ORNL/TM-2015/596

User Manual: Toolkit for Adaptive Stochastic Modeling and Non-Intrusive Approximation (TASMANIAN)

Miroslav Stoyanov

April 2018, version 5.1

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

ORNL/TM-2015/596

Computer Science and Mathematics Division

USER MANUAL: TOOLKIT FOR ADAPTIVE STOCHASTIC MODELING AND NON-INTRUSIVE APPROXIMATION (TASMANIAN)

Miroslav Stoyanov

Date Published: September 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This documents serves to explain the functionality of the Toolkit for Adaptive Stochastic Modeling And Non-Intrusive Approximation (TASMANIAN). The document covers the mathematical background, installation, and the three software components: C/C++ libraries, Python and MATLAB interfaces. Currently TASMANIAN includes two modules: sparse grids surrogate modeling and multidimensional integration, and Bayesian inference using Markov Chain Monte Carlo sampling.

1 Quick Overview

1.1 Sparse grids

Sparse Grids refers to a family of algorithms for approximation of multidimensional functions and integrals, where the approximation operator is constructed as a linear combination of tensors of multiple one dimensional operators [1, 3, 4, 11, 14–18, 20–23, 25–27, 31, 32]. The TASMANIAN sparse grids library implements a wide variety of sparse grids methods with different one dimensional operators and different ways of constructing the linear combination of tensors.

Let $\Gamma^{a_k,b_k} = [a_k, b_k] \subset \mathbb{R}$, for $k = 1, 2, \ldots, d$, indicate a set of one dimensional intervals and let $\Gamma^{\boldsymbol{a},\boldsymbol{b}} = \bigotimes_{k=1}^{d} \Gamma^{a_k,b_k} \subset \mathbb{R}^d$ be a d-dimensional sparse grids domain. A sparse grid consists of a set of points $\{\boldsymbol{x}_i\}_{i=1}^{N} \in \Gamma^{\boldsymbol{a},\boldsymbol{b}}$ and associated numerical quadrature weights $\{\boldsymbol{x}_i\}_{i=1}^{N} \in \mathbb{R}$ or interpolation basis functions $\{\phi_i(\boldsymbol{x})\}_{i=1}^{N} \in \mathbb{C}^0(\Gamma^{\boldsymbol{a},\boldsymbol{b}})$. Usually, a_k and b_k are finite, however, Gauss-Hermite and Gauss-Laguerre rules allow for the use of unbounded domain. Note that Tasmanian constructs grids using the canonical interval $[-1, 1]$ and the result is then translated (via a linear transformation) to the specific $[a_k, b_k]$; also Gauss-Hermite and Gauss-Laguerre rules use canonical intervals $(-\infty, +\infty)$ and $[0, \infty)$ respectively.

Let $f(\boldsymbol{x}) : \Gamma \to \mathbb{R}$ indicate a d-dimensional function, where w.l.o.g. we assume Γ is the canonical domain. We consider two types of approximations, point-wise approximations $\tilde{f}(\boldsymbol{x})$ where $\tilde{f}(\boldsymbol{x}) \approx f(\boldsymbol{x})$ for all $\boldsymbol{x} \in \Gamma$ and numerical integration $Q(f)$ where $Q(f) \approx \int_\Gamma f(\boldsymbol{x})\rho(\boldsymbol{x})d\boldsymbol{x}$. The weight $\rho(\boldsymbol{x})$ is specific to the one dimensional rule that induces the grid; most rules assume uniform weight $\rho(\boldsymbol{x}) = 1$, however, Gauss-Chebyshev, Gegenbauer, Jacobi, Hermite, and Laguerre, rules use different weights (see Table 2). Note: Tasmanian can handle functions with multiple outputs (e.g., vector valued functions), then $\tilde{f}(\boldsymbol{x})$ and $Q(f)$ have a corresponding number of outputs.

Point-wise approximations can be implemented in two different ways, since both ways result in identical $\tilde{f}(\boldsymbol{x})$ there is no official language to distinguish between the two method, hence we'll use the terms *internal* and *adjoint*. The internal form is

$$\tilde{f}(\boldsymbol{x}) = \sum_{i=1}^{N} c_i \phi_i(\boldsymbol{x}), \tag{1}$$

where $\phi_i(\boldsymbol{x})$ are basis functions determined by the one dimensional rule and the chosen set of tensors, and the weights c_i are computed from the values of $f(\boldsymbol{x}_i)$. The term *internal* refers to the fact that the software library needs direct access to the values $f(\boldsymbol{x}_i)$ in order to compute the coefficients c_i. In contrast, the *adjoint* form is given by

$$\tilde{f}(\boldsymbol{x}) = \sum_{i=1}^{N} \psi_i(\boldsymbol{x}) f(\boldsymbol{x}_i), \tag{2}$$

where $\psi_i(\boldsymbol{x})$ depend on the 1-D rule and tensors and can be computed independent from $f(\boldsymbol{x}_i)$. Using the *adjoint* approach, Tasmanian can approximate functions with arbitrary output and arbitrary data-structures, i.e., the library can generate the $\psi_i(\boldsymbol{x})$ weights and the sum can be computed by user written or third party code. Note that (1) and (2) result in point-wise identical approximation, however, in general, the adjoint approach is usually significantly more expensive (computationally). When $\phi_i(\boldsymbol{x})$ are Lagrange polynomials, then $c_i = f(\boldsymbol{x}_i)$ and $\psi_i(\boldsymbol{x}) = \phi_i(\boldsymbol{x})$ and both approximation methods are computationally equivalent.

In general, sparse grids approximations are not interpolatory, however, when the underlying one dimensional rule is *nested* (i.e., the nodes at level l are a subset of the nodes at level $l+1$), then $\tilde{f}(\boldsymbol{x}_i) = f(\boldsymbol{x}_i)$ at all grid points $\{\boldsymbol{x}_i\}_{i=1}^N$. The Gauss rules implemented in Tasmanian (except Gauss-Patterson) and the Chebyshev rule are non-nested, all other rules are nested. In general, nested grids have fewer points which leads to fewer evaluations of $f(\boldsymbol{x}_i)$ and nesting allows the employment of various refinement strategies. Tasmanian implements two types of refinement based on hierarchical surpluses [26] and anisotropic quasi-optimal polynomial spaces [27].

Employing numerical quadrature, the integral of $f(\boldsymbol{x})$ is approximated as

$$\int_\Gamma f(\boldsymbol{x})\rho(\boldsymbol{x})d\boldsymbol{x} \approx Q[f] = \sum_{i=1}^{N} w_i f(\boldsymbol{x}_i), \tag{3}$$

where the points $\{\boldsymbol{x}_i\}_{i=1}^N$ and the weights $\{w_i\}_{i=1}^N$ depend on the one dimensional rule and the selection of tensors. In general, $Q(f)$ can be constructed from $\tilde{f}(\boldsymbol{x})$ by integrating the approximation (i.e., $w_i = \int_\Gamma \psi_i(\boldsymbol{x})d\boldsymbol{x}$), however, Gauss rules allow for better accuracy by selecting the points $\boldsymbol{x}_i$ at the roots of polynomials that are orthogonal with respect to $\rho(\boldsymbol{x})$ (see table 2). Gauss-Patterson and Gauss-Legendre rules use the same uniform $\rho(\boldsymbol{x})$, however, Gauss-Patterson points have the additional constraint of being nested. In one dimension, Gauss-Legendre rule is more accurate than Gauss-Patterson, however, in a multidimensional setting the nested property of Gauss-Patterson leads to better accuracy per number of points. Unlike Gauss-Legendre, the Gauss-Patterson points and weights are very difficult to compute and this library provides only the first 9 levels as hard-coded constants.

Tasmanian implements a variety of different grids and those are grouped into 4 categories:

- **Global Grids**: $\tilde{f}(\boldsymbol{x})$ is constructed using Lagrange polynomials and the grids are suitable for approximating smooth and analytic functions. All Gauss integration rules fall in this category. See §2.1.
- **Sequence Grids**: for a class of rules (namely Leja, $\mathcal{R}$-Leja, $\mathcal{R}$-Leja-Shifted, min/max-Lebesgue and min-Delta, see Table 3) the sequence grids offer an alternative implementation based on Newton polynomials. Sequence grids can evaluate $\tilde{f}(\boldsymbol{x})$ (for a given $\boldsymbol{x}$) much faster, however, speed comes with higher storage overhead as well as higher computational cost for most other operations, especially loading the values and using ajoint interpolation. Note that the difference between global and sequence grids is only in implementation, otherwise a sequence and a global grid with the same rule and points would result in identical $\tilde{f}(\boldsymbol{x})$. See §2.1.
- **Local Polynomial Grids**: suitable for non-smooth functions with locally sharp behavior. Interpolation is based on hierarchical piece-wise polynomials with local support and varying order. See §2.6.
- **Wavelet Grids**: are similar to the local polynomials, however, using wavelet basis. Coupled with local refinement, often times wavelet grids provide the same accuracy with fewer abscissas. See §2.6.

The code consists of three main components:

- *libtasmaniansparsegrids.a/so*: the main component of Tasmanian is the C++ sparse grids library that implements the *TasmanianSparseGrid* class that encapsulates all of the available capabilities. See §5.
- *tasgrid*: an executable that provides a command line interface to the library. The executable reads and writes data to text files and every command generally reads an instance of *TasmanianSparseGrid*

class from a text file, calls a function from the class, and writes the modified class back to a text file, see §7.

- *MATLAB Interface*: which is a series of MATLAB functions that call the executable `tasgrid` and read the result into `MATLAB` matrices. Note: the `MATLAB` interface does not use `.mex` files, thus the library can be compiled with a wider range of compilers than those supported by `MATLAB`, however, the usage of the interface is somewhat different than regular mex files, see §8.
- *Python Interface*: which is a single Python module that implements a Python sparse grids class that mimics closely the behavior of the C++ library. The interface is based on `ctypes`, where a C++ instance of the Tasmanian class is held by a void pointer, accessed via a C interface, and encapsulated by the Python module, see §9.

1.2 DREAM

Starting with version 5.0, Tasmanian includes a module for random sampling based on the DiffeRential Evolution Adaptive Metropolis (DREAM) algorithm. Suppose $\rho(\boldsymbol{x})$ be a non-negative function defined over a domain $\Gamma \subset \mathbb{R}^d$ where we make no assumptions regarding compactness or structure of Γ. We assume that

$$\int_{\Gamma} \rho(\boldsymbol{x}) d\boldsymbol{x} < \infty,$$

in which case normalizing $\rho(\boldsymbol{x})$ will give us a probability density function (PDF). It is not necessary to explicitly compute the normalizing constant and the random sampling algorithm works with unscaled $\rho(\boldsymbol{x})$. The goal of the random sampling procedure is to generate a number of samples $\{\boldsymbol{x}_i\}_{i=1}^N$ that are distributed according to the $\rho(\boldsymbol{x})$ PDF. This is done by iteratively evolving a series of chains and the update algorithm depends on the current distribution of the chains and a random correction factor [6, 13, 29, 30, 33].

Bayesian inference is a common application area for this type of sampling algorithms [6]. In the inference paradigm, we have a model $f : \Gamma \to \mathbb{R}^\mu$ and (potentially noisy) observation data $d \in \mathbb{R}^\mu$, the objective is to assign "belief" to the values of $\boldsymbol{x}$ that correspond to the data. The "belief" is defined by a posterior probability distribution $\rho(\boldsymbol{x})$ defined by Bayes' rule

$$\rho(\boldsymbol{x}) = L(d, f(\boldsymbol{x}))\rho_p(\boldsymbol{x}), \tag{4}$$

where $L(d, f(\boldsymbol{x}))$ is the likelihood function indicating the probability that the discrepancy between d and $f(\boldsymbol{x})$ is due entirely to noise, and $\rho_p(\boldsymbol{x})$ is a probability distribution indicating our prior "belief" regarding the values of $\boldsymbol{x}$. The scaling factor in (4) is omitted. The statistics of $\rho(\boldsymbol{x})$ can be computed from a sufficiently large number of random samples collected by the DREAM algorithm

Accurate statistical analysis requires a huge number of random samples, which is prohibitive when the $f(\boldsymbol{x})$ is expensive to compute. A common practice is to replace the expensive $f(\boldsymbol{x})$ by a cheap to evaluate sparse grid surrogate. The Tasmanian DREAM module can collect samples from an arbitrary user defined $\rho(\boldsymbol{x})$ (not necessarily associated with an inference problem), a sparse grid approximation of the likelihood function or model. The user can provide a custom defined model as well. Currently, Tasmanian includes implementation of Gaussian likelihood, i.e.,

$$L(d, f(\boldsymbol{x})) = \exp\left(-(d - f(\boldsymbol{x}))^T \sigma^{-1} (d - f(\boldsymbol{x}))\right), \tag{5}$$

where $\sigma \in \mathbb{R}^{\mu \times \mu}$ is user defined covariance. Tasmanian also includes priors based on Uniform, Gaussian, truncated Gaussian, Beta, Gamma, and Exponential pdfs. The C++ library makes extensive use of polymorphism and can be easily extended with additional custom prior distributions, likelihood functions, and custom models.

Note: until Tasmanian version 5.0, the main focus of development has been on the sparse grids module. Thus, the DREAM module is less mature than the sparse grid one. There are significant gaps in capability and there is high potential for bugs leading to errors and poor stability. Most notably, the random sampling module lacks Python, `MATLAB`, and command line interfaces, which are particularly challenging due to the extensive use of polymorphism in the C++ library.

1.3 Document organization

In the rest of this document, we first present basic Mathematical background of Sparse Grids and random sampling, followed by detailed description of the libraries, classes and interfaces. In §2.1 and §2.6 we provide a brief description of the construction of sparse grids from global and local rules. In §3.1 we describe in details the random sampling algorithm. In §4 we give a guide to compiling the C++ library and in §5 we describe the *TasmanianSparseGrid* class. In §7 we list the functions of the command line wrapper, and in §8 and §9 we describe the usage of the MATLAB and Python interfaces respectively. Appendix 10 shows the format of a file with a user specified integration or interpolation rule.

2 Sparse Grids

2.1 Global Grids: General construction

Let $\{x_j\}_{j=1}^{\infty} \in \mathbb{R}$ denote a sequence of distinct points (in either a canonical or transformed interval $\Gamma^{a,b}$), and let $m : \mathbb{N} \to \mathbb{N}$ be a strictly increasing *growth function*. We define a one dimensional nested family of interpolants $\{\mathcal{U}^{m(l)}\}_{l=0}^{\infty}$, where $\mathcal{U}^{m(l)}$ is associated with the first $m(l)$ points $\{x_j\}_{j=1}^{m(l)}$ and Lagrange basis functions $\{\phi_j^l(x)\}_{j=1}^{m(l)}$ defined by $\phi_j^l(x) = \prod_{i=1, i\neq j}^{m(l)} \frac{x-x_i}{x_j-x_i}$, i.e.,

$$\tilde{f}^{(l)}(x) = \mathcal{U}^{m(l)}[f](x) = \sum_{j=1}^{m(l)} f(x_j)\phi_j^l(x). \tag{6}$$

The corresponding numerical quadrature is given by

$$\int f(x)\rho(x)dx \approx \mathcal{Q}[f] = \sum_{j=1}^{m(l)} w_j^l f(x_j), \tag{7}$$

where $w_j^l = \int \phi_j^l(x)\rho(x)dx$. In a non-nested case, different nodes are associated with each level, i.e., $\{\{x_j^l\}_{j=1}^{m(l)}\}_{l=0}^{\infty}$ and the basis function and operators are defined accordingly. Examples of nested and non-nested one dimensional rules are listed in Tables 1, 2, and 3.

The point-wise approximation and quadrature construction can be expressed in the same operator notation, hence, we define the surplus operators as

$$\Delta^{m(l)} = \mathcal{U}^{m(l)} - \mathcal{U}^{m(l-1)}, \qquad \text{or} \qquad \Delta^{m(l)} = \mathcal{Q}^{m(l)} - \mathcal{Q}^{m(l-1)} \tag{8}$$

depending on whether we are interested in constructing $\tilde{f}(\boldsymbol{x})$ or $\mathcal{Q}[f]$. We also use the convention that $\Delta^{m(0)} = \mathcal{U}^{m(0)}$ or $\Delta^{m(0)} = \mathcal{Q}^{m(0)}$.

The d-dimensional tenor operators are given by

$$\Delta^{\boldsymbol{m}(\boldsymbol{i})} = \bigotimes_{k=1}^{d} \Delta^{m(i_k)}, \qquad \mathcal{U}^{\boldsymbol{m}(\boldsymbol{i})} = \bigotimes_{k=1}^{d} \mathcal{U}^{m(i_k)}, \qquad \mathcal{Q}^{\boldsymbol{m}(\boldsymbol{i})} = \bigotimes_{k=1}^{d} \mathcal{Q}^{m(i_k)}$$

where we assume standard multi-index notation*. A sparse grid operator is defined as

$$G_{\Theta}[f] = \sum_{\boldsymbol{i}\in\Theta} \Delta^{\boldsymbol{m}(\boldsymbol{i})}, \tag{9}$$

where Θ is a lower set†. An explicit form of the points associated with the sparse grid can be obtained by first defining the tensors

$$\boldsymbol{m}(\boldsymbol{i}) = \bigotimes_{k=1}^{d} m(i_k), \qquad \boldsymbol{x_j} = \bigotimes_{k=1}^{d} x_{j_k},$$

*For the remainder of this document we let $\mathbb{N}$ be the set of natural numbers including zero, and $\Lambda, \Theta \subset \mathbb{N}^d$ will denote set of multi-indexes. For any two vectors, we define $\boldsymbol{x}^{\boldsymbol{\nu}} = \prod_{k=1}^{d} x_k^{\nu_k}$ with the usual convention $0^0 = 1$.

†A set Λ is caller lower or admissible if $\boldsymbol{\nu} \in \Lambda$ implies $\{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{i} \leq \boldsymbol{\nu}\} \subset \Lambda$, where $\boldsymbol{i} \leq \boldsymbol{\nu}$ if and only if $i_k \leq \nu_k$ for all $1 \leq k \leq d$.

then the points associated with (9) are given by

$$\{x_{\boldsymbol{j}}\}_{\boldsymbol{j}\in X(\Theta)}, \qquad \text{where} \quad X(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \{\mathbf{1} \le \boldsymbol{j} \le \boldsymbol{m}(\boldsymbol{i})\}. \tag{10}$$

In the non-nested case, $X(\Theta)$ consists of pairs of multi-indexes $X(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \bigcup_{\mathbf{1}\le\boldsymbol{j}\le\boldsymbol{m}(\boldsymbol{i})} \{(\boldsymbol{i},\boldsymbol{j})\}$, and the points are $\{\boldsymbol{x}_{\boldsymbol{j}}^{\boldsymbol{i}}\}_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)}$ where $\boldsymbol{x}_{\boldsymbol{j}}^{\boldsymbol{i}} = \bigotimes_{k=1}^{d} x_{j_k}^{i_k}$.

For every lower set Θ, there is a set of (integer) weights $\{t_{\boldsymbol{j}}\}_{\boldsymbol{j}\in\Theta(L)}$ that satisfy $\sum_{\boldsymbol{j}\le\boldsymbol{i},\boldsymbol{j}\in\Theta(L)} t_{\boldsymbol{j}} = 1$ for every $\boldsymbol{i} \in \Theta(L)$, i.e., $t_{\boldsymbol{i}}$ solve a linear system of equations. Then,

$$G_\Theta[f] = \sum_{\boldsymbol{i}\in\Theta} \Delta^{\boldsymbol{m}(\boldsymbol{i})} = \sum_{\boldsymbol{i}\in\Theta} t_{\boldsymbol{i}} \mathcal{U}^{\boldsymbol{m}(\boldsymbol{i})}, \tag{11}$$

or in the context of integration $G_\Theta[f] = \sum_{\boldsymbol{i}\in\Theta} t_{\boldsymbol{i}} \mathcal{Q}^{\boldsymbol{m}(\boldsymbol{i})}$. Thus, we explicitly write the Lagrange basis functions and quadrature weights as

$$\phi_{\boldsymbol{j}}(\boldsymbol{x}) = \sum_{\boldsymbol{i}\in\Theta, \boldsymbol{m}(\boldsymbol{i})\ge\boldsymbol{j}} t_{\boldsymbol{i}} \prod_{k=1}^{d} \phi_{j_k}^{i_k}, \tag{12}$$

where each $\phi_{j_k}^{i_k}$ is evaluated at the corresponding k-th component of $\boldsymbol{x}$ and we note that in the nested case $\phi_{\boldsymbol{j}}(\boldsymbol{x}) = \psi_{\boldsymbol{j}}(\boldsymbol{x})$ where $\psi_{\boldsymbol{j}}(\boldsymbol{x})$ are defined in (2). Similarly, the quadrature weights are given by

$$w_{\boldsymbol{j}} = \sum_{\boldsymbol{i}\in\Theta, \boldsymbol{m}(\boldsymbol{i})\ge\boldsymbol{j}} t_{\boldsymbol{i}} \prod_{k=1}^{d} w_{j_k}^{i_k}. \tag{13}$$

Therefore, the explicit form of the sparse grids approximation is given by

$$\tilde{f}_\Theta(\boldsymbol{x}) = \sum_{\boldsymbol{j}\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}) \phi_{\boldsymbol{j}}(\boldsymbol{x}), \qquad Q_\Theta[f] = \sum_{\boldsymbol{j}\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}) w_{\boldsymbol{j}}. \tag{14}$$

For the non-nested case, we have

$$\tilde{f}_\Theta(\boldsymbol{x}) = \sum_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}^{\boldsymbol{i}}) t_{\boldsymbol{i}} \prod_{k=1}^{d} \phi_{j_k}^{i_k}, \qquad Q_\Theta[f] = \sum_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}^{\boldsymbol{i}}) t_{\boldsymbol{i}} \prod_{k=1}^{d} w_{j_k}^{i_k}. \tag{15}$$

Note, that some non-nested rules may share points, e.g., all one dimensional Gauss-Legendre rules with odd number of points include 0, thus, it is possible to have the same point for different index pairs $(\boldsymbol{i},\boldsymbol{j})$. TASMANIAN automatically groups the functions and weights associated with those points and the library uses only unique points.

2.2 Global Grids: Approximation error

First we consider the polynomial space* for which the approximation is exact (i.e., no error). For interpolation $\mathcal{U}^{m(l)}[p] = p$ for all $p \in \mathcal{P}^{m(l)-1}$ and for quadrature rules there is a non-decreasing function $q : \mathbb{N} \to \mathbb{N}$

*$\mathcal{P}^l = span\{x^\nu : \nu \le l\}$ and for a lower multi-index set define $\mathcal{P}_\Lambda = span\{\boldsymbol{x}^{\boldsymbol{\nu}} : \boldsymbol{\nu} \le \boldsymbol{i}\}_{\boldsymbol{i}\in\Lambda}$.

so that $\mathcal{Q}^{m(l)}[p] = p$ for all $p \in \mathcal{P}^{q(l)}$. For Gauss rules $q(l) = 2m(l) - 1$, except Gauss-Patterson where $q(l) = \frac{3}{2}m(l) - \frac{1}{2}$. For other rules generally $q(l) = m(l) - 1$ except for rules with symmetric and odd number of points (e.g., Clenshaw-Curtis), where $q(l) = m(l)$ since any symmetric rule integrates exactly all odd power monomials.

For a general sparse grid point-wise approximation

$$G_\Theta[p] = p, \qquad \text{for all} \quad p \in \mathcal{P}_{\Lambda^m(\Theta)}, \qquad \text{where} \quad \Lambda^m(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \{\boldsymbol{j} : \boldsymbol{j} \le \boldsymbol{m}(\boldsymbol{i}) - \boldsymbol{1}\}. \tag{16}$$

And for numerical quadrature

$$G_\Theta[p] = p, \qquad \text{for all} \quad p \in \mathcal{P}_{\Lambda^q(\Theta)}, \qquad \text{where} \quad \Lambda^q(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \{\boldsymbol{j} : \boldsymbol{j} \le \boldsymbol{q}(\boldsymbol{i})\}^*. \tag{17}$$

Thus, $\Lambda^m(\Theta)$ and $\Lambda^q(\Theta)$ define the polynomial spaces associated with G_Θ.

