ORNL/TM-2015/596

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

Miroslav Stoyanov

August 2018, version 6.0

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[27, 16, 24, 25, 23, 19, 22, 1, 3, 18, 11, 4, 15, 31, 29, 35, 36, 14]. 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 $k = 1, 2, \dots, d$, 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[29] and anisotropic quasi-optimal polynomial spaces[31].

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 §2.7.
- **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 §2.7.

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, 33, 34, 37].

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 §2.7 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 11 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

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

In the non-nested case, $X(\Theta)$ consists of pairs of multi-indexes $X(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \bigcup_{\boldsymbol{1}\leq\boldsymbol{j}\leq\boldsymbol{m}(\boldsymbol{i})} \{(\boldsymbol{i},\boldsymbol{j})\}$, 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 $\sum_{\boldsymbol{i}\leq\boldsymbol{j},\boldsymbol{j}\in\Theta(L)} t_{\boldsymbol{j}} = 1$ 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

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

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

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

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 $q : \mathbb{N} \to \mathbb{N}$

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

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

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

And for numerical quadrature

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

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

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

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 $\psi_{\boldsymbol{j}}^{\boldsymbol{i}}(\boldsymbol{x})$ are defined in (15) with the repeated points grouped together. The error in quadrature approximation is bounded as

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

and for the non-nested case the sum becomes $\sum_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)} |t_{\boldsymbol{i}} w_{\boldsymbol{j}}^{\boldsymbol{i}}|$ 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 $1 \leq k \leq d$, 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

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

log-corrected or *curved* selection[31]

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

[*] 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

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

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 [31] for details). Total degree space

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

or using a log-correction

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

or hyperbolic cross section space

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

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

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

or using a log-correction

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

or hyperbolic cross section space

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

For example, $\Theta^{\boldsymbol{1},q}(L)$ constructed according to (26) will result in $G_{\Theta^{\boldsymbol{1},q}(L)}$ that integrates exactly all polynomials of total degree up to and including L. Similarly, $\Theta^{\boldsymbol{1},-\frac{\boldsymbol{1}}{\boldsymbol{2}},m}(L)$ 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 [31] 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

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

Note that all sparse grids induced by nested one dimensional rules can be written in the Newton form above, but 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 $X(\Theta) = \Theta + \boldsymbol{1}$ 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 [31].

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 [31] 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, 31]
$\mathcal{R}$**-Leja odd:** Centered $\mathcal{R}$-Leja	$m(l) = 2l + 1$	$q(l) = m(l)$	see [7, 31]
$\mathcal{R}$**-Leja double 2:** Centered $\mathcal{R}$-Leja	see (33)	$q(l) = m(l)$	see [7, 31]
$\mathcal{R}$**-Leja double 4:** Centered $\mathcal{R}$-Leja	see (34)	$q(l) = m(l)$	see [7, 31]
$\mathcal{R}$**-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[26] 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

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

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

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

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

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

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 [31]. 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 Fourier Grids

For cases where the interpolant of $f(\boldsymbol{x})$ must be periodic in derivatives as well as function values, Tasmanian implements sparse interpolation with a Fourier basis, for more details see [17, 20, 28]. The one-dimensional (nested) rule assumes a canonical domain of $[0, 1]$ with the nodes

$$x_1 = 0, \qquad x_j = \frac{\left\lfloor \frac{3}{2}\left(j - 1 - 3^{\lfloor \log_3(j-1) \rfloor}\right) \right\rfloor}{3^{\lfloor \log_3(j-1)+1 \rfloor}}, \quad \text{for } j > 1.$$

Note that each level contains 3^l nodes, which allows us to preserve the nested structure, have levels with complete exponent (see below) and use radix-3 Fast-Fourier-Transform (FFT) algorithm. We define the exponential functions

$$\phi_j(x) = \exp\left(2\pi \mathcal{I} x (-1)^{j+1} \lfloor j/2 \rfloor\right),$$

where $\mathcal{I}$ is the unit complex number, i.e., $\mathcal{I}^2 = -1$. The interpolant is real values, which means that the effective basis functions at level l are

$$\left\{ \cos(2\pi\omega x), \sin(2\pi\omega x) \ : \ \omega \in \mathbb{Z}, \ |\omega| \leq \frac{3^l - 1}{2} \right\},$$

and the coefficients of the basis functions are computed using FFT.

Let $f : [0, 1] \to \mathbb{R}$ and consider the 1-D Fourier interpolant at level l, i.e., the interpolant $\mathcal{U}^{3^l}[f](x)$ matching $f(x)$ at nodes x_s for $s \leq 3^l$. The interpolant is given by

$$\mathcal{U}^{3^l}[f](x) = \sum_{j=1}^{3^l} \Re\left(\hat{f}_j \phi_j(x)\right)$$

where $\Re$ indicates the real part of a number and $\hat{f}_j$ is a special reordering of the discrete Fourier coefficients of reordered sequence $f(x_j)$ (i.e., the index has to be reordered twice). First, we define $f_i = f(x_s)$ such that for $i = 1, 2, \cdots, 3^l$ the corresponding x_s values are in ascending order, i.e., we f_i are the function values reordered in domain space according to x_s. Second, we take the discrete Fourier transform (using FFT algorithm) and obtain Fourier coefficients $\hat{f}_i$. Finally, the Fourier coefficients are reordered again to match the basis, i.e.,

$$\hat{f}_j = \hat{f}_i, \qquad \text{where } i = \begin{cases} \lfloor j/2 \rfloor, & (-1)^j < 0, \\ 3^l - \lfloor j/2 \rfloor, & (-1)^j > 0. \end{cases}$$

Internally, Tasmanian implements the re-indexing inline with negligible overhead, but it is noteworthy that there is no strong connection between the coefficient associated with $\phi_j(x)$ and the spacial node x_j, i.e., unlike other types of grids, the coefficient cannot be interpreted as hierarchical surplus.

Extending the one dimensional interpolant to multidimensional context is done analogously to the sparse grids construction with Global rules. We define

$$\boldsymbol{x_j} = \bigotimes_{k=1}^{d} x_{j_k}, \qquad \boldsymbol{\phi_j}(\boldsymbol{x}) = \prod_{k=1}^{d} \phi_{j_k}, \qquad \mathcal{U}^{\boldsymbol{i}} = \bigotimes_{k=1}^{d} \mathcal{U}^{3^{i_k}},$$

and a sparse interpolant

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

where Θ is some lower set and the weights $t_{\boldsymbol{i}}$ are the same as in (11).

At this point, the construction of a Fourier grid uses the same Θ as in the Global case using the same nomenclature as if the indexes correspond to a polynomial space. Rigorous study of the proper optimal power selection is underway, which will also allow for a proper refinement strategy.

2.7 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, 23, 22, 29].

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 2.9. 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

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

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

$$\phi_j^{(0)}(x) = \left\{ \begin{array}{ll} 1, & x \in (x_j - \Delta x_j, x_j + \Delta x_j) \\ 0, & x \notin (x_j - \Delta x_j, x_j + \Delta x_j) \end{array} \right.$$

For linear functions

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

and functions of arbitrary order $p > 1$

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

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 $\phi_{\boldsymbol{j}}(\boldsymbol{x})$

$$s_{\boldsymbol{j}} = f(\boldsymbol{x_j}) - \sum_{\boldsymbol{i} \in A_{\boldsymbol{j}}} s_{\boldsymbol{i}} \phi_{\boldsymbol{i}}(\boldsymbol{x_j}). \tag{40}$$

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

$$X = \{\boldsymbol{j} \in \mathbb{N}^d : h(\boldsymbol{j}) \leq L\}, \tag{41}$$

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

2.8 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

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

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

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

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

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

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

$$\hat{X} = X \bigcup \left(\bigcup_{\boldsymbol{j} \in B} \bigcup_{k \in \{1, 2, \dots, d\}} O_{\boldsymbol{j}}^{(k)} \right). \tag{42}$$

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

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

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.,

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

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

$$\hat{X} = X \bigcup \left(\bigcup_{\boldsymbol{j} \in B} \left(\bigcup_{k \in C_{\boldsymbol{j}} \cap T_{\boldsymbol{j}}} P_{\boldsymbol{j}}^{(k)} \right) \bigcup \left(\bigcup_{k \in C_{\boldsymbol{j}} \setminus T_{\boldsymbol{j}}} O_{\boldsymbol{j}}^{(k)} \right) \right). \tag{45}$$

For more details about the four refinement strategies see [29].

2.9 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

$$x_0 = 0, \qquad x_1 = -1, \qquad x_2 = 1, \qquad \text{for } j > 2 \quad x_j = (2j-1) \times 2^{-\lfloor \log_2(j-1) \rfloor} - 3, \tag{46}$$

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 (43) and (45) 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.

