**ORNL REPORT**

ORNL/TM-2015/596
Unlimited Release
Printed February 2017

# User Manual: TASMANIAN Sparse Grids v4.0

M. Stoyanov

Prepared by
Oak Ridge National Laboratory
One Bethel Valley Road, Oak Ridge, Tennessee 37831

ORNL/TM-2015/596

Computer Science and Mathematics Division

# USER MANUAL: TASMANIAN SPARSE GRIDS V4.0

M. Stoyanov [*]

Date Published: February 2017

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

[*]Computer Science and Mathematics Division, Oak Ridge National Laboratory, One Bethel Valley Road, P.O. Box 2008, MS-6367, Oak Ridge, TN 37831-6164 (`stoyanovmk@ornl.gov`).

## CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ABSTRACT

This documents serves to explain the functionality of the Sparse Grid module of the Toolkit for Adaptive Stochastic Modeling And Non-Intrusive Approximation (TASMANIAN). The document covers the three main components, the *libtasmaniansparsegrids* library, the *tasgrid* wrapper and the MATLAB and Python interfaces.

## ACKNOWLEDGEMENTS

The ORNL is operated by UT-Battelle, LLC, for the United States Department of Energy under Contract DE-AC05-00OR22725.

# 1 Quick Overview

**Sparse Grids** refers to a family of algorithms for approximation of multidimensional functions and integrals, where the approximation operator is constructed as a linear combination of tensors of multiple one dimensional operators [1–3, 9, 11–19, 21–23, 25, 26]. The TASMANIAN sparse grids library (henceforth called "TASMANIAN" or "the 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 $\Gamma$ 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

$$\tilde{f}(\boldsymbol{x}) = \sum_{i=1}^{N} c_i \phi_i(\boldsymbol{x}), \tag{1.1}$$

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

$$\tilde{f}(\boldsymbol{x}) = \sum_{i=1}^{N} \psi_i(\boldsymbol{x}) f(\boldsymbol{x}_i), \tag{1.2}$$

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 (1.1) and (1.2) 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 [22] and anisotropic quasi-optimal polynomial spaces [23].

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

$$\int_\Gamma f(\boldsymbol{x})\rho(\boldsymbol{x})d\boldsymbol{x} \approx Q[f] = \sum_{i=1}^{N} w_i f(\boldsymbol{x}_i), \tag{1.3}$$

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.
- **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.
- **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 §3.
- **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 §3.

The code consists of three main components:

- *libtasmaniansparsegrids.a*: the main component of TASMANIAN is the C++ 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 §6.

- *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 §7.
- *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 §8.

In the rest of this document, in §2 we provide a brief description of the construction of sparse grids from global rules, and in §3 we describe the local rules. In §4 we give a guide to compiling the C++ library and in §5 we describe the *TasmanianSparseGrid* class. In §6 we list the functions of the command line wrapper, and in §7 we describe the installation and usage of the MATLAB interface. Appendix A shows the format of a file with a user specified integration or interpolation rule.

# 2 Global Grids

## 2.1 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.,

$$\tilde{f}^{(l)}(x) = \mathcal{U}^{m(l)}[f](x) = \sum_{j=1}^{m(l)} f(x_j)\phi_j^l(x). \tag{2.1}$$

The corresponding numerical quadrature is given by

$$\int f(x)\rho(x)dx \approx \mathcal{Q}[f] = \sum_{j=1}^{m(l)} w_j^l f(x_j), \tag{2.2}$$

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

$$\Delta^{m(l)} = \mathcal{U}^{m(l)} - \mathcal{U}^{m(l-1)}, \qquad \text{or} \qquad \Delta^{m(l)} = \mathcal{Q}^{m(l)} - \mathcal{Q}^{m(l-1)} \tag{2.3}$$

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[1]. A sparse grid operator is defined as

$$G_{\Theta}[f] = \sum_{\boldsymbol{i}\in\Theta} \Delta^{\boldsymbol{m}(\boldsymbol{i})}, \tag{2.4}$$

where $\Theta$ is a lower set[2]. An explicit form of the points associated with the sparse grid can be obtained by first defining the tensors

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

[1]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 $0^0 = 1$.

