the following fields:

- **idx**: the multi-index $\mathbf{i} \in \mathcal{I}$ corresponding to the current tensor grid;
- **knots**: matrix collecting the knots $\mathcal{T}_{\mathbf{i}}$, each knot being a column vector;
- **weights**: vector of the quadrature weights $\omega^{(\mathbf{j})}_{m(\mathbf{i})}$ corresponding to the knots;
- **size**: size of the tensor grid, i.e. the number of knots $M_{\mathbf{i}} = \prod_{n=1}^{N} m(i_n)$;
- **knots_per_dim**: cell array with N components, each component collecting in an array the set of one-dimensional knots $\mathcal{T}_{n,i_n}$ used to build the tensor grid;
- **m**: vector collecting the number of knots used in each of the N directions $m(\mathbf{i}) = [m(i_1), m(i_2), \ldots, m(i_N)]$;
- **coeff**: the coefficients $c_{\mathbf{i}}$ of the sparse grid in the combination technique formulas (5) and (6).

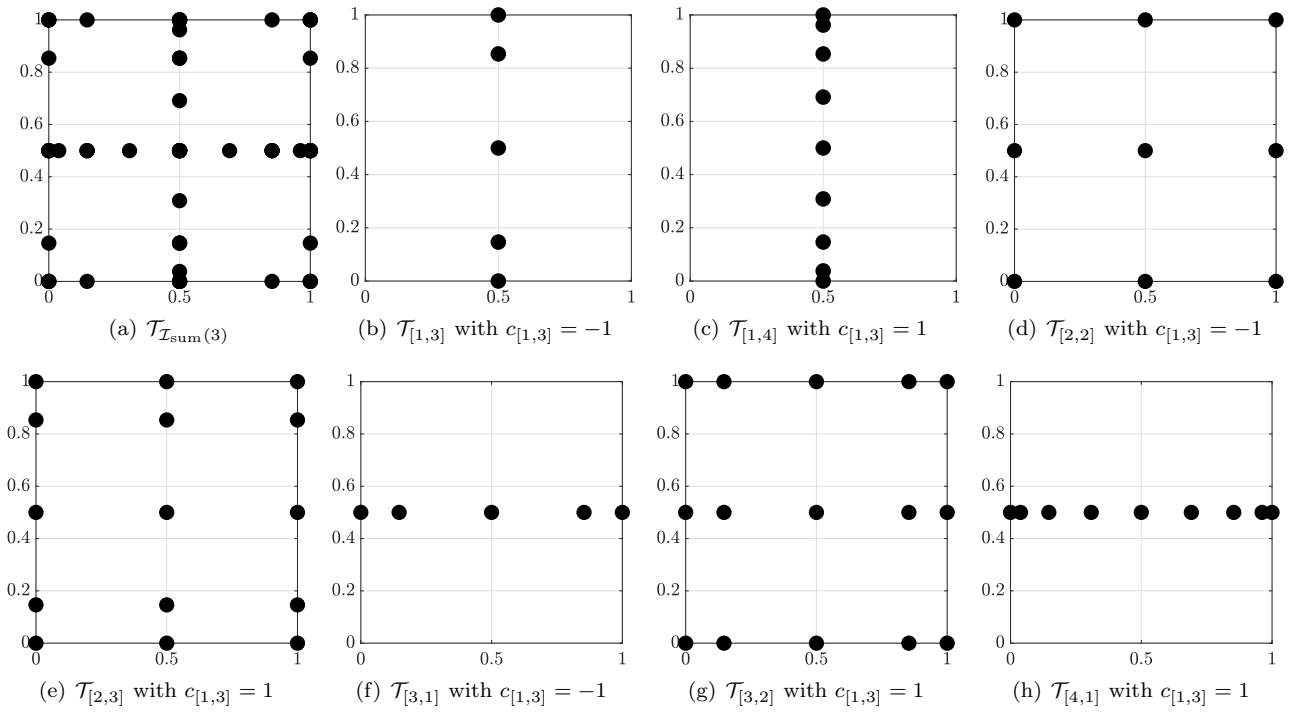
Note in particular that sparse grid knots (and in general knots $\mathbf{y} \in \Gamma$) are always stored in the SGMK as column-vectors (and sets of knots such as **S.knots** are stored as matrices where knots are columns).

The structure array above has seven components, since seven tensor grids are used in the construction of the sparse grid. We report below the first component of the structure array:

```
>> S(1)
ans =

struct with fields:

knots: [2x5 double]
weights: [-0.0333 -0.2667 -0.4000 -0.2667 -0.0333]
size: 5
knots_per_dim: {[0.5000]  [1 0.8536 0.5000 0.1464 0]}
m: [1 5]
coeff: -1
idx: [1 3]
```

In Fig. 5a we plot the sparse grid generated in Listing 3, and in Fig. 5b–h the seven tensor grids that explicitly contribute to it. We can easily observe that some knots appear in multiple tensor grids, and therefore the sparse grids structure **S** contains redundant information. This is a general fact, happening not only when nested knots are used, but also (to a smaller extent) in the case of non-nested knots. Hence, it is convenient to generate a structure containing only the non-repeated knots and a list of corresponding weights, that can be computed taking the linear combination of the quadrature weights of each instance of the repeated knot with the combination coefficient weights in Eq. (5). This can be done calling the function **Sr = reduce_sparse_grid(S)**, that outputs a unique structure containing the following fields:

- **knots**: matrix collecting the list of non-repeated knots, i.e., the set $\mathcal{T}_{\mathcal{I}}$ in Eq. (7);
- **weights**: vector of quadrature weights corresponding to the knots above;
- **size**: size of the sparse grid, i.e. the number of non-repeated knots;
- **m**: index array that maps each knot of **Sr.knots** to their original position in **[S.knots]** (if they have been retained as unique representative of several repeated knots);



(a) $\mathcal{T}_{\mathcal{I}_{\mathrm{sum}}(3)}$ (b) $\mathcal{T}_{[1,3]}$ with $c_{[1,3]} = -1$ (c) $\mathcal{T}_{[1,4]}$ with $c_{[1,3]} = 1$ (d) $\mathcal{T}_{[2,2]}$ with $c_{[1,3]} = -1$

(e) $\mathcal{T}_{[2,3]}$ with $c_{[1,3]} = 1$ (f) $\mathcal{T}_{[3,1]}$ with $c_{[1,3]} = -1$ (g) $\mathcal{T}_{[3,2]}$ with $c_{[1,3]} = 1$ (h) $\mathcal{T}_{[4,1]}$ with $c_{[1,3]} = 1$

Figure 5: The sparse grid generated by Listing 3 and the seven tensor grids that generate it.

- **n**: index array that maps each knot of **[S.knots]** to **Sr.knots**.

For instance, the sparse grid **S** above consists of 67 knots

```
>> size([S.knots])
ans =

2    67
```

and the corresponding reduced sparse grid results in 29 non-repeated knots:

```
>> Sr =

struct with fields:

knots: [2x29 double]
m: [29x1 double]
weights: [1x29 double]
n: [67x1 double]
size: 29
```

Both extended and reduced formats are useful for working with sparse grids and should always be stored in memory, although this implies a certain redundancy in memory storage; see [2, Section 3] for some considerations and numerical tests about the performance of **reduce_sparse_grid**.

Finally, for quick generation of a sparse grid (both extended and reduced formats), the SGMK provides also the convenience function **create_sparse_grid_quick_preset**, which generates Smolyak grids of level w with Clenshaw–Curtis knots in the default interval $[-1, 1]^N$, and only takes as inputs N and w

```
>> [S,Sr] = create_sparse_grid_quick_preset(N,w);
```

3.4.2 Adaptive sparse grids

The SGMK provides also the possibility to construct adaptive sparse grids, i.e. sparse grids where the multi-index set, and hence the approximations in Eqs. (5) and (6), are constructed in an iterative way, relying on some heuristic criteria (called profit indicators in the following) based on the values of the function currently being interpolated/integrated.

name	short description	default value
`'nested'`	Specifies nestendess of knots	none (must always be set)
`'max_pts'`	Maximum number of evaluations of f allowed	1000
`'prof_tol'`	Stops algorithm when all profits are below this threshold	10^{-14}
`'paral'`	Controls interaction with Matlab parallel toolbox	`NaN` (i.e. serial computations)
`'recycling'`	Can speed up computations when non-nested knots are used	`'priority_to_evaluations'`
`'prof'`	Sets the profit to be used (see Tab. 8)	`'Linf/new_points'`
`'op_vect'`	In case of vector-valued f, specifies the vector norm to be used in the computation of the profits to account for the different components of f	Euclidean norm
`'pts_tol'`	Tolerance to detect identical knots when calling `reduce_sparse_grid` internally	1e-14
`'pdf'`	Needed for certain profits (see Tab. 8)	none (must be set if required by profit)
`'var_buffer_size'`	Controls buffer size	N (i.e., no buffer)
`'plot'`	Plots current multi-index set and candidate set at each iteration ($N = 2$ only)	`false`

Table 7: Main control options of the implementation of **adapt_sparse_grids** and their default value.