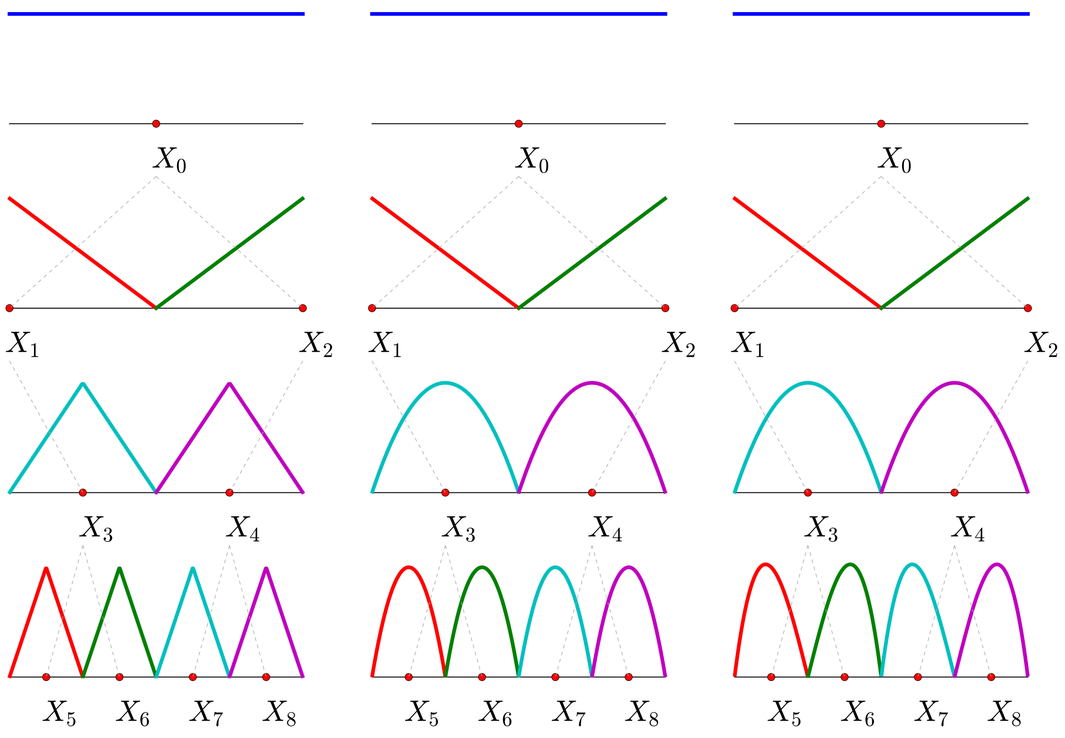


Figure 1. Local polynomial points (*rule_localp*) and functions, left to right: linear, quadratic, and cubic functions.

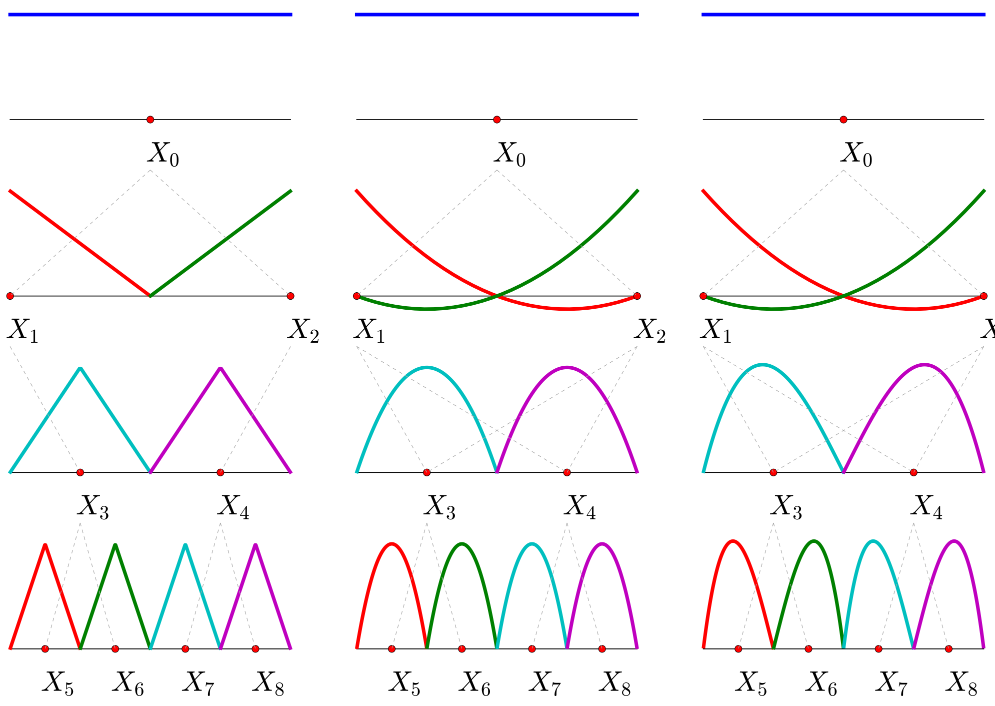


Figure 2. Semi-local polynomial points (*rule_semilocalp*) and functions, left to right: linear, quadratic, and cubic functions.

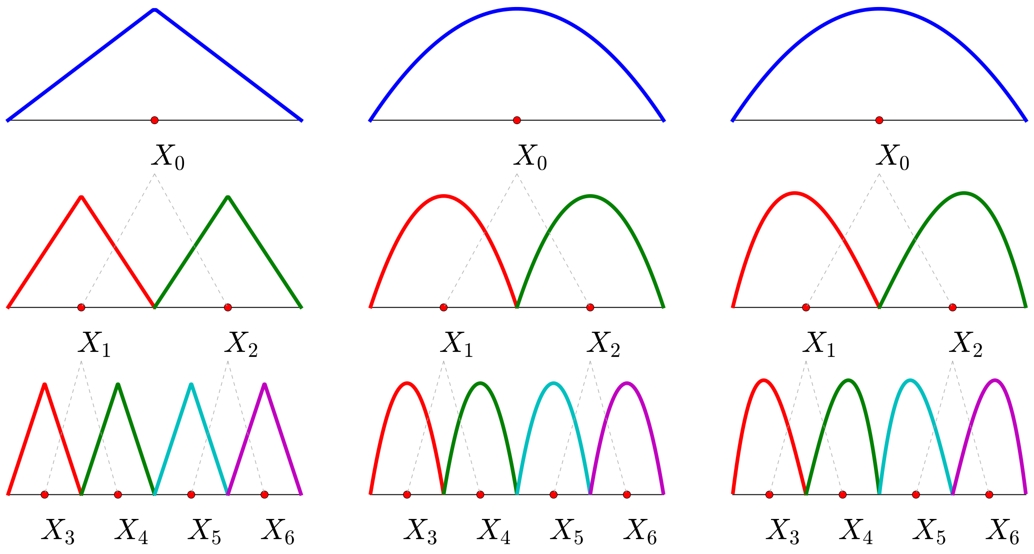


Figure 3. Semi-local polynomial points (*rule_localp0*) and functions, left to right: linear, quadratic, and cubic functions.

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

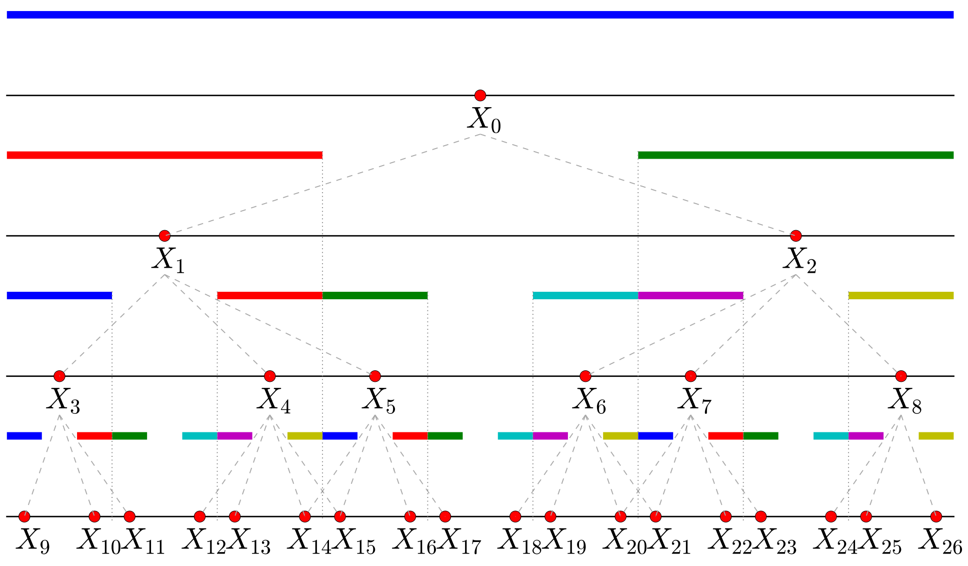
$$x_0 = 0, \qquad \text{for } j > 0 \quad x_j = (2j+3) \times 2^{-\lfloor \log_2(j+1) \rfloor} - 3, \tag{47}$$

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.

A rule with piece-wise constant (and discontinuous) basis is also provided within Tasmanian. Figure 4 shows the first four levels and the associated parents-offspring relations, see [30] for details.



X_0
X_1
X_2
X_3
X_4
X_5
X_6
X_7
X_8
X_9 X_{10} X_{11} X_{12} X_{13} X_{14} X_{15} X_{16} X_{17} X_{18} X_{19} X_{20} X_{21} X_{22} X_{23} X_{24} X_{25} X_{26}

Figure 4. Semi-local polynomial points (*rule_localp*) rule with order 0.

2.10 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 5.

- 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, 19, 32].