[2]A set $\Lambda$ 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 (2.4) are given by

$$\{x_{\boldsymbol{j}}\}_{\boldsymbol{j}\in X(\Theta)}, \qquad \text{where} \quad X(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \{\mathbf{1} \leq \boldsymbol{j} \leq \boldsymbol{m}(\boldsymbol{i})\}. \tag{2.5}$$

In the non-nested case, $X(\Theta)$ consists of pairs of multi-indexes $X(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \bigcup_{\mathbf{1}\leq\boldsymbol{j}\leq\boldsymbol{m}(\boldsymbol{i})} \{(\boldsymbol{i}, \boldsymbol{j})\}$, 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 $\Theta$, there is a set of (integer) weights $\{t_{\boldsymbol{j}}\}_{\boldsymbol{j}\in\Theta(L)}$ that satisfy $\sum_{\boldsymbol{j}\leq\boldsymbol{i},\boldsymbol{j}\in\Theta(L)} t_{\boldsymbol{j}} = 1$ for every $\boldsymbol{i} \in \Theta(L)$, i.e., $t_{\boldsymbol{i}}$ solve a linear system of equations. Then,

$$G_{\Theta}[f] = \sum_{\boldsymbol{i}\in\Theta} \Delta^{\boldsymbol{m}(\boldsymbol{i})} = \sum_{\boldsymbol{i}\in\Theta} t_{\boldsymbol{i}} \mathcal{U}^{\boldsymbol{m}(\boldsymbol{i})}, \tag{2.6}$$

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

$$\phi_{\boldsymbol{j}}(\boldsymbol{x}) = \sum_{\boldsymbol{i}\in\Theta,\boldsymbol{m}(\boldsymbol{i})\geq\boldsymbol{j}} t_{\boldsymbol{i}} \prod_{k=1}^{d} \phi_{j_k}^{i_k}, \tag{2.7}$$

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 (1.2). Similarly, the quadrature weights are given by

$$w_{\boldsymbol{j}} = \sum_{\boldsymbol{i}\in\Theta,\boldsymbol{m}(\boldsymbol{i})\geq\boldsymbol{j}} t_{\boldsymbol{i}} \prod_{k=1}^{d} w_{j_k}^{i_k}. \tag{2.8}$$

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

$$\tilde{f}_{\Theta}(\boldsymbol{x}) = \sum_{\boldsymbol{j}\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}})\phi_{\boldsymbol{j}}(\boldsymbol{x}), \qquad Q_{\Theta}[f] = \sum_{\boldsymbol{j}\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}) w_{\boldsymbol{j}}. \tag{2.9}$$

For the non-nested case, we have

$$\tilde{f}_{\Theta}(\boldsymbol{x}) = \sum_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}^{\boldsymbol{i}}) t_{\boldsymbol{i}} \prod_{k=1}^{d} \phi_{j_k}^{i_k}, \qquad Q_{\Theta}[f] = \sum_{(\boldsymbol{i},\boldsymbol{j})\in X(\Theta)} f(\boldsymbol{x}_{\boldsymbol{j}}^{\boldsymbol{i}}) t_{\boldsymbol{i}} \prod_{k=1}^{d} w_{j_k}^{i_k}. \tag{2.10}$$

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 Approximation error

First we consider the polynomial space[3] 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 $q : \mathbb{N} \to \mathbb{N}$ 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

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

$q(l) = \frac{3}{2}m(l) - \frac{1}{2}$. 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

$$G_\Theta[p] = p, \qquad \text{for all} \quad p \in \mathcal{P}_{\Lambda^m(\Theta)}, \qquad \text{where} \quad \Lambda^m(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \{\boldsymbol{j} : \boldsymbol{j} \leq \boldsymbol{m}(\boldsymbol{i}) - \boldsymbol{1}\}. \tag{2.11}$$

And for numerical quadrature

$$G_\Theta[p] = p, \qquad \text{for all} \quad p \in \mathcal{P}_{\Lambda^q(\Theta)}, \qquad \text{where} \quad \Lambda^q(\Theta) = \bigcup_{\boldsymbol{i}\in\Theta} \{\boldsymbol{j} : \boldsymbol{j} \leq \boldsymbol{q}(\boldsymbol{i})\}^4. \tag{2.12}$$

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

Let $C^0(\Gamma)$ be the space of all continuous functions $f : \Gamma \to \mathbb{R}$ imbued with sup (or $L^\infty$) 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

$$\|f - G_\Theta[f]\|_{C^0(\Gamma)} \leq \left(1 + \|G_\Theta\|_{C^0(\Gamma)}\right) \inf_{p\in\mathcal{P}_{\Lambda^m(\Theta)}} \|f - p\|_{C^0(\Gamma)}, \tag{2.13}$$

where $\|G_\Theta\|_{C^0(\Gamma)}$ is the operator norm of $G_\Theta$ (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 (2.7) and (2.9), and for the non-nested case $\psi_{\boldsymbol{j}}^{\boldsymbol{i}}(\boldsymbol{x})$ are defined in (2.10) with the repeated points grouped together. The error in quadrature approximation is bounded as

$$\left| \int_\Gamma f(\boldsymbol{x})\rho(\boldsymbol{x})d\boldsymbol{x} - G_\Theta[f] \right| \leq \left( \int_\Gamma \rho(\boldsymbol{x})d\boldsymbol{x} + \sum_{\boldsymbol{j}\in X(\Theta)} |w_{\boldsymbol{j}}| \right) \inf_{p\in\mathcal{P}_{\Lambda^q(\Theta)}} \|f - p\|_{C^0(\Gamma)}, \tag{2.14}$$