Let $C^0(\Gamma)$ be the space of all continuous functions $f : \Gamma \to \mathbb{R}$ imbued with sup (or L^∞) norm $\|f\|_{C^0(\Gamma)} = \max_{\boldsymbol{x}\in\Gamma} |f(\boldsymbol{x})|$. The point-wise approximation error of a sparse grid is bounded by

$$\|f - G_\Theta[f]\|_{C^0(\Gamma)} \le \left(1 + \|G_\Theta\|_{C^0(\Gamma)}\right) \inf_{p\in\mathcal{P}_{\Lambda^m(\Theta)}} \|f - p\|_{C^0(\Gamma)}, \tag{18}$$

where $\|G_\Theta\|_{C^0(\Gamma)}$ is the operator norm of G_Θ (also called the Lebesgue constant)

$$\|G_\Theta\|_{C^0(\Gamma)} = \sup_{g\in C^0(\Gamma)} \frac{\|G_\Theta[g]\|_{C^0(\Gamma)}}{\|g\|_{C^0(\Gamma)}} = \max_{\boldsymbol{x}\in\Gamma} \sum_{\boldsymbol{j}\in X(\Theta)} |\psi_{\boldsymbol{j}}(\boldsymbol{x})|.$$

For the nested case $\psi_{\boldsymbol{j}}(\boldsymbol{x})$ are defined in (12) and (14), and for the non-nested case $\psi^{\boldsymbol{i}}_{\boldsymbol{j}}(\boldsymbol{x})$ are defined in (15) with the repeated points grouped together. The error in quadrature approximation is bounded as

$$\left| \int_\Gamma f(\boldsymbol{x})\rho(\boldsymbol{x})d\boldsymbol{x} - G_\Theta[f] \right| \le \left(\int_\Gamma \rho(\boldsymbol{x})d\boldsymbol{x} + \sum_{\boldsymbol{j}\in X(\Theta)} |w_{\boldsymbol{j}}| \right) \inf_{p\in\mathcal{P}_{\Lambda^q(\Theta)}} \|f - p\|_{C^0(\Gamma)}, \tag{19}$$

and for the non-nested case the sum becomes $\sum_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)} |t_{\boldsymbol{i}} w^{\boldsymbol{i}}_{\boldsymbol{j}}|$ where weights corresponding to the same points are grouped together before taking the absolute value. Note, even if the one dimensional rule inducing the sparse grid has positive quadrature weights, since $t_{\boldsymbol{i}}$ can be negative, some of $w_{\boldsymbol{j}}$ can be negative.

The classical approach for sparse grids construction is to pre-define Θ according to some formula. Let $\boldsymbol{\xi}, \boldsymbol{\eta} \in \mathbb{R}^d$ be anisotropic weight vectors such that $\xi_k > 0$ for all $1 \le k \le d$, and let L indicate the "level" of the sparse grid approximation (the word "level" here is used loosely as the value of L has meaning only relative to $\boldsymbol{\xi}$). The classical anisotropic case takes

$$\Theta^{\boldsymbol{\xi}}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{i} \le L\}^\dagger, \tag{20}$$

log-corrected or *curved* selection [27]

$$\Theta^{\boldsymbol{\xi},\boldsymbol{\eta}}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{i} + \boldsymbol{\eta} \cdot \log(\boldsymbol{i} + \boldsymbol{1}) \le L\}^\ddagger, \tag{21}$$

*as with $\boldsymbol{m}(\boldsymbol{i})$ we take $\boldsymbol{q}(\boldsymbol{i}) = \bigotimes_{k=1}^d q(i_k)$

†Here $\cdot$ indicates the standard vector dot product $\boldsymbol{i} \cdot \boldsymbol{j} = \sum_{k=1}^d i_k j_k$.

‡Here $\log(\boldsymbol{i}) = \bigotimes_{k=1}^d \log(i_k)$

hyperbolic cross section

$$\Theta^{\boldsymbol{\xi}}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : (\boldsymbol{i}+\mathbf{1})^{\boldsymbol{\xi}} \leq L\}. \tag{22}$$

Alternatively, the multi-index set Θ can be selected as the smallest lower set that results in a $\Lambda^m(\Theta)$ (or $\Lambda^q(\Theta)$) that includes a desired polynomial space (see [27] for details). Total degree space

$$\{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{j} \leq L\} \subset \Lambda^m(\Theta), \quad \Rightarrow \quad \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{m}(\boldsymbol{i}-\mathbf{1}) \leq L\}^*, \tag{23}$$

or using a log-correction

$$\begin{aligned} & \{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{j} + \boldsymbol{\eta}\cdot\log(\boldsymbol{j}+\mathbf{1}) \leq L\} \subset \Lambda^m(\Theta), \quad \Rightarrow \\ \Theta^{\boldsymbol{\xi},\boldsymbol{\eta},m}(L) \;=\; & \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{m}(\boldsymbol{i}-\mathbf{1}) + \boldsymbol{\eta}\cdot\log(\boldsymbol{m}(\boldsymbol{i}-\mathbf{1})+\mathbf{1}) \leq L\}, \end{aligned} \tag{24}$$

or hyperbolic cross section space

$$\{\boldsymbol{j} \in \mathbb{N}^d : (\boldsymbol{j}+\mathbf{1})^{\boldsymbol{\xi}} \leq L\} \subset \Lambda^m(\Theta), \quad \Rightarrow \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : (\boldsymbol{m}(\boldsymbol{i}-\mathbf{1})+\mathbf{1})^{\boldsymbol{\xi}} \leq L\}. \tag{25}$$

Tensor selection types (23), (24) and (25) target corresponding polynomial spaces associated with point-wise approximation, the corresponding quadrature formulas use q in place of m, i.e., for total degree space

$$\{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{j} \leq L\} \subset \Lambda^q(\Theta), \quad \Rightarrow \quad \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{q}(\boldsymbol{i}-\mathbf{1})+\mathbf{1} \leq L\}^\dagger, \tag{26}$$

or using a log-correction

$$\begin{aligned} & \{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot\boldsymbol{j} + \boldsymbol{\eta}\cdot\log(\boldsymbol{j}+\mathbf{1}) \leq L\} \subset \Lambda^q(\Theta), \quad \Rightarrow \\ \Theta^{\boldsymbol{\xi},\boldsymbol{\eta},q}(L) \;=\; & \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi}\cdot(\boldsymbol{q}(\boldsymbol{i}-\mathbf{1})+\mathbf{1}) + \boldsymbol{\eta}\cdot\log(q(\boldsymbol{i}-\mathbf{1})+\mathbf{2}) \leq L\}, \end{aligned} \tag{27}$$

or hyperbolic cross section space

$$\{\boldsymbol{j} \in \mathbb{N}^d : (\boldsymbol{j}+\mathbf{1})^{\boldsymbol{\xi}} \leq L\} \subset \Lambda^q(\Theta), \quad \Rightarrow \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : (\boldsymbol{q}(\boldsymbol{i}-\mathbf{1})+\mathbf{1})^{\boldsymbol{\xi}} \leq L\}. \tag{28}$$

For example, $\Theta^{\mathbf{1},q}(L)$ constructed according to (26) will result in $G_{\Theta^{\mathbf{1},q}(L)}$ that integrates exactly all polynomials of total degree up to and including L. Similarly, $\Theta^{\mathbf{1},-\frac{\mathbf{1}}{\mathbf{2}},m}(L)$ will result in the dominant polynomial space defined in Proposition 8 and equation (8) in [5]. For more information about optimal and quasi-optimal polynomial approximation see [27] and references therein.

2.3 Global Grids: Sequence Grid

A sequence grid is constructed from a one dimensional nested rule with $m(l) = l+1$. The theoretical properties, i.e., (18) and (19), are identical to the global grid, however, the sequence grid uses representation in terms of Newton (as opposed to Lagrange) polynomials. Let

$$\phi_1(x) = 1, \quad \text{for } j > 1, \quad \phi_j(x) = \prod_{i=1}^{j-1} \frac{x - x_i}{x_j - x_i}, \quad \text{and for } \boldsymbol{j} \in \mathbb{N}^d, \quad \phi_{\boldsymbol{j}}(\boldsymbol{x}) = \prod_{k=1}^{d} \phi_{j_k},$$

*Here for notational convenience we assume that $m(-1) = 0$.

†Here for notational convenience we assume that $q(-1) = -1$.

where each ϕ_{j_k} is evaluated at the corresponding k-th component of $\boldsymbol{x}$. Then $G_\Theta[f]$ can be written as

$$G[f](\boldsymbol{x}) = \sum_{\boldsymbol{j} \in X(\Theta)} s_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}), \tag{29}$$

where the surplus coefficients $s_{\boldsymbol{j}}$ satisfy the linear system of equation

$$\sum_{\boldsymbol{1} \le \boldsymbol{j} \le \boldsymbol{i}} s_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}_{\boldsymbol{i}}) = f(\boldsymbol{x}_{\boldsymbol{i}}), \quad \text{for every } \boldsymbol{i} \in X(\Theta). \tag{30}$$

Note that all sparse grids induced by nested one dimensional rules can be written in the Newton form above, however, TASMANIAN implements sequence grids only for the case when $m(l) = l + 1$.

Computing and storing the coefficients $s_{\boldsymbol{j}}$ is more expensive then the weights $t_{\boldsymbol{i}}$, especially when $f(\boldsymbol{x})$ is a vector valued function where each output dimension of $f(\boldsymbol{x})$ requires a separate set of coefficients. However, computing the surpluses is a one time cost, followup evaluations of a sequence approximation are much cheaper since Newton polynomials are easier to construct. Thus, sequence grids are faster when a large number of evaluations of $G_\Theta[f]$ are desired.

2.4 Global Grids: Refinement

Global and sequence grids implemented in TASMANIAN support two types of refinement based on surpluses and anisotropic coefficient decay. Given $G_\Theta[f]$ for some index set Θ, the goal of a refinement procedure is to produce an updated $\hat{\Theta}$ (with $\Theta \subset \hat{\Theta}$) such that $G_{\hat{\Theta}}[f]$ is more accurate and the additional indexes included in $\hat{\Theta}$ are "optimal" with respect to properties of $f(\boldsymbol{x})$ that are "inferred" from $G_\Theta[f]$. Note that refinement is supported only for grids induced by nested rules.

The surplus refinement is implemented only for grids induced by rules with $m(l) = l + 1$ (sequence and global grids alike). In that case $X(\Theta) = \Theta + \mathbf{1}$ and the refinement strategy considers the hierarchical surpluses (30). The set Θ is then expanded with indexes that are "close" to the indexes associated with large relative surpluses. Specifically:

$$\hat{\Theta} = \Theta \bigcup \left(\bigcup_{\boldsymbol{j} \in X(\Theta), |s_{\boldsymbol{j}}| > \epsilon \cdot f_{\max}} \left\{ \boldsymbol{i} \in \mathbb{N}^d : \sum_{k=1}^{d} |i_k - j_k - 1| = 1 \right\} \right), \tag{31}$$

where $f_{\max} = \max_{\boldsymbol{j} \in X(\Theta)} |f(\boldsymbol{x}_{\boldsymbol{j}})|$ and $\epsilon > 0$ is user specified tolerance. In the case when $f(\boldsymbol{x})$ has multiple outputs, if using a global grids (i.e., with Lagrange representation) then the user must specify one output to be used by the refinement criteria. The surpluses and $f_{\max}$ will be computed only for that one output. In contrast, a sequence grid computes and stores the surpluses for all outputs, thus, refinement can be easily done with either one output or all outputs simultaneously, in which case we refine for those $\boldsymbol{j} \in X(\Theta)$ such that $|s_{\boldsymbol{j}}| > \epsilon \cdot f_{\max}$ for any of the outputs. Here the purpose of the $f_{\max}$ is used to normalize the surpluses in case a vector valued function has outputs with significantly different scaling.

The second type of refinement is labeled *anisotropic*, and it is a two stage process. First, $G_\Theta[f]$ is expresses in terms of orthogonal multivariate Legendre polynomials, then anisotropic weights $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ are inferred from the decay rate of the coefficients. The refinement set $\hat{\Theta}$ is constructed according to (23) or (24) so that $G_{\hat{\Theta}}$ includes a desired minimum number of new points, where the minimum number of new points

exploits parallelism in computing the values of $f(\boldsymbol{x_j})$. Legendre expansion is computationally expensive, hence grids induced by rules with growth $m(l) = l + 1$ use hierarchical surpluses in place of the Legendre coefficients. As before, when $f(\boldsymbol{x})$ has multiple outputs, sequence and global grids can focus on a single output, and sequence grids can considers the largest normalized surplus, i.e., largest $|s_{\boldsymbol{j}}|/f_{\max}$ among all outputs. For more details on this type of refinement, see [27].

2.5 Global Grids: One dimensional rules

2.5.1 Chebyshev rules

Roots and extrema of Chebyshev polynomials are a common choice of one dimensional interpolation and integration rules and TASMANIAN implements several Chebyshev based rules. The non-nested Chebyshev points are placed at the roots of the polynomials and the growth is either $m(l) = l+1$ or $m(l) = 2l+1$. The Clenshaw-Curtis [9] and Clenshaw-Curtis-zero (latter assumes the $f(\boldsymbol{x})$ is zero at $\partial\Gamma$) use only the nested Chebyshev points and $m(l)$ grows exponentially. The nested Fejer type 2 [12] points use the extrema of the Chebyshev polynomials and also have exponential $m(l)$.

In addition, the library includes the more recently developed $\mathcal{R}$-Leja points [7]. Define $\{\theta_j\}_{j=1}^{\infty}$ as

$$\theta_1 = 0, \quad \theta_2 = \pi, \quad \theta_3 = \frac{\pi}{2}, \quad \text{for } j > 3,\ \theta_j = \begin{cases} \theta_{j-1} + \pi, & j \text{ is odd} \\ \frac{1}{2}\theta_{\frac{j}{2}+1}, & j \text{ is even} \end{cases} \tag{32}$$

then the $\mathcal{R}$-Leja points are given by $x_j = \cos(\theta_j)$ and the centered $\mathcal{R}$-Leja points start at $x_1 = 0$, $x_2 = 1$, $x_3 = -1$, and $x_j = \cos(\theta_j)$ for $j > 3$. The growth of the $\mathcal{R}$-Leja rule is $m(l) = l + 1$ and the centered rule allows for multiple definitions, namely odd rules $m(l) = 2l + 1$, the $\mathcal{R}$-Leja double-2 growth defined by

$$m(0) = 1, \quad m(1) = 3, \quad \text{for } l > 1, \quad m(l) = 2^{\lfloor \frac{l}{2} \rfloor + 1} \left(1 + \frac{l}{2} - \left\lfloor \frac{l}{2} \right\rfloor\right) + 1, \tag{33}$$

and the $\mathcal{R}$-Leja double-4 rule defined by

$$m(l) = 1, \quad m(l) = 3, \quad \text{for } l > 1, \quad m(l) = 2^{\lfloor \frac{l-2}{4} \rfloor + 2} \left(1 + \frac{l-2}{4} - \left\lfloor \frac{l-2}{4} \right\rfloor\right) + 1, \tag{34}$$

where $\lfloor x \rfloor = \max\{z \in \mathbb{Z} : z \leq x\}$ is the *floor* function, see [27] for more details.

TASMANIAN also includes a shifted $\mathcal{R}$-Leja sequence defined by

$$x_1 = -\frac{1}{2}, \quad x_2 = \frac{1}{2}, \quad \text{for } j > 2, \quad x_j = \begin{cases} \sqrt{\frac{1+x_{(j+1)/2}}{2}}, & j \text{ is odd} \\ -x_{j-1}, & j \text{ is even} \end{cases} \tag{35}$$

which comes with growth $m(l) = l + 1$ or $m(l) = 2(l + 1)$. Table 1, summarizes all Chebyshev rules.

2.5.2 Gauss rules

The roots of orthogonal polynomials are a common choice for points for numerical integration due to the high level of precision. Orthogonality is defined with respect to a specific integration weight that often

Points	$m(l)$	$q(l)$	Note:
Chebyshev: Non-nested Chebyshev roots	$m(l) = l + 1$	$q(l) = l - 1 + (l \mod 2)$	very low Lebesgue constant
Clenshaw-Curtis: Nested Chebyshev roots	$m(0) = 1, m(l) = 2^l + 1$	$q(l) = m(l)$	very low Lebesgue constant
Clenshaw-Curtis-Zero: Nested Chebyshev roots	$m(l) = 2^{l+1} - 1$	$q(l) = 2^l$	assumes $f(\boldsymbol{x}) = 0$ at $\partial\Gamma$
Fejer type 2: Nested Chebyshev extrema	$m(l) = 2^{l+1} - 1$	$q(l) = 2^l$	no points at $\partial\Gamma$
$\mathcal{R}$-Leja: See (32)	$m(l) = l + 1$	$q(l) = l - 1 + (l \mod 2)$	see [7, 27]
$\mathcal{R}$-Leja odd: Centered $\mathcal{R}$-Leja	$m(l) = 2l + 1$	$q(l) = m(l)$	see [7, 27]
$\mathcal{R}$-Leja double 2: Centered $\mathcal{R}$-Leja	see (33)	$q(l) = m(l)$	see [7, 27]
$\mathcal{R}$-Leja double 4: Centered $\mathcal{R}$-Leja	see (34)	$q(l) = m(l)$	see [7, 27]
$\mathcal{R}$-Leja shifted: See (35)	$m(l) = l + 1$	$q(l) = m(l) - 1$	see [8]
$\mathcal{R}$-Leja shifted even: See (35)	$m(l) = 2(l + 1)$	$q(l) = 2l + 1$	see [8]

Table 1. Summary of the available Chebyshev rules.

times requires additional parameters α and/or β. The Gauss rules also include the Hermite and Laguerre polynomials that assume unbounded domain. Gauss rules are usually non-nested, have growth $m(l) = l+1$, and precision $q(l) = 2l + 1$. Odd versions of the rules use growth $m(l) = 2l + 1$ and $q(l) = 4l + 1$, and when coupled with *qpcurved* or *qptotal* tensor selection the odd versions of the Gauss rules usually result in sparse grids with fewer points.

Gauss-Patterson [24] points are a notable exception in most ways. The Patterson construction uses the Legendre orthogonal polynomials and imposes the additional requirement that the points are nested, which leads to a rule with growth $m(l) = 2^{l+1} - 1$ and precision $q(l) = \frac{3}{2}m(l) - \frac{1}{2} = 3 \cdot 2^l - 2$. Note that the construction of the Gauss-Patterson points and weights is a computationally expensive and ill-conditioned problem, TASMANIAN does not include code that computes the point and weight, instead the first 9 levels are hard-coded into the library. The 9 levels should give sufficient precision for most applications, while the custom rule capabilities of the library can be used to extend beyond that limit, assuming the user provides Gauss-Patterson points and weights for higher levels. Summary of all Gauss rules is listed in Table 2.

2.5.3 Greedy rules

TASMANIAN implements a number of rules using sequences of points that are based on greedy optimization. The most well known rule uses the Leja points [10], where

$$x_1 = 0, \qquad \text{for } j > 1 \quad x_{j+1} = \operatorname*{argmax}_{x \in [-1,1]} \prod_{i=1}^{j} \big|x - x_i\big|. \tag{36}$$

Name	Generalized Integral	Notes
Gauss-Patterson:	$\int_a^b f(x)dx$	The only nested rule Canonical: $a=-1, b=1$
Gauss-Legendre:	$\int_a^b f(x)dx$	Highest 1-D exactness Canonical: $a=-1, b=1$
Gauss-Chebyshev type 1:	$\int_a^b f(x)(b-x)^{-0.5}(x-a)^{-0.5}dx$	Canonical: $a=-1, b=1$
Gauss-Chebyshev type 2:	$\int_a^b f(x)(b-x)^{0.5}(x-a)^{0.5}dx$	Canonical: $a=-1, b=1$
Gauss-Gegenbauer:	$\int_a^b f(x)(b-x)^{\alpha}(x-a)^{\alpha}dx$	Must specify α Canonical: $a=-1, b=1$
Gauss-Jacobi:	$\int_a^b f(x)(b-x)^{\alpha}(x-a)^{\beta}dx$	Must specify α, β Canonical: $a=-1, b=1$
Gauss-Laguerre:	$\int_a^{\infty} f(x)(x-a)^{\alpha}e^{-b(x-a)}dx$	Must specify α Canonical: $a=0, b=1$
Gauss-Hermite:	$\int_{-\infty}^{\infty} f(x)(x-a)^{\alpha}e^{-b(x-a)^2}dx$	Must specify α Canonical: $a=0, b=1$

Table 2. Summary of the available Gauss rules.

Similar construction can be done using the extrema of the Lebesgue function

$$x_1 = 0, \qquad \text{for } j > 1 \quad x_{j+1} = \underset{x\in[-1,1]}{\text{argmax}} \sum_{j'=1}^{j} \prod_{i=1, i\neq j'}^{j} \left| \frac{x - x_i}{x_{j'} - x_i} \right|. \tag{37}$$

We can greedily minimize the norm of $\mathcal{U}^{m(j+1)}$, where $x_1 = 0$ and for $j > 1$

$$x_{j+1} = \underset{x\in[-1,1]}{\text{argmin}} \max_{y\in[-1,1]} \prod_{i=1}^{j} \left| \frac{y - x_i}{x - x_i} \right| + \sum_{j'=1}^{j} \left| \frac{y - x}{x_{j'} - x} \right| \prod_{i=1, i\neq j'}^{j} \left| \frac{y - x_i}{x_{j'} - x_i} \right| \tag{38}$$

or minimizing the norm of the surplus operator $\Delta^{m(j+1)}$, where $x_1 = 0$ and for $j > 1$

$$x_{j+1} = \underset{x\in[-1,1]}{\text{argmin}} \max_{y\in[-1,1]} \left(1 + \sum_{i=1}^{j} \prod_{j'=1, j'\neq i}^{j} \left| \frac{x - x_{j'}}{x_i - x_{j'}} \right| \right) \prod_{j'=1}^{j} \left| \frac{y - x_{j'}}{x - x_{j'}} \right|. \tag{39}$$

In all cases the growth can be set to $m(l) = l + 1$ or $m(l) = 2l + 1$. However, unlike the $\mathcal{R}$-Leja points, the odd rules here do not result in symmetric distribution of the points, hence $q(l) = m(l) - 1$ (and $q(0) = 1$). For a numerical survey of the properties of interpolants constructed from the above sequences, see [27]. Note that quadrature rules using the above sequences can potentially result in zero weights (i.e., $w_j = 0$ for some j), TASMANIAN does NOT automatically check if the weights are zero. The greedy rules are intended for interpolation purposes and are not the best rules to use for numerical integration. A list of the greedy rules is given in Table 3.

Name	Points	$m(l)$
Leja: **Leja odd:**	See (36)	$m(l) = l + 1$ $m(l) = 2l + 1$
Max-Lebesgue: **Max-Lebesgue odd:**	See (37)	$m(l) = l + 1$ $m(l) = 2l + 1$
Min-Lebesgue: **Min-Lebesgue odd:**	See (38)	$m(l) = l + 1$ $m(l) = 2l + 1$
Min-Delta: **Min-Delta odd:**	See (39)	$m(l) = l + 1$ $m(l) = 2l + 1$

Table 3. Summary of the available greedy sequence rules.

2.6 Local Polynomial Grids: Hierarchical interpolation rule

Local polynomial grids are constructed from equidistant points and use functions with support restricted to a neighborhood of each point. The local support of the functions allow the employment of locally adaptive strategies and thus local grids are suitable for approximating functions with sharp behavior, e.g., large fluctuation of the gradient. Similar to the global grids, local grids are constructed from tensors of points and functions in one dimension. In contrast to global grids, local grids use functions with local support and very strict hierarchy. For in depth analysis of the properties of the local grids see [16, 20, 21, 26].

Let $\{x_j\}_{j=0}^{\infty} \in [-1, 1]$ be a sequence of nodes (w.l.o.g., we assume that we are working on the canonical domain $[-1, 1]$) and let $\{\Delta x_j\}_{j=0}^{\infty}$ indicate the "resolution" of our approximation at point x_j, i.e., the support of the associated function. In addition, we have the hierarchy defined by the *parents* and *children* sets

$$\begin{aligned} P_j &= \{i \in \mathbb{N} : x_i \text{ is a parent of } x_j\}, \\ O_j &= \{i \in \mathbb{N} : x_i \text{ is a child (offspring) of } x_j\}, \end{aligned}$$

where P_j can have more than one element. For a particular example of such hierarchies, see Section 2.8. We assume that P_j and O_j define a partial order of the points and let $h : \mathbb{N} \to \mathbb{N}$ map each point to a place in the hierarchy also called *level*, i.e.,

$$h(j) = \begin{cases} 0, & P_j = \emptyset \\ h(i) + 1, & \text{for any } i \in P_j \end{cases}$$

We define the ancestry set A_j

$$A_j = \{i \in \mathbb{N} : h(i) \leq h(j) \quad \text{and} \quad (x_i - \Delta x_i, x_i + \Delta x_i) \cap (x_j - \Delta x_j, x_j + \Delta x_j) \neq \emptyset\}$$

In order to construct the basis functions, for each x_j we consider the set of p nearest ancestors

$$F_j^{(p)} = \underset{F \subset A_j, \#F = p}{\operatorname{argmin}} \sum_{i \in F} |x_i - x_j|,$$

where $\#F$ indicates the number of elements of F. Note that $F_j^{(p)}$ is defined only for $p \leq \#A_j$.