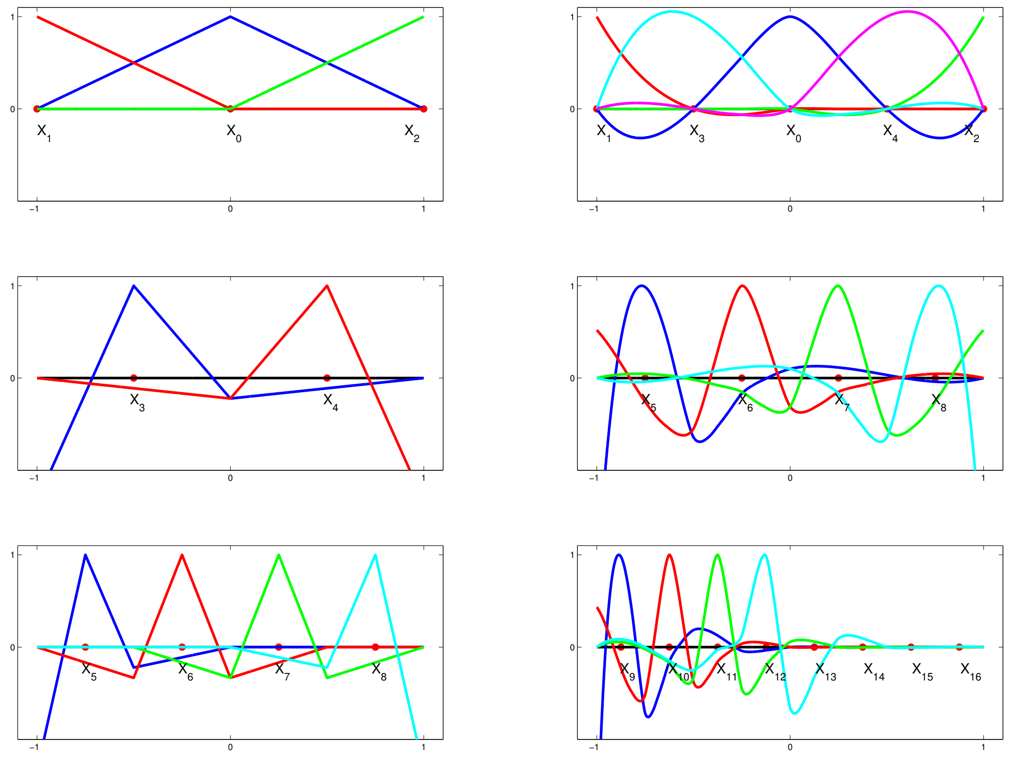


Figure 5. The first three levels for wavelets of order 1 (left) and 3 (right). The functions associated with x_{13}, x_{14}, x_{15}, and x_{16} are purposely omitted to reduce the clutter on the plot, since the funcitons are mirror images of the those associated with x_{12}, x_{11}, x_{10}, and x_9 respectively.

2.11 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.

2.11.1 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[21, 2].

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.

2.12 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

$$A\boldsymbol{c} = \boldsymbol{f}. \tag{48}$$

In the case of standard sparse grids construction with a nested rule, (48) 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 (48) 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

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

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

$$\boldsymbol{x}_{c,i+1} = \begin{cases} \boldsymbol{x}_{c,i} + \boldsymbol{g}_{c,i}, & \frac{\rho(\boldsymbol{x}_i + \boldsymbol{g}_{c,i})}{\rho(\boldsymbol{x}_i)} \geq u_i, \\ \boldsymbol{x}_{c,i} & \text{othwerwise.} \end{cases} \tag{49}$$

The updates are chosen as

$$\boldsymbol{g}_{c,i} = \gamma(\boldsymbol{x}_{c_1,i} - \boldsymbol{x}_{c_2,i}) + \boldsymbol{r}_{c,i}, \tag{50}$$

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	5, 6, 7, 8	any
clang	4.0, 5.0, 6.0	4.0 or 5.0
intel	18.0	18.0
cmake	3.5.1, 3.6.3, 3.7.2, 3.8.2, 3.9.6, 3.10.2, 3.11.4, 3.12	3.10.2
python	2.7, 3.5, 3.6	3.5 or 3.6
OpenBlas	0.2.18, 0.2.20	0.2.18 or 0.2.20
CUDA	7.5, 9.0, 9.1, 9.2	9.1 or 9.2
libiomp	5.0	5.0
MAGMA	2.3, 2.4	2.4

Table 5. Tested and recommended features.

4 Installation

The recommended way to install Tasmanian is to use `cmake`, which is supported for Linux, MacOSX and MS Windows. Under UNIX platforms (e.g., Linux and MacOSX), Tasmanian also comes with a convenient `install` script that wraps around `cmake`. If only basic Tasmanian install is desired, then simple GNU `make` script is also available, but note that acceleration features such as BLAS and CUDA are not supported through simple `make`. For more details regarding MS Windows install see §4.8.

Note: the install tree structure has changed in version 6.0. The content of `bin`, `lib` and `include` remain the same, but all Python and MATLAB files, environment setup scripts, and log files are now installed in `share/Tasmanian`. Thus, it is possible (albeit not recommended) to install Tasmanian directly into `/usr` or `/usr/local`.

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 and CTypes 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 basic python matplotlib support. The gfortran and ifort compilers are also supported for the Fortran module. Still in experimental phase are 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 or untar the archive which will create the `TASMANIAN-X.Y` folder, i.e., the root folder of all source code. From a terminal, run the included install script:

Executing the script manually:

```
cd TASMANIAN-6.0   (assuming TASMANIAN-6.0 is the folder with source code)
./install <install-path> <matlab-work-folder> <options>
```

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 folder.
- Using absolute paths for the install and MATLAB paths 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.
- A summary of all options is stored in `<install-path>/share/Tasmanian/Tasmanian.log`.
- The install script supports additional options, use the "`-help`" switch or see §4.2.1.

The `install` script will attempt to build the library with OpenMP, BLAS and Python support, in addition CUDA, Fortran, MAGMA and MPI capabilities can be enabled with the appropriate options (see below). Both static and shared libraries will be created and the `-O3` optimization flag will be used. Tasmanian supports both Python 2 and 3, and cmake will attempt to detect installed versions automatically (usually defaults to 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 Python, the corresponding option will be automatically disabled. At the end of the configure stage, cmake writes 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 enable/disable OpenMP, BLAS, CUDA, MAGMA, Python, shared and static libraries. Note that Python requires shared libraries. 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.

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

4.3 Installation folder structure

```
<install-path>/bin/                        (tagrid and tasdream executables)
<install-path>/lib/                        (shared and static libraries)
<install-path>/lib/Tasmanian/              (cmake package-config files)
<install-path>/lib/pythonX.Y/              (python module)
<install-path>/include/                    (headers .h and hpp, and Fortran .mod)
<install-path>/share/Tasmanian             (bash env scripts, install log, table)
<install-path>/share/Tasmanian/examples/  (reference examples)
<install-path>/share/Tasmanian/matlab/    (matlab scripts)
<install-path>/share/Tasmanian/python/    (sym-link to <install-path>/lib/pythonX.Y)
```

Additional notes:

- The Python module is version independent, i.e., the same file works with all tested versions, hence the version independent sym-link `share/Tasmanian/python` which allows to use the Python interface regardless of which version of Python was used during install.
- The default location of the `MATLAB` scripts has changed since version 5.1.
- Under MS Windows the shared library (e.g., the .dll files) are installed in `bin`.

4.4 Quick Build: `make`

```
cd TASMANIAN-6.0   (assuming TASMANIAN-6.0 is the folder with source code)
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)
```

Additional notes:

- Unzip/untar the archive from github will create folder `TASMANIAN-X.Y` where `X.Y` is the version number. Folder name is not important, it can be renames for convenience.
- The `make` command will install the libraries, executables, and Python module in `TASMANIAN-6.0`, and the headers in `TASMANIAN-6.0/include`.
- OpenMP is enabled by default under Linux, BLAS, CUDA and MPI could be enabled by editing `Config/Makefile.in`, but this is neither tested nor recommended. The correct way to enable BLAS, CUDA, and MPI is to use `cmake` or the `install` script.
- OpenMP is disabled by default under MacOSX, it can be enabled by editing `Config/Makefile.in` or using `cmake`.

- If Python is missing, then `make test` will fail. The manual C++ tests can be run with commands `./gridtest` and `./tasdream -test`.
- The `MATLAB` interface is set in `TASMANIAN-6.0/InterfaceMATLAB`, the `MATLAB` work folder will be set to `TASMANIAN-6.0/tsgMatlabWorkFolder/`, another folder can be used by manually editing the `tsgGetPaths.m` script, see §8.
- Fortran is currently tested with `gfortran` only, the .mod file will be installed alongside the headers.
- Multi-threaded build is supported, `make -j` is the command used in all Tasmanian testing.

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 Tasmanian_ENABLE_RECOMMENDED:BOOL=OFF \
 -D Tasmanian_ENABLE_OPENMP:BOOL=OFF \
 -D Tasmanian_ENABLE_BLAS:BOOL=OFF \
 -D Tasmanian_ENABLE_PYTHON:BOOL=OFF \
 -D Tasmanian_ENABLE_CUDA:BOOL=OFF \
 -D Tasmanian_ENABLE_MAGMA:BOOL=OFF \
 -D Tasmanian_MATLAB_WORK_FOLDER:PATH=<matlab-work-folder-path> \
 -D Tasmanian_ENABLE_FORTRAN:BOOL=OFF \
 -D Tasmanian_ENABLE_MPI:BOOL=OFF \
 <path-to>/TASMANIAN-6.0/
```

Standard `cmake` commands are accepted to select specific compiler and/or flags. By default both static and shared libraries are created, `BUILD_SHARED_LIBS` can be used to specify only shared or only static libraries.