The function provided to this end is called **adapt_sparse_grids** and a minimal working example of its use is provided in Listing 4.

```
f = @(y) exp(sum(y)); % @-function, takes as input a column vector for y and returns a scalar
                      % or column-vector
N = 2;
knots = @(n) knots_CC(n,0,1);
lev2knots = @lev2knots_doubling;
controls = struct('nested',true); % each field of this struct specifies an optional argument
                                  % to control the algorithm
Ad = adapt_sparse_grids(f,N,knots,lev2knots,[],controls);
```

Listing 4: Basic creation of an adaptive sparse grid.

The implemented adaptive algorithm for sparse grid generation is an extension of the classical Gerstner–Griebel dimension-adaptive algorithm [11] (see [2, Section 3.1] for the mathematical background), and can accommodate for non-nested knots, vector-valued functions, several profit definitions, and the so-called "dimension-buffering". In the following, we discuss the options available in the SGMK, which can be passed to the code in the typical Matlab way by suitably setting the fields of a structure, i.e. **controls = ('option-name',<option-value>,..)** and then passing such structure as input to **adapt_sparse_grids**, cf. Listing 4. The available options are reported in Tab. 7; we discuss the main ones below and refer readers to the help of **adapt_sparse_grids** for a full discussion on the other ones.

The algorithm iteratively explores the set of multi-indices and at each iteration adds one multi-index to the current multi-index set $\mathcal{I}$, choosing from a set of candidate multi-indices (the so-called *reduced margin* of $\mathcal{I}$) the one with the largest profit. The profit indicator of a given multi-index $\mathbf{i}$ quantifies the "net contribution" of $\mathbf{i}$ to the sparse grid and is computed by dividing an "error contribution" estimator $\mathcal{E}_\mathbf{i}$ (i.e., how much the sparse grid approximation / quadrature of f would improve if $\mathbf{i}$ was added to $\mathcal{I}$) by a "work contribution" $\mathcal{W}_\mathbf{i}$ (i.e., how many new evaluations f would be needed if $\mathbf{i}$ was added to $\mathcal{I}$). Of course, several error indicators are possible, and the work contribution depends on whether the knots used are nested or not, which means that several profit definitions are possible. The profits implemented in the SGMK are listed in Tab. 8[2], where:

$$\mathcal{E}_\mathbf{i}^{\mathcal{Q}} = \|\mathcal{Q}_{\mathcal{I}\cup\{\mathbf{i}\}} - \mathcal{Q}_\mathcal{I}\|_V, \tag{12}$$

$$\mathcal{E}_\mathbf{i}^{\mathcal{U},\xi} = \max_{\mathbf{y}\in\mathcal{H}} \left(\|\mathcal{U}_{\mathcal{I}\cup\{\mathbf{i}\}}(\mathbf{y}) - \mathcal{U}_\mathcal{I}(\mathbf{y})\|_V \xi(\mathbf{y})\right), \tag{13}$$

$$\mathcal{W}_\mathbf{i} = \begin{cases} \prod_{n=1}^N \left(m(i_n) - m(i_n - 1)\right) & \text{for nested knots} \\ \prod_{n=1}^N m(i_n) & \text{for non-nested knots} \end{cases} \tag{14}$$

[2]Note that some profit indicators actually do not consider the work contribution and therefore are not profits, strictly speaking.

profit name	Definition
deltaint	$\mathcal{E}^{\mathcal{Q}}$
deltaint/new_points	$\mathcal{E}^{\mathcal{Q}}/\mathcal{W}$
Linf	$\mathcal{E}^{\mathcal{U},1}$
Linf/new_points (default)	$\mathcal{E}^{\mathcal{U},1}/\mathcal{W}$
weighted Linf (requires **'pdf'** option to be set)	$\mathcal{E}^{\mathcal{U},\rho}$
weighted Linf/new_points (requires **'pdf'** option to be set)	$\mathcal{E}^{\mathcal{U},\rho}/\mathcal{W}$

Table 8: Profits and corresponding names, cf. Eqs. (12)–(16).

with $\|\cdot\|_V$ the Euclidean norm in $\mathbb{R}^V$ (see below for other choices) and

$$\xi(\mathbf{y}) = \begin{cases} 1 & \text{for } \mathbf{y} \text{ distributed as uniform random variables} \\ \xi(\mathbf{y}) = \rho(\mathbf{y}) & \text{for the other supported random variables,} \end{cases} \tag{15}$$

$$\mathcal{H} = \begin{cases} \mathcal{T}_{\mathcal{I}\cup\{\mathbf{i}\}} \setminus \mathcal{T}_{\mathcal{I}} & \text{for nested knots, i.e. the new knots added by } \mathbf{i}, \text{ cf. Eq. (7)} \\ \mathcal{T}_{\mathbf{i}} & \text{for non-nested knots, i.e., the tensor grid generated by } \mathbf{i}. \end{cases} \tag{16}$$

As already hinted, the profit to be used can be specified as a field of the structure array **controls** (Line 6 of Listing 4), such as **controls = struct(..,'prof','Linf')**. The weight ξ in Eq. (13) can be provided as a function handle **controls = struct(..,'pdf',@(y) ..)**, and the default norm $\|\cdot\|_V$ can also be changed (e.g. to an ℓ^p norm or a Mahalanobis distance), by specifying a suitable **@**-function in the field **'op_vect'** of **controls**. The formula for the work contribution and the testing set $\mathcal{H}$ are automatically selected based on the field **'nested'** of the structure array, which can take value <**true/false**> and is the only mandatory field to be provided.

The adaptive algorithm terminates when one or more suitable stopping criterion is met. Common choices are checking that the computational work or the profits of the multi-indices in the reduced margin are respectively above or below a certain tolerance. These termination criteria can be controlled by setting further fields in the **controls** structure array: **controls = struct(..,'max_pts',<value>, 'prof_tol',<value>)**.

Finally, the so-called *buffering* of dimensions is a way to speed up the computations when f depends on a large number of random variables, $N \gg 1$, in which case the set of candidates to be examined is very large. In this scenario, it is possible to start the adaptive algorithm considering a reduced set of random variables, and gradually including the remaining ones as the algorithm evolves. In particular, the algorithm is forced to consider at most N_{buf} "non-activated"/"buffered" random variables at each iteration, whereas the non-buffered version would always consider the whole set of random variables. This behavior is obtained by setting to N_{buf} the field **'var_buffer_size'** of the **controls** structure, **controls = struct(..,'var_buffer_size',<value>)**.

The output of the function **adapt_sparse_grids** is a structure, which contains in separate fields both the extended and the reduced version of the grid, as well as additional information such as the values of the function f evaluated at the collocation knots, the current index set, the reduced margin and others.

```
>> Ad
Ad =

struct with fields:

N: 2
S: [1x7 struct]
Sr: [1x1 struct]
f_on_Sr: [1x257 double] % values of f at the sparse grid knots
nb_pts: 257 % nb. of knots in Sr
nested: 1
nb_pts_visited: 257 % number of knots considered during the construction.
                    % For nested knots, this is equal to nb_pts; for non-nested knots,
                    % this will be larger than nb_pts, because some knots enter and then exit
                    % the grid when the corresponding idx exits from the combination technique.
num_evals: 257 % actual nb of evaluations of f required to build the sparse grid.
               % This is not necessarily equal to nb_pts_visited because for speed reasons
               % some function evaluations might be taken more than once (e.g. when
               % if evaluating f is faster than checking whether the point has been
               % already evaluated)
```

```
intf: 2.9525 % expected value of f
private: [1x1 struct] % contains more detailed info on the status of the adaptive algorithm
```

3.4.3 Grid recycling

We close this section pointing out a useful feature of the functions to build sparse grids. For efficiency reasons, all these functions can take as optional input another grid already available in memory and recycle as many computations as possible from there, in terms of e.g. generating the multi-index set, computing the coefficients of the combination technique, and generating the tensor grids needed for the the *extended* version of their storage. The mechanisms implemented to this end are a bit different depending whether one is working with a-priori or adaptive sparse grids:

1. the commands **create_sparse_grid** and **create_sparse_grid_multiidx_set** take as input an *extended* version of a previous grid:

```
I = [1 1;
     1 2;
     2 1;
     3 1];
S = create_sparse_grid_multiidx_set(I,knots,lev2knots);
I_new = [I;
         1 3;
         2 2;
         4 1];
I_new = sortrows(I_new); % multi-indices must be sorted in lexicographic order
S_new = create_sparse_grid_multiidx_set(I_new,knots,lev2knots,S);
```