The functions associated with a hierarchy can have various polynomial order $p \geq 0$. For constant functions

$$\phi_j^{(0)}(x) = \begin{cases} 1, & x \in (x_j - \Delta x_j, x_j + \Delta x_j) \\ 0, & x \notin (x_j - \Delta x_j, x_j + \Delta x_j) \end{cases}$$

For linear functions

$$\phi_j^{(1)}(x) = \begin{cases} 1 - \frac{|x - x_j|}{\Delta x_j} & x \in (x_j - \Delta x_j, x_j + \Delta x_j) \\ 0, & x \notin (x_j - \Delta x_j, x_j + \Delta x_j) \end{cases}$$

and functions of arbitrary order $p > 1$

$$\phi_j^{(p)}(x) = \begin{cases} \prod_{i \in F_j^{(p)}} \frac{x - x_i}{x_j - x_i}, & x \in (x_j - \Delta x_j, x_j + \Delta x_j) \\ 0, & x \notin (x_j - \Delta x_j, x_j + \Delta x_j) \end{cases}$$

Note that a function can have order p only if the corresponding $F_j^{(p)}$ exists, i.e., $h(j)$ is large enough. TASMANIAN constructs local polynomial grids by automatically using the largest p available for each $\phi_j^{(p)}(x)$, optionally the library can be restricted p to a maximum user defined value. In the rest of this discussion, we would omit p.

We extend the one dimensional hierarchy to a d-dimensional context using multi-index notation*

$$\boldsymbol{x_j} = \bigotimes_{k=1}^{d} x_{j_k}, \qquad \phi_{\boldsymbol{j}}(\boldsymbol{x}) = \prod \phi_{j_k}, \qquad supp\{\phi_{\boldsymbol{j}}(\boldsymbol{x})\} = \bigotimes_{k=1}^{d} (x_{j_k} - \Delta x_{j_k}, x_{j_k} + \Delta x_{j_k}),$$

where each $\prod \phi_{j_k}$ is evaluated at the corresponding k-th entry of $\boldsymbol{x}$ and $supp\{\phi_{\boldsymbol{j}}(\boldsymbol{x})\}$ indicate the support of $\phi_{\boldsymbol{j}}(\boldsymbol{x})$. Parents and children are associated with different directions

$$P_j^{(k)} = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{i} \underset{k}{=} \boldsymbol{j}^{\dagger} \text{ and } i_k \in P_{j_k}\} \qquad O_j^{(k)} = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{i} \underset{k}{=} \boldsymbol{j} \text{ and } i_k \in O_{j_k}\}$$

and the level of a multi-index is $h(\boldsymbol{j}) = \sum_{k=1}^{d} h(j_k)$. The multidimensional ancestry set is

$$A_{\boldsymbol{j}} = \left\{ \boldsymbol{i} \in \mathbb{N}^d : h(\boldsymbol{i}) \leq h(\boldsymbol{j}) \quad \text{and} \quad supp\{\phi_{\boldsymbol{i}}(\boldsymbol{x})\} \bigcap supp\{\phi_{\boldsymbol{j}}(\boldsymbol{x})\} \neq \emptyset \right\}$$

For $f : \Gamma \to \mathbb{R}$, a multi-dimensional interpolant of $f(\boldsymbol{x})$ is defined by a set of points X so that

$$G_X[f] = \sum_{\boldsymbol{j} \in X} s_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}),$$

where the surplus coefficients $s_{\boldsymbol{j}}$ are chosen such that $G_X[f](\boldsymbol{x_i}) = f(\boldsymbol{x_i})$ for all $\boldsymbol{i} \in X$, specifically, by definition of $\phi_{\boldsymbol{j}}(\boldsymbol{x})$

$$s_{\boldsymbol{j}} = f(\boldsymbol{x_j}) - \sum_{\boldsymbol{i} \in A_{\boldsymbol{j}}} s_{\boldsymbol{i}} \phi_{\boldsymbol{i}}(\boldsymbol{x_j}).$$

In the case when $f(\boldsymbol{x})$ is a vector valued function, a separate set of surplus coefficients is computed for each output. When TASMANIAN first creates a local polynomial grid, the set of points is chosen so that

$$X = \{\boldsymbol{j} \in \mathbb{N}^d : h(\boldsymbol{j}) \leq L\}, \tag{40}$$

for some use specified L.

*Similar to the global grids, $\mathbb{N}$ indicates the set of non-negative integers, and $W, F, A, P, O, B, X \subset \mathbb{N}^d$ denote sets of multi-indexes.

†Here by $\boldsymbol{i} \underset{k}{=} \boldsymbol{j}$ we mean that $\boldsymbol{i}$ and $\boldsymbol{j}$ have the same components in all but the k-th direction

2.7 Local Polynomial Grids: Adaptive refinement

Locally adaptive grids are best utilized with an appropriate refinement strategy. Suppose we have constructed $G_X[f]$ for some X and consider an updated $\hat{X}$ so that new points are added only in the region of Γ where $G_X[f]$ sharply deviates from $f(\boldsymbol{x})$. The surpluses $s_{\boldsymbol{j}}$ are a good local error indicator, and thus we define $\hat{X}$ that contains only indexes that are parents or children of indexes $\boldsymbol{j}$ associated with large $s_{\boldsymbol{j}}$.

First, we define the set of large surpluses

$$B = \left\{ \boldsymbol{j} \in X : \frac{|s_{\boldsymbol{j}}|}{f_{\max}} > \epsilon \right\},$$

where $\epsilon > 0$ is desired tolerance and $f_{\max} = \max_{\boldsymbol{i} \in X} |f(\boldsymbol{x}_{\boldsymbol{i}})|$. When $f(\boldsymbol{x})$ is a vector valued function, an index $\boldsymbol{j}$ is included in B if any of the outputs has normalized surpluses larger than ϵ. TASMANIAN implements 4 different refinement strategies, where $\hat{X}$ is selected by including parents and/or children of $\boldsymbol{j} \in B$ in different directions. This is done based on consideration of "orphan" directions and directional surpluses.

For each index in $\boldsymbol{j}$, we define the "orphan" directions

$$T_{\boldsymbol{j}} = \left\{ k \in \{1, 2, \ldots, d\} : P^{(k)}_{\boldsymbol{j}} \not\subset X \right\},$$

thus, $T_{\boldsymbol{j}}$ contains the directions where we have missing parents. We also consider directional surpluses, let

$$W^{(k)}_{\boldsymbol{j}} = \left\{ \boldsymbol{i} \in X : \boldsymbol{i} \underset{k}{=} \boldsymbol{j} \right\}, \qquad G_{W^{(k)}_{\boldsymbol{j}}}[f] = \sum_{\boldsymbol{i} \in W^{(k)}_{\boldsymbol{j}}} c^{(k)}_{\boldsymbol{i}} \phi_{\boldsymbol{i}}(\boldsymbol{x}),$$

where we have a set of the one directional surpluses $c^{(k)}_{\boldsymbol{i}}$ associated with each index $\boldsymbol{j}$, however, we focus our attention only to $c^{(k)}_{\boldsymbol{j}}$. The set of large one directional surpluses is

$$C_{\boldsymbol{j}} = \left\{ k \in \{1, 2, \ldots, d\} : \frac{\left| c^{(k)}_{\boldsymbol{j}} \right|}{f_{\max}} > \epsilon \right\}.$$

The classical refinement strategy constructs $\hat{X}$ by adding the children of $\boldsymbol{j} \in B$, i.e.,

$$\hat{X} = X \bigcup \left(\bigcup_{\boldsymbol{j} \in B} \bigcup_{k \in \{1,2,\ldots,d\}} O^{(k)}_{\boldsymbol{j}} \right). \tag{41}$$

However, the classical strategy can lead to instability around orphan points, hence, the parents-first approach adds parents before the children

$$\hat{X} = X \bigcup \left(\bigcup_{\boldsymbol{j} \in B} \left(\bigcup_{k \in T_{\boldsymbol{j}}} P^{(k)}_{\boldsymbol{j}} \right) \bigcup \left(\bigcup_{k \notin T_{\boldsymbol{j}}} O^{(k)}_{\boldsymbol{j}} \right) \right). \tag{42}$$

Large surplus signifies large local error, however, refinement doesn't have to be done in all directions, thus, the directional refinement uses $k \in C_{\boldsymbol{j}}$, i.e.,

$$\hat{X} = X \bigcup \left(\bigcup_{\boldsymbol{j} \in B} \bigcup_{k \in C_{\boldsymbol{j}}} O_{\boldsymbol{j}}^{(k)} \right). \tag{43}$$

Combining the parents-first and directional approach leads to the family-direction-selective (FDS) method

$$\hat{X} = X \bigcup \left(\bigcup_{\boldsymbol{j} \in B} \left(\bigcup_{k \in C_{\boldsymbol{j}} \cap T_{\boldsymbol{j}}} P_{\boldsymbol{j}}^{(k)} \right) \bigcup \left(\bigcup_{k \in C_{\boldsymbol{j}} \setminus T_{\boldsymbol{j}}} O_{\boldsymbol{j}}^{(k)} \right) \right). \tag{44}$$

For more details about the four refinement strategies see [26].

2.8 Local Polynomial Grids: One dimensional rules

TASMANIAN implements three specific one dimensional hierarchical rules: standard rule with Δx_j decreasing by 2 at each level, a semi-local rule where global basis is used for levels 0 and 1, and a modified rule that assumes $f(\boldsymbol{x}) = 0$ at $\partial\Gamma$.

The standard local rule is given by

$$x_0 = 0, \qquad x_1 = -1, \qquad x_2 = 1, \qquad \text{for } j > 2 \quad x_j = (2j-1) \times 2^{-\lfloor \log_2(j-1) \rfloor} - 3, \tag{45}$$

where $\lfloor x \rfloor = \max\{z \in \mathbb{Z} : z \leq x\}$ is the *floor* function. The parent sets are

$$P_0 = \emptyset, \qquad P_1 = \{0\}, \qquad P_2 = \{0\}, \qquad P_3 = \{1\}, \qquad \text{for } j > 3 \quad P_j = \left\{ \left\lfloor \frac{j+1}{2} \right\rfloor \right\},$$

and the offspring sets are

$$O_0 = \{1, 2\}, \qquad O_1 = \{3\}, \qquad O_2 = \{4\}, \qquad \text{for } j > 2 \quad O_j = \{2j-1,\, 2j\}.$$

The level function is

$$h(j) = \begin{cases} 0, & j = 0, \\ 1, & j = 1, \\ \lfloor \log_2(j-1) \rfloor + 1, & j > 1, \end{cases}$$

and the resolution Δx_j is given by $\Delta x_0 = 1$ and for $j > 0$ we have $\Delta x_j = 2^{-h(j)+1}$. Figure 1 shows the first four levels of the linear, quadratic, and cubic functions.

A modification to the standard rule uses the same points, however, functions at level $l = 1$ with degree higher than linear will have global support, i.e., if $p > 1$ then $\Delta x_1 = \Delta x_2 = 2$. In addition, for the purpose of parents refinement (42) and (44) we use $P_3 = P_4 = \{1, 2\}$. The modified rule sacrifices resolution and gains higher polynomial order, thus, the semi-local approach is better suited for functions with "smoother" behavior. Figure 2 shows the linear, quadratic, and cubic semi-local functions. Note: there is no difference between the linear versions of the local and semi-local rules.

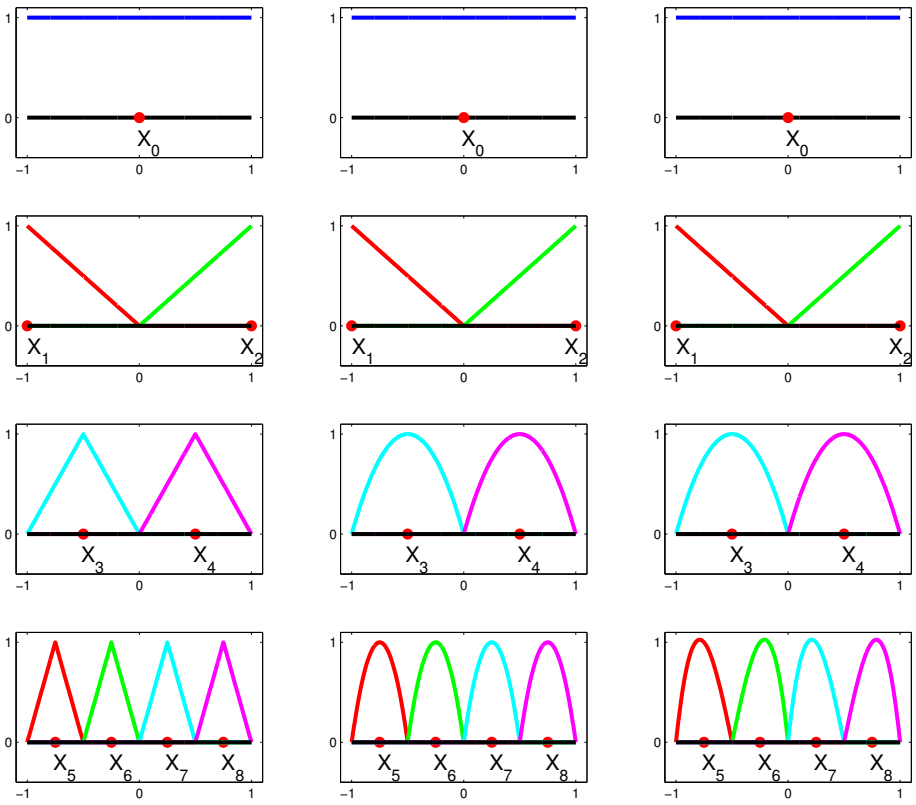

Figure 1. Local polynomial points (*rule_localp*) and functions, left to right: linear, quadratic, and cubic functions.

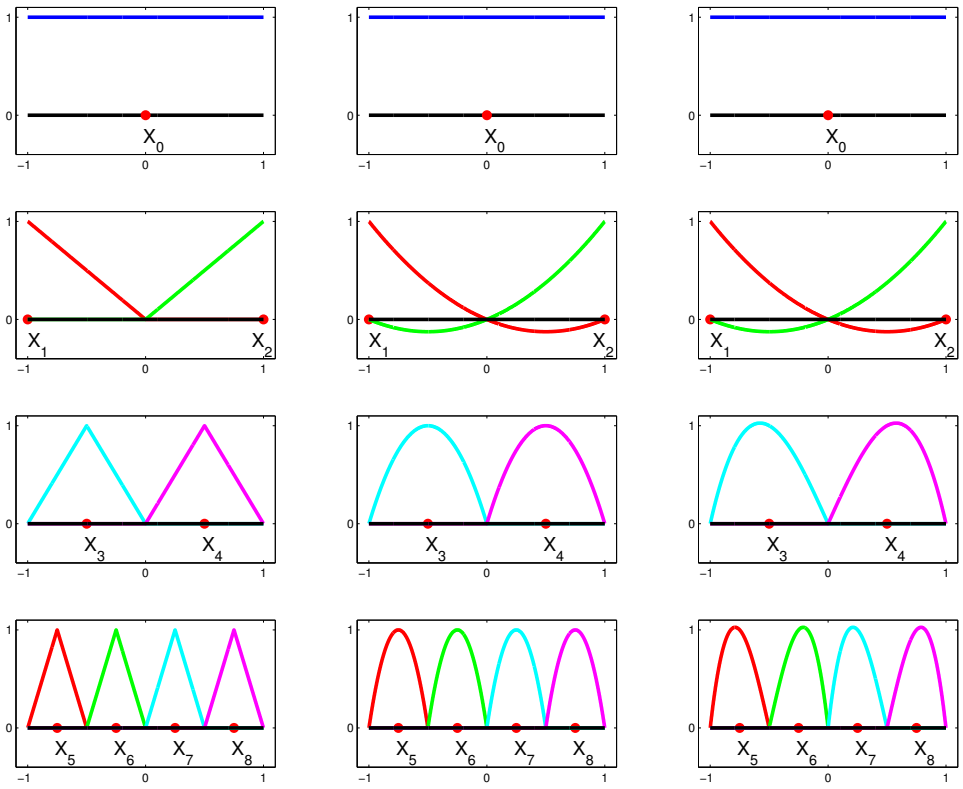

Figure 2. Semi-local polynomial points (*rule_semilocalp*) and functions, left to right: linear, quadratic, and cubic functions.

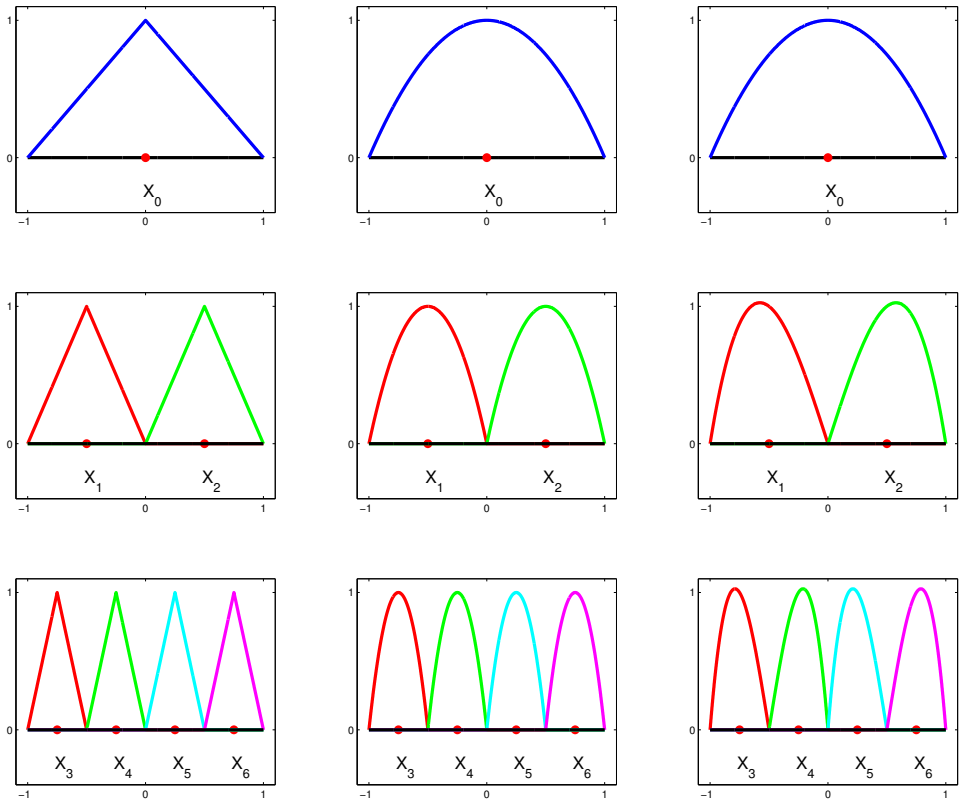

Figure 3. Semi-local polynomial points (*rule_localp0*) and functions, left to right: linear, quadratic, and cubic functions.

An alternative local rule does not put points on the boundary and implicitly assumes that $f(\boldsymbol{x}) = 0$ at $\partial\Gamma$. The hierarchy is defined as

$$x_0 = 0, \qquad \text{for } j > 0 \quad x_j = (2j+3) \times 2^{-\lfloor \log_2(j+1) \rfloor} - 3, \tag{46}$$

The parent sets are

$$P_0 = \emptyset, \qquad \text{for } j > 0 \quad P_j = \left\{ \left\lfloor \frac{j-1}{2} \right\rfloor \right\},$$

and the offspring sets are $O_j = \{2j+1, 2j+2\}$. The level function is $h(j) = \lfloor \log_2(j+1) \rfloor$ and the resolution Δx_j is given by $\Delta x_0 = 2^{-h(j)}$. Figure 3 shows the first three levels of the linear, quadratic, and cubic functions.

2.9 Wavelets

TASMANIAN, in addition to the local polynomial rules, also implements wavelet rules with order 1 and 3. The hierarchy followed by the wavelets as well as the refinement strategies are very similar to the local grids. The differences are as follows:

- The zeroth levels of wavelet rules of order 1 and 3, have 3 and 5 points respectively. This is a sharp contrast to the single point of of the polynomial rules, since level 0 wavelet grid has 3^d (or 5^d) points in d-dimensions (as opposed to a single point). See Figure 4.
- Wavelet rules have larger Lebesgue constant, which is due to the large magnitude of the boundary wavelet functions. This can lead to instability of the wavelet interpolant around the boundary of the domain.
- The linear system of equations associated with the wavelet surpluses is not triangular, hence a sparse matrix has to be inverted every time values are loaded into the interpolant. This leads to a significantly higher computational cost in manipulating the wavelet grids, especially in loading values and performing direction selective refinement.
- Wavelets form a Riesz basis, which over-simplistically means that the wavelet surpluses are much sharper indicators of the local error and hence wavelet based refinement strategy "could" generate a grid that is more accurate and has fewer points. The quotations around the word "could" relate to the point about the Lebesgue constant.
- For more details about wavelets, see [16, 18, 28].

2.10 Domain Transformation

Sparse grids are build on canonical 1D domain $[-1, 1]$, with the exception of Gauss-Laguerre and Gauss-Hermite rules that use $[0, \infty)$ and $(-\infty, \infty)$ respectively. Linear transformation can be applied to translate $[-1, 1]$ to an arbitrary interval $[a, b]$, for unbounded domain we can apply shift a and scaling b. This simple linear transformation will not affect the properties of the grid, i.e., function space spanned by the basis or the Lebesgue constant. Thus, the a and b parameters are used to simplify implementation and generate a grid on a domain consistent with the range of the input of an arbitrary function $f(\boldsymbol{x})$. However, non-linear transformation can also be used with the goal of accelerating convergence.

2.10.1 Conformal Map

For simplicity, assume that $f(x)$ is a one dimensional function defined on $[-1, 1]$. Then a conformal map is any monotonic strictly increasing $g(x)$ such that

$$g : [-1, 1] \to [-1, 1], \qquad g(-1) = -1, \qquad \text{and} \quad g(1) = 1,$$

Then, instead of constructing a sparse grids rule that integrates or interpolates $f(x)$, we construct a rule for $f(g(x))$, with the hope that the composed function will be easier to approximate, e.g., have larger region of analyticity [2, 19].

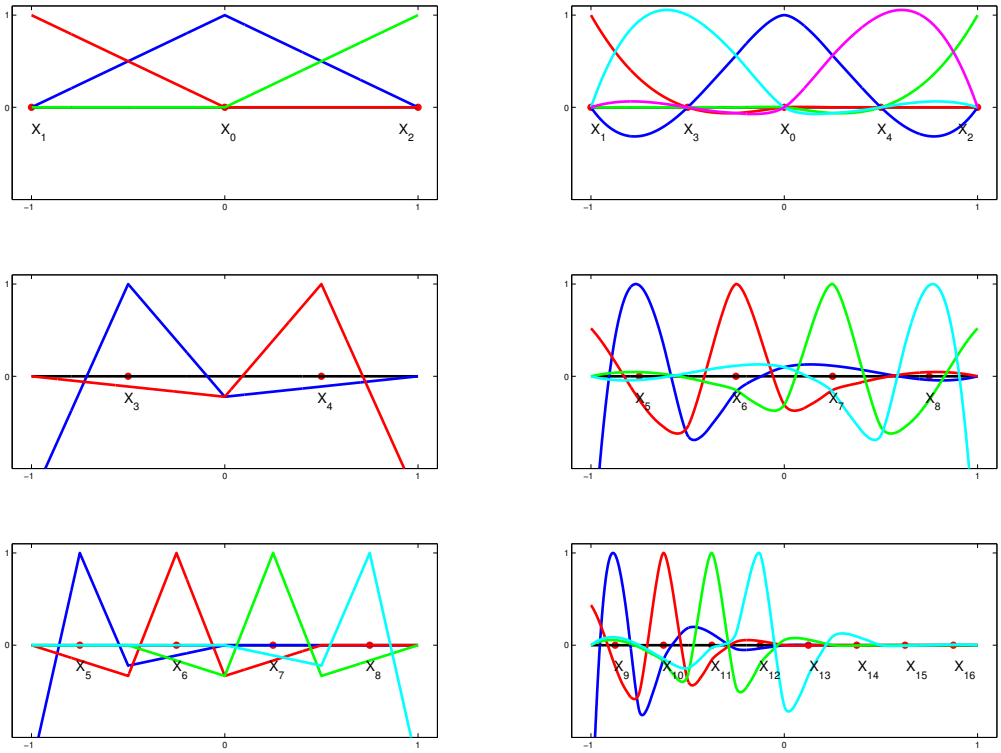

Figure 4. The first three levels for wavelets of order 1 (left) and 3 (right). The functions associated with x_{13}, x_{14}, x_{15}, and x_{16} are purposely omitted to reduce the clutter on the plot, since the funcitons are mirror images of the those associated with x_{12}, x_{11}, x_{10}, and x_9 respectively.