```
 -D BUILD_SHARED_LIBS:BOOL=ON
```

Recommended features are OpenMP, BLAS, and Python. If using `Tasmanian_ENABLE_RECOMMENDED=ON`, Tasmanian will automatically search for available OpenMP, BLAS and Python, and enable the corresponding options. If any of those cannot be found, the build process will continue without the corresponding options. In addition, the `-O3` flag will be set for both C++ and Fortran.

The OpenMP option will enable multi-threading which is ubiquitous through Tasmanian and thus strongly recommended. However, if using `cmake` prior to version 3.9.6, the library dependence for OpenMP is not properly recorded in the `cmake` package-config (due to `cmake` limitations). See the installed example build script `share/Tasmanian/examples/CMakeLists.txt` for a work-around.

The Python option uses the `cmake` included command `find_package(PythonInterp)` and will also check for available `numpy` and `ctypes` modules. If multiple versions of Python are present, specific executable can be selected with

```
  -D PYTHON_EXECUTABLE:PATH=<path-to-python>
```

The BLAS option will use `find_package(BLAS)` to find suitable BLAS implementation. Specific libraries can be selected with

```
  -D BLAS_LIBRARIES:PATH=<path-to-blas>
```

The CUDA option will search for available installation of Nvidia CUDA with `find_package(CUDA)`. Additional CUDA kernels will be compiled and Tasmanian will link to Nvidia cuBlas and cuSparse. Specific CUDA installation can be selected with

```
  -D CUDA_TOOLKIT_ROOT_DIR:PATH=<path-to-cuda>
```

The MAGMA option will search for available installation of UTK MAGMA. Search will be performed for both shared and static libraries for both dense and sparse linear algebra. Using MAGMA within Tasmanian requires also enabling CUDA. Specific installation of MAGMA can be given with

```
  -D Tasmanian_MAGMA_ROOT_DIR:PATH=<path-to-magma>
```

Alternatively, variable `MAGMA_ROOT_DIR` is accepted without the `Tasmanian` prefix.

The MATLAB installation is enabled with `Tasmanian_MATLAB_WORK_FOLDER` (unlike version 5.1 there is no explicit enable matlab option). 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. See §8 for more information.

The Fortran option will search for a suitable Fortran 90/95 compiler and build a module interface to the C++ library. Currently, `gfortran` and `ifort` are tested for the corresponding versions of `gcc` and `icc`. Compiler can be manually specified with

```
  -D CMAKE_Fortran_COMPILER=<path-to-Fortran-compiler>
```

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

```
   -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>
```

Testing (by default) will use all available OpenMP threads and all available GPUs (if OpenMP and CUDA are enabled). Testing can be restricted to a specific number of threads or specific GPU with the options:

```
  -D Tasmanian_TESTS_OMP_NUM_THREADS=<number-of-threads-for-testing>
  -D Tasmanian_TESTS_GPU_ID=<cuda-id-for-the-gpu-to-use>
```

The C++ 2011 support is required, there is no option to enable/disable C++ 2011 since version 6.0.

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

- 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 extra options should not be needed under most circumstances.

4.6 Testing

The testing commands used by Tasmanian are

```
 ./SparseGrids/gridtest
 ./DREAM/tasdream -test
 ./Python/testTSG.py      (optional: only if python is enabled)
 ./Fortran/fortester      (optional: only if Fortran is enabled)
 ./test_post_install.sh
```

The first four commands will be called from `make test`. The last command will be called from `make test_install` which will make sure the executable, libraries, Python, Fortran, and CMake modules are properly installed.

Both `tasgream` and `gridtest` tests rely on random number generation for error checking (especially the DREAM module). Thus, the outcome of many 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. Tests can be manually invoked with the system time as a random seed:

```
 ./SparseGrids/gridtest random
 ./DREAM/tasdream -test random
```

to reset the random seed based on `time()`.

4.7 Package Config: Link to Tasmanian using `cmake`

Tasmanian version 6.0 comes with a `cmake` package-config file, which will be installed in `lib/Tasmanian`. The package-config will extract from the build process all the information needed by `cmake` to correctly link to the installed Tasmanian libraries. An external project can import Tasmanian `cmake` targets with the command:

```
find_package(Tasmanian 6.0.0 PATHS "<Tasmanian-install-path>")
```

The imported targets will be called:

```
Tasmanian_libsparsegrid_shared  Tasmanian_libsparsegrid_static
Tasmanian_libdream_shared       Tasmanian_libdream_static
Tasmanian_libfortran90_shared   Tasmanian_libfortran90_static
```

Depending on the selected options, not all targets may be available, e.g., using `BUILD_SHARED_LIBS=ON` will disable the static targets. In order to simplify the user code, the package-config will also create `cmake` interface targets without the shared/static suffixes:

```
Tasmanian_libsparsegrid    Tasmanian_libdream    Tasmanian_libfortran90
```

These interface targets will always depend on a valid imported target, and if both shared and static libraries are present, the static libraries will be chosen by default.

The Tasmanian package-config may also set `cmake` variable `Tasmanian_CXX_FLAGS`. Older versions of `cmake` do not set the OpenMP libraries as a dependence of a target build with the OpenMP flags; this, if Tasmanian was build with OpenMP support and an older version of `cmake`, the `Tasmanian_CXX_FLAGS` will be set to the OpenMP flags used by Tasmanian. A project linking to Tasmanian will need to also use the same compiler and the specified flags. If OpenMP is not enabled or if newer `cmake` is used, `Tasmanian_CXX_FLAGS` is not needed and the variable will not be defined.

See the installed `CMakeLists.txt` file in `share/Tasmanian/examples` for comments and proper use of the Tasmanian package-config.

4.8 Build on Windows using Mircosoft Visual C++ 2015 and 2017

Tasmanian 6.0 comes with full `cmake` support for MS Visual C++ 2015 and 2017, including BLAS, Python, MATLAB, and CUDA support. The recommended way of installing Tasmanian under Windows is to use the CMake GUI program to select the source, build, and install folder, as well as all relevant `cmake` options. The options work the same as under UNIX, and `Tasmanian_ENABLE_RECOMMENDED` will set optimization flags for both Release and Debug configurations. After `Configure` and `Generate` have been called from the CMake GUI, Tasmanian can be build, tested and installed with

```
cd <folder-to-build-binaries>               (folder is chosen in the CMake GUI)
cmake --build . --config Release            (compiles, can use Debug in place of Release)
ctest -C Release                            (run all tests for Release)
cmake --build . --config Release --target install (install Tasmanian)
```

Alternatively, the Visual Studio editor can open the project files `ALL_BUILD.vcxproj`, `RUN_TESTS.vcxproj` and `INSTALL.vcxproj`.

NOTE: in the current version of Tasmanian, the Python function `getGPUName()` does not work when CUDA is enabled (segfault is encountered). The corresponding C++ method works correctly, only the Python interface is affected. As this is a minor issue, it will be addressed at a later time, for now, do not use the faulty method.

Previous versions of Tasmanian included Windows .bat scripts that can build the libraries from the Development Prompt. The commands are:

```
WindowsMake.bat
WindowsMake.bat test
WindowsMake.bat clean (optional: reset the project)
```

Detailed instructions are given in the included `WindowsREADME.txt`, but the .bat scripts are now deprecated and will be removed in the next major release. Also note that the simple script is not compatible with CMake, i.e., executing `WindowsMake.bat` will alter the source code and may conflict with a CMake build.

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. By design, each object (instance of the class) is a stand-alone unit operating independently from other objects without the need for global library initialization.

5.1 Error Handling

The class has multiple overloaded interface functions, error checking (e.g., size of the input) is performed only on the interface using STL vectors. Most functions throw exceptions

```
std::invalid_argument("Message")
std::runtime_error("Message")
```

where `Message` is a human readable short explanation of the nature of the problem. Overall the error-checking interface is still incomplete, but functional and stable.

5.2 STL vector interface

The Tasmanian interface has been updated to accept STL vectors for all functions where previously arrays were uses. Overloaded functions can accept either all arrays or all vectors, where empty vectors are used in place of `NULL` pointers. There is no performance difference between arrays and vectors (they use identical algorithms at the back-end), but arrays offer opportunity for error checking, e.g., exceptions are thrown when the vector has incorrect size, as opposed to segfault errors in the case of arrays.

5.3 Default Constructor/Destructor TasmanianSparseGrid()

```
TasmanianSparseGrid();
~TasmanianSparseGrid();
```

The default constructor makes an empty grid which can access only the `make*Grid()` and `read()` methods (and of course the static members such as `getVersion()`. The destructor frees all resources.

5.4 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.

5.5 getLicense()

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

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

5.6 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, OpenMP will be used at all stages of the sparse grids, i.e., not just evaluations.

5.7 makeGlobalGrid()

```
void makeGlobalGrid( int dimensions,
                     int outputs,
                     int depth,
                     TypeDepth type,
                     TypeOneDRule rule,
                     const int *anisotropic_weights = 0,  (can be const std::vector<int> &)
                     double alpha = 0,
                     double beta = 0,
                     const char *custom_rule_filename = 0,
                     const int *level_limits = 0);        (can be const std::vector<int> &)
```

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

Note: the custom tabulated rule requires `custom_rule_file`, see below as well as Appendix 11.

`anisotropic` (`anisotropic_weights`) is either `NULL` (empty when vector) 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 11 for more information about the file format of the custom file.

`level_limits` is either `NULL` (empty when vector) 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,  (can be const std::vector<int> &)
                     const int *level_limits = 0 );       (can be const std::vector<int> &)
```

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`

Mathematically the Sequence and Global grids do not differ in properties; however, the Sequence grids use optimized internal data structures which leads to massive increase in speed when calling `evaluate*()` functions, at the expense of increased storage and increased cost of `loadNeededPoints()`. Run the sparse grid examples to see a simple benchmark.