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 $\Theta$ 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

$$\Theta^{\boldsymbol{\xi}}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{i} \leq L\}^5, \tag{2.15}$$

log-corrected or *curved* selection [23]

$$\Theta^{\boldsymbol{\xi},\boldsymbol{\eta}}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{i} + \boldsymbol{\eta} \cdot \log(\boldsymbol{i} + \boldsymbol{1}) \leq L\}^6, \tag{2.16}$$

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

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

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

hyperbolic cross section

$$\Theta^{\boldsymbol{\xi}}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : (\boldsymbol{i}+\boldsymbol{1})^{\boldsymbol{\xi}} \leq L\}. \tag{2.17}$$

Alternatively, the multi-index set $\Theta$ 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 [23] for details). Total degree space

$$\{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{j} \leq L\} \subset \Lambda^m(\Theta), \quad \Rightarrow \quad \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{m}(\boldsymbol{i}-\boldsymbol{1}) \leq L\}^{7}, \tag{2.18}$$

or using a log-correction

$$\begin{aligned} & \{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{j} + \boldsymbol{\eta} \cdot \log(\boldsymbol{j}+\boldsymbol{1}) \leq L\} \subset \Lambda^m(\Theta), \quad \Rightarrow \\ \Theta^{\boldsymbol{\xi},\boldsymbol{\eta},m}(L) \quad = \quad & \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{m}(\boldsymbol{i}-\boldsymbol{1}) + \boldsymbol{\eta} \cdot \log(\boldsymbol{m}(\boldsymbol{i}-\boldsymbol{1})+\boldsymbol{1}) \leq L\}, \end{aligned} \tag{2.19}$$

or hyperbolic cross section space

$$\{\boldsymbol{j} \in \mathbb{N}^d : (\boldsymbol{j}+\boldsymbol{1})^{\boldsymbol{\xi}} \leq L\} \subset \Lambda^m(\Theta), \quad \Rightarrow \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : (\boldsymbol{m}(\boldsymbol{i}-\boldsymbol{1})+\boldsymbol{1})^{\boldsymbol{\xi}} \leq L\}. \tag{2.20}$$

Tensor selection types (2.18), (2.19) and (2.20) 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

$$\{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{j} \leq L\} \subset \Lambda^q(\Theta), \quad \Rightarrow \quad \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{q}(\boldsymbol{i}-\boldsymbol{1}) + \boldsymbol{1} \leq L\}^{8}, \tag{2.21}$$

or using a log-correction

$$\begin{aligned} & \{\boldsymbol{j} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot \boldsymbol{j} + \boldsymbol{\eta} \cdot \log(\boldsymbol{j}+\boldsymbol{1}) \leq L\} \subset \Lambda^q(\Theta), \quad \Rightarrow \\ \Theta^{\boldsymbol{\xi},\boldsymbol{\eta},q}(L) \quad = \quad & \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{\xi} \cdot (\boldsymbol{q}(\boldsymbol{i}-\boldsymbol{1})+\boldsymbol{1}) + \boldsymbol{\eta} \cdot \log(q(\boldsymbol{i}-\boldsymbol{1})+\boldsymbol{2}) \leq L\}, \end{aligned} \tag{2.22}$$

or hyperbolic cross section space

$$\{\boldsymbol{j} \in \mathbb{N}^d : (\boldsymbol{j}+\boldsymbol{1})^{\boldsymbol{\xi}} \leq L\} \subset \Lambda^q(\Theta), \quad \Rightarrow \Theta^{\boldsymbol{\xi},m}(L) = \{\boldsymbol{i} \in \mathbb{N}^d : (\boldsymbol{q}(\boldsymbol{i}-\boldsymbol{1})+\boldsymbol{1})^{\boldsymbol{\xi}} \leq L\}. \tag{2.23}$$

For example, $\Theta^{\boldsymbol{1},q}(L)$ constructed according to (2.21) 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 [4]. For more information about optimal and quasi-optimal polynomial approximation see [23] and references therein.

## 2.3 Sequence Grid

A sequence grid is constructed from a one dimensional nested rule with $m(l) = l+1$. The theoretical properties, i.e., (2.13) and (2.14), are identical to the global grid, however, the sequence grid uses representation in terms of Newton (as opposed to Lagrange) polynomials. Let

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

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

[8] Here for notational convenience we assume that $q(-1) = -1$.