2. **adapt_sparse_grids** takes as input the result of a previous computation, to resume the computation where the previous one stopped:

```
% stop the adaptive algorithm when profits < 1e-5
controls = struct('nested',true,'prof_tol',1e-5);
Ad = adapt_sparse_grid(f,N,knots,lev2knots,[],controls);
% run the algorithm up to default tolerance
controls_next = struct('nested',true); % default tolerance 1e-14
Next = adapt_sparse_grid(f,N,knots,lev2knots,Ad,controls_next);
```

3. When the new grid differs from the previous one by just one multi-index, the code can be further optimized, and an ad-hoc function in this case is available:

```
I_in = [1 1;
        1 2;
        2 1;
        3 1];
S_in = create_sparse_grid_multiidx_set(I_in,knots,lev2knots);
coeff_in = combination_technique(I_in);
new_idx = [4 1];
S_new = create_sparse_grid_add_multiidx(new_idx,S_in,I_in,coeff_in,knots,lev2knots);
```

4 Operating on sparse grids

After having discussed how to generate a sparse grid, the aim of this section is to present the main functionalities of the SGMK. In Sect. 4.1 we first discuss basic tasks, i.e. evaluation, quadrature and interpolation of the function f, as well as plotting of the sparse grid approximation. The following Sect. 4.2 covers some advanced features that can be optionally activated to save computational time (evaluation recycling, parallelization). We then move to tools to manipulate the sparse grid approximation: Sect. 4.3 covers computation of gradients and Hessians, and Sect. 4.4 computation of Sobol indices by conversion to polynomial chaos expansion. Most of these functions work for vector-valued $f : \Gamma \subset \mathbb{R}^N \to \mathbb{R}^V$, in which case the function is applied component-wise to f. Finally, in Sect. 4.5 we discuss how to interface the SGMK with external software, either by saving grids on file or using the *UM-Bridge* protocol.

4.1 Basic operations: function evaluation, quadrature, interpolation, and plotting

Once a sparse grid is built, the basic operations of interest are: evaluating a function f for each point of the grid, $f(\mathbf{y}_j)$, $\forall \mathbf{y}_j \in \mathcal{T}_{\mathcal{I}}$, approximating the integral (expected value) $\int_\Gamma f(\mathbf{y})\rho(\mathbf{y})\mathrm{d}\mathbf{y}$, evaluating and plotting the sparse grid approximation $\mathcal{U}_{\mathcal{I}}(\mathbf{y})$ for $\mathbf{y} \in \Gamma$; cf. Eqs. (5), (6), and (7). The corresponding functions of the SGMK are:

- **evaluate_on_sparse_grid**, that evaluates the function f at the sparse grid knots;
- **quadrature_on_sparse_grid**, that computes integrals of f by quadrature formulas collocated at the sparse grid knots, cf. Eq. (6);
- **interpolate_on_sparse_grid**, that evaluates the sparse grids approximation as defined in Eq. (5)
- **plot_sparse_grids_interpolant**, that plots such sparse grids approximation.

We illustrate them with the help of the following snippets. First, the values of f at the sparse grid knots are obtained as follows:

```
f = @(y) exp(sum(y));
% S as in Listing 3
Sr = reduce_sparse_grid(S);
f_on_Sr = evaluate_on_sparse_grid(f,Sr);
```

We plot the result in Fig. 6a. Once the values of f at the sparse grid knots are available we can compute the approximated value of the integral of f (see Eq. (6)), as well as construct and evaluate the sparse grid approximation defined in Eq. (5). The snippet below shows how to compute approximated integrals of f and of f^2:

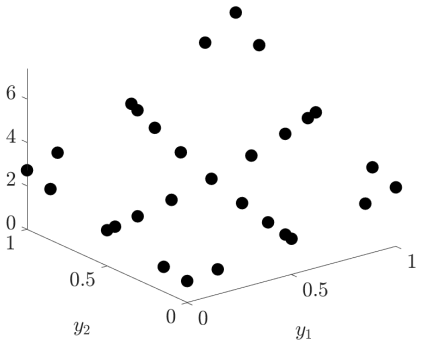
```
q = quadrature_on_sparse_grid(f_on_Sr,Sr);
q2= quadrature_on_sparse_grid(f_on_Sr.^2,Sr);
```

Finally, the following piece of code shows how to evaluate the sparse grid approximation at a uniform cartesian grid of 15×15 points. The resulting surface is reported in Fig. 6b. Note that for this operation both the *extended* and *reduced* versions of the sparse grid are needed (see [2, Section 4.4] for a discussion on why this is the case).

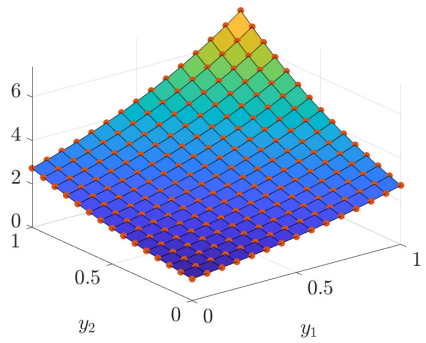
```
% building the cartesian grid of 15x15 equispaced points
y = linspace(0,1,15);
[Y1,Y2] = meshgrid(y,y);
% rearrange the points such that each point is a column
eval_points = [Y1(:)';Y2(:)']; % matrix of dim. 2x15
% S, Sr and f_on_Sr as above
f_vals = interpolate_on_sparse_grid(S,Sr,f_on_Sr,eval_points);
surf(Y1,Y2,reshape(f_vals,15,15))
```

The snippet above is actually very close to the implementation for the case $N = 2$ of the last function mentioned at the beginning of this section, i.e. **plot_sparse_grids_interpolant**, that plots the sparse grids approximation. For $N = 3$, this function would instead plot a number of contour lines colored according to the value of the interpolant, stacked over the same axis; finally, when $N > 3$ a number of bi-dimensional cuts of the domain Γ are considered, and for each of them the surface (i.e., the plot for $N = 2$) will be generated, keeping all other y_n constant to their mid-value. Examples for cases $N = 3$ and $N = 4$ are reported in the listing below, and the corresponding plots are shown in Fig. 7 and 8, respectively. We refer the readers to the help of **plot_sparse_grids_interpolant** for the full description of the optional controls to generate plots like these. Of course, the SGMK contains also functions to plot the knots of a sparse grid: **plot_sparse_grid** (that we have already used multiple times in previous snippets) and **plot3_sparse_grid** (see snippet below). The former can be used to plot two-dimensional sparse grids, or two-dimensional projections of a higher-dimensional sparse grid; the latter can be used instead to plot three-dimensional sparse grids, or three-dimensional projections.

```
N = 3;
f = @(x) exp(sum(x));
% S as in Listing 3 with N = 3, Sr its reduced version
plot3_sparse_grid(S)
f_on_Sr = evaluate_on_sparse_grid(f,Sr);
domain = [0 0 0; 1 1 1]; % domain where to plot
plot_sparse_grids_interpolant(S,Sr,domain,f_on_Sr,'nb_contourfs',5,'nb_countourf_lines',10);

N = 4;
f = @(x) exp(x(1,:)+0.5*x(2,:)+1.5*x(3,:)+2*x(4,:))
```

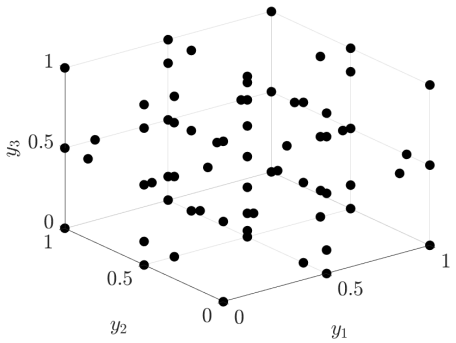


(a) Values of f at the sparse grid knots

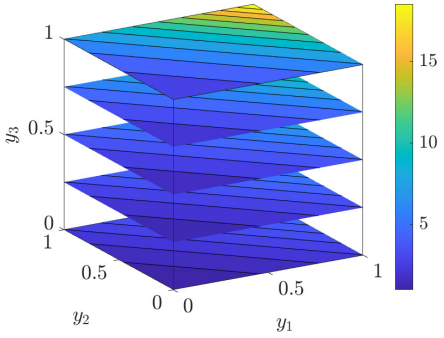


(b) Sparse grid approximation of f evaluated at the points of a uniform cartesian grid(the values are marked in red)

Figure 6: Evaluations of $f(\mathbf{y}) = \exp(y_1 + y_2)$ at the sparse grid knots and corresponding sparse grid approximation.