5.9 makeFourierGrid()

```
void makeFourierGrid( int dimensions,
                      int outputs,
                      int depth,
                      TypeDepth type,
                      const int* anisotropic_weights = 0,  (can be const std::vector<int> &)
                      const int* level_limits = 0);        (can be const std::vector<int> &)
```

Creates a Fourier grid that uses trigonometric function basis and guarantees that the interpolant satisfies periodic boundary conditions.

`dimensions` same as `makeGlobalGrid()`

`outputs` same as `makeGlobalGrid()`

`depth` controls the density of the nodes in conjunction with `type`

`type` uses algorithms for tensor selection identical to the Global grids with frequency number used in place for polynomial order

`anisotropic` same as `makeGlobalGrid()`

`level_limits` same as `makeGlobalGrid()`

5.10 makeLocalPolynomialGrid()

```
void makeLocalPolynomialGrid( int dimensions,
                              int outputs,
                              int depth,
                              int order,
                              TypeOneDRule rule = rule_localp,
                              const int *level_limits = 0 ); (can be const std::vector<int> &)
```

Creates a grid based on one of the local hierarchical piece-wise polynomial rules described in section 2.7. 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 (41).

`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`, `rule_localpb`.

`level_limits` same as `makeGlobalGrid()`

5.11 makeWaveletGrid()

```
void makeWaveletGrid( int dimensions,
                      int outputs,
                      int depth,
                      int order = 1,
                      const int *level_limits = 0 );   (can be const std::vector<int> &)
```

Creates a grid based on local hierarchical wavelet basis, see 2.10.

`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.

5.12 updateGlobalGridGrid(), updateSequenceGrid()

```
void updateGlobalGrid( int depth,
                       TypeDepth type,
                       const int *anisotropic_weights = 0 ); (can be const std::vector<int> &)
void updateSequenceGrid( int depth,
                       TypeDepth type,
                       const int *anisotropic_weights = 0 ); (can be const std::vector<int> &)
```

The inputs a the same as in `makeGlobalGrid()/makeSequenceGrid()`, 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()/makeSequenceGrid()` 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 (i.e., set union) to the old index set. This corresponds to refinement with user specified `depth` and `anisotropic_weights`.

5.13 Copy Constructor, copyGrid()

```
  TasmanianSparseGrid(const TasmanianSparseGrid &source);
  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.

5.14 write()

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

Writes out the grid in either ASCII text or binary format to the `ofstream` or a file with given `filename`. The first function will open the file and call the second, while the second function allows for multiple grids to be stored in the same file.

5.15 read()

```
void read(const char* filename);
void read(std::ifstream &ifs, bool binary = false);
```

Reads a grid that has already been written to the file with `filename` or the stream. Runtime exception is raised is a problem is encountered with the file format. The first function automatically distinguishes between ASCII and binary formats.

5.16 setDomainTransform()

```
void setDomainTransform( const double a[],
                         const double b[] );
void setDomainTransform( const std::vector<double> &a,
                         const std::vector<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 and Fourier grids that default to $[0, 1]$). 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/vector of size `getNumDimensions()` that defines the a_k parameter associated with every direction.

`b` is an array/vector of size `getNumDimensions()` that defines the b_k parameter associated with every direction.

5.17 isSetDomainTransform()

```
bool isSetDomainTransform() const;
void clearDomainTransform();
void getDomainTransform( double a[], double b[] ) const;
void getDomainTransform( std::vector<double> &a, std::vector<double> &b ) const;
```

`isSet` returns `True` is a transform is set, `False` otherwise;

`clear` cancels the currently set transformation, note that this will effectively invalidate any point and values used by the grid (the points and values are still there, but the points will change and the values will not longer refelct the correct points);

`get` returns the currently loaded transform, the arrays must have size equal to `getNumDimensions()`, the vectors will be allocated to such size (if no transform has been set, the vectors will be empty).

5.18 getNumDimensions(), getNumOutputs()

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

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

5.19 getOneDRule()

```
TypeOneDRule getOneDRule() const;
const char* OneDimensionalMeta::getHumanString( TypeOneDRule rule );
```

Returns the value of the `rule` parameter in the `make***Glid()` function call, for a wavelet grids this returns `rule_wavelet` and Fourier grids return `rule_fourier`. The second function can be used to get a human readable description of any rule.

5.20 getCustomRuleDescription()

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

Returns the custom rule description string, see Appendix 11. If **rule** was not set to `rule_customtabulated`, then this function will return `NULL`.

5.21 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.

5.22 getOrder()

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

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

5.23 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.

5.24 get***Points()

```
double* getLoadedPoints() const;
void getLoadedPoints( double *x ) const;
void getLoadedPoints( std::vector<double> &x ) const;

double* getNeededPoints() const;
void getNeededPoints( double *x ) const;
void getNeededPoints( std::vector<double> &x ) const;

double* getPoints() const;
void getPoints( double *x ) const;
void getPoints( std::vector<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 as an array, it must be at least of size `getNumDimensions()` $\times$ `getNum***()`; vectors will be resized. The number of points corresponds to the output of `getNum***()`. 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.

5.25 getQuadratureWeights()

```
double* getQuadratureWeights() const;
void getQuadratureWeights( double weights[] ) const;
void getQuadratureWeights( std::vector<double> &weights ) const;
```

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

5.26 getInterpolationWeights()

```
double* getInterpolationWeights( const double x[] ) const;
void getInterpolationWeights( const double x[], double weights[] ) const;
void getInterpolationWeights( const std::vector<double> &x,
                              std::vector<double> &weights ) const;
```

Returns the interpolation 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 weights.

weights an array of size getNumPoints() (vectors get resized), weights returns 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.

5.27 loadNeededPoints()

```
void loadNeededPoints( const double vals[] );
void loadNeededPoints( const std::vector<double> &vals );
```

Provides the values of the function to be interpolated evaluated at the corresponding abscissas. The values are copied in the Tasmanian internal data structures. If getNumNeeded() is 0, this function will overwrite all currently stored values.

vals must be an array or vector of size getNumOutputs() $\times$ getNumNeeded() providing the values of the model at the needed points. If getNumNeeded() $= 0$, then the size must correspond to all currently stored data getNumOutputs() $\times$ getNumPoints(). The first getNumOutputs() entries correspond to the outputs of the interpolated function at the first grid point (either loaded or needed point). The second set of getNumOutputs() entries correspond to the second point and so on.

5.28 evaluate(), evaluateFast()

```
void evaluate( const double x[], double y[] ) const;
void evaluate( const std::vector<double> &x, std::vector<double> &y ) const;
void evaluateFast( const double x[], double y[] ) const;
void evaluateFast( const std::vector<double> &x, std::vector<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/vector of size getNumDimensions() that indicate the point where the interpolant should be evaluated.

y an already allocated array of size getNumOutputs() or a vector that will be resized. On exit, the entries of y will contain the values of the interpolant at the point x.

The Fast and not Fast versions give the same result, but Fast uses the enabled acceleration. The Fast function is potentially much faster, especially when working with models with many outputs, but Fast is not thread safe, i.e., if two threads simultaneously call evaluateFast() on the same object it will create a race condition. CUDA and GPU based evaluations are always thread unsafe, BLAS evaluations could be thread save depending on the BLAS implementation, e.g., OpenBLAS can be either safe or unsafe depending on compile flags. The conservative assumptions is that `evaluateFast()` is not thread safe.

Note: thread safety relates to calling two functions associated with the same object; in Tasmanian, function calls to different objects never create race conditions (unless such problems come from some of the third party libraries).

5.29 evaluateBatch()

```
void evaluateBatch(const double x[], int num_x, double y[]) const;
void evaluateBatch(const std::vector<double> &x, std::vector<double> &y) const;
```

Evaluate the approximation at multiple points with a single command.

- `x` an array/vector 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. In the vector interface, `num_x` is inferred from the size of `x`.
- `y` an already allocated array of size `getNumOutputs()` $\times$ `num_x` or vector that will be resized. 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 `Batch` function gives the same output as multiple calls to `evaluate()`, but Tasmanian will leverage OpenMP, CUDA kernels, and third-party libraries for accelerated matrix-matrix multiplication. The performance boost is massive even without GPU acceleration.

5.30 integrate()

```
void integrate( double y[] ) const;
void integrate( std::vector<double> &q ) const;
```

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

- `y` an already allocated array of size `getNumOutputs()` or a vector that will be resized. On exit, the entries of `y` are overwritten with the values of the integral of the interpolant over the domain.

5.31 is/Global/Sequence/Fourier/LocalPolynomial/Wavelet()

```
bool isGlobal() const;
bool isSequence() const;
bool isFourier() 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`.

5.32 is/set/get/clearConformalTransformASIN()

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

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 use in the truncation, i.e., truncation 4 will result in 7-th order polynomial ($\arcsin(x)$ has only odd powers in the series). **NOTE**: setting or clearing 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.

5.33 clear/getLevelLimits()

```
void clearLevelLimits();
void getLevelLimits( int *limits ) const;
void getLevelLimits( std::vector<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 function can be used to read the limits. The input array `limits` must have length `getNumDimensions()`, the vector will be resized.