In case of a quadrature rule, we note that

$$\int f(x)dx = \int f(g(x))g'(x)dx,$$

thus if we have a quadrature rule $\int f(g(x))dx \approx \sum_{\boldsymbol{i}\in X(\theta)} \omega_{\boldsymbol{i}} f(g(x_{\boldsymbol{i}}))$, then

$$\int f(x)dx \approx \sum_{\boldsymbol{i}\in X(\theta)} \omega_{\boldsymbol{i}} g'(\boldsymbol{x}_{\boldsymbol{i}}) f(g(x_{\boldsymbol{i}})) = \sum_{\boldsymbol{i}\in X(\theta)} \hat{\omega}_{\boldsymbol{i}} f(\hat{x}_{\boldsymbol{i}}).$$

The transformed quadrature nodes are $\hat{x}_{\boldsymbol{i}} = g(\boldsymbol{x}_{\boldsymbol{i}})$ and the corresponding quadrature weights are $\hat{\omega}_{\boldsymbol{i}} = \omega_{\boldsymbol{i}} g'(\boldsymbol{x}_{\boldsymbol{i}})$. Similarly, if $G_\theta[f \circ g](x) \approx f(g(x))$, then

$$f(x) = f(g(g^{-1}(x))) \approx G_\theta[f \circ g](g^{-1}(x)),$$

and the sparse grids nodes associated with $f(x)$ are again $\hat{x}_{\boldsymbol{i}} = g(\boldsymbol{x}_{\boldsymbol{i}})$. Note, in the interpolation case the function basis used to approximate $f(x)$ is a composition between the standard basis (polynomials or wavelets) and $g^{-1}(x)$.

Appropriate choice of $g(x)$ can significantly accelerate convergence, but a wrong choice can severely deteriorate accuracy. As an experimental feature, Tasmanian allows for non-linear transformation of the integration/interpolation domain with $g(x)$ based on the truncated Maclaurin series of $\arcsin(x)$. Different degree of truncation can be chosen in each direction and conformal mapping can be composed with standard linear a-b transformation to obtain optimal rule over any arbitrary domain. Note: this feature will not work with unbounded rules, such as Gauss-Laguerre and Gauss-Hermite.

2.11 Alternative coefficient construction

The sparse grids approximation can be generalized as

$$G_\Theta[f](\boldsymbol{x}) = \sum_{i=j}^{n} c_j \phi_j(\boldsymbol{x}),$$

where c_j is a set of scalar or vector coefficients depending whether $f(\boldsymbol{x})$ has scalar or vector output, and ϕ_j is a set basis functions. The approximation is related to the best fit of $f(\boldsymbol{x})$ in the span of $\phi_j(\boldsymbol{x})$ with a penalty constant (e.g., Lebesgue constant). Here, for simplicity, we suppress the multi-index notation and assume linear ordering of the nodes and basis functions. In the standard SG algorithms, the n coefficients c_j are derived from n samples of $f(\boldsymbol{x}_j)$ collected at specially chosen nodes $\boldsymbol{x}_j$. The choice of $\boldsymbol{x}_j$ is performed in a way that minimizes the penalty, but it also leads to a significant drawback, i.e., the target function $f(\boldsymbol{x})$ must be evaluated at exactly the selected set of nodes. In some applications, this is either impractical or even infeasible, e.g., the domain of $f(\boldsymbol{x})$ is not a hypercube but rather a blob of some shape contained within a hypercube. In order to utilize the flexible function spaces associated with sparse grids and in order to take advantage of the advanced adaptive approximation algorithms, a different approach is needed to construct c_j from an arbitrary set of realizations of $f(\boldsymbol{x})$.

Let $\{f(\boldsymbol{s}_i)\}_{i=1}^m$ indicate m samples of $f(\boldsymbol{x})$ for an arbitrary set of sample points $\boldsymbol{s}_i$, where for simplicity we assume that $f(\boldsymbol{x})$ is scalar valued. Define the basis matrix A and data vector f

$$A = \{a_{i,j}\} \in \mathbb{R}^{m\times n}, \quad \text{where} \quad a_{i,j} = \phi_j(\boldsymbol{s}_i), \qquad \boldsymbol{f} = \{f_i\}, \quad \text{where} \quad f_i = f(\boldsymbol{s}_i).$$

Similarly, we can arrange the coefficients c_j in a vector $\boldsymbol{c}$, and we seek $\boldsymbol{c}$ such that

$$A\boldsymbol{c} = \boldsymbol{f}. \tag{47}$$

In the case of standard sparse grids construction with a nested rule, (47) has exact solution, i.e., either $\boldsymbol{c} = \boldsymbol{f}$ for global grids or $\boldsymbol{c}$ are the hierarchical coefficients of the sequence or local grids. In the case of non-nested grids, the coefficients $\boldsymbol{c}$ have a more complex nature and (47) is not satisfied for all rows, but the "solution" $\boldsymbol{c}$ is found according to the direct sum of tensors formula. In the general case, when the samples come from an arbitrary set, an exact solution cannot be found and since $m \neq n$ the system of equations is either under or over determined.

3 Random Sampling

3.1 DREAM: General algorithm

Let $\Gamma \subset \mathbb{R}^d$ and $\rho : \Gamma \to \mathbb{R}^+$ be a non-negative function with

$$\int_\Gamma \rho(\boldsymbol{x}) d\boldsymbol{x} < \infty,$$

then scaling $\rho(\boldsymbol{x})$ gives us a probability density function and the goal of the random sampling algorithm is to generate points $\{\boldsymbol{x}_i\}$ with the said distributions.

Standard Metropolis-Hastings algorithm creates a chain of samples, by iteratively proposing a new sample followed by an accept/reject test. In short, given $\boldsymbol{x}_i$, we obtain a random perturbation $\boldsymbol{g}_i$ (with distribution symmetric around $\mathbf{0}$) and we set

$$\boldsymbol{x}_{i+1} = \begin{cases} \boldsymbol{x}_i + \boldsymbol{g}_i, & \frac{\rho(\boldsymbol{x}_i+\boldsymbol{g}_i)}{\rho(\boldsymbol{x}_i)} \geq u_i, \\ \boldsymbol{x}_i & \text{othwerwise,} \end{cases}$$

where u_i is a random sample from uniform distribution over $[0, 1]$. Regardless of the initial $\boldsymbol{x}_0$, in the limit as $i \to \infty$, the distribution of $\boldsymbol{x}_i$ matches the one defined by the pdf $\rho(\boldsymbol{x})$. In practice, a finite set of $\boldsymbol{x}_i$ are computed and an initial batch of samples is discarded (a process called the burn-up).

The DiffeRential Evolution Adaptive Metropolis (DREAM) algorithm simultaneously evolves a number of chains and the probability distribution for the correction is informed by all samples in the chain. Specifically, the chain state is

$$\{\boldsymbol{x}_{1,i},\, \boldsymbol{x}_{2,i},\, \ldots,\, \boldsymbol{x}_{C,i}\},$$

where C is the total number of chains. Each chain is updated according to the accept/reject criteria

$$\boldsymbol{x}_{c,i+1} = \begin{cases} \boldsymbol{x}_{c,i} + \boldsymbol{g}_{c,i}, & \frac{\rho(\boldsymbol{x}_i+\boldsymbol{g}_{c,i})}{\rho(\boldsymbol{x}_i)} \geq u_i, \\ \boldsymbol{x}_{c,i} & \text{othwerwise.} \end{cases} \tag{48}$$

The updates are chosen as

$$\boldsymbol{g}_{c,i} = \gamma(\boldsymbol{x}_{c_1,i} - \boldsymbol{x}_{c_2,i}) + \boldsymbol{r}_{c,i}, \tag{49}$$

where c_1 and c_2 are random integers in the range $[1, C]$, γ is a jump scale constant (usually in $[0, 1]$), and $\boldsymbol{r}_{c,i}$ is a small correction sampled from a distribution that is symmetric around $\mathbf{0}$.

Compared to the standard Metropolis-Hastings method, the DREAM algorithm has several practical advantages

- Using a large number of chains allows better initial coverage of Γ, which limits the dependence on the initial guess.
- The proposal is constantly updated based on the current chain state, which accelerates convergence.
- In the single chain algorithm, once the state reaches a high-probability regions, it is very unlikely that the chain would jump out of that region and reach a second one. Thus, Metropolis-Hastings struggles when dealing with multi-modal distributions. In contrast, when DREAM uses a sufficiently large number of chains some chains will reach every high probability region.

Distribution	Domain	Density	Defining parameters
Uniform	$[a, b]$	$\frac{1}{b-a}$	a, b
Gaussian	$(-\infty, \infty)$	$\frac{1}{\sqrt{2\pi\sigma}} \exp\left(-\frac{1}{2\sigma}(x-\mu)^2\right)$	σ, μ
Truncated Gaussian	$[a, b]$	$\frac{\exp\left(-\frac{1}{2\sigma}(x-\mu)^2\right)}{\tilde{C}}$	σ, μ, a, b
Exponential	$[a, \infty)$	$\lambda \exp\left(-\lambda(x-a)\right)$	λ, a
Beta	$[a, b]$	$\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \frac{(x-a)^{\alpha-1}(b-x)^{\beta-1}}{(b-a)^{\alpha+\beta-2}}$	a, b, α, β
Gamma	$[a, \infty)$	$\frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} \exp\left(-\beta(x-a)\right)$	a, α, β

Table 4. Probability distributions included in Tasmanian. The normalization constant for the Truncated Gaussian distribution is $\tilde{C} = \left(\sqrt{2\pi\sigma} - \int_{(-\infty,a)\cup(b,\infty)} \exp\left(-\frac{1}{2\sigma}(\xi-\mu)^2\right) d\xi\right)$, and $\Gamma(\alpha)$ us the gamma funciton.

- Metropolis-Hastings can handle multiple modes if the high probability region of the update distribution is sufficiently large; however, this is seldom practical as wide spread of the updates leads to very low acceptance rate, which in turn leads to poor mixing.* DREAM largely circumvents this limitation and can handle multiple modes without sacrificing acceptance rate.
- Evolving multiple chains simultaneously allows the use of batched evaluations of $\rho(\boldsymbol{x})$, which can be performed much more efficiently than sequential evaluations.

Note that setting $\gamma = 0$ reduces the DREAM algorithm to multiple independent chains of standard Metropolis-Hastings.

3.2 Supported probability distributions

Tasmanian includes 6 probability distributions that can be used as priors in a context of Bayesian inference (e.g., see 1.2). The pdfs and the associated parameters are listed in Table 4. In addition, Tasmanian implements Gaussian likelihood of form (5), where the covariance could be diagonal with constant or non-constant diagonal entries, or a general dense matrix.

*Mixing is a numerical phenomena where multiple iterates of the same chain have identical values, which is not desirable when the chains are used for statistical analysis.

Feature	Tested	Recommended
gcc	4.8, 5, 6, 7	5 or newer
clang	3.8, 4.0, 5.0	4.0 or newer
cmake	2.8, 3.5, 3.9, 3.10	3.0.2 or newer
python	2.7, 3.5, 3.6	2.7 or newer
OpenBlas	0.2.18, 0.2.20	0.2.18 or newer
libiomp	5.0	5.0 or newer

Table 5. Tested and recommended features.

4 Installation

Under UNIX platforms (e.g., Linux and MacOSX), Tasmanian supports two build engines, `cmake` and standard GNU `make`. Under MS Windows, the code includes a batch script that compiles the library from "Developer Command Prompt" for MS Visual C++, see §4.9.

NOTE: in versions 5.x, the build engine underwent significant overhaul. Do not ignore this section.

4.1 Required and supported software

At the bare minimum, Tasmanian requires a C/C++ compiler (e.g., gcc or clang) and either cmake or GNU Make engines. Additionally, we recommend Python with NumPy package, BAsic Linear Algebra Subroutine (BLAS) implementation, and OpenMP implementation (libgomp is included with gcc and libiomp can be installed and used with clang). Optionally, Tasmanian provided acceleration using Nvidia CUDA libraries with custom kernels, as well as python matplotlib. Still in experimental phase are the Fortran interface and the MPI capabilities of the DREAM module. See Table 5 for a list of supported compilers and interpreters.

4.2 Quick Build: `install` with `cmake` backend

Unzip the archive which will create the `Tasmanian` folder, i.e., the root folder of all source code. Run the included install script, on most Linux distribution the script will run if started from the graphical environment by executing itself in a graphical terminal. However, depending on the security policies and potentially non-standard setup, the graphical command may fail.

Executing the script manually:

```
cd Tasmanian
./install <install-root-folder> <matlab-work-folder>
```

If the install root folder is omitted, i.e., the script is called with no parameters, then the installer will ask for the installation and MATLAB work folders.

- The script must be executed from the Tasmanian source root folder.
- Using absolute paths for the install and MATLAB folders is **strongly recommended**.

- If the MATLAB work folder is not provided, the MATLAB interface is disabled, for more details on the MATLAB interface see §8.
- The script will create the `Build` sub-folder, invoke `cmake`, and call the `make`, `make test`, `make install` and `make test_install` commands, followed by post install tests.
- A summary of the installation is stored in `<install-root-folder>/config/Tasmanian.log`.
- The install command support additional options, use the "`-help`" switch, see §4.2.1.

The `install` script will attempt to build the library with all option, except `MPI`, `Fortran` and custom `CUDA` kernels, which are still in experimental stage. Both static and shared libraries will be created and the `cmake` engine will attempt to find OpenMP, BLAS, and CUDA/cuBLAS. In addition interfaces will be created for all selected languages. Tasmanian supports both Python 2 and 3, and cmake will attempt to detect installed versions automatically (usually defaults is Python 2), to manually specify python interpreter use the `-python=<interpreter>`. If `cmake` fails to find any of the libraries needed by OpenMP, BLAS, or Nvidia CUDA, then the corresponding option will be automatically disabled. At the end of the configure stage, cmake will write out a list of the options.

4.2.1 Additional install options

A list of options will be displayed by

```
./install -help
```

Options are included to disable OpenMP, BLAS, CUDA/CuBLAS, Python, shared or static libraries. Note that Python requires shared libraries and at least one type of libraries has to be build. In addition:

- `-notest` disables the included automatic test, which may be useful if the tests fail for a reason other than a problem with the build, e.g., non-standard implementation of Python;
- `-debug` enables the debug build flag, as opposed to the default `Release`;
- `-verbose` forces a large amount of output;
- `-make-j` passes the `-j` options to make to enable parallel compilation;
- `-noinstall` do not call `make install` or the post install test;
- `-nobashrc` skips the `.bashrc` setup at the end;
- `-holdend` wait for a enter at the end.

More fine grained control can be gained by invoking a more advanced `cmake` command, see §4.5 for details.

4.3 Installation folder structure

The install command will create a number of sub-folders in the `<install-root-folder>`:

`bin` contains the `tasgrid` and `tasdream` executables

`lib` contains the shared and/or static libraries and the python interface in pythonX.Y subfolder, see `python` below

`include` contains the C++ headers, ending in `.hpp`, and the C header `TasmanianSparseGrid.h`

`examples` contains the Python and Fortran examples (if either is enabled) and the C++ examples source code with the corresponding `CMakeLists.txt`. The three sparse grids examples execute identical operations, but are written in the corresponding languages to demonstrate proper Tasmanian calls.

`matlab` contains the files needed by the MATLAB interface (if MATLAB is enabled). To enable the Tasmanian from within the MATLAB environment, you must use the command:

```
addpath('<install-root-folder>/matlab/')
```

where you must substitute the proper root folder. The `make install` command will write out the proper string to both console output and `<install-root-folder>/config/Tasmanian.log`.

`python` is a symlink to `lib/pythonX.Y` contains the files needed by the Python interface (if Python is enabled). To import the Tasmanian module in Python, you need the commands

```
import sys
sys.path.append("<install-root-folder>/python")
import TasmanianSG
```

where you must substitute the proper root folder. The `make install` command will write out the proper string to both console output and `<install-root-folder>/config/Tasmanian.log`.

`config` contains two `bash` scripts that can be sources to gain access to Tasmanian capabilities. The folder also contains the `cmake` include scripts. The `TasmanianENVsetup.sh` script sets the `PATH` and `LD_LIBRARY_PATH` environmental variables, the `TasmanianDEVsetup.sh` script sets the development environmental variables `C_INCLUDE_PATH`, `CPLUS_INCLUDE_PATH` and `LIBRARY_PATH`. The included `cmake` scripts allows the import the six build targets and all dependencies: two executables, two shared and two static libraries (if enabled). The `make install` command will list the available targets. Note that the DEV script is not needed if using the `cmake` include file.

4.4 Quick Build: `make`

Unzip the archive which will create the `Tasmanian` folder, i.e., the root folder of all source code. As in the previous versions of Tasmanian, we support the simple GNU make engine that can be invoked by the make command:

```
cd Tasmanian
make
make test
```

The command will likely succeed on most Unix systems and will create both static and shared libraries. In addition, OpenMP will be enabled on Linux platforms. BLAS, CUDA and MPI are disabled by default, but can be enabled by editing `Config/Makefile.in`, see §4.6.

The `make` command will copy the libraries and executables to `Tasmanian` source root folder:

tasgrid	tasdream
libtasmaniansparsegrid.a	libtasmaniandream.a
libtasmaniansparsegrid.so	libtasmaniandream.so

The Python module `TasmanianSG.py` and the Python and C++ examples will also be copied.

The test target will call

```
./tasgrid -test
./tasdream -test
./testTSG.py
```

If python is not set on the system (see below), then the `testTSG.py` will fail. While python is not required to use Tasmanian through another interface, the python test checks important library functionality, such as basic disc I/O and various error catching options.

The Python interface in Tasmanian is compatible with both Python 2 and 3, by default the scripts call `/usr/bin/env python`. Depending on your system, this can be either python 2, python 3, or missing. To switch the default python env command from `python` to `python3`:

```
make python3
```

Alternatively, you can manually change the top of the python scripts to call any desired python version.

Header files for C/C++ will be copied to `Tasmanian/include`, thus only one folder must be included for any of the Tasmanian modules and headers.

The files needed for the MATLAB interface are located in `Tasmanian/InterfaceMATLAB`. You can now use the command

```
make matlab
```

that will automatically edit the `tsgGetPaths.m` file and set the path to `tasgrid` executable and the MATLAB work folder to `Tasmanian/tsgMatlabWorkFolder/`. You can also skip this and manually edit the the MATLAB interface, see section §8.

```
make fortran
```

will compile the Fortran library, provided gfortran can be found on the system.

The simple make engine also has `examples` target:

```
make examples
```

which will compile the two C++ examples. This make command will display the proper compiler command to link to the Tasmanian libraries. The sparse grids examples have three versions, C++, python and MATLAB. The examples are identical and contrast the use of the three Tasmanian interfaces.

The full set of simple make commands:

```
make
make test
make matlab      (optional: sets work folder Tasmanian/tsgMatlabWorkFolder/)
make python3     (optional: sets #!/usr/bin/env python3)
make fortran     (optional: compile libtasmanianfortran.a/so)
make examples
./example_sparse_grids
./example_sparse_grids.py   (optional: if python is enabled)
./fortester
./example_dream
make clean                  (optional: restart the build process)
```

Note: on most systems you can use `make -j` to enable parallel compilation.

4.5 Advanced build options: `cmake`

In this section, we present a list of all `cmake` compile options that allow selecting of individual features and specifying third-party libraries. Note that we **strongly recommend using out-of-source build** as in-source call to `cmake` will interfere with the basic make engine and potentially fail. The default build command is given below, i.e., the primary user options and the corresponding default values:

```
cmake \
 -D CMAKE_BUILD_TYPE:STRING=Debug \
 -D CMAKE_INSTALL_PREFIX:PATH=<install-prefix-with-full-path> \
 -D USE_XSDK_DEFAULTS:BOOL=OFF \
 -D Tasmanian_STRICT_OPTIONS:BOOL=OFF \
 -D Tasmanian_ENABLE_BLAS:BOOL=ON \
 -D Tasmanian_ENABLE_PYTHON:BOOL=ON \
 -D Tasmanian_ENABLE_MATLAB:BOOL=ON \
 -D Tasmanian_MATLAB_WORK_FOLDER:PATH=<matlab-work-folder-path> \
 -D Tasmanian_ENABLE_OPENMP:BOOL=ON \
 -D Tasmanian_ENABLE_FORTRAN:BOOL=OFF \
 -D Tasmanian_ENABLE_CUBLAS:BOOL=ON \
 -D Tasmanian_ENABLE_CUDA:BOOL=OFF \
 -D Tasmanian_ENABLE_MPI:BOOL=OFF \
 -D Tasmanian_SHARED_LIBRARY:BOOL=ON \
 -D Tasmanian_STATIC_LIBRARY:BOOL=ON \
 <path-to>/Tasmanian/
```

Standard `cmake` commands are accepted. In addition to `CMAKE_BUILD_TYPE` and `CMAKE_INSTALL_PREFIX`, the engine recognizes:

`CMAKE_C_COMPILER`	`CMAKE_CXX_COMPILER`
`CMAKE_CXX_FLAGS`	`BUILD_SHARED_LIBS`

where `BUILD_SHARED_LIBS` is identical to selecting only one of shared/static libs.

Options will be auto-adjusted if any of the selected features cannot be automatically discovered by cmake, e.g., if cmake `find_package(BLAS)` fails then BLAS will be switched off even if it has been selected.

However, if Tasmanian_STRICT_OPTIONS is enabled, then the project will refuse to build unless all selected options are satisfied, which is fail-safe mechanism to ensure consistency with the user preferences.

The OpenMP option will search for an OpenMP flag for the selected compiler and attempt to set it. The automatic search can be bypassed by manually specifying the OpenMP_CXX_FLAGS variable. In addition, Tasmanian tests use OpenMP and as many threads as set by default in the environment, which can be adjusted with the option Tasmanian_TESTS_OMP_NUM_THREADS, i.e., the tests will be limited to the number of threads specified in by the variable. Note: Tasmanian_TESTS_OMP_NUM_THREADS does not affect the behavior of the library outside of testing.

```
cmake \
 -D Tasmanian_ENABLE_OPENMP:BOOL=ON \
 -D OpenMP_CXX\_FLAGS:STRING=<flags-used-by-openmp> \
 -D Tasmanian_TESTS_OMP_NUM_THREADS:INTEGER=<number-of-threads-for-testing> \
 <path-to>/Tasmanian/
```

Shared and static libraries can be compiled, if static libraries are enabled the executables will be compiled statically, which means that the MATLAB and cli interfaces can be used without modification to the system library search path. The python interface requires the dynamic libraries.