(a) Sparse grid



(b) Sparse grid approximation

Figure 7: Sparse grid and corresponding approximation of $f(\mathbf{y}) = \exp(y_1 + y_2 + y_3)$.

```
% S as in Listing 3 with N=4, Sr its reduced version
f_on_Sr = evaluate_on_sparse_grid(f,Sr);
domain = [0 0 0 0; 1 1 1 1]; % domain where to plot
plot_sparse_grids_interpolant(S,Sr,domain,f_on_Sr,'two_dim_cuts',[1 2 3 4 1 4])
% cut for (y1,y2),(y3,y4),(y1,y4)
```

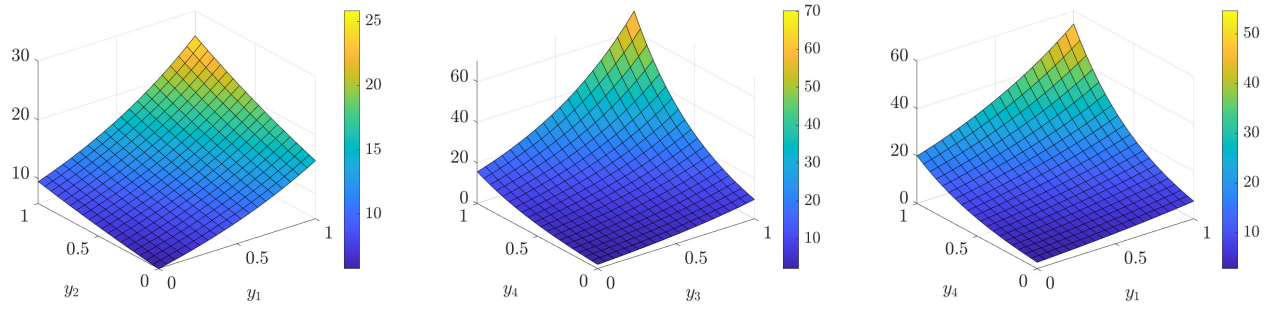
4.2 Enhancing computational performance

The function **evaluate_on_sparse_grids** is in practice a wrapper for looping through the knots of a sparse grid and calling the evaluation of f. It is convenient especially because it implements a couple of advanced features that can be used to improve the computational efficiency: evaluation recycling and parallelization. We discuss them below.

4.2.1 Evaluation recycling

When multiple grids have to be built and the function f evaluated at the corresponding knots (for instance, when the user would like to refine the current sparse grid approximation of f adding more collocation knots to the grid, or when several sparse grids are being compared), significant computational savings can be achieved by recycling the evaluations of f that have been already performed. This can be obtained by passing to **evaluate_on_sparse_grids** the previous grids and the corresponding evaluations as additional inputs, as shown in the next listing where we generate a sequence of grids with increasing w in a for loop.

```
N = 2;
f = @(y) exp(sum(y));
knots = @(n) knots_CC(n,0,1);
lev2knots = @lev2knots_doubling;
```



(a) Cut for (y_1, y_2) (b) Cut for (y_3, y_4) (c) Cut for (y_1, y_4)

Figure 8: Sparse grid approximation of $f(\mathbf{y}) = \exp(y_1 + \frac{1}{2}y_2 + \frac{3}{2}y_3 + 2y_4)$.

```
rule = @(ii) sum(ii-1);
% consider a sequence of grids with increasing level
w_vec = 1:6;
S_old = []; % initialize the grid
Sr_old = []; % initialize the reduced grid
evals_old = []; % initialize the set of evaluations
for w = w_vec
  % recycle tensor grids from S_old
  S = create_sparse_grid(N,w,knots,lev2knots,rule,S_old);
  Sr = reduce_sparse_grid(S);
  % recycle evaluations on S_old
  f_evals = evaluate_on_sparse_grid(f,S,Sr,evals_old,S_old,Sr_old);
  ..
  % do something with f_evals
  ..
  % then update the containers
  S_old = S;
  Sr_old = Sr;
  evals_old = f_evals;
end
```

Note that it is also possible to pass as input a generic list of points (i.e. not a sparse grid):

```
f_evals = evaluate_on_sparse_grid(f,S,Sr,evals_old,[],M)
```

where **M** is a matrix containing the list of points at which the evaluations of f are available, and **evals_old** contains such evaluations. As discussed in [2, Section 4.1], the computational overhead to detect the evaluations that can be recycled is larger in this case than in the previous snippet, where the evaluations are recycled from another sparse grid.

4.2.2 Parallelization of evaluations by Matlab `parpool`

The function **evaluate_on_sparse_grids** also acts as wrapper to calls of the Matlab parallel toolbox construct **parfor**, to speed-up the evaluations of f by running the evaluations on multiple cores. This functionality is activated by first opening a Matlab parallel session (**parpool** command) and then specifying one further input in the call to the function:

```
f_evals =   evaluate_on_sparse_grid(f,S,Sr,evals_old,S_old,Sr_old,paral);
```

where **Sr,evals_old,S_old,Sr_old** can be set to **[]** if no grid / point recycling is required. Here **paral** is an integer number that specifies the minimum number of evaluations of f above which **parfor** should replace the regular **for** when looping over the knots of the sparse grid. This apparently cumbersome call is due to the fact that communication between workers of a **parpool** takes some time. Therefore, unless a sufficient number of evaluations of f are requested, serial evaluation might be faster, depending on the time it takes to perform one evaluation of f. The default value of **paral** is **NaN**, which means that no parallel is used. Note that **evaluate_on_sparse_grid** will not open a **parpool** session, but will check if one is open and throw an error if not.

Remark. *Note that recycling and paralllelization of evalutions are available also for* ***quadrature_on_sparse_grid****. Indeed, the quadrature function can also take as input a* **@***-function (instead of the evaluations of f as explained in Sect. 4.1)*

name	orthonormal pdf	function
Legendre	uniform	**lege_eval_multidim**
Hermite	normal	**herm_eval_multidim**
Laguerre	exponential	**lagu_eval_multidim**
generalized Laguerre	gamma	**generalized_lagu_eval_multidim**
probabilistic Jacobi[3]	beta	**jacobi_prob_eval_multidim**
Chebyshev (1^{st} kind)	Chebyshev	**cheb_eval_multidim**

Table 9: Orthonormal polynomials, corresponding pdfs, and function name.

```
[q,f_on_Sr] = quadrature_on_sparse_grid(f,Sr);
```

In this way, the call to **quadrature_on_sparse_grid** *embeds the one to* **evaluate_on_sparse_grids** *and, as such, all the functionalities of the latter are inherited by the former.*

4.3 Gradients and Hessian

Computing gradients and Hessians of the sparse grids approximation of f can be useful in applications, such as the inverse UQ problem discussed in Sect. 5.2, optimization or local sensitivity analysis studies. To this end, the SGMK provides the functions **derive_sparse_grid** and **hessian_sparse_grid**, that respectively compute gradients and Hessians of the sparse grid approximation of a *scalar-valued* function $f : \mathbb{R}^N \to \mathbb{R}$ by centered finite differences schemes; gradients and Hessians of vector-valued functions must be computed component-wise calling repeatedly the functions **derive_sparse_grid** and **hessian_sparse_grid**. More specifically, **derive_sparse_grid** returns the gradient of f evaluated at *multiple points* as a matrix, each column being the gradient in one point; **hessian_sparse_grid** conversely computes the Hessian matrix at a *single point*. The step-size h of the computation along each variable y_k is defaulted to $(b-a)/10^5$ (other values can be specified if needed), where a, b are the boundaries of Γ along y_k (or of the subset of Γ_k where derivatives are needed, in case of unbounded domains).

4.4 Polynomial chaos expansion and Sobol indices computation

The generalized Polynomial Chaos Expansion (gPCE) is the expansion of a function f over multi-variate ρ-orthonormal polynomials, that in practice gets truncated as follows:

$$f(\mathbf{y}) \approx \sum_{\mathbf{p}\in\Lambda} d_{\mathbf{p}} \mathcal{P}_{\mathbf{p}}(\mathbf{y}),$$

where $\Lambda \subset \mathbb{N}^N$ is a multi-index set, and $\mathcal{P}_{\mathbf{p}} = \prod_{n=1}^{N} P_{p_n}(y_n)$ are products of N univariate ρ_n-orthonormal polynomials of degree p_n. The SGMK provides functions to evaluate a number of different orthonormal polynomials in the folder

```
tools
└─ polynomials functions
```

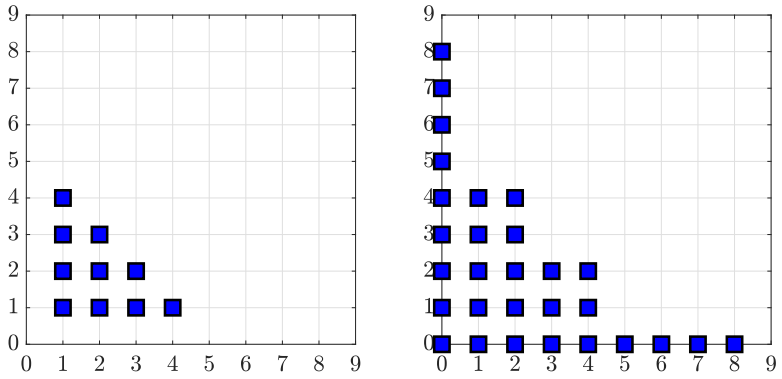
The available polynomials with the corresponding orthonormal pdf and the associated function names are listed in Tab. 9.