5.34 setAnisotropicRefinement()

```
void setAnisotropicRefinement( TypeDepth type,
                               int min_growth,
                               int output,
                               const int *level_limits = 0 ); (can be const std::vector<int> &)
```

Implements the anisotropic refinement strategy described briefly in section 2.4 and in more details in [31]. 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, which is useful in controlling the number of points for the next iteration;

`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` sets new limits, by default (`NULL` or empty), the limits used by `make**Grid()` are used for all refinement calls.

5.35 setSurplusRefinement() - global version

```
void setSurplusRefinement( double tolerance, int output,
                           const int *level_limits = 0 );  (can be const std::vector<int> &)
```

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` sets new limits, by default (`NULL` or empty), the limits used by `make**Grid()` are used for all refinement calls.

5.36 setSurplusRefinement() - local version

```
void setSurplusRefinement( double tolerance,
                           TypeRefinement criteria,
                           int output,
                           const int *level_limits = 0,
                           const double *scale_correction = 0 );
void setSurplusRefinement( double tolerance,
                           TypeRefinement criteria,
                           int output,
                           const std::vector<int> &,
                           const std::vector<double> & );
```

Implements the surplus refinement strategy described briefly in section 2.8 and in more details in [29]. 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 (42), (43), (44), (45);

`criteria` specifies the refinement strategy

`refine_classic`, see (42)	`refine_direction_selective`, see (44)
`refine_parents_first`, see (43)	`refine_fds`, see (45)

`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;

`level_limits` sets new limits, by default (`NULL` or empty), the limits used by `make**Grid()` are used for all refinement calls;

`scale_correction` defines a non-negative number to multiply the corresponding hierarchical coefficient before comparing to the tolerance. This can be used to set/ignore refinement in certain regions based on specific problem interest or prior knowledge, as opposed to treating all coefficients identically. If all outputs are to be used, then `scale_correction` must have length `getNumPoints()` $\times$ `getNumOutputs()`, if only one output is to be used, then the length is only `getNumPoints()`.

5.37 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 to any of the set refinement functions. Note that `clearRefinement()` will have no effect if `getNumNeeded()` is zero, i.e., a refinement cannot be undone after `loadNeededPoints()` is called. The purpose of this function is to reduce the memory footprint of the grid in case the user decides not to ignore the last refinement.

5.38 mergeRefinement()

```
void mergeRefinement();
```

Every `set***Refinement()` function expands the grid to include a new set of points. However, the new points cannot be used unless Tasmanian has assess to the associated model values. The `mergeRefinement()` combines the old and new points into a single grid without using model values; instead, all old and new values are overwritten with zeros. 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 coefficients are set to zero.

5.39 getHierarchicalCoefficients()

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

Prior to Tasmanian 5.1, this function was called `getSurpluses()`. The function returns a pointer to the hierarchical coefficients associated with equation (1). Note that modifying the content of the pointer will result in undefined behavior.

The structure of the coefficients depends on the type of the grid:

Global The coefficients are just the loaded model values for all the grid points following the same order as in `getPoints()`.

Sequence The coefficients are the s_j coefficients at (29) and (30), there is one coefficient per point per output, the coefficients associated with the first point is at the first `getNumOutputs()` entries, the second point follows at the second set of `getNumOutputs()` entries, etc. The order of the points matches the order from `getNumPoints()`.

LocalPoly The coefficients are essentially the same as in (40). The order is identical to the `Sequence` case.

Wavelet The wavelet coefficients that solve the linear system of equations (i.e., the coefficients that guarantee that the interpolant will match the data at the sparse grid points). The order is the same as in the `Sequence` case.

Fourier The Fourier coefficients are not directly associated with sparse grids nodes, instead each coefficient is associated with a tensor of complex (as in complex numbers) exponential basis functions corresponding to a specific frequency. The Fourier coefficients are complex numbers and the format is that the first `getNumPoints()` $\times$ `getNumOutputs()` entries are the real values, while the second set contains the complex part (the total length of the array is twice the size of the other grids). The order of the coefficients will match the output of `evaluateHierarchicalFunctions()` and the real/complex split has been chosen for performance reasons.

5.40 evaluateHierarchicalFunctions()

```
void evaluateHierarchicalFunctions( const double x[], int num_x, double y[] ) const;
void evaluateHierarchicalFunctions( const std::vector<double> &x,
                                    std::vector<double> &y ) const;
```

x is an array of size `getNumDimensions()` $\times$ `num_x` indicating the points of interest where the functions should be evaluated. In the vector case, `num_x` is inferred from the size of `x`. The format of `x` is identical to the one used in `evaluateBatch()`;

y is a vector of size `num_x` $\times$ `getNumPoints()` where the values of the function basis corresponding to each `x` are stored.

For `Global` grids, 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 §2.12. In the case of Fourier grids, the basis values are actually complex numbers, the size of y is $2\times$ `num_x` $\times$ `getNumPoints()` and the real and complex parts are interlaced (as opposed to the block format of the hierarchical coefficients). Thus, the following is a valid data conversion:

```
grid.makeFourierGrid( ... )
std::vector<double> y, x = { ... };
grid.evaluateHierarchicalFunctions(x, y);
std::complex<double> *y_complex = (std::complex<double> *) y.data();
```

5.41 evaluateSparseHierarchicalFunctions()

```
void evaluateSparseHierarchicalFunctions( const double x[], int num_x,
                                          int* &pntr, int* &indx, double* &vals) const;
void evaluateSparseHierarchicalFunctions( const std::vector<double> &x,
                                          std::vector<int> &pntr,
                                          std::vector<int> &indx,
                                          std::vector<double> &vals ) const;
```

x is an array of size getNumDimensions() $\times$ num_x indicating the points of interest where the functions should be evaluated, in the vector case num_x is inferred from the size of x. The format of x is identical to the one used in evaluateBatch().

The matrix associated with the hierarchical basis, when working with local polynomial and wavelet grids, is usually sparse. The evaluateHierarchicalFunctions() function generates the dense representation of the matrix and evaluateSparseHierarchicalFunctions() generates the sparse representation. On exit, 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. Note: do not call this function for Global, Sequence or Fourier grids.

5.42 setHierarchicalCoefficients()

```
void setHierarchicalCoefficients( const double c[] );
void setHierarchicalCoefficients( const std::vector<double> &c );
```

The hierarchical coefficients are set to the ones specified by c. For all but Fourier grids, the size of c must be getNumPoints() $\times$ getNumOutputs() and the order is identical to that in loadNeededPoints() and getHierarchicalCoefficients(). In the Fourier case, the size of c is double, i.e., $2\times$ getNumOutputs() $\times$ getNumPoints(), where the first half of the entries correspond to the real values of the coefficients while the second corresponds to the complex part.

The function loadNeededPoints() sets the model values and computes the coefficients, the converse happens with setHierarchicalCoefficients(), the coefficients are loaded and the values are inferred.

5.43 Acceleration Functions

```
void enableAcceleration( TypeAcceleration acc );
TypeAcceleration getAccelerationType() const;
static bool isAccelerationAvailable( TypeAcceleration acc );
void favorSparseAcceleration( bool favor );
```

Tasmanian offers the following types of acceleration

accel_none	accel_gpu_magma
accel_gpu_cublas	accel_cpu_blas

`accel_gpu_cuda`	`accel_gpu_default`

All but `accel_none` require that Tasmanian is build with a corresponding CMake option. The `Available()` function can be used to check whether the option was used or not. The `enableAcceleration()` function can be called with any options, regardless of the CMake options. If the specified acceleration is not available, Tasmanian will default to the *next best options* in the following order:

`accel_gpu_magma` → `accel_gpu_cuda`	`accel_gpu_cublas` → `accel_cpu_blas`
`accel_gpu_cuda` → `accel_cpu_blas`	`accel_cpu_blas` → `accel_none`

The fallback options make it possible to use the same API calls to Tasmanian regardless of the CMake options, which allows for better performance portability. Since Tasmanian 6.0, `accel_gpu_default` is identical to `accel_gpu_magma` and falls-back to `cuda`, `blas` and finally `none`).

The names of the acceleration types relate to the algorithm and libraries used when calling `evaluateFast()` and `evaluateBatch()`. Evaluations proceed in two stages, first compute the values of the basis functions for the different points in `x`, then multiply the resulting matrix by the hierarchical coefficients. The first stage can be done either on the CPU (always using multi-threading with OpenMP, if available) or on the GPU with custom CUDA kernels. The second stage is a sparse or dense matrix multiplied by the dense matrix of the coefficients, which can be performed with any of the accelerated linear algebra libraries: BLAS, Nvidia cuBlas/cuSparse, or UTK MAGMA/MAGMA-sparse. Refer to Table 6 for details.

The acceleration used by the Local Polynomial grids have additional variations, since the back-end can use either sparse or dense algorithms. By default, Tasmanian will automatically choose a suitable algorithm where the decision is based on data collected from a large number of in-house tests. However, the decision does not take into account the memory usage (usually higher for dense matrices), and the tests were performed on Nvidia Tesla GPUs that come with very fast double-precision capabilities. Thus, the default Tasmanian decision may not be optimal for every situation, e.g., GPUs with less memory, or reduced double-precision capabilities, or new and untested architecture. Tasmanian includes a function that will overwrite the automatic process and always select either dense or sparse back-end methods, `favorSparseAcceleration()` can be called with either `True` (sparse) or `False` (dense). The function can be called repeatedly; calling `True` after `False` or `False` after `True` will reset the algorithm to auto, and calling `True` after `True` or `False` after `False` will have no effect. This function has no effect when called for grids that are not Local Polynomial.