The Python option will install TasmanianSG.py module, the example and the test scripts. The option uses the find_package(Python) module within cmake which will search for any installed python version. Note that once installed, TasmanianSG.py will work with either Python 2 or Python 3, but only one will be used for testing during installation. The PYTHON_EXECUTABLE option can be used to specify a specific version of python. In addition, the python module also requires the shared libraries.

```
cmake \
 -D Tasmanian_ENABLE_PYTHON:BOOL=ON \
 -D PYTHON_EXECUTABLE:PATH=<path-to-python> \
 <path-to>/Tasmanian/
```

The MATLAB option will copy the .m files into the matlab install sub-folder and automatically adjust the tsgGetPaths.m file with the paths to the tasgrid executable and Tasmanian_MATLAB_WORK_FOLDER. The MATLAB interface needs that addpath command executed in the MATLAB environment, see installed Tasmanian.log file for the correct command. In order for the interface to work properly, it is strongly recommended to use full system paths. Note that if static libraries are not compiled, then the dynamic library will have to be in the system LD_LIBRARY_PATH in order for the MATLAB interface to work.

```
cmake \
 -D Tasmanian_ENABLE_MATLAB:BOOL=ON \
 -D Tasmanian_MATLAB_WORK_FOLDER:PATH=<path-to-work-folder> \
 <path-to>/Tasmanian/
```

The BLAS option will automatically search for a BLAS implementation and auto-disable the option if no implementation is found. Specific library can be selected with the BLAS_LIBRARIES option, which will bypass find_package(BLAS), e.g.,

```
 cmake \
  -D Tasmanian_ENABLE_BLAS:BOOL=ON \
  -D BLAS_LIBRARIES:PATH=<path-to-desired-blas-implementation> \
  <path-to>/Tasmanian/
```

The cuBLAS option will automatically search for a CUDA implementation and auto-disable the option if no implementation is found. Optionally, search path can be specified with the CUDA_TOOLKIT_ROOT_DIR variable, e.g.,

```
cmake \
  -D Tasmanian_ENABLE_CUBLAS:BOOL=ON \
  -D CUDA_TOOLKIT_ROOT_DIR:PATH=<path-to-cuda> \
  <path-to>/Tasmanian/
```

Similar to BLAS, the find_package(CUDA) can be bypassed with manual selection of the required libraries, at the minimum CUDA_CUBLAS_LIBRARIES is required, and additional CUDA options may also be needed depending on the user system.

```
   cmake \
    -D Tasmanian_ENABLE_CUBLAS:BOOL=ON \
    -D CUDA_CUBLAS_LIBRARIES=<cuda-cublas-library> \
    -D CUDA_INCLUDE_DIRS:PATH=<path-to-cuda-include> \
    -D CUDA_LIBRARIES:STRING=<generic-cuda-libs-such-as-cudart> \
    -D CUDA_cusparse_LIBRARY:STRING=<cuda-cusparse-library> \
    <path-to>/Tasmanian/
```

Note: see the remark about Tasmanian_TESTS_GPU_ID below.

The CUDA option also searches for the cuda implementation, but instead of just linking to the cuBlas libraries it also compiles custom CUDA kernels. While somewhat experimental, the kernels can significantly improve performance. Note, when using the CUDA options, there is no way to bypass the find_package(CUDA), which requires supported version of cmake and the C++ compiler.

In principle, it is possible to compile the library with CUDA but without CUBLAS. However, this is supported only for testing and research purposes and should not be used in any form of a "production" install. Also note that disabling CUBLAS will have an overall negative impact on the CUDA performance, which is another reason to avoid using CUDA just by itself.

Enabling either CUBLAS or CUDA options will also enable tests that require a compatible Nvidia GPU. By default, Tasmanian will run tests on all available GPUs and the rests will requires about 512MB - 1GB or GPU RAM. The option Tasmanian_TESTS_GPU_ID allows specifying the GPU for testing, where the ID corresponds to the index given by CUDA. The command tasgrid -v will list all GPUs discovered on the system with the corresponding ID. The Tasmanian_TESTS_GPU_ID options is useful on systems with one weak GPU dedicated to system graphics, while other powerful GPUs are reserved for computing. Note: this options has not effect on the internals of the library, only the tests are restricted to the specified GPU, the library can still use any device visible to the CUDA engine.

```
   cmake \
    -D Tasmanian_ENABLE_CUBLAS:BOOL=ON \
    -D Tasmanian_ENABLE_CUDA:BOOL=ON \
    -D Tasmanian_TESTS_GPU_ID=<gpu-id-for-testing> \
    <path-to>/Tasmanian/
```

The MPI option will automatically search for an MPI implementation, which is used only by the DREAM module and likewise is still in an experimental stage. Manually specifying MPI_CXX_LIBRARIES will bypass the automatic find_package(MPI) and other options may also be needed depending on the system

```
cmake \
 -D Tasmanian_ENABLE_MPI:BOOL=ON \
 -D MPI_CXX_LIBRARIES:STRING=<mpi-libraries> \
 -D MPI_CXX_INCLUDE_PATH:PATH=<path-to-mpi-headers> \
 -D MPI_COMPILE_FLAGS:STRING=<mpi-compile-flags> \
 -D MPI_LINK_FLAGS:STRING=<mpi-link-flags> \
 <path-to>/Tasmanian/
```

The C++ 2011 support is not required, but Tasmanian is fully compatible with the standard. The advanced options, MPI and CuBLAS/CUDA, will automatically enable the CXX_STANDARD 11 options. The `Tasmanian_ENABLE_CXX_11` option will manually set the standard and bypass the automatic process, but note that recent MPI or CUDA versions may require the standard and disabling C++ 2011 may lead to build failure. Furthermore, `Tasmanian_ENABLE_CXX_11` only works on `cmake 3.x`, for older versions it has to be manually set in compile flags.

```
cmake \
 -D Tasmanian_ENABLE_CXX_11:BOOL=ON \
 <path-to>/Tasmanian/
```

The XSDK option refers to the xSDK project and the software development policies promoted by the community. For example, when xSDK compatibility is enabled, Tasmanian_STRICT_OPTIONS will be set to on and the `Tasmanian_ENABLE_***` options will be replaced by corresponding naming convention consistent with the xSDK policies. In addition, the xSDK policies mandate that the by default the library does not produce error messages to the command line. Visit the xSDK web-page for more information: https://xsdk.info/policies/.

Additional options are also available for special situations to ease the build process in exotic environments:

- `Tasmanian_EXTRA_CXX_FLAGS` will append additional flags to the project;
- `Tasmanian_EXTRA_INCLUDE_DIRS:STRING` option will specify additional include directories;
- `Tasmanian_EXTRA_LIBRARIES:STRING` option will append additional libraries to the default selected by Tasmanian;
- `Tasmanian_EXTRA_LINK_DIRS:STRING` option will append additional link folders to the default selected by Tasmanian.

The extra options allow for find grained control of the build environment similar to the `Makefile.in` capability, where almost anything can be written without concern of some automated system overwriting the commands.

For example,

```
cmake ... \
 -D Tasmanian_ENABLE_OPENMP:BOOL=OFF \
 -D Tasmanian_ENABLE_BLAS:BOOL=ON \
 -D BLAS_LIBRARIES:PATH=/opt/acml/gfortran64_mp/lib/acml_mp.a \
 -D Tasmanian_EXTRA_LIBRARIES:STRING="gfortran\;gomp\;pthread" \
 <path-to>/Tasmanian/
```

will disable OpenMP within the Tasmanian library, but will link to OpenMP implementation of ACML (i.e., AMD BLAS) which cannot be found automatically and requires the `libgomp` and `libgfortran`. **Note**: the additional options should not be needed under regular circumstances.

The full set of commands to use in the `cmake` module are:

```
cmake ... \
 <path-to>/Tasmanian/
make
make test
make install
make test_install
```

The `make` command can also accept the `-j` option to enable parallel compilation. For more information on the post install testing, see §4.7.

Development options are designed to be used only by the Tasmanian development team. The options are:

- `Tasmanian_DEVELOPMENT_BACKWARDS` sets the *.in.* files and overwrites the corresponding non-in files in the `source` folder! This changes the source and it is used to keep one source three compatible with both `cmake` and GNU make build engines.
- `Tasmanian_ENABLE_DEVELOPMENT_DEFAULTS` build the examples in stage 1 of the build process, i.e., in the build folder and enables error checking in the compiler. This makes editing the examples easier.

4.6 Advanced build options: simple `make`

The advanced capabilities of BLAS, cuBLAS and MPI can be enabled in the simple `make` build engine by manually editing `Tasmanian/Config/Makefile.in` and `Tasmanian/Config/tasmanianConfig.hpp`. See the included comments in the files, but also note that the Tasmanian team does not provide any official support for advanced options in the simple make engine. If you want advanced options, **use cmake** or the included **install** script (which wraps around cmake).

4.7 Testing

The testing commands used by the Tasmanian are

```
./tasgrid -test
./tasdream -test
./testTSG.py            (optional: only if python is enabled)
./fortester             (optional: only if Fortran is enabled)
./test_post_install.sh  (optional: only if using cmake)
```

The first three options are called by `make test` in both `cmake` and GNU make engines. The last command will check if `tasgrid` can be called form the install folder, compile and run the C++ examples using `cmake`, run the Python example (if python is enabled) and compile a very simple C++ example to link to the Tasmanian libraries.

Both `tasgream` and `tasgrid` tests rely on random number generation for error checking (especially the DREAM module). Thus, the outcome of many of the tests depends on the random seed. Currently, the

executables (not the libraries) have a hard-coded random seed, which is used for testing, but this may fail depending on the random number generator used by the compiler. To use a different random seed, use the commands

```
./tasgrid -test random
./tasdream -test random
```

to force the executable to use a random seed based on `time()`.

4.8 Known Build Problems

On MacOSX, OpenMP is not fully supported by all compilers and sometimes does not scale well. The simple `make` engine disables OpenMP on all non-Linux platforms. The `cmake` engine enables it by default, but if poor performance is encountered, consider disabling OpenMP.

4.9 Build on Windows using Mircosoft Visual C++ 2015

Starting with version 3.1, TASMANIAN can be compiled using MS Visual C++ with either `cmake` or the included `WindowsMake.bat` script. See the included `WindowsREADME.txt` file.

All of the advanced `cmake` options listed above should work well under Windows; however, cuBLAS and BLAS have not been tested yet. It is best to use the `cmake` graphical interface to select source and binary folders and enable/disable different options. The libraries and executables can be compiled using the `Developer Command Prompt`

```
msbuild.exe ALL_BUILD.vcxproj
msbuild.exe INSTALL.vcxproj
```

The install command will generate the appropriate folders and files identical to `make install` on Unix, with the exception that the `.dll` files will be stored in the `bin` folder.

The equivalent of the simple `make` command is the `WindowsMake.bat` script. The commands are

```
WindowsMake.bat
WindowsMake.bat test
```

The clean options is `WindowsMake.bat clean`.

5 Library: Tasmanian Sparse Grids

All of the sparse grids functionality is included in the `libtasmaniansparsegrid` C++ library. Code that interfaces with the library should include the `TasmanianSparseGrid.hpp`, which introduces the `TasGrid` namespace and the definition of the `TasmanianSparseGrid` class.

WARNING: The code performs little sanity check on the validity of input. Wrong input would result in incorrect output and most likely a crash.

5.1 Constructor TasmanianSparseGrid()

```
TasmanianSparseGrid();
```

This is the only class constructor (called by default), makes an empty grid. Before any operations can be performed, a grid has to be made with one of the functions:

- `makeGlobalGrid()`
- `makeSequenceGrid()`
- `makeLocalPolynomialGrid()`
- `makeWaveletGrid()`

Alternatively the grid can be read from a stream/file using the `read()` functions (in order to read a grid, it must first be written to the file with the `write()` function). The `getVersion()` and `getLicense()` functions can be called at any time. Calling any other function will result in a segfault.

5.2 Destructor TasmanianSparseGrid()

```
~TasmanianSparseGrid();
```

This is the destructor that releases any memory used by the class.

5.3 getVersion*()

```
static const char* getVersion() const;
static int getVersionMajor() const;
static int getVersionMinor() const;
```

Returns the version of the library, either as a simple hard-coded string or integers indicating the major and minor parts.

5.4 getLicense()

```
static const char* getLicense() const;
```

Returns a short string indicating the license of the library. This is a simple hard-coded string.

5.5 isOpenMPEnabled()

```
static bool isOpenMPEnabled() const;
```

Hard-coded booleans indicating whether the library is build with OpenMP support. Note that OpenMP is not considered "acceleration" in the context of Tasmanian, when enabled on compile time, OpenMP will be used at all stages of the sparse grids, i.e., not just evaluations.

5.6 *Log()

```
void setErrorLog(std::ostream *os);
void disableLog();
```

By default, an instance of Tasmanian Sparse Grids will write errors to the `cerr` stream. However, all error messages can be send to any `std::ostream` using the above command. If the log is set to a `NULL` stream, then all errors will be silent. Note: if the `XSDK_DEFAULTS` options is enabled in `cmake`, then the default behavior is to produce no errors.

5.7 makeGlobalGrid()

```
void makeGlobalGrid( int dimensions,
                     int outputs,
                     int depth,
                     TypeDepth type,
                     TypeOneDRule rule,
                     const int *anisotropic_weights = 0,
                     double alpha = 0,
                     double beta = 0,
                     const char *custom_rule_filename = 0,
                     const int *level_limits = 0) );
```

This function creates a sparse grid induced by one of the global quadrature and interpolation rules. See Section 2.1 for a full list of the rules. The parameters are described as follows:

`dimensions` is a positive integer specifying the dimension of the grid. There is no hard restriction on how big the dimension can be, however, for large dimensions, the number of points of the sparse grid grows fast (this is called the curse of dimensionality) and hence the grid may require prohibitive amount of memory.

`outputs` is a non-negative integer specifying the number of outputs for the function that would be interpolated. If `outputs` is zero, then the grid can only generate quadrature and interpolation weights, i.e., problems (3) and (2). There is no hard restriction on how many outputs can be handled, however, note that the code requires at least `outputs` $\times$ `number of points` in storage and hence for large number of `outputs` memory management may have adverse effect on performance.

`depth` is a non-negative (or strictly positive) integer that controls the density of abscissa points. This is the L parameter in tensor selection (20) - (28). There is no hard restriction on how big `depth` can be, however, it has direct effect on the number of points and hence performance and memory requirements.

`type` is an enumerated type indicating the tensor selection strategy.

- `type_level`: see (20)
- `type_curved`: see (21)
- `type_hyperbolic`: see (22)
- `type_iptotal`: see (23)
- `type_ipcurved`: see (24)
- `type_iphyperbolic`: see (25)
- `type_qptotal`: see (26)
- `type_qpcurved`: see (27)
- `type_qphyperbolic`: see (28)
- `type_tensor`: creates a full (not sparse) tensor grid in the notation of §2.1, $G = \bigotimes_{k=1}^{d} \mathcal{U}^{m(L\cdot\xi_k)}$.
- `type_iptensor`: creates the smallest full tensor grid that will interpolate exactly all polynomials in $span\{\boldsymbol{x}^{\boldsymbol{\nu}} : \boldsymbol{\nu} \leq L \cdot \boldsymbol{\xi}\}$
- `type_iptensor`: creates the smallest full tensor grid that will integrate exactly all polynomials in $span\{\boldsymbol{x}^{\boldsymbol{\nu}} : \boldsymbol{\nu} \leq L \cdot \boldsymbol{\xi}\}$

`rule` is an enumerated type from any of the global rules in Tables 1, 2 and 3. Those are:

`rule_chebyshev`
`rule_chebyshevodd`
`rule_clenshawcurtis`
`rule_clenshawcurtis0`
`rule_fejer2`
`rule_rleja`
`rule_rlejadouble2`
`rule_rlejadouble4`
`rule_rlejaodd`
`rule_rlejashifted`
`rule_rlejashiftedeven`
`rule_leja`
`rule_lejaodd`
`rule_maxlebesgue`
`rule_maxlebesgueodd`
`rule_minlebesgue`
`rule_minlebesgueodd`
`rule_mindelta`
`rule_mindeltaodd`
`rule_gausslegendre`
`rule_gausslegendreodd`
`rule_gausspatterson`
`rule_gausschebyshev1`
`rule_gausschebyshev1odd`
`rule_gausschebyshev2`
`rule_gausschebyshev2odd`
`rule_gaussgegenbauer`
`rule_gaussgegenbauerodd`
`rule_gaussjacobi`
`rule_gaussjacobiodd`
`rule_gausslaguerre`
`rule_gausslaguerreodd`
`rule_gausshermite`
`rule_gausshermiteodd`
`rule_customtabulated`

Note: the custom tabulated rule requires `custom_rule_file`, see below as well as Appendix 10.

`anisotropic` (`anisotropic_weights`) is either `NULL` or an array of integers `dimensions` or $2\times$ `dimensions`, specifying the $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ anisotropic weights. If the pointer is `NULL`, then TASMANIAN assumes $\boldsymbol{\xi} = \mathbf{1}$ and $\boldsymbol{\eta} = \mathbf{0}$, otherwise, the entries 0 to `dimension`-1 of the vector specify the components in $\boldsymbol{\xi}$ and the following `dimension` to $2\times$ `dimension` entries specifies $\boldsymbol{\eta}$ (if `type` is not set to one of the "*curved" ones, then the second set of entries is not used). Note that in the literature, the weights are assumed to be real numbers, however, Tasmanian assumes that the weights are normalized rational numbers, i.e., the library uses $\boldsymbol{\xi} = \boldsymbol{\xi}/\max_k \xi_k$ and $\boldsymbol{\eta} = \boldsymbol{\eta}/\max_k \xi_k$ (no typo here $\max_k \xi_k$ is used in both cases).

`alpha` specifies the α parameter of $\rho(x)$, this is used only if `rule` requires the α parameter. See Table 2.

`beta` specifies the β parameter of $\rho(x)$, this is used only if `rule` requires the β parameter. See Table 2.

`custom_rule_file` is either `NULL` or the path to a file describing a custom rule. Custom rules are described via tables provided in a text file format. See Appendix 10 for more information about the file format of the custom file.

`level_limits` is either `NULL` or an array of size `dimensions` that indicates a limit for the grid level in a given direction. If limits are specified, the no points will be added to the grid beyond the given level, e.g., Clenshaw-Curtis rule of level 1 has 3 points, if `leve_limit` for dimensions 0 is set to 1, then all grid points will align with those 3 points regardless of the coordinates of other dimensions. To inddicate no limit for a given direction, either use a very large limit or set `level_limits` to -1.

5.8 makeSequenceGrid()

```
void makeGlobalGrid( int dimensions,
                     int outputs,
                     int depth,
                     TypeDepth type,
                     TypeOneDRule rule,
                     const int *anisotropic_weights = 0,
                     const int *level_limits = 0 );
```

Creates a global grid using the representation described in section 2.3. The `rule` is restricted to one of the nested rules with growth $m(l) = l + 1$, namely:

`rule_rleja`	`rule_leja`	`rule_minlebesgue`
`rule_rlejashifted`	`rule_maxlebesgue`	`rule_mindelta`

5.9 makeLocalPolynomialGrid()

```
void makeLocalPolynomialGrid( int dimensions,
                              int outputs,
                              int depth,
                              int order,
                              TypeOneDRule rule = rule_localp,
                              const int *level_limits = 0 );
```

Creates a grid based on one of the local hierarchical piece-wise polynomial rules described in section 2.6. Local grids can be used for integration, however, in many cases, this would result in points associated with zero weights.

`dimensions` same as `makeGlobalGrid()`

`outputs` same as `makeGlobalGrid()`, however, due to the non-trivial form of the surplus coefficients s_j, large number of outputs comes with bigger computational cost in addition to the larger storage cost of more than $2 \times$ `outputs` $\times$ `number of points`.

`depth` is a positive integer that specifies the initial number of levels for the grid, namely the L in (40).

`order` is an integer no smaller than -1, which specifies the largest order of polynomial to be used (i.e., the p parameter). If `order` is set to -1, the largest possible order would be selected automatically "on the fly".

`rule` is specifies one of the three local polynomial rules `rule_localp`, `rule_semilocalp`, `rule_localp0`.

`level_limits` same as `makeGlobalGrid()`

5.10 makeWaveletGrid()

```
void makeWaveletGrid( int dimensions,
                      int outputs,
                      int depth,
                      int order = 1,
                      const int *level_limits = 0 );
```

Creates a grid based on local hierarchical wavelet basis, see 2.9.

`dimensions` same as in `makeGlobalGrid()` and `makeLocalPolynomialGrid()`

`outputs` same as in `makeLocalPolynomialGrid()`

`depth` same as in `makeLocalPolynomialGrid()`

`order` an integer equal to either 1 or 3.

5.11 makeFullTensorGrid()

Since Tasmanian version 3.0, this function is removed. In order to create a full tensor grid, use function `makeGlobalGrid()` with `type` set to tensor.

5.12 recycle***Grid()

Those functions were removed in Tasmanian version 3.0, see the `update***Grid()` functions, but note that those are not the same as the old functions.

5.13 updateGlobalGridGrid()

```
void updateGlobalGrid( int depth,
                       TypeDepth type,
                       const int *anisotropic_weights = 0 );
```

The inputs a the same as in `makeGlobalGrid()`, thus function should only be called for a grid with a nested rules (i.e., among the non-Gauss rules only `rule_chebyshev` is non-nested, among the Gauss rules only `rule_gausspatterson` is nested). If the grid has no outputs or no values have been loaded, then this function is equivalent to calling `makeGlobalGrid()` with the new `depth`, `type` and `anisotropic_weights` but using the old `dimensions`, `outputs` and `rule`. If values have been loaded, then a new tensor index set Θ_{new} is created according to the formula specified by `type` and the new index set is added to the old index set. This corresponds to refinement with user specified `depth` and `anisotropic_weights`.

5.14 updateSequenceGrid()

```
void updateSequenceGrid( int depth,
                       TypeDepth type,
                       const int *anisotropic_weights = 0 );
```

Same as `updateGlobalGrid()`, but called for a sequence grid.

5.15 copyGrid()

```
void copyGrid(const TasmanianSparseGrid *source);
```

Copy the source grid into the current one. Points, weights, dimensions, domain transforms and active refinement are copied. Acceleration options are not copied, i.e., the new grid falls to the default acceleration mode.

5.16 write()

```
void write(std::ofstream &ofs, bool binary = false) const;
```

Writes out the grid in either text or binary format to the `ofstream`.

5.17 read()

```
bool read(std::ifstream &ifs, bool binary = false);
```

Reads a grid that has already been written to the stream. The function returns `TRUE` if the reading was successful or `FALSE` if errors with the file format were encountered. The function will write error information to `std::cerr` stream. When usinjg `ifstream` the binary or ascii format has to be specified.

5.18 write()

```
void write(const char *filename, bool binary = false) const;
```

Opens a file with `filename` and calls `void write( std::ofstream &ofs, binary ) const;` with the associated stream. In the end, the file is closed.

5.19 read()

```
bool read( const char* filename );
```

Opens a file with `filename` and calls `bool read( std::ifstream &ifs ) const;` with the associated stream. In the end, the file is closed. This function automatically differentiates between binary and text format.

5.20 setTransformAB()

```
void setTransformAB( const double *a,
                     const double *b );
```

Since Tasmanian version 3.0, this function is renamed to `setDomainTransform()`.

5.21 setDomainTransform()

```
void setDomainTransform( const double a[],
                         const double b[] );
```

By default integration and interpolation are performed on a canonical interval $[-1, 1]$ (with the exception of a few Gauss rules descried in Table 2). Optionally, the library can transform the canonical interval into a custom one defined by the a and b parameters for every direction. The transformation is applied as a post-processing step to the abscissas and weights.

- `a` is an array of real numbers of size `getNumDimensions()` that defines the a_k parameter associated with every direction.
- `b` is an array of real numbers of size `getNumDimensions()` that defines the b_k parameter associated with every direction.

5.22 isSetDomainTransform()

```
bool isSetDomainTransform() const;
```

Returns `True` if `setDomainTransform()` has been called since the last `make***Grid()`, `False` if the grid is set to the default canonical domain.

5.23 clearTransformAB()

```
void clearTransformAB();
```

Since Tasmanian version 3.0, this function is renamed to `clearDomainTransform()`.

5.24 clearTransformAB()

```
void clearDomainTransform();
```

Removed the transform set with `setDomainTransform()` and the points are no longer transformed during calls to `get***Points()` functions.

5.25 getTransformAB()

```
void getTransformAB( double* &a,
                     double* &b ) const;
```

Since Tasmanian version 3.0, this function is replaced by `getDomainTransform()`, however, note that in the new function **a** and **b** cannnot be `NULL`, they have to be pre-allocated.

5.26 getDomainTransform()

```
void getTransformAB( double a[],
                     double b[] ) const;
```

Returns the transform parameters.

- `a` the first `getNumDimensions()` entries of `a` are overwritten with the a_k parameters of the transform.
- `b` the first `getNumDimensions()` entries of `b` are overwritten with the b_k parameters of the transform.

5.27 getNumDimensions()

```
int getNumDimensions() const;
```

Returns the value of the `dimension` parameter used by the `make***Glid()` function call.

5.28 getNumOutputs()

```
int getNumOutputs() const;
```

Returns the value of the `outputs` parameter used by the `make***Glid()` function call.

5.29 getOneDRule()

```
TypeOneDRule getOneDRule() const;
```

Returns the value of the `rule` parameter in the `make***Glid()` function call, for a wavelet grids this returns `rule_wavelet`.