Thus, a gPCE is completely determined upon prescribing a multi-index set Λ and the vector containing the corresponding expansion coefficients $d_{\mathbf{p}}$:

- the multi-index set Λ can be constructed with the same procedures detailed in Sect. 3.3: the only difference is that now the entries of each multi-index indicate a polynomial degree, therefore entries with value zero are allowed. This can be obtained e.g. by means of the option **base=0** in the function **multiidx_gen**.

```
N = 2;
rule = @(ii) sum(ii);
w = 3;
base = 0;
Lambda = multiidx_gen(N,rule,w,base);
```

[3]These polynomials are slightly different from the classical Jacobi polynomials: the former are orthonormal with respect to the beta pdf, cf. Tab. 1, whereas the latter are orthonormal with respect to the weight function $w(y) = (y-a)^{\beta}(b-y)^{\alpha}$ – note the switch in the role of the parameters α, β.



(a) Sparse grid multi-index set $\mathcal{I}$ (b) PCE multi-index set Λ

Figure 9: Sparse grid multi-index set and corresponding PCE multi-index set.

- The coefficients $d_{\mathbf{p}}$ can be computed in several ways, e.g. by quadrature, least squares fitting or compressed sensing approaches. The strategy provided by the SGMK is briefly described below.

Once the multi-index set is defined and gPCE coefficients are available, the gPCE can be evaluated in a straightforward way as follows:

```
PCE_coeffs = [..]; % a vector of coefficients obtained e.g. by least square or quadrature
eval_points = rand(N,10); % points where to evaluate the gPCE of f
PCE_vals = zeros(1,size(eval_points,2));
for i = 1:length(PCE_coeffs)
    PCE_vals = PCE_vals+PCE_coeffs(i)*lege_eval_multidim(eval_points,Lambda(i,:),0,1);
    % 0,1 are shape param and indicate that the Legendre polynomials are defined over [0,1]^N
end
```

Coming to the computation of the gPCE coefficients, the SGMK provides a function to compute them given the evaluations of f over a sparse grid. Such function is called **convert_to_modal** and it supports transformation to all polynomials listed in Tab. 9, and, roughly speaking, works by converting the sparse-grid surrogate model into its equivalent gPCE by performing a change of polynomial basis. The function **convert_to_modal** returns a vector containing the gPCE coefficients **PCE_coeffs** and a matrix **Lambda** containing the multi-index set of the gPCE:

```
f = @(y) exp(sum(y));
w = 3;
base = 1;
knots = @(n) knots_CC(n,0,1);
lev2knots = @lev2knots_doubling;
rule = @(ii) sum(ii-1);
I = multiidx_gen(N,rule,w,base);
S = create_sparse_grid_multiidx_set(I,knots,lev2knots);
Sr = reduce_sparse_grid(S);
f_on_Sr = evaluate_on_sparse_grid(f,Sr);
domain = [0 0; 1 1];
[PCE_coeffs,Lambda] = convert_to_modal(S,Sr,f_on_Sr,domain,'legendre');
```

Note that the multi-index set of the PCE is completely determined by the choices of the level-to-knots function and of the multi-index set of the sparse grid from which the conversion procedure begins. In Fig. 9 we plot both multi-index sets from the snippet above.

The conversion to gPCE is useful e.g. to compute the indices of the Sobol decomposition of f: to this end, the SGMK provides a wrapper function **compute_sobol_indices_from_sparse_grid** which calls **convert_to_modal** and performs some algebraic manipulations of the gPCE coefficients of f to compute the Sobol indices of f; we give an example of its usage in Sect. 5.1.

4.5 Interfacing the SGMK with other software

Interfacing the SGMK with other software implementing the evaluation of f can be done in (at least) two ways: by saving sparse grids on file and then using external tools to evaluate f at each point, or through the interface provided by the software UM-Bridge [1], available at `https://um-bridge-benchmarks.readthedocs.io/en/docs/`.

4.5.1 Saving grid on file

Saving a grid (reduced format) on file can be obtained by the command **export_sparse_grid_to_file(Sr,filename)**, which generates a ASCII file (default name **points.dat**) that has the following format

```
<Nb of points of Sr>  N
<coord 1 of point 1> <coord 2 of point 1> <coord 3 of point 1> ..
<coord 1 of point 2> <coord 2 of point 2> <coord 3 of point 2> ..
..
```

Optionally, one additional column with the quadrature weight of each collocation point can be added.

4.5.2 Using the UM-Bridge interface

The software UM-Bridge provides a means to interfacing UQ software with complex software for evaluating $f(\mathbf{y})$ (e.g. when $f(\mathbf{y})$ is the numerical solution of a PDE) by providing a standardized protocol between two layers:

1. the first one (*client* side) converts requests of evaluations of f at collocation points from, e.g., the SGMK, such as those occurring when **evaluate_on_sparse_grids** is called, into HTTP messages;

2. the second one (*server* side) converts back the client requests to actual calls to the software that implements the evaluation of $f(\mathbf{y})$.

The fact that communication is via HTTP messages opens the way to several possibilities on the server side: the software implementing the evaluation of $f(\mathbf{y})$ can be an executable running either on the same machine of the client, or on a remote server. UM-Bridge clients are currently available in Python, C++, Matlab, R, while servers are available only in Python and C++. In particular, the UM-Bridge Matlab client is fully compatible with the SGMK. For instance, the following snippet evaluates over a sparse grid the function $f(\mathbf{y})$ provided by a service running on a remote server:

```
% uri of the service running the server
uri = 'http://xxx.xxx.xx.xxx';
% HTTPModel is an object provided by the UM-Bridge Matlab client.
model = HTTPModel(uri,'forward');
% model.evaluate(y) sends a request to the server to evaluate the model at y.
% Wrap it in an @-function
f = @(y) model.evaluate(y);
N=2;
w=5;
[S,Sr]=create_sparse_grid_quick_preset(N,w);
f_evals = evaluate_on_sparse_grid(f,Sr);
```

Note that once the call to **HTTPModel.evaluate** is encapsulated in an anonymous function, UM-Bridge is fully transparent to SGMK, and it would be e.g. possible to enable parallel evaluations of **f** as discussed in Sect. 4.2.2, without any change to the snippet above, beside activating a **parpool** session and passing a value for **paral** to **evaluate_on_sparse_grid** (of course, the server machine must be able to process parallel requests).

5 Working example

The aim of this section is to give a comprehensive overview on the functionalities of the SGMK for forward and inverse UQ problems. To this end we consider the following PDE:

$$\begin{cases} -\partial_x(a(x,\mathbf{y})\partial_x u(x,\mathbf{y})) = 1, & \text{for } x \in (0,1) \\ u(0,\cdot) = u(1,\cdot) = 0 \end{cases} \tag{17}$$

with piecewise constant diffusion coefficient $a(x,\mathbf{y}) : [0,1] \times \Gamma \to \mathbb{R}$, $\Gamma = [-\sqrt{3},\sqrt{3}]^N$ of the following form

$$a(x,\mathbf{y}) = \mu + \sum_{n=1}^{N} s_n\, y_n \chi_{[\bar{x}_{n-1},\bar{x}_n]}(x), \tag{18}$$

where $\mu = 1$, $\bar{x}_n = \frac{n}{N}$ for $0 \le n \le N$, y_n are independent uniform random variables $y_n \sim \mathcal{U}(-\sqrt{3},\sqrt{3})$ with zero mean and unit variance, and $s_n > 0$ such that $\mu - \sqrt{3}s_n > 0$.

5.1 Forward UQ analysis

We focus first on the forward UQ analysis, i.e. the study of the propagation of the randomness in the diffusion coefficient $a(x, \mathbf{y})$ to a functional of the solution of Eq. (17), often called *quantity of interest*, that we denote in the following by $I(\mathbf{y})$. In this example we consider as quantity of interest the spatial integral of the solution u, i.e.

$$I(\mathbf{y}) = \int_0^1 u(x, \mathbf{y})\, \mathrm{d}x, \tag{19}$$

and aim to estimate its expected value $\mathbb{E}[I] = \int_\Gamma I(\mathbf{y})\rho(\mathbf{y})\, \mathrm{d}\mathbf{y}$, its variance $\mathrm{Var}[I] = \mathbb{E}[I^2] - \mathbb{E}[I]^2$, and its pdf. In the following we choose $N = 2$ (which allows plotting the sparse grid approximation as a surf plot), and set the coefficients s_1 and s_2 in Eq. (18) to 0.5 and 0.1, respectively.