The dense GPU algorithms will need enough GPU RAM to store three dense matrices with different sizes:

1. size `getNumOutputs()` × `getNumPoints()` for the loaded values;
2. size `getNumPoints()` × `num_x` for the values of the basis functions;
3. size `getNumOutputs()` × `num_x` for the output of `evaluateBatch()`.

The first matrix is the same for all calls to `evaluateFast()` and `evaluateBatch()`, thus the matrix will be kept persistently in memory unless another function is called to modify the values or coefficients (i.e., `loadNeededPoints()` or `setHierarchicalCoefficients()`) or the acceleration type of GPU-id are updated. The memory associated with the second and third matrices will be deleted after the call to evaluate. In addition, when using custom CUDA kernels for CUDA and MAGMA accelerations, the kernels need the

Acceleration	Stage 1	Stage 2
accel_none	CPU (OpenMP)	CPU (OpenMP)
accel_cpu_blas	CPU (OpenMP)	BLAS
accel_gpu_cublas	CPU (OpenMP)	cuBlas/cuSparse
accel_gpu_cuda	GPU (CUDA)	cuBlas/cuSparse
accel_gpu_magma	GPU (CUDA)	MAGMA

Table 6. Acceleration types and the corresponding algorithms.

input points x as well as some additional meta data required to form the second matrix. Nevertheless, the total memory usage is dominated by the three matrices.

The sparse GPU algorithms use a sparse presentation to the basis matrix. Depending on the grid structure, the sparse matrix could have dramatically lower memory footprint, but generally (in most cases) it is of order roughly $\log(N)^d \times$ num_x. However, because of limitations of the cuSparse API, the sparse algorithm requires another matrix of size equal to the output (third matrix) in order to compute explicit transpose. Thus, the CUDA and cuBlas acceleration modes could result in larger memory footprint, but MAGMA does not suffer from this limitation. Also note that transpose is very cheap operation and larger memory usage does not imply slower performance, the performance difference between sparse and dense comes from the number of non-zeros and the sparsity pattern.

5.44 set/set/GPUID()

```
void setGPUID( int in_gpuID );
int getGPUID() const;
static int getNumGPUs();
static int getGPUmemory( int gpu );
static char* getGPUname( int gpu );
```

Tasmanian can be used on a multi-GPU systems, where each Tasmanian object can be associated with a different GPU. The set and get functions allow for a GPU to be selected for each object. The static functions that check the number of available GPUs, the available memory, and the name, are just wrappers around the standard CUDA API (where GPUs are referred to as CUDA devices). The GPUs use zero-indexing, the memory reported is in megabytes (2^{20} bytes), and the name is null-terminated C style of string which must be deleted by the user. Note that each call to evaluateBatch() and evaluateFast() will also trigger cudaSetDevice() to the GPU set for the corresponding object.

5.45 evaluateHierarchicalFunctionsGPU(), evaluateSparseHierarchicalFunctionsGPU()

```
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;
```

Identical to `evaluateHierarchicalFunctions()` and `evaluateSparseHierarchicalFunctions()` with the difference that both input and output arrays reside on the GPU.

5.46 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.

5.47 printStats()

```
void printStats( std::ostream &os = std::cout ) const;
```

Prints short description of the sparse grid to the selected stream. The printed information is in human readable format.

5.48 getNeededIndexes() and getPointIndexes()

```
const int* getNeededIndexes() 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 `getPointIndexes()` returns an array with multi-indexes, for local polynomial and wavelet grids the function returns X, for Global, Sequence, and Fourier grids returns $X(\theta)$.

5.49 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 §11.

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();
    ~TasmanianDREAM();
    void overwriteBaseUnifrom( const BaseUniform *new_uniform );
```

Default constructor and a function to overwrite the default uniform random number generator, see §6.1.

6.2.2 Version Information

```
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 and 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 §50. 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 §50. The `setCorrectionAll()` function will set identical correction for all directions. The second function allows corrections to be set per-direction. The `TasmanianDREAM` class will hold an alias to the objects given here but will not call `delete` on the pointers.

6.2.8 collectSamples()

```
double* collectSamples( int num_burnup, int num_samples, bool useLogForm = false );
void collectSamples( int num_burnup, int num_samples,
                     double *samples, bool useLogForm = false );
void collectSamples( int num_burnup, int num_samples,
                     std::vector<double> &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 returned in the samples array/vector. Note that the total number of samples is `num_samples` $\times$ `num_chains`. The array has to be pre-allocated, the vector is resized.
`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 natural logarithm of the `setProbabilityWeightFunction()`, which could improve stability (depending on the problem).

The first sample is stored in the first `getNumDimensions()` entries, the second sample in the second set of `getNumDimensions()` entries, and so on.

6.2.9 setChainState()

```
void setChainState( const double* state );
void setChainState( const std::vector<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;
void getPDFHistory( std::vector<double> &history ) 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;
virtual void getDomainBounds( std::vector<bool> &lower, std::vector<bool> &upper ) = 0;
virtual void getDomainBounds( std::vector<double> &lower, std::vector<double> &upper ) = 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 vectors **must be** resized. 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.

Either the vector or array functions can be implemented in the custom class, if both are present, the DREAM will default to the vector implementation. Note that the boolean and double functions have to match, i.e., cannot mix boolean vectors and double arrays.

6.4.4 evaluate()

```
virtual void evaluate( int num_points, const double x[], double y[], bool useLogForm ) = 0;
virtual void evaluate( const std::vector<double> &x,
                       std::vector<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. The vector function infers `num_points` as the size of `x` divided by `getNumDimensions()`.

`x` is an array or vector 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 or vector that will contain the values of the unscaled pdf. The array will have sufficient size to fit all values, the vector **must be** resized to match `num_points`.

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

Either the vector or array functions can be implemented in the custom class, if both are present, the DREAM will default to the vector implementation.

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 );
    ~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 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 );
    PosteriorFromModel( const CustomModelWrapper *model );
    ~PosteriorFromModel();
    void overwritePDF( int dimension, BasePDF* pdf );
```

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 void getLikelihood( int num_model, const double *model,
                                std::vector<double> &likelihood,
                                int num_data = 0, const double *data = 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. Note that the `likelihood` vector must be resized.

6.8 class GaussianLikelihood

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

    void getLikelihood( int num_model, const double *model,
                        std::vector<double> &likelihood,
                        int num_data = 0, const double *data = 0,
                        bool useLogForm = true );
```

Defines the Gaussian likelihood currently included in Tasmanian.

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

For the constructor

- `outputs` is the number of outputs used by the model, i.e., the range of $f(\boldsymbol{x})$;
- `likelihood` (constructor) 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;
- `likelihood` (get function) is a vector that will be resized to `num_model` and will contain the values of the likelihood for each model output;

- 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 $\{d_1, \dots, d_s\}$ 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;
    virtual void evaluate(const std::vector<double> &x, std::vector<double> &y) const;
};
```

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.

The commands for `tasgrid` correspond to calls to the C++ API, where scalar inputs are given as command line arguments, and vector/array parameters are given as matrix files, see the end of this section for the matrix file format.

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. See the help subsection for how to find which command needs which options.

The `tasgrid` tool has some 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 Example commands

```
./tasgrid -mq -dim 4 -depth 2 -type qptotal -1d gauss-legendre -p
```

Make quadrature rule in 4 dimensions that can integrate exactly all quadratic polynomials, print the result to the screen. Note that the first column is the weight.

```
./tasgrid -mg -dim 3 -out 2 -depth 4 -type iptotal -1d clenshaw-curtis -gf example_grid_file
./tasgrid -l -gf example_grid_file -vf file_with_values
./tasgrid -e -gf example_grid_file -xf file_with_points -of result_file
```

Make global grid in 3 dimensions with clenshaw-curtis points that interpolates exactly all polynomials of order 4. The grid is stored in `example_grid_file`. On the second command, model values are read from the `file_with_values` and loaded into the grid. In the final command, the interpolant is evaluated at the points specified in `file_with_points` and the result is stored in the last file.

7.3 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.

7.4 Commands and C++ functions