5.30 getOneDRuleDescription()

```
const char *getOneDRuleDescription() const;
```

Since Tasmanian version 3.0, this function is removed. If this functionality if desired, then use

```
const char* s = TasGrid::OneDimensionalMeta::getHumanString( grid->getRule() );
```

where `grid` is the active instance of the `TasmanianSparseGrid` class.

5.31 getCustomRuleDescription()

```
const char *getCustomRuleDescription() const;
```

Returns the custom rule description string, see Appendix 10. If **rule** was not set to `rule_customtabulated`, then this function will return `NULL`.

5.32 getAlpha()/getBeta()

```
double getAlpha() const;
double getBeta() const;
```

Returns the `alpha` and `beta` parameters used in the call to `makeGlobalGrid()`. For all other grids, these functions return 0.

5.33 getOrder()

```
int getOrder() const;
```

Returns the `order` parameter used in the call to `makeLocalPolynomialGrid()` or `makeWaveletGrid()`, for global and sequence grids this function returns -1.

5.34 getNum***()

```
int getNumLoaded() const;
int getNumNeeded() const;
int getNumPoints() const;
```

Returns the number of points. The loaded points are ones that have already been associated with values via the `loadNeededPoints()` function. Right after the call to `make***Gird()` the needed points are all the points in the grid, otherwise the needed points are those generated by the refinement procedures. If no points have been loaded, then `getNumPoints()` returns the same as `getNumNeeded()`, otherwise, `getNumPoints()` returns the same as `getNumLoaded()`.

Note: if a grid is created with zero `outputs`, then `getNumNeeded()` always returns 0 and `getNumPoints()` returns the same as `getNumLoaded()`, i.e., no points are needed and all points are considered loaded.

Note: as compared to the interface of Tasmanian version 2.0, `getNumPoints()` is the same, and `getNumNeeded()` is just a renamed version of `getNumNeededPoints()`.

5.35 get***Points()

```
double* getLoadedPoints() const;
double* getNeededPoints() const;
double* getPoints() const;
void getLoadedPoints(double *x) const;
void getNeededPoints(double *x) const;
void getPoints(double *x) const;
```

If no argument is given, returns an array of length `getNumDimensions()` $\times$ `getNum***()` of values that represent the points of the grid. If `x` is specified, it must be at least of size `getNumDimensions()` $\times$ `getNum***()`. The number of points corresponds to the above `getNum***()` functions. The first point is located in the first `getNumDimensions()` number of entries, the second point is located in the second `getNumDimensions()` number of entries, and so on.

Note: as compared to the interface in version 2.0, `getPoints()` and `getNeededPoints()` are the same, albeit with different syntax.

5.36 getWeights()

```
void getWeights( double* &weights ) const;
```

Since version 3.0, this function has been renamed to `getQuadratureWeights()`.

5.37 getQuadratureWeights()

```
double* getQuadratureWeights() const;
void getQuadratureWeights(double weights[]) const;
```

If no argumenta are given to the funciton, it returns an array of size `getNumPoints()` of the quadrature weights associated with the points. If `x` is specified, then it must be at least as big as `getNumPoints()`. The first weight is associated with the first point returned by `getPoints()`, the second weight is associated with the second point and so on.

5.38 getInterpolantWeights()

```
void getInterpolantWeights( const double x[],
                            double* &weights ) const;
```

Since version 3.0, this function has been renamed to `getInterpolationWeights()`.

5.39 getInterpolationWeights()

```
double* getInterpolationWeights( const double x[] ) const;
void getInterpolationWeights(const double x[], double weights[]) const;
```

Returns the interpolantion weights associated with the point **x**, as in equation (2). For global and sequence grids with nested rules this function returns the multivariate Legendre polynomials evaluated at point **x**. For global grids with non-nested rules, this returns a linear combination of tensors of Legendre polynomials (note that non-nested grids do not generate interpolants). For all grids other than Global, computing the `getInterpolationWeights()` is very expensive and should be avoided (if possible).

`x` is an array of dimension `getNumDimensions()` representing the point of interest to evaluate the interpolant.

`returns` an array of size `getNumPoints()` of the interpolation weights associated with the grid points. The first weight is associated with the first points returned by `getPoints()`, the second weight is associated with the second point and so on.

5.40 getNumNeededPoints()

```
int getNumNeededPoints() const;
```

Since version 3.0, this has been renamed to `getNumNeeded()`.

5.41 loadNeededPoints()

```
void loadNeededPoints( const double vals[] );
```

Provides the values of the function to be interpolated evaluated at the corresponding abscissas.

`vals` is an array of size `getNumOutputs()` $\times$ `getNumNeeded()`. The first `getNumOutputs()` entries correspond to the outputs of the interpolated function at the first grid point. The second set of `getNumOutputs()` entries correspond to the second point and so on.

5.42 evaluate()

```
void evaluate( const double x[], double y[] ) const;
```

Finds the value of the interpolant (or point-wise approximation) at the provided point `x` as defined by equation (1). The result is written into `y`.

- `x` an array of size `getNumDimensions()` that indicate the point where the interpolant should be evaluated.
- `y` an already allocated array of size `getNumOutputs()`. On exit, the entries of `y` are overwritten with the values of the interpolant at the point `x`.

5.43 evaluateFast()

```
void evaluateFast( const double x[], double y[] ) const;
```

Same as evaluate, but using the enabled acceleration. **Pros**: this is potentially much faster than evaluate, especially when working with functions with many outputs. **Cons**: this function is not thread-safe, i.e., if two threads simultaneously call `evaluateFast()` this will result in a race condition. CUDA and GPU based evaluations are alway thread unsafe, BLAS evaluations could be thread save depending on the BLAS implementation, e.g., OpenBLAS needs a special flag enabled at compile time. The safe assumptions is that `evaluateFast()` is not thread safe.

Note: thread safety relates to calling two functions associated with the same object, in Tasmanian, funciton calls to different objects vener create race conditions (unless such problem comes from some of the third party libraries).

5.44 evaluateBatch()

```
void evaluateBatch(const double x[], int num_x, double y[]) const;
```

Evaluate the approximation at multiple points with a single command.

- `x` an array of size `getNumDimensions()` $\times$ `num_x` that indicate the point where the interpolant should be evaluated, the first set of `getNumDimensions()` entries indicate the first point, the second set of `getNumDimensions()` entries indicates the second point, and so on.
- `y` an already allocated array of size `getNumOutputs()` $\times$ `num_x`. On exit, the entries of `y` are overwritten with the values of the interpolant at the point **x**. The first set of `getNumOutputs()` entries indicate the outputs at the first point, the second set of `getNumOutputs()` entries is the second set of outputs and so on.

This function uses acceleration and similarly to `evaluateFast()`, it is not thread safe. The advantage of the `evaluateBatch()` function is that it can take advantage of accelerated matrix-matrix multiplication. When the interpolated function has many outputs and `num_x` is large, `evaluateBatch()` could work orders of magnitude faster than `evaluate()` or even `evaluateFast()`.

5.45 integrate()

```
void integrate( double y[] ) const;
```

Integrates the interpolant over the domain and returns the result in y.

- y an already allocated array of size `getNumOutputs()`. On exit, the entries of y are overwritten with the values of the integral of the interpolant over the domain.

5.46 is/Global/Sequence/LocalPolynomial/Wavelet()

```
bool isGlobal() const;
bool isSequence() const;
bool isLocalPolynomial() const;
bool isWavelet() const;
```

The function corresponding to the last call to `make***Grid()` returns `true`, all other functions return `false`. If `make***Grid()` has not been called, then all functions return `false`.

5.47 is/set/getConformalTransformASIN()

```
void setConformalTransformASIN(const int truncation[]);
void getConformalTransformASIN(int truncation[]) const;
bool isSetConformalTransformASIN() const;
```

Allow to manipulate the conformal transform based on the truncated $\arcsin(x)$ Maclaurin series. The truncation is a set of integers with dimension `getNumDimensions()` that indicate the number of terms to us in the truncation, i.e., truncation 4 will result in 7-th order polynomial ($\arcsin(x)$ has only off powers in the series). **NOTE**: setting the transform will change the points and weights associated with the grid and any currently loaded values of $f(\boldsymbol{x})$ will not longer be valid.

5.48 clearConformalTransform()

```
void clearConformalTransform();
```

Removes any conformal transform currently used by the grid. **NOTE**: setting the transform will change the points and weights associated with the grid and any currently loaded values of $f(\boldsymbol{x})$ will not longer be valid.

5.49 clear/getLevelLimits()

```
void clearLevelLimits();
void getLevelLimits(int *limits) const;
```

When make***Grid() is called with level limits, the limits are stored within the grid and used in all refinement calls. The clear function can be used to delete any set limits and the get funciton can be used to read the limits. The input `limits` must have length mathcing getNumDimensions().

5.50 setRefinement()

```
void setRefinement( double tolerance, TypeRefinement criteria );
```

Since version 3.0, this function is replaced by `setSurplusRefinement()`, see below.

5.51 setAnisotropicRefinement()

```
void setAnisotropicRefinement( TypeDepth type,
                               int min_growth,
                               int output,
                               const int *level_limits = 0 );
```

Implements the anisotropic refinement strategy described briefly in section 2.4 and in more details in [27]. This function can only be called for Global and Sequence grids. Note that refinement cannot be used if the grid has no outputs or before values have been loaded, i.e., `loadNeededPoints()` has been called.

`type` specifies the type of refinement to use, this can be any type described in `makeGlobalGrid()`, with the exception of the tensor and hyperbolic types.

`min_growth` forces the new "refined" grid to have a minimum number of new (needed) points.

`output` specifies the output to use in the refinement strategy and only computes orthogonal expansion or surpluses for that specific output. Sequence grids store all surpluses anyway, hence all outputs can be easily used together in the refinement strategy, to achieve that set `output` to -1.

`level_limits` the limits set by make**Grid() stored and used by default for all refinement calls. Limits specified here will overwrite the onece set by the make***Grid() command. The purspe of the limits is the same, no points will be added beyond the level for the given direction.

5.52 setSurplusRefinement() - global version

```
void setSurplusRefinement( double tolerance, int output, const int *level_limits = 0 );
```

Implements the surplus refinement strategy described in equation (31) in section 2.4. This function can only be called for Sequence grids and Global grids with sequence rules. Note that refinement cannot be used if the grid has no outputs or before values have been loaded, i.e., `loadNeededPoints()` has been called.

`tolerance` specifies the cutoff threshold, no refinement will be performed for surpluses with relative magnitude smaller than `tolerance`.

`output` specifies the output to use in the refinement strategy and only computes surpluses for that specific output. Sequence grids store all surpluses anyway, hence all outputs can be easily used together in the refinement strategy, to achieve that set `output` to -1.

`level_limits` the limits set by make**Grid() stored and used by default for all refinement calls. Limits specified here will overwrite the onece set by the make***Grid() command. The purspe of the limits is the same, no points will be added beyond the level for the given direction.

5.53 setSurplusRefinement() - local version

```
void setSurplusRefinement( double tolerance,
                           TypeRefinement criteria,
                           int output,
                           const int *level_limits = 0,
                           const double *scale_correction = 0 );
```

Implements the surplus refinement strategy described briefly in section 2.7 and in more details in [26]. This function can only be called for Local polynomial and Wavelet grids. Note that refinement cannot be used if the grid has no outputs or before values have been loaded, i.e., `loadNeededPoints()` has been called.

`tolerance` specifies the cutoff threshold, i.e., the ϵ parameter in equations (41), (42), (43), (44).

`criteria` specifies the refinement strategy

- `refine_classic`, see (41)
- `refine_direction_selective`, see (43)
- `refine_parents_first`, see (42)
- `refine_fds`, see (44)

`output` specifies the output to use in the refinement strategy and only consider surpluses for that specific output. Optionally, `output` can be set to -1 in which case all surpluses will be considered, i.e., for each point the code will consider the output with largest relative surplus. Note, that -1 corresponds to the default behavior of Tasmanian 3.0.

`level_limits` the limits set by make**Grid() stored and used by default for all refinement calls. Limits specified here will overwrite the onece set by the make***Grid() command. The purspe of the limits is the same, no points will be added beyond the level for the given direction.

`scale_correction` defines a number to be used to multiply the corresponding hierarchical coefficient before comparing to the tolerance. This can be used to set/ignore refinement in certain regions based on prior knowledge as opposed to relying entirely on the coefficients. If all outputs are to be used, then `scale_correction` must have lenght getNumPoints() $\times$ getNumOutputs(), if only one output is to be used, then only getNumPoints() number of weights are needed.

5.54 clearRefinement()

```
void clearRefinement();
```

Every `set***Refinement()` function generates a new set of potential points for the sparse grid. The `clearRefinement()` function removes the needed points and all internal data structures associated with the last call any of the set refinement functions. Note that once function `loadNeededPoints()` is called for the new set of points, then the refinement cannot be undone. The purpose of this function is to reduce the memory footprint of the grid in case the user decides not to load the new points from the last refinement.

5.55 mergeRefinement()

```
void mergeRefinement();
```

Every `set***Refinement()` function expands the grid to include a new set of points. However, the new points cannot be used until `loadNeededPoints()` has been called. The `mergeRefinement()` combines the old and new points into a single grid, but removes all old and new loaded values. This is useful when grids are constructed from random set of points and hierarchical coefficients are computed outside of the TasmanianSparseGrid class (see the next four functions). Note: after `mergeRefinement()` all values and hierarchical coeffcients are set to zero.

5.56 getHierarchicalCoefficients()

```
const double* getHierarchicalCoefficients() const;
```

In Tasmanian 5.0 and earlier verions, this function was called getSurpluses(). This returns a pointer to the hierarhcial coefficients associated with eqaution (1).

5.57 evaluateHierarchicalFunctions()

```
void evaluateHierarchicalFunctions(const double x[], int num_x, double y[]) const;
```

- `x` is a vector of size `getNumDimensions()` $\times$ `num_x` indicating the points of interest where the functions should be evaluated. The format is idnetical to the one used in `evaluateBatch()`
- `y` is a vector of size `getNumDimensions()` $\times$ `getNumPoints()` where the functions corresponding to each `x` are stored.

For GlobalGrids, this is the same as `getInterpolantionWeights()`. For other types of grids, this gives the values of the hierarchical functions at the given points, i.e., matrix A in §2.11.

5.58 evaluateSparseHierarchicalFunctions()

```
void evaluateSparseHierarchicalFunctions(const double x[], int num_x,
                                         int* &pntr, int* &indx, double* &vals) const;
```

- `x` is a vector of size `getNumDimensions()` $\times$ `num_x` indicating the points of interest where the functions should be evaluated. The format is idnetical to the one used in `evaluateBatch()`

The matrix associated with hierarchial function values, when using local polynomial and wavelet grids, is often times sparse. This function generates a corresponding compressed sparse representation of the matrix, which is much faster to compute and much easier to store. The size of `pnts` is `num_x+1` and it indicates the offsets of each group of non-zeros, where each group is in turn associated with a single `x` value. The `indx` stores the index of the non-zero value, each entry in `indx` goes from 0 to `getNumPoints() -1`, and `vals` is the corresponding value. If the `pntr`, `indx` and `vals` are looked at as a row-compressed sparse matrix, and

if `getNumOutputs()` is 1, then values of the interpolant for each point in `x` are the entreis of the produce of the matrix times the hierarchical coefficients.

5.59 evaluateHierarchicalFunctions()

```
void setHierarchicalCoefficients(const double c[]);
```

The hierarhcial coefficients are set to the ones specified by `c`. The size of `c` must be `getNumPoints()` $\times$ `getNumOutputs()` and the order is identical to that in `loadNeededPoints()`. Unlike `loadNeededPoints()`, this function sets the coefficients and recomputes the values that match with the coefficients (the load function computes the coefficients as a function fo the values).

5.60 getGlobalPolynomialSpace()

```
int* getPolynomialSpace( bool interpolation, int &n ) const;
```

Computes the polynomial associated with the grid, see Λ^m and Λ^q in equations (16) and (17). Returns a list of integers that stores the multi-indexes.

`interpolation` specifies whether to consider the polynomial space associated with interpolation or integration, i.e., (16) and (17).

`n` returns the number of multi-indexes in the list.

`returns` an array of integers of length `getNumDimensions()` $\times$ `n`, where the first `getNumDimensions()` entries give the first multi-index, the second multi-index is in the second `getNumDimensions()` entries, etc.

5.61 printStats()

```
void printStats() const;
void printStatsLog() const;
```

Prints short description of the sparse grid. The output of `printStats()` is written to standard output (i.e., `cout`), the output of `printStatsLog()` is given to the provided log stream (by default it is `cerr`).

5.62 getSurpluses() and getPointIndexes()

```
const double* getSurpluses() const;
const int* getPointIndexes() const;
```

Those functions exist primarily for debugging and testing purposes. The functions expose internal data structures, modifying the content of the pointers will result in undefined bahavior. Function `getSurpluses()` returns a pointer to the s_j coefficients for sequence, local grids. Function `getPointIndexes()` returns an array with multi-indexes, for local polynomial and wavelet grids the function returns X, for global and sequence grids returns $X(\theta)$. Note that in all cases indexing on the points starts form zero.

5.63 enableAcceleration()

```
void enableAcceleration(TypeAcceleration acc);
```

Sets the preferred acceleration type for this instance of TasmanianSparseGrid class. If the selected acceleration is not available, then the grid will use the next available option. Currently, Tasmanian supports three options:

- `accel_none`: disable all acceleration (other than OpenMP). In this mode, all evaluation function are thread safe, and `evaluateBatch()` uses OpenMP to launch multiple calls to `evaluate()`. The "none" acceleration is usually the slowest, but also guarantees the smallest memory footprint.
- `accel_cpu_blas`: uses BLAS level 2 and level 3 functions in `evaluateFast()` and `evaluateBatch()`. If BLAS is enabled on compile time, this is the default acceleration mode. Functions in this mode are as thread safe as the specific implementation of BLAS, for example, OpenBLAS is not thread safe unless compiled with a specific flag. In general, this mode is assumed not thread safe.
- `accel_gpu_cublas`: used cuBLAS accelerated library for Nvidia GPUs in `evaluateFast()` and `evaluateBatch()` function. This is not a thread safe mode! When working with global and sequence matrices, the data for the loaded values will be loaded into the GPU memory, hence this mode is limited by the available GPU RAM. The amount of GPU RAM needed for this mode must fit three matrices of doubles:
  - `getNumOutputs()` $\times$ `getNumPoints()` for the loaded values;
  - `getNumOutputs()` $\times$ `num_x` for the output of `evaluateBatch()`;
  - `getNumPoints()` $\times$ `num_x` for the values of the basis functions.

  When applied to a local polynomial grid, the hierarchical matrix is sparse, while the exact number of non-zeros cannot be predicted in the general case, for grids constructed without refinement or level limits, it is of order $\log(N)^d \times$ `num_x` for the values of the basis functions, where N is `getNumPoints()`. In addition, cuBLAS uses additional memory for internal variables. If the GPU selected for acceleration is also powering the desktop graphical environment, this would put even more strain in the memory. In the case of `evaluateFast()`, `num_x` is just 1.
- `acceel_gpu_cuda` works similar to cuBLAS, but it also uses a series of custom cuda kernels to compute the hierarchial basis functions. Currently, cuda work only for local polynomial grids, but it provides potentially significant speedup over just using cuBLAS. This mode is not enabled by default, as it is still in somewhat experimental phase. It can be enabled on compile time with the `Tasmanian_ENABLE_CUDA` option for cmake or the `-cuda` switch for the install script.
- `accel_gpu_fullmemory`: was the name used in the previous version 5.0, now it is simply an alias to `accel_gpu_cublas` and will be removed in future versions.
- `accel_gpu_default`: is currently an alias for `accel_gpu_cublas`

Currently, acceleration options affect only Sequence and Global grids, in the future acceleration will be enabled for all types of grids and there will be options for more detailed GPU memory management.

5.64 forceSparseAlgorithmForLocalPolynomials()

```
void forceSparseAlgorithmForLocalPolynomials();
```

If the sparcity pattern, of the matrix associated with batch evaluations of local polynomial grids, is not sufficiently spase, then using a dense matrix with BLAS and cuBLAS will yield better performance. Tasmanian automatically swithes between different evaluation modes depending on the number of non-zeros and the number of points in the sparse grid. However, using dense evaluations leads to increased memory usage, which is expecially important for the GPU. The memory requirement can be reduced by splitting the batch into several small batches, but this may not be desirable in all situations. Furthermore, graphics cards with low double-precision capabilities (e.g., Nvidia GTX cards) may not benefit from the dense algorithm. The command above will force the spase algorithm in all cases.

5.65 getAccelerationType()

```
TypeAcceleration getAccelerationType() const;
```

Returns the acceleration type set by `enableAcceleration()`.

5.66 isAccelerationAvailable()

```
static bool isAccelerationAvailable(TypeAcceleration acc)
```

Returns `True` if the specified acceleration type has been enabled on compile time. Note, that regardless of the answer to the above, any acceleration type can be enabled at any time, Tasmanian will simply internally switch to the "next best" acceleration closest to the one selected by the user, e.g., if CUDA switches to cuBLAS, cuBLAS to CUDA, if both CUDA and cuBLAS are missing, the default is BLAS, which falls back to "none".

5.67 set/set/GPUID()

```
void setGPUID(int in_gpuID);
int getGPUID() const;
static int getNumGPUs();
static int getGPUmemory(int gpu); // returns the MB of a given GPU
static const char* getGPUname(int gpu);
```

Those functions allow for management of multi-GPU setup when GPU based acceleration is requested. The number of GPUs, memory and name are just wrappers around corresponding CUDA functions, those are provided for convenience to allow comprehensive control through just one interface. Each grid comes with associated GPUID and the CUDA native `cudaSetDevice()` functions is called with every call to `evaluateBatch()` and `evaluateFast()`.

5.68 isAccelerationAvailable()

```
void evaluateHierarchicalFunctionsGPU(const double gpu_x[], int cpu_num_x,
                                      double gpu_y[]) const;
void evaluateSparseHierarchicalFunctionsGPU(const double gpu_x[], int cpu_num_x,
                                            int* &gpu_pntr, int* &gpu_indx,
                                            double* &gpu_vals, int &num_nz) const;
```

Experimental functions that return result identical to the non GPU functions, with the difference that all arrays reside on the CUDA device. This can only be called if `Tasmanian_ENABLE_CUDA` has been used on compile time.

5.69 Examples

The file `example_sparse_grids.cpp` in the `Examples/` folder has sample code that demonstrates proper use of the `TasmanianSparseGrid` class. The example can be compiled with the included `CMakeLists.txt` (when using `cmake`) or with the `make exmaples` command (when using the GNU make build engine).

6 Library: Tasmanian DREAM

In this section we describe the main classes associated with the Tasmanian DREAM module. Sometime C++ classes are best described with C++ code, hence we have included the dream example which demonstrates the use of each class, see §10.

6.1 class BaseUniform

```
class BaseUniform{
public:
    BaseUniform();
    virtual ~BaseUniform();

    virtual double getSample01() const = 0;
};
```

By default, Tasmanian uses the pseaudo-random number generator (RNG) build into the C++ compiler. Often, more sophisticated RNG algorithms are desired and this class allows for the default RNG to be replaced by a used provided one. All classes that rely on randomness accept an instance of `BaseUniform` that would be used in place of the default. The `getSample01()` function should return a random number uniformly distributed in $(0, 1)$.

Tasmanian is written with the assumption that `getSample01()` is computationally expensive and special attention has been given to ensure there are no extraneous call to this function. Nevertheless, random sampling requires a large number of calls to `getSample01()`, thus computational cost is of potential consideration.

6.2 class TasmanianDREAM

The class providing the sampling update and accept/reject algorithm is called `TasmanianDREAM`.

6.2.1 Constructor

```
class TasmanianDREAM{
public:
    TasmanianDREAM(std::ostream *os = 0);
    ~TasmanianDREAM();
    void setErrorLog(std::ostream *os);
    void overwriteBaseUnifrom(const BaseUniform *new_uniform);
```

The constructor takes a log stream where error messages would be written out, the logging mechanism defaults to `cerr` if the library is compiled without the `XSDK_DEFAULTS` flag. Logging can be disabled on run-time with a call to `setErrorLog(0)`.

The `overwriteBaseUnifrom()` function allows for user specified RNG to be used, see §6.1.

6.2.2 Version control

```
static const char* getVersion();
static int getVersionMajor();
static int getVersionMinor();
static const char* getLicense();
```

Same as in Tasmanian Sparse Grid, those functions allow run-time access to the version number of license.