Both the estimation of the moments and of the pdf of I require evaluations of the quantity of interest at the sparse grids knots. It is then convenient to define a wrapper for the call to the PDE solver and the estimation of I. Note that we use here a piece-wise linear finite element solver using 200 elements, but of course any other solver can be used.

```
mu = 1; %  mean of the diffusion coefficient, cf. Eq. (18)
s = [0.5, 0.1]; % s_1, s_2 in Eq. (18)
PDE_rhs = @(x) ones(size(x)); % PDE right-hand side, cf. Eq. (17)
N_el = 200; % number of FEM elements
xx = linspace(0,1,N_el+1); % discretization of the physical domain
I = @(y) QoI(xx,y,mu,s,PDE_rhs);
```

To perform the UQ analysis we choose to work with Smolyak grids of Clenshaw–Curtis knots, since the random variables y_n are uniform. As the level of approximation w determines the accuracy of the sparse grid approximation and quadrature, we test $w = 2, \ldots, 8$ and compare the corresponding estimates of the expected value with respect to a very accurate result obtained with $w = 10$ (see Fig. 10a).

```
N = 2;
knots = @(n) knots_CC(n,-sqrt(3),sqrt(3));
lev2knots = @lev2knots_doubling;
rule = @(ii) sum(ii-1);
% initializations for recycling
S_old = []; Sr_old = []; evals_old = [];
for w = 2:8 % testing different values of w
    S = create_sparse_grid(N,w,knots,lev2knots,rule,S_old);
    Sr = reduce_sparse_grid(S);
    [exp_I,evals] = quadrature_on_sparse_grid(I,S,Sr,evals_old,S_old,Sr_old);
    S_old = S; Sr_old = Sr;  evals_old = evals;
end
```

The plot shows that $w = 4$ gives already very accurate results in terms of expected value, and we expect errors in the other quantities (variance, pdf) to be roughly of the same size, due to the "simple" structure of the problem at hand. Hence, for the following numerical results we consider $w = 4$. The expected value is estimated to

```
>> exp_I
exp_I =

0.0935
```

For the computation of the variance of I we proceed as follows

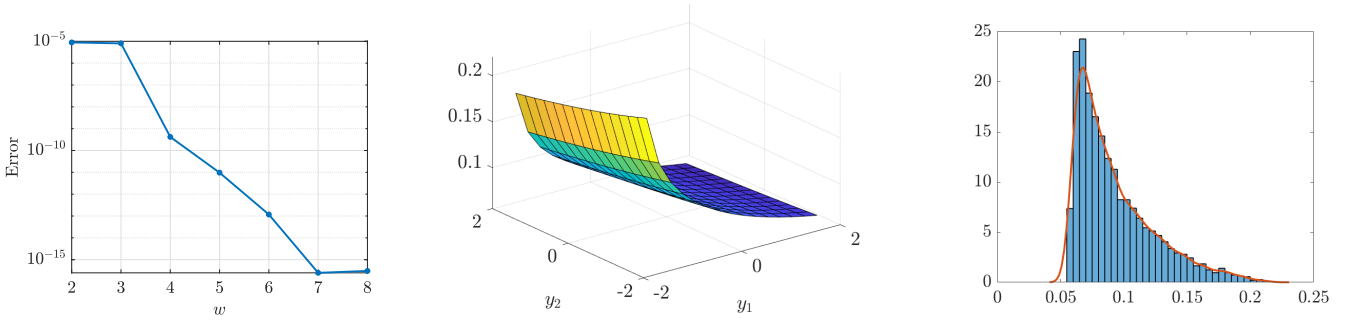
```
I_on_Sr = evaluate_on_sparse_grid(I,Sr);
int_I2 = quadrature_on_sparse_grid(I_on_Sr.^2,Sr); % computing E[I^2]
var_I = int_I2-exp_I^2;
```

and obtain

```
>> var_I
var_I =

0.0010
```

Further, the sparse grid approximation of I (cf. Eq. (5)) is plotted in Fig. 10b.

```
domain = [-sqrt(3),-sqrt(3);sqrt(3),sqrt(3)];
plot_sparse_grids_interpolant(S,Sr,domain,I_on_Sr) % we recall that this command embeds the
% call to interpolate_on_sparse_grid that actually constructs the sparse grid approximation
```



(a) Error in the estimate of $\mathbb{E}[I]$ for increasing values of w

(b) Sparse grid surface for $w = 4$

(c) pdf estimation by histogram and kernel density estimation (curve in red)

Figure 10: Forward UQ. Sparse grid approximation of I.

Then, the estimate of the pdf of I can be done sampling such sparse grid approximation: we evaluate the sparse grid approximation at $M = 5000$ uniformly distributed samples of Γ and estimate the pdf by means of an histogram or a kernel density estimator. The results are reported in Fig. 10c.

```
M = 5000;
y_samples = -sqrt(3)+2*sqrt(3)*rand(N,M); % map the uniform samples on [0,1]^2
                                          % generated by rand to [-sqrt(3),sqrt(3)]^2
I_vals_rand = interpolate_on_sparse_grid(S,Sr,I_on_Sr,y_samples);
H = histogram(I_vals_rand,'Normalization','pdf');
[pdf_vals,pdf_pts] = ksdensity(I_vals_rand);
plot(pdf_pts,pdf_vals,'LineWidth',2)
```

Finally, we check the principal and total Sobol indices of I. The principal Sobol indices are two values between 0 and 1 which quantify the variability of I due to y_1 and y_2 alone, respectively. The total Sobol indices quantify instead the variability of I due to each random variable considering both its individual contributions as well as its contributions in combination with any other variable, and are again numbers between 0 and 1 (note that, on the contrary, their sum can be larger than 1). In other words, the first total Sobol index quantifies the variability of I due to y_1 alone and combined with y_2, and vice versa for the second one.

```
domain = [-sqrt(3) -sqrt(3);sqrt(3) sqrt(3)];
[Sob_idx, Tot_Sob_idx]=compute_sobol_indices_from_sparse_grid(S,Sr,I_on_Sr,domain,'legendre');
```

They result to be

```
>> Sob_idx
Sob_idx =

    0.9709
    0.0244

>> Tot_Sob_idx
Tot_Sob_idx =

    0.9756
    0.0291
```

This clearly shows that the first random variable is more influential on I than the second one. This could also be inferred from the plot of the sparse grid approximation in Fig. 10b, where we can observe that the variability of I with respect to y_1 is larger than with respect to y_2.

5.2 Inverse UQ analysis

In this section we address inverse UQ analysis, i.e. "adjusting" the pdf of the stochastic input $\mathbf{y}$ given a set of noisy measurements (data) of the solution of Eq. (17), or of functionals thereof. Such pdf is usually called *data-informed* or *posterior* pdf, ρ_{post}, and can be derived using the Bayes theorem, which can be informally stated as

$$\text{pdf}(\mathbf{y} \text{ given data}) = \text{pdf}(\text{data given } \mathbf{y}) \times \text{pdf}(\mathbf{y}) \times \frac{1}{\text{pdf}(\text{data})}, \tag{20}$$

where "pdf($\mathbf{y}$ given data)" is the posterior pdf ρ_{post} that we aim at computing, "pdf(data given $\mathbf{y}$)" is the so-called *likelihood function*, which is denoted in the following by $\mathcal{L}(\mathbf{y})$, and "pdf($\mathbf{y}$)" is the pdf of the random variables based only on a-priori information on them, denoted by ρ_{prior}. The "pdf(data)" can be simply considered to be the normalization constant such that the posterior pdf is actually a pdf (i.e., its integral is equal to 1). Thus, we can write in more formal terms that

$$\rho_{post}(\mathbf{y}) \propto \mathcal{L}(\mathbf{y})\rho_{prior}(\mathbf{y}). \tag{21}$$

Then, let us consider as data a set of values $\widetilde{u}_k$, $k = 1, \dots, K$ of the solution of the PDE in Eq. (17) at different measurement points $x_1, x_2, \dots, x_K$, for an unknown value of the inputs $\mathbf{y}^*$. Moreover, we assume that these observations are corrupted by additive random errors $\varepsilon_1, \dots, \varepsilon_K$ (noise), which we assume for simplicity to be independent, identically distributed normal random variables with zero mean and variance σ_ε^2, i.e.

$$\begin{cases} \widetilde{u}_k = u(x_k, \mathbf{y}^*) + \varepsilon_k, \quad k = 1, \dots, K \\ \varepsilon_k \sim \mathcal{N}(0, \sigma_\varepsilon^2). \end{cases} \tag{22}$$