```
./tasgrid -makeglobal         ->  makeGlobalGrid()
./tasgrid -makesequence       ->  makeSequenceGrid()
./tasgrid -makelocalpoly      ->  makeLocalPolynomialGrid()
./tasgrid -makewavelet        ->  makeWaveletGrid()
./tasgrid -makefourier        ->  makeFourierGrid()
./tasgrid -makequadrature     ->  (one of the grids above, see comments)
./tasgrid -makeupdate         ->  updateGlobalGrid()/updateSequenceGrid()
./tasgrid -setconformal       ->  setConformalTransformASIN()
./tasgrid -getquadrature      ->  getQuadratureWeights()/getPoints()
./tasgrid -getinterweights    ->  getInterpolationWeights()
./tasgrid -getpoints          ->  getPoints()
./tasgrid -getneededpoints    ->  getNeededPoints()
./tasgrid -loadvalues         ->  loadNeededPoints()
./tasgrid -evaluate           ->  evaluateBatch()
./tasgrid -evalhierarchyd     ->  evaluateHierarchicalFunctions()
./tasgrid -evalhierarchys     ->  evaluateSparseHierarchicalFunctions()
./tasgrid -integrate          ->  integrate()
./tasgrid -getanisotropy      ->  estimateAnisotropicCoefficients()
./tasgrid -refineaniso        ->  setAnisotropicRefinement()
./tasgrid -refinesurp         ->  setSurplusRefinement()
./tasgrid -refine             ->  setAnisotropicRefinement()/setSurplusRefinement()
./tasgrid -cancelrefine       ->  clearRefinement()
./tasgrid -mergerefine        ->  mergeRefinement()
./tasgrid -getcoefficients    ->  getHierarchicalCoefficients()
./tasgrid -setcoefficients    ->  setHierarchicalCoefficients()
./tasgrid -getpoly            ->  getGlobalPolynomialSpace()
./tasgrid -summary            ->  printStats()
./tasgrid <command> help -> show more info for this command
```

Additional notes:

- The domain types for all grids are set during the `make*` command, domains cannot be changed with the `tasgrid` executable since domain changes always change the nodes and effectively generates a new grid.
- Make quadrature creates a grid with zero outputs with types that is based on the one dimensional rule.
- The `-makeupdate` grid will automatically detect sequence or global grids.
- The `-getquadrature` command will generate larger matrix, where the first column is the weights and the rest correspond to the points.
- The `-getinterweights` command can work with multiple points at a time, the call will use OpenMP (if available).
- The `-refine` command will call anisotropic refinement on Global and Sequence grids, and surplus refinement otherwise.
- The coefficients and hierarchical functions for Fourier grids work with complex numbers, meaning that each pair of consecutive numbers correspond to one complex number (real and complex parts). This the matrices have twice as many columns. Note that this also applies to the coefficients as inputs and outputs (which differs from the C++ API).

- The `-evaluate` command accepts `-gpuid` options, which allows to select a CUDA device to use for acceleration. If the option is omitted, GPU acceleration will not be used.

7.5 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.

7.6 Command: -version or -info

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

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

7.7 Command: -test

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

Since Tasmanian 6.0 the sparse grids testing is moved to a different executable, i.e., `gridtest`. The test method of `tasgrid` is still included but it covers only a sub-set of the tests. Also, the tests take longer, especially when CUDA is enables, since large reference solutions have to be computed on the slow CPU. The `gridtest` executable takes the `random` and `verbose` switches, but does not need the `-test` command.

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.

7.8 Matrix File Format

The matrix files have two formats, binary and ascii. The simple text file describes a two dimensional array of real (double-precision) 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
```

In binary format, the file starts with three characters `TSG` indicating that this is a binary Tasmanian file. The characters are followed by two integers and the double-precision numbers that correspond to the matrix being read left-to-right top-to-bottom.

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 double-precision numbers that have integer values. The `-customrulefile` has special format is described in Appendix 11.

The default mode is to use binary files for all calls to `tasgrid`, but ASCII files are easier to debug and potentially easier to import to external codes. This `tasgrid` has an option `-ascii` that can be added to any command and will force the resulting output to be written in ASCII format.

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 List of MATLAB functions and corresponding C++ API

```
tsgCancelRefine.m                      ->  clearRefinement()
tsgEstimateAnisotropicCoefficients.m   ->  estimateAnisotropicCoefficients()
tsgEvaluateHierarchy.m                 ->  evaluateHierarchicalFunctions()
tsgEvaluate.m                          ->  evaluateBatch()
tsgGetHCoefficients.m                  ->  getHierarchicalCoefficients()
tsgGetInterpolationWeights.m           ->  getInterpolationWeights()
tsgGetNeededPoints.m                   ->  getNeededPoints()
tsgGetPoints.m                         ->  getPoints()
tsgGetPolynomialSpaces.m               ->  getGlobalPolynomialSpace()
tsgGetQuadrature.m                     ->  getQuadratureWeights()/getPoints()
tsgIntegrate.m                         ->  integrate()
tsgLoadHCoefficients.m                 ->  setHierarchicalCoefficients()
tsgLoadValues.m                        ->  loadNeededPoints()
tsgMakeFourier.m                       ->  makeFourierGrid()
tsgMakeGlobal.m                        ->  makeGlobalGrid()
tsgMakeLocalPolynomial.m               ->  makeLocalPolynomialGrid()
tsgMakeQuadrature.m                    ->  (see tasgrid -makequadrature)
tsgMakeSequence.m                      ->  makeSequenceGrid()
tsgMakeWavelet.m                       ->  makeWaveletGrid()
tsgMergeRefine.m                       ->  mergeRefinement()
tsgRefineAnisotropic.m                 ->  setAnisotropicRefinement()
tsgRefineSurplus.m                     ->  setSurplusRefinement()
tsgSummary.m                           ->  printStats()
```

The MATLAB functions wrap around `tasgrid`, thus what applies to one applies to the other. See the `tasgrid` notes about the order of complex Fourier coefficients and make quadrature.

8.2 function tsgCoreTests()

```
tsgCoreTests()
```

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

8.3 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.

8.4 functions tsgReadMatrix() and tsgWriteMatrix()

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

8.5 functions tsgCleanTempFiles()

Those functions are used internally to clean the temporary files.

8.6 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()`.

8.7 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()`.

8.8 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 8.15. 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> )`.

8.9 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.

8.10 function tsgWriteCustomRuleFile()

Writes a file with a custom quadrature or interpolation rule, see Appendix 11 and the function help for more details.

8.11 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.

8.12 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.

8.13 GPU acceleration

```
  lGrid = tsgMakeLocalPolynomial( \cdots )
  lGrid.gpuDevice = 0;
  result = tsgEvaluate(lGrid, \cdots)
```

If the `lGrid` object has a `gpuDevice` field, then the corresponding GPU will be used for the evaluations. Run `tsgCoreTest()` to see a list of detected CUDA devices.

8.14 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.

8.15 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 of Tasmanian. The C interface uses a series of functions that take a void pointer which 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 hash-bang 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 Fortran Interface

Tasmanian 6.0 comes with semi-stable Fortran 90/95 module that wraps around the C++ sparse grids library and gives access to all of the core functionality. The interface uses integers that reference entries of an array of pointers to `TasmanainSparseGrid` objects. A new ID is generated by `tsgNewGridID()`, which works similar to a C++ constructor, then the ID can be used in all follow on calls to the Fortran module. Once work is done with the grid, `tsgFreeGridID()` should be called to free all memory used by this grid. And `tsgClearAll()` can be called to delete all memory associated with all grids.

```
function tsgNewGridID() result(newid)
  integer :: newid
  ...
subroutine tsgFreeGridID(gridID)
  integer :: gridID
  ...
subroutine tsgClearAll()
```

The enumerate types used by the C++ interface are replaced by hard-coded constant integers, i.e., the module defines a parameter integer `tsg_clenshaw_curtis`, which can be used to call `tsgMakeGlobalGrid()`. All vectors are replaced by Fortran vectors and 2D matrixes, similar to the calls to `tasgrid`. However, note that Fortran is using column-major storage format, hence the row-column format is reversed to what you see in Python or `tasgrid` calls, i.e., when the C++ interface calls "the first point is stored in the first set of `getNumDimensions()` entries" that means the first column of the Fortran matrix.

All defined integer parameters start wit `tsg_` and all functions/subroutines start with `tsg`. A full list is provided at the top of `TasmanianSG.f90` file, at the moment the manual will not list all names as they are practically identical to the C++ interface.

The Fourier grids use complex numbers and hence special functions are provides:

```
tsgEvaluateComplexHierarchicalFunctions(...)
tsgGetComplexHierarchicalCoefficients(...)
tsgGetComplexHierarchicalCoefficientsStatic(...)
```

The interface comes with negligible overhead, except for the case of sparse hierarchical functions, where extra overhead is needed since there is no "clean" way to pass pointer to dynamic memory created in C++ to Fortran 90/95.

11 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 grids, the code will raise a run-time exception if the custom rule does not provide a sufficient number of points.
- The exactness constants are used only if `qptotal` or `qpcurved` types are used and the indexes of the polynomial space, i.e., `getPolynomialIndexes()`. If quadrature rules are 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 `setDomainTransform()`, 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 `setDomainTransform()` 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. 24

[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, 15, 22

[17] M. Griebel and J. Hamaekers, *Fast Discrete Fourier Transform on Generalized Sparse Grids*, Lecture Notes in Computational Science and Engineering, Springer International Publishing Switzerland, 2014, ch. 4, pp. 75–107. 14

[18] 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

[19] 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, 22

[20] K. Hallatschek, *Fouriertransformation auf dünnen gittern mit hierarchischen basen*, Numerische Mathematik, 63 (1992), pp. 83–97. 14

[21] P. Jantsch and C. G. Webster, *Sparse grid quadrature rules based on conformal mappings*, Sparse Grids and Applications, - (to appear), pp. –. 24

[22] 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, 15

[23] 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, 15

[24] 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

[25] 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

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

[27] 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

[28] F. Sprengel, *A class of periodic function spaces and interpolation on sparse grids*, Numerical Functional Analysis and Optimization, 21 (2000), pp. 273–293. 14

[29] M. Stoyanov, *Hierarchy-direction selective approach for locally adaptive sparse grids*, tech. rep., ORNL/TM-2013/384, Oak Ridge National Laboratory., 2013. 2, 3, 15, 18, 48

[30] M. Stoyanov, *Adaptive sparse grid construction in a context of local anisotropy and multiple hierarchical parents*, in Sparse Grids and Applications-Miami 2016, Springer, 2018, pp. 175–199. 21

[31] 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, 47

[32] W. SWELDENS AND P. SCHRÖDER, *Building your own wavelets at home*, in Wavelets in the Geosciences, Springer, 2000, pp. 72–107. 22

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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