6.2.3 setProbabilityWeightFunction()

```
void setProbabilityWeightFunction(ProbabilityWeightFunction *probability_weight);
```

Defines the probability function $\rho(\boldsymbol{x})$, this function should be called before setting chains or any other problem parameters. Calling this function will `delete` the current chain state. The `TasmanianDREAM` class will hold an alias to this class, but will not call `delete` when it is destroyed.

6.2.4 get/setNumChains()

```
void setNumChains(int num_dream_chains);
int getNumChains() const;
```

Defines the number of chains to be used in the DREAM algorithm. Must be called after the call to `setProbabilityWeightFunction()`.

6.2.5 get/setJumpScale()

```
void setJumpScale(double jump_scale);
double getJumpScale();
```

Defines the jump scale parameter, i.e., γ defined in §49. This function must be called after the call to `setProbabilityWeightFunction()`.

6.2.6 getNumDimensions()

```
int getNumDimensions() const;
```

The number of dimensions of $\boldsymbol{x}$. This value is loaded form the probability weight specified in the call to `setProbabilityWeightFunction()`.

6.2.7 get/setCorrection()

```
void setCorrectionAll(BasePDF *correct);
void setCorrection(int dim, BasePDF *correct);
const BasePDF* getCorrection(int dim);
```

Defines the correction parameters, i.e., $\boldsymbol{r}$ defined in §49. The `setCorrectionAll()` function will set identical correction for all directions. Individual directions can also be specified. The `TasmanianDREAM` class will hold an alias to the classed given here, but will not call `delete` when it is destroyed.

6.2.8 collectSamples()

```
double* collectSamples(int num_burnup, int num_samples, bool useLogForm = false);
```

The most computationally expensive function, it performs the actual sampling and has to be called after all other parameters have been set.

`num_burnup` indicates the number of initial iterations that should be discarded.

`num_samples` indicates the number of iterations to be collected. Note that the total number of samples is `num_samples` $\times$ `num_chains`.

`useLogForm` indicates whether to perform sampling in regular or log form. Many likelihood functions and probability distributions include exponential terms that can lead to numerical instability, e.g., comparing exponentials of large negative numbers. The update and accept/reject steps of the DREAM algorithm can be expressed through the log of the `setProbabilityWeightFunction()`, which could improve stability (depending on the problem).

The output of the function is an array of samples with dimensions `getNumDimensions()` $\times$ `getNumChains()` $\times$ `num_samples`. The first sample is stored in the first `getNumDimensions()` entries, the second sample in the second set of `getNumDimensions()` entries, etc.

6.2.9 setChainState()

```
void setChainState(const double* state);
```

Overwrites the current chain state. This allows to manually select the initial samples $\boldsymbol{x}_{c,0}$ and potentially avoid repeating a burn-up process. The format of the `state` files matches the output of `collectSamples()` with `num_samples`$= 1$.

6.2.10 getPDFHistory()

```
double* getPDFHistory() const;
```

Returns the values of the `probability_weight` at the sample points returned by `collectSamples()`, using either regular or log form matching the `useLogForm` variable.

6.3 class BasePDF

The `BasePDF` class describes a general probability function designed to be used as a prior to a Bayesian inference problem or a correction in the DREAM sampling.

6.3.1 Constructor

```
class BasePDF{
public:
    BasePDF();
    virtual ~BasePDF();
    virtual void overwriteBaseUnifrom(const BaseUniform *new_uniform) = 0;
```

The `overwriteBaseUnifrom()` allows the sue of custom RNG, see §6.1.

6.3.2 is/getBoundedAbove/Below()

```
    virtual bool isBoundedBelow() const = 0;
    virtual bool isBoundedAbove() const = 0;
    virtual double getBoundBelow() const = 0;
    virtual double getBoundAbove() const = 0;
```

The `isBounded*()` functions return `True` if the support of the pdf is restricted to a domain bounded above or below. The `getBound*()` returns the actual bounds. With respect to the pdfs defined in Table 4, the `isBounded*()` functions indicate whether the domain contains ∞ either above or below, and the `getBound*()` functions return the values of a and b.

6.3.3 getSample()

```
    virtual double getSample() const = 0;
```

Returns a sample randomly generated by the pdf. This function is called to initialize the chain state of the `TasmanianDREAM` class. Hence, it is possible to return a number inconsistent with the probability density function (although this is not advisable).

6.3.4 getDensity/Log()

```
    virtual double getDensity(double x) const = 0;
    virtual double getDensityLog(double x) const = 0;
```

Returns the probability density at point x, or the $\log$ of the PDF.

6.3.5 TypeDistribution()

```
    virtual TypeDistribution getType() const = 0;
```

Returns an enumerated type corresponding to the distribution. The full list of enumerated types is:

`dist_uniform`, `dist_exponential`

`dist_gaussian`, `dist_truncated_gaussian`

`dist_beta`, `dist_gamma`

`dist_custom`

where additional classes should specify the `dist_custom` type.

6.4 class ProbabilityWeightFunction

The class that described $\rho(\boldsymbol{x})$ and Γ for the DREAM sampling procedure. Unlike the `BasePDF` class, this class is defined in a multidimensional context, there is no assumption that the function integrates to 1, and there is no explicit way to generate random samples with the associated PDF.

6.4.1 Constructor

```
class ProbabilityWeightFunction{
public:
    ProbabilityWeightFunction();
    virtual ~ProbabilityWeightFunction();
```

The properties of this class are intended for inheritance, hence the base constructor is empty.

6.4.2 getNumDimensions()

```
    virtual int getNumDimensions() const = 0;
```

Returns the number of dimensions of $\boldsymbol{x}$.

6.4.3 getDomainBounds()

```
    virtual void getDomainBounds(bool* lower_bound, bool* upper_bound) = 0;
    virtual void getDomainBounds(double* lower_bound, double* upper_bound) = 0;
```

The arrays `lower_bound` and `upper_bound` have dimensions equal to `getNumDimensions()` and return whether or not the support of the `ProbabilityWeightFunction` is finite or infinite, as well as the numerical value of the upper and lower bound. The DREAM sampler assumes that the pdf is 0 outside the bounds specified here and no sample will ever be evaluated for such point not would it be accepted. In practice, the

PDF could be supported on a subset of the hypercube defined by the domain bounds, but `evaluate()` will be called for those samples.

6.4.4 evaluate()

```
virtual void evaluate(int num_points, const double x[], double y[], bool useLogForm) = 0;
```

Returns the values of the pdf at the specified points.

`num_points` is the number of points that need to be evaluated. Note that this number will never exceed the number of chains, however, it may be less as samples outside of the domain bounds will not be evaluated.

`x` is an array where the first set of `getNumDimensions()` entries corresponds to the first point, the second set of `getNumDimensions()` entries corresponds to the second point, etc.

`y` is an output array that will contain the values of the unscaled pdf.

`useLogForm` indicates whether to return the pdf of the $\log$ of the pdf. Note: when `useLogForm=True` the values in y could be negative.

6.4.5 getInitialSample()

```
void getInitialSample(double x[]);
```

The vector `x` has size `getNumDimensions()` and is overwritten with an initial guess for the pdf. This is used to automatically initialize the chain state, however, if the chains state is set directly with the function `TasmanianDREAM::setChainState()`, then this function will be ignored.

6.5 class ProbabilityWeightFunction

```
class LikelihoodTSG : public ProbabilityWeightFunction{
public:
    LikelihoodTSG(const TasGrid::TasmanianSparseGrid *likely,
                  bool savedLogForm, std::ostream *os = 0);
    ~LikelihoodTSG();
    void setPDF(int dimension, BasePDF* &pdf);
```

This class is designed to solve a Bayesian inference problem where the likelihood function has been approximated with a sparse grid method, i.e.,

$$G_\theta[L](\boldsymbol{x}) \approx L(d, f(\boldsymbol{x})).$$

The `savedLogForm` indicates whether the construction was done using $L(d, f(\boldsymbol{x})$ or $\log(L(d, f(\boldsymbol{x})))$, as it is often times easier to interpolate the $\log$ of a complex pdf, especially when using Gaussian likelihood in high dimensions. The `ostream` is used for logging of error messages similar to other uses of logging.

The `LikelihoodTSG` class assumes that the likelihood is associated with priors based on the domain of the sparse grid, i.e.,

- `rule_gausslaguerre` is associated with Exponential distribution.
- `rule_gausshermite` is associated with Gaussian distribution.
- `rule_localp0` and `rule_clenshawcurtis0` are associated with Beta distribution with $\alpha = \beta = 2$.
- All other rules are associated with a uniform distribution.

The domain bounds are taken from the ones specified by the grid. The entries of the initial sample are taken form the priors.

In many cases, the default pdf selection is undesirable, then the default prior for each `dimension` can be overwritten with `void setPDF()`. Note that in this case the priors will be deleted when the class is destroyed.

6.6 class PosteriorFromModel

This class is designed to perform Bayesian inference with a model that is either provided by the user in a wrapper class or $f(\boldsymbol{x})$ has been approximated with a sparse grid.

6.6.1 Constructor and core functions

```
class PosteriorFromModel : public ProbabilityWeightFunction{
public:
    PosteriorFromModel(const TasGrid::TasmanianSparseGrid *model, std::ostream *os = 0);
    PosteriorFromModel(const CustomModelWrapper *model, std::ostream *os = 0);
    ~PosteriorFromModel();
    void overwritePDF(int dimension, BasePDF* pdf);
    void setErrorLog(std::ostream *os);
```

If the constructor is called with a sparse grid, then default priors will be selected based on the grid rule and domain similar to `ProbabilityWeightFunction`, no priors are selected for a custom model. When using `overwritePDF()` the pointers will not be deleted when the class is destroyed. The set `setErrorLog()` works the same as in other cases.

6.6.2 setLikelihood()/setData()

```
    void setLikelihood(BaseLikelihood *likelihood);
    void setData(int num_data_samples, const double *posterior_data);
```

Those functions allow the user to specify the likelihood function for the inference as well as manipulate the data. Currently, Tasmanian implements only Gaussian likelihood with covariance that is either constant diagonal, diagonal, or general dense matrix.

Evaluating the `PosteriorFromModel` first generates a call to the model or sparse grids, then the output (i.e., values of $f(\boldsymbol{x})$) are given to the evaluate function of the likelihood class. Depending on the way the likelihood is set, the data may be included in the class when the class is created, or taken as input to the evaluate function. If the data is stored in the class, then `setData()` doesn't need to be called.

6.7 class BaseLikelihood

```
class BaseLikelihood{
public:
    BaseLikelihood();
    virtual ~BaseLikelihood();
    virtual double* getLikelihood(int num_model, const double *model,
                                  int num_data, const double *data,
                                  double *likelihood = 0, bool useLogForm = true) = 0;
};
```

This class defined a function of the form $L(d, f(\boldsymbol{x}))$. The `num_model` indicates the number of points where the likelihood needs to be evaluated and `model` indicates the model values. The number of model outputs has to be build into the class, e.g., when calling the constructor of the inheriting class. The `nun_data` indicates the number of data entries and `data` holds the values. The format of the vectors matches the output of the sparse grids `evaluateBatch()` function. The result is stored in `likelihood` and `useLogForm` indicates whether to use the log of the likelihood.

6.8 class GaussianLikelihood

```
class GaussianLikelihood : public BaseLikelihood{
public:
    GaussianLikelihood(int outputs, TypeLikelihood likelihood, const double covariance[],
                       int data_entries, const double data[]);
    ~GaussianLikelihood();

    double* getLikelihood(int num_model, const double *model,
                          int num_data = 0, const double *data = 0,
                          double *likelihood = 0, bool useLogForm = true);
```

Defines the Gaussian likelihood currently included in Tasmanian.

$$L(\{d_1, \ldots, d_s\}, f(\boldsymbol{x})) = \exp\left(-\sum_{i=1}^{s} (d_i - f(\boldsymbol{x}))^T \Sigma^{-1} (d_i - f(\boldsymbol{x})) \right)$$

For the constructor

- `outputs` is the number of outputs used by the model, i.e., the range of $f(\boldsymbol{x})$;
- `likelihood` indicates one of the three types: `likely_gauss_scale`, `likely_gauss_diagonal`, and `likely_gauss_dense`, where the scale likelihood corresponds to a covariance with constant diagonal;
- `covariance` is a single number for `likely_gauss_scale`, a vector of size `outputs` when using `likely_gauss_diagonal`, and an `outputs` $\times$ `outputs` symmetric positive definite matrix Σ for `likely_gauss_dense`;
- `data_entries` is the number of data entries, i.e., s;
- `data` the data entries, i.e., the matrix $\{d_1, \ldots, d_s\}$ in column major format.

The data is stored in the class and the calls to `getLikelihood()` may contain default data entries. There is no need to use the `setData()` function for the `PosteriorFromModel` class.

6.9 class CustomModelWrapper

```
class CustomModelWrapper{
public:
    CustomModelWrapper();
    virtual ~CustomModelWrapper();
    virtual int getNumDimensions() const = 0;
    virtual int getNumOutputs() const = 0;
    virtual void evaluate(const double x[], int num_points, double y[]) const = 0;
};
```

This class can be used to define a custom model, not necessarily associated with sparse grids. The model should communicate the umber of inputs and outputs and evaluate batches of points, identical to the sparse grids function `evaluateBatch()`.

6.10 MPI and distributed memory

Tasmanian includes a class that allows the use of multiple sparse grids models in a distributed computing environment using MPI. Each sparse grid model is associated with local data and the likelihood is the product of all likelihood across all MPI processes. This class is still in a highly experimental stage and very feature poor. Use at your own peril.

7 TASGRID

The `tasgrid` executable is a command line interface to `libtasmaniansparsegrid`. It provides the ability to create and manipulate sparse grids, save and load them into files and optionally interface with another program via text files. For the most part, `tasgrid` reads a grid from a file, calls one or more of the functions described in the previous section and then saves the resulting grid. In addition, `tasgrid` provides a set of basic functionality tests.

7.1 Basic Usage

```
./tasgrid <command> <option1> <value1> <option2> <value2> ....
```

The first input to the executable is the command that specifies the action that needs to be taken. The command is followed by options and values.

Every command is associated with a number of options. If other options are provided, then they are ignored.

`Tasgrid` has some basic error checking and if it encounters an error in the input, `tasgrid` will print a short message specifying the error and then exit.

7.2 Command: -h, help, -help, –help

```
./tasgrid --help
./tasgrid -makequadrature help
```

Prints information about the usage of `tasgrid`. In addition, writing `help` after any command will print information specific to that command. Thus, `help` is a universal option.

7.3 Command: -listtypes

```
./tasgrid -listtypes
```

List the available one dimensional quadrature and interpolation rules as well as the different types of grids, refinement and conformal mapping types. Use this command to see the correct spelling of all string options.

7.4 Command: -version or -info

```
./tasgrid -version
./tasgrid -v
./tasgrid -info
```

Prints the version of the library and the available acceleration options.

7.5 Command: -test

```
./tasgrid -test
./tasgrid -test random
./tasgrid -test verbose
```

Performs a series of basic functionality tests. For different grids, different parameters and all possible quadrature rules, `tasgrid` will perform a test to make sure that it can integrate or interpolate appropriate functions to a high degree of precision. The output of the command should be a list of the tests and the `Pass` or `Fail` result. A failure of a test in an indication that something went wrong in the build process or there is a bug in the code.

Note that the test of the custom rule requires that `GaussPattersonRule.table` file be present in the execution folder.

On a Intel 3.2Ghz 6-core Sandy Bridge-E CPU all tests take about 8 seconds (with OpenMP enabled), if the test take much longer on your machine this is an indication of either a slow CPU or problem with the build or code.

The tests rely on random number generation to estimate the accuracy of computed interpolants. If the test fails, this may be indication of a problem with the hard-coded random seed. Using the `random` option will reset the seed on every run and will provide more statistically significant results (also will likely fail a few times).

The `verbose` will print more detailed output. This affects only the successful tests, failed tests always print verbose information.

7.6 Command: -makegrid

This is a deprecated command, will be removed in future releases. Right now, `makegrid` calls `makeglobal` for global rules, `makelocalpoly` for local polynomial rules, and `makewavelet` for wavelet rules.

7.7 Command: -makeglobal

```
./tasgrid -makeglobal <option1> <value1> <option2> <value2> ....
```

Calls `makeGlobalGrid()` with the specified set of options. Accepted options are:

`-dimensions` the dimensions parameter

`-outputs` the outputs parameter

`-depth` the depth parameter

`-onedim` is a string specifying the `rule` parameter of `makeGlobalGrid()`. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-type` is a string specifying the `type` parameter of `makeGlobalGrid()`. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-alpha` the alpha parameter

`-beta` the beta parameter

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` number of columns. The first points will be on the first row, the second on the second row and so on.

`-gridfile` is an optional file. The grid can be saved in this file for future use.

`-anisotropyfile` is an optional matrix file, however, unlike regular matrix files the entries `!must be integers!`, otherwise the behavior of the code becomes unpredictable. The matrix file must have one column and either `-dimensions` number of rows for non-"curved" rules or double `-dimensions` number of rows for "curved". Basically, this specifies the `anisotropic_weights` input to `makeGlobalGrid()`.

`-customrulefile` must be specified when `-onedim custom-tabulated` is used. The given filename must provide the description of a custom rule. See Appendix 10 for details on the custom file format.

`-transformfile` is an optional matrix file that specifies the transformation from the canonical domain to a custom domain. The matrix file should have `dimensions` number of rows and 2 columns. The first column is the a_k parameter and the second column is the b_k parameter and each row corresponds to one dimension. For detail on the matrix file format see subsection 7.29. Note: this option used to be called `-inputfile`.

`-conformaltype` set the type of the conformal map, only `asin` is currently supported.

`-conformalfile` gives a matrix file with a column of `-dimensions` number of integers corresponding to the number of terms in the truncated Maclaurin series.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.8 Command: -makesequence

```
./tasgrid -makesequence <option1> <value1> <option2> <value2> ....
```

Calls `makeSequenceGrid()` with the specified set of options. Accepted options are:

`-dimensions` the dimensions parameter

`-outputs` the outputs parameter

`-depth` the depth parameter

`-onedim` is a string specifying the `rule` parameter of `makeSequenceGrid()`. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-type` is a string specifying the `type` parameter of `makeSequenceGrid()`. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` number of columns. The first points will be on the first row, the second on the second row and so on.

`-gridfile` is an optional file. The grid can be saved in this file for future use.

`-anisotropyfile` same as in `-makeglobal`

`-transformfile` same as in `-makeglobal`

`-conformaltype` same as in `-makeglobal`

`-conformalfile` same as in `-makeglobal`

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.9 Command: -makelocalpoly

```
./tasgrid -makelocalpoly <option1> <value1> <option2> <value2> ....
```

Calls `makeLocalPolynomialGrid()` with the specified set of options. Accepted options are:

`-dimensions` the dimensions parameter

`-outputs` the outputs parameter

`-depth` the depth parameter

`-order` the order parameter

`-onedim` is a string specifying the `rule` parameter of `makeLocalPolynomialGrid()`. To see a list of accepted types, use command `./tasgrid -listtypes`

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` number of columns. The first points will be on the first row, the second on the second row and so on.

`-gridfile` is an optional file. The grid can be saved in this file for future use.

`-transformfile` same as in `-makeglobal`. Note: this option used to be called `-inputfile`.

`-conformaltype` same as in `-makeglobal`

`-conformalfile` same as in `-makeglobal`

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.10 Command: -makewavelet

```
./tasgrid -makewavelet <option1> <value1> <option2> <value2> ....
```

Calls `makeWaveletGrid()` with the specified set of options. Accepted options are:

`-dimensions` the dimensions parameter

`-outputs` the outputs parameter

`-depth` the depth parameter

`-order` the order parameter

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` number of columns. The first points will be on the first row, the second on the second row and so on.

`-gridfile` is an optional file. The grid can be saved in this file for future use.

`-transformfile` same as in `-makeglobal`. Note: this option used to be called `-inputfile`.

`-conformaltype` same as in `-makeglobal`

`-conformalfile` same as in `-makeglobal`

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.11 Command: -makequadrature

```
./tasgrid -makequadrature <option1> <value1> <option2> <value2> ....
```

Based on the value of `-onedim`, this calls to one of: `-makeglobal`, `-makelocalpoly`, and `-makewavelet`. The accepted parameters are the same with these exceptions:

`-outputs` is NOT accepted, the outputs are set to 0

`-gridfile` is NOT accepted, if a gridfile is desired, call the corresponding `-make***` command with `-outputs 0`.

`-outputfile` has different format. At the end of the program, `tasgird` will write in the file the quadrature weights and points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` plus one number of columns. The first abscissa will be on the first row, the second on the second row and so on. On each row, the first column is the weight and the rest of the columns are the entries of the associated point.

`-print` has the same format as `-outputfile`, but using `cout` stream.

7.12 Command: -recycle

```
./tasgrid -recycle
```

Removed in version 3.0.

7.13 Command: -makeupdate

```
./tasgrid -makeupdate <option1> <value1> <option2> <value2> ...
```

Calls `updateGlobalGird()` or `updateSequenceGrid()`.

`-gridfile` this is the file with an already created grid, must be either Global or Sequence.

`-depth` the depth parameter

`-type` is a string specifying the `type` parameter. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-anisotropyfile` is an optional matrix file, however, unlike regular matrix files the entries **!must be integers!**, otherwise the behavior of the code becomes unpredictable. The matrix file must have one column and either `-dimensions` number of rows for non-"curved" rules or double `-dimensions` number of rows for "curved". Basically, this specifies the `anisotropic_weights` input to `updateGlobalGird()` and `updateSequenceGrid()`.

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the new (needed) points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` number of columns. The first points will be on the first row, the second on the second row and so on.

`-print` write out the same data as in the `outputfile` but to the `cout` stream.

7.14 Command: -getquadrature

```
./tasgrid -getquadrature <option1> <value1> <option2> <value2> ...
```

Calls `getQuadratureWeights()`.

`-gridfile` this is the file with an already created grid.

`-outputfile` has different format. At the end of the program, `tasgird` will write in the file the quadrature weights and points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `-dimensions` plus one number of columns. The first abscissa will be on the first row, the second on the second row and so on. On each row, the first column is the weight and the rest of the columns are the entries of the associated point.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.15 Command: -getpoints

```
./tasgrid -getpoints <option1> <value1> <option2> <value2> ....
```

Calls `getPoints()`.

`-gridfile` this is the file with an already created grid.

`-outputfile` is an optional matrix file. The program will write in the file the points associated with the grid. The matrix file will have `getNumPoints()` number of rows and `getNumDimensions()` number of columns. The first points will be on the first row, the second on the second row and so on.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.16 Command: -getinterweights

```
./tasgrid -getinterweights <option1> <value1> <option2> <value2> ....
```

Calls `getInterpolationWeights()` for every point specified by the `-xfile`. The result is written to an output matrix file

`-gridfile` this is the file with an already created grid and loaded values.

`-xfile` is a matrix file with points of interest. The file can have arbitrary number of rows and `getNumDimensions()` number of columns. Each row corresponds to one point of interest.

`-outputfile` is an optional matrix file that is written on exit. The file contains the interpolation weights associated with the points provided by the `-xfile`. The file has the same number of rows and `getNumPoints()` number of columns. Each row contains the interpolation weights associated with the corresponding point of interest.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.17 Command: -getneededpoints

```
./tasgrid -getneededpoints <option1> <value1> <option2> <value2> ....
```

Calls `getNeeded()`.

`-gridfile` this is the file with an already created grid.

`-outputfile` is an optional matrix file. The program will write in the file the points associated with the grid that are not yet associated with values of the interpolated function. The matrix file will have `getNumPoints()` number of rows and `getNumDimensions()` number of columns. The first points will be on the first row, the second on the second row and so on.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.18 Command: -loadvalues

```
./tasgrid -loadvalues <option1> <value1> <option2> <value2> ....
```

Calls `loadNeededPoints()`.

`-gridfile` this is the file with an already created grid. On exit, it will contain the grid with loaded values.

`-valsfile` is a matrix file with `getNumNeededPoints()` number of rows and `getNumOutputs()` number of columns. The first row contains the values of the interpolated function associated with the first needed abscissa. The second row corresponds to the second abscissa and so on.

7.19 Command: -evaluate

```
./tasgrid -evaluate <option1> <value1> <option2> <value2> ....
```

Calls `evaluateBatch()` and used BLAS and OpenMP acceleration, if those are enabled on compile time. For now GPU acceleration is not available, most notably due to the overhead of reading/writing matrix files, which is far greater then the cost of evaluations and hence makes acceleration much less valuable.