We also introduce the misfits between such data and the values obtained by the PDE model in Eq. (17) for any value of $\mathbf{y}$, i.e.

$$\mathcal{M}_k(\mathbf{y}) := \widetilde{u}_k - u(x_k, \mathbf{y}), \quad k = 1, \dots, K.$$

The next step for the computation of the posterior pdf of $\mathbf{y}$ is to write a computable expression for the likelihood function $\mathcal{L}(\mathbf{y})$ in Eq. (21). Due to the assumptions on the measurement errors ε_k, we can write

$$\mathcal{L}(\mathbf{y}) = \prod_{k=1}^{K} \frac{1}{\sqrt{2\pi\sigma_\varepsilon^2}} \mathrm{e}^{-\frac{(\widetilde{u}_k - u(x_k, \mathbf{y}))^2}{2\sigma_\varepsilon^2}} = \prod_{k=1}^{K} \frac{1}{\sqrt{2\pi\sigma_\varepsilon^2}} \mathrm{e}^{-\frac{\mathcal{M}_k^2(\mathbf{y})}{2\sigma_\varepsilon^2}}.$$

Finally, we make the quite strong (but often reasonable) assumption that the posterior pdf ρ_{post} can be well approximated by a normal distribution. This assumption is reasonable in presence of numerous data with small measurement noise, and when the observation functional (in this case the evaluation of u at x_k) is adequately sensitive to the input variables, such that the actual posterior is unimodal and well peaked. In case any of these requirements is not valid, a more appropriate approach would be to apply e.g. Markov-Chain Monte Carlo methods, which are agnostic with respect to the shape of ρ_{post}, see e.g. [12] and references therein. Under normality assumption of the posterior, we only have to choose its mean and covariance matrix, see again e.g. [12]:

- The mean can be taken as the mode of the posterior pdf (i.e., its maximum), which we denote as $\mathbf{y}_{\mathrm{MAP}}$ (Maximum A-Posteriori); note that $\mathbf{y}_{\mathrm{MAP}}$ can be interpreted as a reasonable approximation of the true value of $\mathbf{y}$ that generated the data, i.e., $\mathbf{y}^* \approx \mathbf{y}_{\mathrm{MAP}}$, cf. Eq. (22).

  From a computational point of view, it is convenient to compute $\mathbf{y}_{\mathrm{MAP}}$ by taking the negative logarithm of Eq. (21) and computing the minimum of such quantity. Moreover, since we assumed a uniform prior pdf for $\mathbf{y}$, the operation just described finally amounts to computing the minimum of the so-called *negative log-likelihood* function (NLL):

$$\mathrm{NLL}(\mathbf{y}) := -\log\left(\mathcal{L}(\mathbf{y})\right) = \sum_{k=1}^{K} \frac{\mathcal{M}_k^2(\mathbf{y})}{2\sigma_\varepsilon^2} + K\log\sigma_\varepsilon^2 + K\log\sqrt{2\pi}, \tag{23}$$

$$\mathbf{y}_{\mathrm{MAP}} := \operatorname*{argmin}_{\mathbf{y}\in\Gamma} \mathrm{NLL}(\mathbf{y}). \tag{24}$$

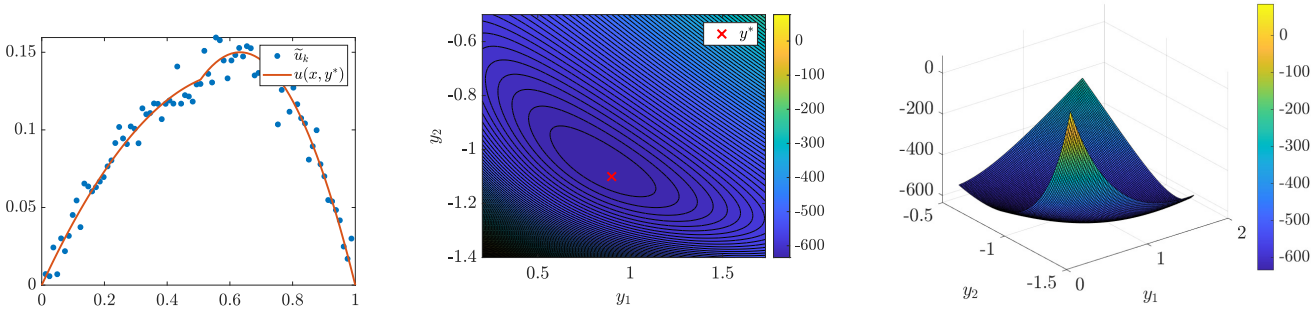
  Note that this minimization is actually independent of σ_ε^2, and it is in practice equivalent to the classical least-squares approach for calibrating $\mathbf{y}$, i.e. to the minimization of the sum of the squared errors:

$$\mathrm{LS}(\mathbf{y}) = \sum_{k=1}^{K} \mathcal{M}_k^2(\mathbf{y}), \tag{25}$$

$$\mathbf{y}_{\mathrm{MAP}} = \operatorname*{argmin}_{\mathbf{y}\in\Gamma} \mathrm{LS}(\mathbf{y}). \tag{26}$$

  An approximation of σ_ε^2 can actually be recovered after having determined $\mathbf{y}_{\mathrm{MAP}}$, by using the standard sample variance estimator:

$$\sigma_\varepsilon^2 \approx \frac{1}{K} \sum_{k=1}^{K} \mathcal{M}_k^2(\mathbf{y}_{\mathrm{MAP}}).$$



(a) Data and "true" values

(b) Contour of the NLL in a neighborhood of y^*

(c) NLL in a neighborhood of y^*

Figure 11: Inverse UQ.

- The covariance matrix Σ_{post} can be taken as the inverse of the Hessian of the NLL at $\mathbf{y}_{\text{MAP}}$. A convenient approximation for Σ_{post} is

$$\Sigma_{\text{post}} \approx \sigma_\varepsilon^2 (J_{\mathbf{y}} u^{\mathrm{T}} J_{\mathbf{y}} u)^{-1},$$

  where $J_{\mathbf{y}} u$ is the Jacobian matrix of $u(x_1, \mathbf{y}), \ldots, u(x_K, \mathbf{y})$ with respect to $\mathbf{y}$, i.e. $(J_{\mathbf{y}} u)_{k,\ell} = \frac{\partial}{\partial y_\ell} u(x_k, \mathbf{y})$.

For the numerical tests reported in this section we fix $s_1 = s_2 = 0.5$ in Eq. (18). We consider a FEM discretization with $K + 2 = 82$ knots, and generate a set of K synthetic data (one for each internal knot of the mesh), i.e., we use Eq. (22) with $x_k = k\frac{1}{K+1}$, $k = 1, \ldots, K$, $\mathbf{y}^* = [0.9, -1.1]$ and $\sigma_\varepsilon = 0.01$.

```
K = 80;
x_k = linspace(0,1,K+2);
s = [0.5, 0.5];
y_star = [0.9,-1.1];
sigma_eps = 0.01;
u_star = PDE_solver(x_k,y_star,mu,s,PDE_rhs);
u_star = u_star(2:end-1); % select the values at the internal knots of the mesh
u_tilde = u_star + sigma_eps*randn(K,1);
```

The data thus generated are plotted in Fig. 11a. The negative log-likelihood function is defined as

```
u = @(y) PDE_solver(x_k,y,mu,s,PDE_rhs);
misfits = @(y) [0;u_tilde;0] - u(y); % misfits are computed also at the boundary to keep
                                     % the code compact, but their contribution is zero
NLL = @(y) sum(misfit(y).^2/(2*sigma_eps^2))+(K-2)*log(sigma_eps^2)+(K-2)*0.5*log(2*pi);
```

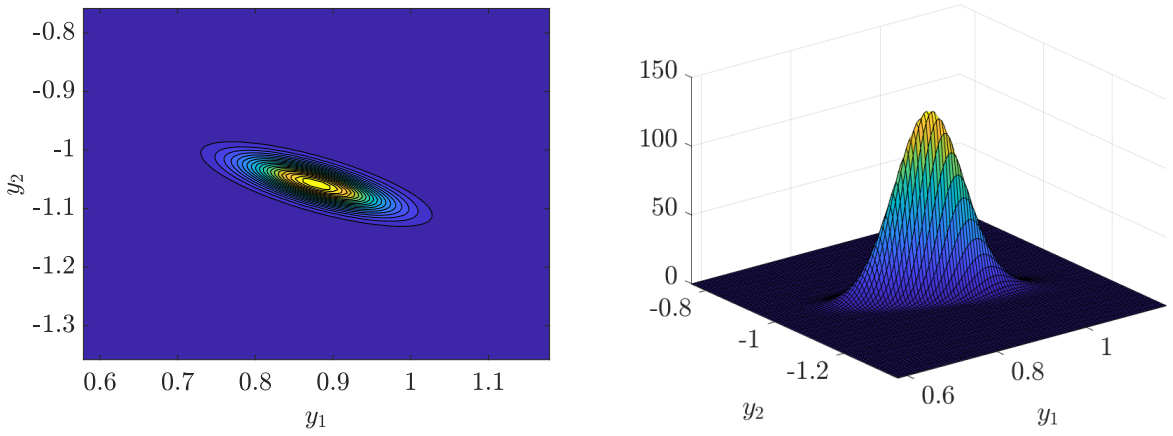
A plot of NLL in a neighborhood of $\mathbf{y}^*$ is given in Fig. 11b,c. Its isolines are approximately ellipses, which indicates that the normal approximation of ρ_{post} is appropriate in this case.

The minimization in Eq. (24) or (26) can be computationally expensive, especially when sampling the values $u(\cdot, \mathbf{y})$ calls a complex model (this is actually not the case for our current example, but we illustrate this step nonetheless for the sake of generality). Hence, the exact quantity of interest u can be replaced in Eq. (23) by its sparse grid approximation, which is in general cheaper to evaluate. We consider here the sparse grid built in Sect. 5.1 (third snippet) corresponding to $w = 5$ and proceed as follows

```
u_on_Sr = evaluate_on_sparse_grid(u,Sr); % u_on_Sr has dimension (K-2) x Sr.size:
                                         % the ith row contains u(x_i,Sr.knots)
u_approx = @(y) interpolate_on_sparse_grid(S,Sr,u_on_Sr,y);
% redefine misfits to use u_approx instead of u
misfits = @(y) [0;u_tilde;0] - u_approx(y);
% note that interpolate_on_sparse_grids supports vector-valued functions,
% so in one call we get the values of u at all nodes x_i
LS = @(y) sum(misfits(y).^2);
```

For the minimization we resort to the Matlab function **fminsearch**

```
% initial guess of the minimization: center of Gamma
y_start = [0; 0];
y_MAP = fminsearch(LS,y_start);
sigma_eps_approx = sqrt(mean(misfits(y_MAP).^2));
```



(a) Contour (b) Surface

Figure 12: Inverse UQ. Gaussian approximation of the posterior distribution of $\mathbf{y}$.

which returns

```
>> y_MAP
y_MAP =

    0.8787
   -1.0581

>> sigma_eps_approx
sigma_eps_approx =

    0.0084
```

They are reasonably good approximations of $\mathbf{y}^*$ and σ_ε^2.Now, for the computation of an approximation of the covariance matrix we need the Jacobian matrix of u with respect to $\mathbf{y}$. Each row of the Jacobian matrix can then be computed by the function **derive_sparse_grid** as shown in the following

```
Jac_at_MAP = zeros(K-2,N);
domain = [-sqrt(3), -sqrt(3); sqrt(3), sqrt(3)];
for i = 1:(K-2)
    Jac_at_MAP(i,:) = derive_sparse_grid(S,Sr,u_on_Sr(i,:),domain,y_MAP)';
end
Sigma_post = sigma_eps_approx^2*inv(Jac_at_MAP'*Jac_at_MAP);
```

which returns

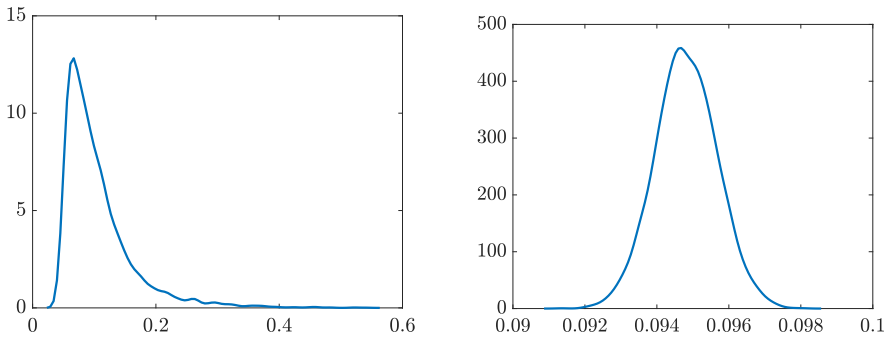
```
>> Sigma_post

Sigma_post =

    0.0036   -0.0013
   -0.0013    0.0009
```

The resulting normal approximation of the posterior distribution of $\mathbf{y}$ is plotted in Fig. 12.

Finally, having the posterior distribution of $\mathbf{y}$ at hand, we can perform the forward UQ analysis of the quantity of interest I defined in Eq. (19) based on such pdf. The procedure is analogous to what done in Sect. 5.1, but the sparse grid generation needs to be further discussed due to the fact that this time y_1 and y_2 are no longer independent. A possible solution is to introduce the transformation

$$\mathbf{y} = \mathbf{y}_{\text{MAP}} + H^{\text{T}}\mathbf{z} \tag{27}$$

with H being the Choleski factor of Σ_{post} (i.e. $H^{\text{T}}H = \Sigma_{\text{post}}$) and $\mathbf{z} \sim \mathcal{N}(0, I)$, which allows to move from independent variables z_1, z_2 to y_1, y_2 dependent random variables. Hence, we first construct a sparse grid according to the distribution of $\mathbf{z}$, for example



(a) Prior-based pdf (b) Posterior-based pdf

Figure 13: Inverse UQ. Prior and data-informed pdf of I. Note the difference in the support of the two pdfs.

```
N = 2;
w = 4;
knots = @(n) knots_normal(n,0,1);
rule = @(ii) sum(ii-1);
lev2knots = @lev2knots_lin;
S = create_sparse_grid(N,w,knots,lev2knots,rule);
Sr = reduce_sparse_grid(S);
```

and then apply the change of variables in Eq. (27) to the sparse grid knots before calling the PDE solver:

```
H = chol(Sigma_post);
I = @(z) QoI(xx,H'*z+y_MAP,mu,s,PDE_rhs); % moving from z to y
I_on_Sr = evaluate_on_sparse_grid(I,Sr);
```

Similarly, for the estimation of the distribution of I we have to draw samples according to the distribution of `z`. This time we do not need to apply the change of variables in Eq. (27), since it is embedded in the definition of `I` in the previous snippet:

```
M = 5000;
z_samples = randn(N,M);
I_vals_randn = interpolate_on_sparse_grid(S,Sr,I_on_Sr,z_samples);
[pdf_vals,pdf_pts] = ksdensity(I_vals_randn);
```

In Fig. 13 we compare the posterior-based and prior-based pdfs of I. We can immediately observe that the variability of I is strongly reduced in the posterior-based case (note the support of the posterior-based pdf of I, which is much smaller than the prior-based one).

References

[1] L. Seelinger, V. Cheng-Seelinger, A. Davis, M. Parno, and A. Reinarz. UM-Bridge: Uncertainty quantification and modeling bridge. *Journal of Open Source Software*, 8(83):4748, 2023.

[2] C. Piazzola and L. Tamellini. The Sparse Grids Matlab kit - a Matlab implementation of sparse grids for high-dimensional function approximation and uncertainty quantification. *ArXiv*, (2203.09314), 2022.

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

[4] S. De Marchi. On Leja sequences: some results and applications. *Appl. Math. Comput.*, 152:621–647, 2004.

[5] F. Nobile, L. Tamellini, and R. Tempone. Comparison of Clenshaw–Curtis and Leja Quasi-Optimal Sparse Grids for the Approximation of Random PDEs. In R. M. Kirby, M. Berzins, and J. S. Hesthaven, editors, *Spectral and High Order Methods for Partial Differential Equations - ICOSAHOM '14*, volume 106 of *Lecture Notes in Computational Science and Engineering*, pages 475–482. Springer International Publishing, 2015.

[6] A. Chkifa. On the Lebesgue constant of Leja sequences for the complex unit disk and of their real projection. *Journal of Approximation Theory*, 166(0):176 – 200, 2013.

[7] A. Narayan and J. D. Jakeman. Adaptive Leja Sparse Grid Constructions for Stochastic Collocation and High-Dimensional Approximation. *SIAM Journal on Scientific Computing*, 36(6):A2952–A2983, 2014.

[8] L. N. Trefethen. Is Gauss quadrature better than Clenshaw-Curtis? *SIAM Rev.*, 50(1):67–87, 2008.

[9] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer-Verlag, Berlin, second edition, 2007.

[10] A. Genz and B. D. Keister. Fully symmetric interpolatory rules for multiple integrals over infinite regions with Gaussian weight. *J. Comput. Appl. Math.*, 71(2):299–309, 1996.

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

[12] C. Piazzola, L. Tamellini, and R. Tempone. A note on tools for prediction under uncertainty and identifiability of SIR-like dynamical systems for epidemiology. *Mathematical Biosciences*, 332:108514, 2021.