`-gridfile` this is the file with an already created grid and loaded values.

`-xfile` is a matrix file with points of interest. The file can have arbitrary number of rows and `getNumDimensions()` number of columns. Each row corresponds to one point of interest.

`-outputfile` is an optional matrix file that is written on exit. The file contains the values of the interpolant at the points provided by the `-xfile`. The file has the same number of rows and `getNumOutputs()` number of columns. Each row contains the values of the interpolant at the corresponding point of interest.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.20 Command: -integrate

```
./tasgrid -integrate <option1> <value1> <option2> <value2> ....
```

Calls `integrate()`

`-gridfile` this is the file with an already created grid and loaded values.

`-outputfile` is an optional matrix file that is written on exit. The file contains the integrals of the interpolant over the domain. The file has one row and `getNumOutputs()` number of columns.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.21 Command: -getanisotropy

```
./tasgrid -getanisotropy <option1> <value1> <option2> <value2> ....
```

Calls `estimateAnisotropicCoefficients()`

`-gridfile` this is the file with an already created grid.

`-type` is a string specifying the `type` parameter of `estimateAnisotropicCoefficients()`. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-refout` specifies the `output` parameter of `estimateAnisotropicCoefficients()`.

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the values of the estimated coefficients.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.22 Command: -refine

```
./tasgrid -refine <option1> <value1> <option2> <value2> ....
```

Note: the behavior of this command has changed in version 3.0, when applied to Global grids this command will use anisotropic as opposed to surplus refinement.

For Global and Sequence grids calls `setAnisotropicRefinement()` and for Local Polynomial and Wavelet grids calls `setSurplusRefinement()`. See commands `-refineaniso` and `-refinesurp`.

7.23 Command: -refineaniso

```
./tasgrid -refineaniso <option1> <value1> <option2> <value2> ....
```

Calls `setAnisotropicRefinement()`

`-gridfile` this is the file with an already created grid and loaded values.

`-type` is a string specifying the `type` parameter of `setAnisotropicRefinement()`. See `./tasgrid -listtypes` for the list of accepted strings, it's pretty self-explanatory.

`-mingrowth` specifies the `min_growth` parameter of `setAnisotropicRefinement()`.

`-refout` specifies the `output` parameter of `setAnisotropicRefinement()`.

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the needed points, i.e., the ones that are not yet associated with values of the interpolated function. The matrix file will have `getNumPoints()` number of rows and `getNumDimensions()` number of columns. The first points will be on the first row, the second on the second row ...

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.24 Command: -refinesurp

```
./tasgrid -refinesurp <option1> <value1> <option2> <value2> ....
```

Calls `setSurplusRefinement()`

`-gridfile` this is the file with an already created grid and loaded values.

`-tolerance` specifies the `tolerance` parameter of `setSurplusRefinement()` command.

`-refout` specifies the `output` parameter of `setAnisotropicRefinement()`.

`-reftype` is a string specifying the refinement criteria. See `./tasgrid -listtypes` for accepted values for this option.

`-outputfile` is an optional matrix file. At the end of the program, `tasgird` will write in the file the needed points, i.e., the ones that are not yet associated with values of the interpolated function. The matrix file will have `getNumPoints()` number of rows and `getNumDimensions()` number of columns. The first points will be on the first row, the second on the second row ...

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.25 Command: -cancelrefine

```
./tasgrid -cancelrefine <option1> <value1> <option2> <value2> ....
```

Calls `clearRefinement()`

`-gridfile` this is the file with an already created grid.

7.26 Command: -getpoly

```
./tasgrid -getpoly <option1> <value1> <option2> <value2> ....
```

Calls `getGlobalPolynomialSpace()`

`-gridfile` this is the file with an already created grid.

`-type` is a string specifying the `type` parameter of `makeGlobalGrid()`. See `./tasgrid -listtypes` for the list of accepted strings, any type starting with `i` sets the `interpolation` parameter to `True`, any type starting with `q` sets `interpolation` parameter to `False`.

`-outputfile` is an optional matrix file. The list of multi-indexes is written to the file.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.27 Command: -summary

```
./tasgrid -summary -gridfile <filename>
```

Reads the grid in the provided file and prints short summary about the grid.

`-gridfile` this is the file with an already created grid.

7.28 Commands: -getsurpluses, -getpointindexes

```
./tasgrid -getsurpluses/-getpointindexes <option1> <value1> ....
```

Calls `getSurpluses()` or `getPointIndexes()`.

`-gridfile` this is the file with an already created grid.

`-outputfile` is an optional matrix file. The list of surpluses or multi-indexes is written to the file.

`-print` write out the same data as in the `-outputfile` but to the `cout` stream.

7.29 Matrix File Format

A matrix file is a simple text file that describes a two dimensional array of real numbers. The file contains two integers on the first line indicating the number of rows and columns. Those are followed by the actual entries of the matrix one row at a time.

The file containing

```
3 4
1.0  2.0  3.0  4.0
5.0  6.0  7.0  8.0
9.0 10.0 11.0 12.0
```

represents the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

A matrix file may contain only one row or column, e.g.,

```
1 2
13.0 14.0
```

All files used by `tasgrid` have the above format with three exceptions. The `-gridfile` option contains saved sparse grids and it is not intended for editing outside of the `tasgrid` calls. The `-anisotropyfile` requires a matrix with one column and it should contain only integers. The `-customrulefile` has special format is described in Appendix 10.

8 MATLAB Interface

The MATLAB interface to `tasgrid` consists of several functions that call various `tasgrid` commands and read and write matrix files. Unlike most MATLAB interfaces, this is code does not use .mex files, but rather system commands and text files. In a nut shell, MATLAB `tsgMake***` functions take a user specified name and create a MATLAB object and a file generated by `tasgrid` option `-gridfile` (or `TasmanianSparseGrid::write()` function). The MATLAB object is used to reference the specific grid file and is needed by most other functions. Here are some notes to keep in mind:

- The MATLAB interface requires that MATLAB is able to call external commands and the `tasgrid` executable in particular.
- The MATLAB interface also requires access to a folder where the files can be written.
- The MATLAB work folder option in the `install.sh` scrip as well as `cmake` allows you to automatically specify where the temporary MATLAB files will be stored. The `make matlab` target in the GNU make engine sets the work folder in a sub-folder of the Tasmanian source folder. In either case, the default folder can be changed by manually editing `tsgGetPaths.m`.
- Each grid has a user specified name, that is a string which gets appended at the beginning of the file name.
- The `tsgDeleteGrid()`, `tsgDeleteGridByName()` and `tsgListGridsByName()` functions allow for cleaning the files in the temporary folder.
- Every MATLAB function corresponds to one `tasgrid` command.
- Every function comes with help comments that can be accessed by typing

```
help tsgFunctionName
```

- Note that it is recommended to add the folder with the MATLAB interface to your MATLAB path, otherwise you have to use the `addpath` command every time you want Tasmanian after `MATLAB` restart.
- All input variables follow naming convention where the first character specifies the type of the variable:

  i stands for integer

  s stands for string

  f stands for real number

  l stands for list

  v stands for vector, i.e., row or column matrix

  m stands for matrix, i.e., two dimensional array

8.1 function tsgCoreTests()

```
tsgCoreTests()
```

Performs a series of tests of the MATLAB interface. Failing test are indication of wrong installation.

8.2 function tsgGetPaths()

```
[ sFiles, sTasGrid ] = tsgGetPaths()
```

This function returns two strings:

- `sTasGrid` is a string containing the path to the `tasgrid` executable (including the name of the executable).
- `sFiles` is the path to a folder where MATLAB has read/write permission. Files will be created and deleted in this folder.

8.3 functions tsgReadMatrix() and tsgWriteMatrix()

Those functions are used internally to read from or write to matrix files. Those functions should not be called directly.

8.4 functions tsgCleanTempFiles()

Those functions are used internally to clean the temporary files.

8.5 function tsgListGridsByName()

Scans the work folder and lists the existing grids regardless whether those are currently associated with MATLAB objects. The names can be used for calls to `tsgDeleteGridByName()` and `tsgReloadGrid()`.

8.6 function tsgDeleteGrid()/tsgDeleteGridByName()

Deleting the MATLAB object doesn't remove the files from the work folder, thus `tsgDeleteGrid()` has to be explicitly called to remove the files associated with the grid. If the MATLAB object has been lost (i.e., cleared by accident), then the grid files can be deleted by specifying just the name for `tsgDeleteGridByName()`, see also `tsgListGridsByName()`.

8.7 function tsgReloadGrid()

Creates a new MATLAB object file for a grid with existing files in the work folder. This function can restore access to a grid if the grid object has been lost. This function can also create aliases between two grids which can be dangerous, see section 8.13. This function can also be used to gain access to a file generated by `tasgrid -gridfile` option or `TasmanianSparseGrid::write()` function, just generate the file, move it to the work folder, rename it to `<name>_FileG`, and call `lGrid = tsgReloadGrid( <name> )`.

8.8 function tsgCopyGrid()

Creates a duplicate of an existing grid, this function creates a new MATLAB object and a new grid file in the work folder.

8.9 function tsgWriteCustomRuleFile()

Writes a file with a custom quadrature or interpolation rule, see Appendix 10 and the function help for more details.

8.10 function tsgExample()

```
tsgExample()
```

This function contains sample code that replicated the C++ example. This is a demonstration on the proper way to call the MATLAB functions.

8.11 Other functions

All other functions correspond to calls to `tasgrid` with various options. The names are self-explanatory. Use the MATLAB help command to see the syntax of each function.

8.12 Saving a Grid

You can save the `lGrid` object just like any other MATLAB object. However, a saved grid has two components, the `lGrid` object and the files associated with the grid that are stored in the folder specified by `tsgGetPath()`. The files in the temporary folder will be persistent until either `tsgDeleteGrid()` is called or the files are manually deleted. The only exception is that the `tsgExample()` function will overwrite any grids with names starting with `_tsgExample1` through `_tsgExample10`. Note that modifying `tsgGetPath()` may result in the code not being able to find the needed files and hence the grid object may be invalidated.

8.13 Avoiding Some Problems

- Make sure to call `tsgDeleteGrid()` as soon as you are done with a grid, this will avoid clutter in the temporary folder.
- If you clear an `lGrid` object without calling `tsgDeleteGrid()` (i.e., you exit MATLAB without saving), then make sure to use `tsgListGridsByName()` and `tsgDeleteGridByName()` to safely delete the "lost" grids.
- Working with the MATLAB interface is very similar to working with dynamical memory, where the data is stored on the disk as opposed to the RAM and the `lGrid` object is the pointer. Also, the grids are associated by name as opposed to a memory address.
- If multiple users are sharing the same temporary folder, then it would be useful if they come up with a naming convention that prevents two users from using the same grid name. For example, instead of both users creating a grid named `mygrid1`, the users should name their grids `johngrid1` and `janegrid1`.
- All of the grid data for all of the grids is stored in the same folder. Anyone with access to the temporary folder has full access to all of the sparse grid data.
- If two users have separate copied of `tsgGetPaths()`, then they can use separate storage folders without any of the multi-user considerations. This is true even if all other files are shared, including the `tasgrid` executable and `libtsg` library.

9 Python Interface

The Python interface uses `c_types` and links to the C interface to TASMANIAN. The C interface uses a series of functions that take a void pointer that is an instance of the C++ class. The Python module takes a hold of the C void pointer and encapsulates it into a Python class. This allows for the usage of Python in a way very similar to C++.

Both Python 2 and 3 are supported and the Tasmanian module is fully compatible with both versions. By default, the installer puts `/usr/bin/env python` in the hashbang command and this can be overwritten using the appropriate `cmake` or `make` command, see §4.

Required Python modules:

- `c_types`
- `numpy`
- `matplotlib.pyplot` (optional)

Every Python function that accepts input checks the validity of the inputs and trows a `TasmanianInputError` exception with two strings

- `sVariable`: pointing to the variable where the error is encountered
- `sMessage`: gives a short explanation of the error encountered

In addition, every function in the `TasmanianSparseGrid` class comes with short description that can be invoked with the Python `help` command.

The names of the functions in the Python class match the names in the C++ library. The input and output C++ arrays, i.e., `double* double[] int[]`, are replaced by numpy 1D and 2D arrays. The enumerated inputs are replaced by strings with the same syntax as the `tasgrid` command line tool. Refer to the help for the specifics for each function.

One point of difference in the evaluate functions is that the default Python `evaluate()` corresponds to C++ `evaluateBatch()` and is not thread safe by default. Python scripts are usually sequential, hence the faster thread unsafe option is chosen here. Thread safe evaluations are available with the `evaluateThreadSafe()` function. The GPU acceleration options are also available via the Python interface.

Finally, if `matplotlib.pyplot` is available on execution time, the Python module gives two plotting functions that can be called for grids with two dimensions.

- `plotPoints2D`: plots the nodes associated with the grid
- `plotResponse2D`: plots a color image corresponding to the response surface

10 Examples

Tasmanian comes with four example files written in C++, Python and `MATLAB` that demonstrate how to use the libraries. The examples are a good self explanatory illustration for the libraries are intended to be used and a very good starting point for people new to Tasmanian. Make sure to look at the source files.

The `MATLAB` example is called `tsgExample()` and is located in the `MATLAB` install folder, which is needed for the script to interact with the rest of the `MATLAB` interface. The other three examples are located in the `<install folder>/examples` (if using `install.sh` or `cmake`) or the source root folder (if using GNU make). The C++ examples come with `CMakeLists.txt` or a `make examples` target, depending on the build engine.

The three sparse grids example comes in all three languages and executes identical operations, hence all three can be viewed side by side for comparison. The source is split into independent sections, allowing to view each example independently. The examples are designed to be simple to illustrate capabilities and not to be a comprehensive numerical study of sparse grids. Only benchmark 5 comes with a mock-up benchmark to contrast Global and Sequence grids.

The single DREAM example is written only in C++ and like the rest of the module is still in a testing stage.

Custom Rule Specification

The custom rule functionality allows the creation of a sparse grid using a rule other than the ones implemented in the code. The custom rule is defined via a file with tables that list the levels, number of points per level, exactness of the quadrature at each level, points and their associated weights. Currently, the custom rules work only with global grids and hence the interpolant associated with the rule is a global interpolant using Lagrange polynomials.

The custom rule is defined via custom rule file, with the following format:

`line 1:` should begin with the string `description:` and it should be followed by a string with a short description of the rule. This string is used only for human readability purposes.

`line 2:` should begin with the string `levels:` followed by an integer indicating the total number of rule levels defined in the file.

After the description and total number of levels have been defined, the file should contain a sequence of integers describing the number of points and exactness, followed by a sequence of floating point numbers listing the points and weights.

`integers:` is a sequence of integer pairs where the first integer indicates the number of points for the current level and the second integer indicates the exactness of the rule. For example, the first 3 levels of the Gauss-Legendre rule will be described via the sequence $1\,1\,2\,3\,3\,5$, while the first 3 levels of the Clenshaw-Curtis rule will be described via $1\,1\,3\,3\,5\,5$.

`floats:` is a sequence of floating point pairs describing the weights and points. The first number of the pair is the quadrature weight, while the second number if the abscissa. The points associated with the first level are listed in the first pairs. The second set of pairs lists the points associated with the second level and so on.

Here is an example of Gauss-Legendre 3 level rule for reference purposes:

```
description: Gauss-Legendre rule
levels: 3
1 1 2 3 3 5
2.0 0.0
1.0 -0.5774 1.0 0.5774
0.5556 -0.7746 0.8889 0.0 0.5556 0.7746
```

Similarly, a level 3 Clenshaw-Curtis rule can be defined as

```
description: Clenshaw-Curtis rule
levels: 3
1 1 3 3 5 5
2.0 0.0
0.333 1.0 1.333 0.0 0.333 -1.0
0.8 0.0 0.067 -1.0 0.067 1.0 0.533 -0.707 0.533 0.707
```

Several notes on the custom rule file format:

- TASMANIAN works with double precision and hence a custom rule should be defined with the corresponding number of significant digits. The examples above are for illustrative purposes only.

- The order of points within each level is irrelevant. TASMANIAN will internally index the points.
- Points that are within distance of 10^{-12} of each other will be treated as the same point. Thus, repeated (nested) points can be automatically handled by the code. The tolerance can be adjusted in `tsgEnumerates.hpp` by modifying the `NUM_TOL` constant,
- Naturally, TASMANIAN cannot create a sparse grid that requires a one dimensional rule with level higher than what is provided in the file. Predicting the required number of levels can be hard in the case of anisotropic `iexact/qexact` grids, the code will print a warning message if the custom rule does not provide a sufficient number of points.
- The exactness constants are used only if `qexact` is used for `makeGlobalGrid()` and the indexes of the polynomial space, i.e., `getPolynomialIndexes()`. If `qexact` is not used, then the exactness integers can be set to 0.
- The quadrature weights are used only if integration is performed. If no quadrature or integration is used, then the weights can all be set to 0.
- If a custom rule is used together with `TransformAB`, then the transform will assume that the rule is defined on the canonical interval $[-1, 1]$. A custom rule can be defined on any arbitrary interval, however, for any interval different from $[-1, 1]$ the `TransformAB` functions should not be used.
- The code comes with an example custom rule file that defines 9 levels of the Gauss-Legendre-Patterson rule, a.k.a., nested Gauss-Legendre rule.

References

[1] S. ACHARJEE AND N. ZABARAS, *A non-intrusive stochastic galerkin approach for modeling uncertainty propagation in deformation processes*, Computers & structures, 85 (2007), pp. 244–254. 2

[2] B. ADCOCK AND R. B. PLATTE, *A mapped polynomial method for high-accuracy approximations on arbitrary grids*, SIAM Journal on Numerical Analysis, 54 (2016), pp. 2256–2281. 21

[3] N. AGARWAL AND N. R. ALURU, *A domain adaptive stochastic collocation approach for analysis of mems under uncertainties*, Journal of Computational Physics, 228 (2009), pp. 7662–7688. 2

[4] V. BARTHELMANN, E. NOVAK, AND K. RITTER, *High dimensional polynomial interpolation on sparse grids*, Advances in Computational Mathematics, 12 (2000), pp. 273–288. 2

[5] J. BECK, F. NOBILE, L. TAMELLINI, AND R. TEMPONE, *Convergence of quasi-optimal stochastic galerkin methods for a class of pdes with random coefficients*, Computers & Mathematics with Applications, 67 (2014), pp. 732–751. 9

[6] G. E. BOX AND G. C. TIAO, *Bayesian inference in statistical analysis*, vol. 40, John Wiley & Sons, 2011. 4

[7] M. A. CHKIFA, *On the Lebesgue constant of Leja sequences for the complex unit disk and of their real projection*, Journal of Approximation Theory, 166 (2013), pp. 176–200. 11, 12

[8] ——, *On the lebesgue constant of a new type of $\mathcal{R}$-leja sequences*, tech. rep., ORNL/TM-2015/657, Oak Ridge National Laboratory., 2015. 12

[9] C. W. CLENSHAW AND A. R. CURTIS, *A method for numerical integration on an automatic computer*, Numerische Mathematik, 2 (1960), pp. 197–205. 11

[10] S. DE MARCHI, *On Leja sequences: some results and applications*, Applied Mathematics and Computation, 152 (2004), pp. 621–647. 12

[11] M. ELDRED, C. WEBSTER, AND P. CONSTANTINE, *Evaluation of non-intrusive approaches for wiener-askey generalized polynomial chaos*, in Proceedings of the 10th AIAA Non-Deterministic Approaches Conference, number AIAA-2008-1892, Schaumburg, IL, vol. 117, 2008, p. 189. 2

[12] L. FEJÉR, *On the infinite sequences arising in the theories of harmonic analysis, of interpolation, and of mechanical quadratures*, Bulletin of the American Mathematical Society, 39 (1933), pp. 521–534. 11

[13] D. GAMERMAN AND H. F. LOPES, *Markov chain Monte Carlo: stochastic simulation for Bayesian inference*, CRC Press, 2006. 4

[14] T. GERSTNER AND M. GRIEBEL, *Numerical integration using sparse grids*, Numerical algorithms, 18 (1998), pp. 209–232. 2

[15] ——, *Dimension–adaptive tensor–product quadrature*, Computing, 71 (2003), pp. 65–87. 2

[16] M. GRIEBEL, *Adaptive sparse grid multilevel methods for elliptic pdes based on finite differences*, Computing, 61 (1998), pp. 151–179. 2, 14, 21

[17] M. GUNZBURGER, C. TRENCHEA, AND C. WEBSTER, *A generalized stochastic collocation approach to constrained optimization for random data identification problems*, Tech. Rep. ORNL/TM-2012/185, Oak Ridge National Laboratory, 2012. 2

[18] M. GUNZBURGER, C. WEBSTER, AND G. ZHANG, *An adaptive wavelet stochastic collocation method for irregular solutions of partial differential equations with random input data*, Tech. Rep. ORNL/TM-2012/186, Oak Ridge National Laboratory, 2012. 2, 21

[19] P. JANTSCH AND C. G. WEBSTER, *Sparse grid quadrature rules based on conformal mappings*, Sparse Grids and Applications, - (to appear), pp. –. 21

[20] A. KLIMKE AND B. WOHLMUTH, *Algorithm 847: Spinterp: piecewise multilinear hierarchical sparse grid interpolation in matlab*, ACM Transactions on Mathematical Software (TOMS), 31 (2005), pp. 561–579. 2, 14

[21] X. MA AND N. ZABARAS, *An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations*, Journal of Computational Physics, 228 (2009), pp. 3084–3113. 2, 14

[22] F. NOBILE, R. TEMPONE, AND C. G. WEBSTER, *An anisotropic sparse grid stochastic collocation method for partial differential equations with random input data*, SIAM Journal on Numerical Analysis, 46 (2008), pp. 2411–2442. 2

[23] F. NOBILE, R. TEMPONE, AND C. G. WEBSTER, *A sparse grid stochastic collocation method for partial differential equations with random input data*, SIAM Journal on Numerical Analysis, 46 (2008), pp. 2309–2345. 2

[24] T. C. PATTERSON, *The optimum addition of points to quadrature formulae*, Mathematics of Computation, 22 (1968), pp. 847–856. 12

[25] S. A. SMOLYAK, *Quadrature and interpolation formulas for tensor products of certain classes of functions*, Dokl. Akad. Nauk SSSR, 4 (1963), pp. 240–243 (English translation). 2

[26] M. STOYANOV, *Hierarchy-direction selective approach for locally adaptive sparse grids*, tech. rep., ORNL/TM-2013/384, Oak Ridge National Laboratory., 2013. 2, 3, 14, 17, 51

[27] M. K. STOYANOV AND C. G. WEBSTER, *A dynamically adaptive sparse grid method for quasi-optimal interpolation of multidimensional analytic functions*, arXiv preprint arXiv:1508.01125, (2015). 2, 3, 8, 9, 11, 12, 13, 50

[28] W. SWELDENS AND P. SCHRÖDER, *Building your own wavelets at home*, in Wavelets in the Geosciences, Springer, 2000, pp. 72–107. 21

[29] J. A. VRUGT, C. TER BRAAK, C. DIKS, B. A. ROBINSON, J. M. HYMAN, AND D. HIGDON, *Accelerating markov chain monte carlo simulation by differential evolution with self-adaptive randomized subspace sampling*, International Journal of Nonlinear Sciences and Numerical Simulation, 10 (2009), pp. 273–290. 4

[30] J. A. VRUGT, C. J. TER BRAAK, M. P. CLARK, J. M. HYMAN, AND B. A. ROBINSON, *Treatment of input uncertainty in hydrologic modeling: Doing hydrology backward with markov chain monte carlo simulation*, Water Resources Research, 44 (2008). 4

[31] G. ZHANG AND M. GUNZBURGER, *Error analysis of a stochastic collocation method for parabolic partial differential equations with random input data*, SIAM Journal on Numerical Analysis, 50 (2012), pp. 1922–1940. 2

[32] G. ZHANG, M. GUNZBURGER, AND W. ZHAO, *A sparse grid method for multi-dimensional backward stochastic differential equaitons*, Journal of Computational Mathematics, 31 (2013), pp. 221–248. 2

[33] G. ZHANG, D. LU, M. YE, M. GUNZBURGER, AND C. WEBSTER, *An adaptive sparse-grid high-order stochastic collocation method for bayesian inference in groundwater reactive transport modeling*, Water Resources Research, 49 (2013), pp. 6871–6892. 4