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

$$G[f](\boldsymbol{x}) = \sum_{\boldsymbol{j} \in X(\Theta)} s_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}), \tag{2.24}$$

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

$$\sum_{\boldsymbol{1} \leq \boldsymbol{j} \leq \boldsymbol{i}} s_{\boldsymbol{j}} \phi_{\boldsymbol{j}}(\boldsymbol{x}_{\boldsymbol{i}}) = f(\boldsymbol{x}_{\boldsymbol{i}}), \quad \text{for every } \boldsymbol{i} \in X(\Theta). \tag{2.25}$$

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

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

## 2.4 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 $\Theta$, 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 (2.25). The set $\Theta$ is then expanded with indexes that are "close" to the indexes associated with large relative surpluses. Specifically:

$$\hat{\Theta} = \Theta \bigcup \left( \bigcup_{\boldsymbol{j} \in X(\Theta), |s_{\boldsymbol{j}}| > \epsilon \cdot f_{\max}} \left\{ \boldsymbol{i} \in \mathbb{N}^d : \sum_{k=1}^{d} |i_k - j_k - 1| = 1 \right\} \right), \tag{2.26}$$

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 (2.18) or (2.19) 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 $f(\boldsymbol{x}_{\boldsymbol{j}})$. 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 [23].

## 2.5 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 [7] 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 [10] 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 [5]. Define $\{\theta_j\}_{j=1}^{\infty}$ as

$$\theta_1 = 0, \quad \theta_2 = \pi, \quad \theta_3 = \frac{\pi}{2}, \quad \text{for } j > 3,\ \theta_j = \begin{cases} \theta_{j-1} + \pi, & j \text{ is odd} \\ \frac{1}{2}\theta_{\frac{j}{2}+1}, & j \text{ is even} \end{cases} \tag{2.27}$$

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

$$m(0) = 1, \quad m(1) = 3, \quad \text{for } l > 1, \quad m(l) = 2^{\lfloor \frac{l}{2} \rfloor + 1}\left(1 + \frac{l}{2} - \left\lfloor \frac{l}{2} \right\rfloor\right) + 1, \tag{2.28}$$

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

$$m(l) = 1, \quad m(l) = 3, \quad \text{for } l > 1, \quad m(l) = 2^{\lfloor \frac{l-2}{4} \rfloor + 2}\left(1 + \frac{l-2}{4} - \left\lfloor \frac{l-2}{4} \right\rfloor\right) + 1, \tag{2.29}$$

where $\lfloor x \rfloor = \max\{z \in \mathbb{Z} : z \leq x\}$ is the *floor* function, see [23] for more details.

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

$$x_1 = -\frac{1}{2}, \quad x_2 = \frac{1}{2}, \quad \text{for } j > 2, \quad x_j = \begin{cases} \sqrt{\frac{1+x_{(j+1)/2}}{2}}, & j \text{ is odd} \\ -x_{j-1}, & j \text{ is even} \end{cases} \tag{2.30}$$

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 times requires additional parameters $\alpha$ and/or $\beta$. The Gauss rules also include the Hermite and Laguerre

| Points | $m(l)$ | $q(l)$ | Note: |
|---|---|---|---|
| **Chebyshev:** *rule_chebyshev*, *chebyshev* | | | |
| Non-nested Chebyshev roots | $m(l) = l + 1$ | $q(l) = l - 1 + (l \mod 2)$ | very low Lebesgue constant |
| **Clenshaw-Curtis:** *rule_clenshawcurtis*, *clenshaw-curtis* | | | |
| Nested Chebyshev roots | $m(0) = 1, m(l) = 2^l + 1$ | $q(l) = m(l)$ | very low Lebesgue constant |
| **Clenshaw-Curtis-Zero:** *rule_clenshawcurtis0*, *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:** *rule_fejer2*, *fejer2* | | | |
| Nested Chebyshev extrema | $m(l) = 2^{l+1} - 1$ | $q(l) = 2^l$ | no points placed at $\partial\Gamma$ |
| $\mathcal{R}$**-Leja:** *rule_rleja*, *rleja* | | | |
| See (2.27) | $m(l) = l + 1$ | $q(l) = l - 1 + (l \mod 2)$ | see [5, 23] |
| $\mathcal{R}$**-Leja odd:** *rule_rlejaodd*, *rleja-odd* | | | |
| Centered $\mathcal{R}$-Leja | $m(l) = 2l + 1$ | $q(l) = m(l)$ | see [5, 23] |
| $\mathcal{R}$**-Leja double 2:** *rule_rlejadouble2*, *rleja-double2* | | | |
| Centered $\mathcal{R}$-Leja | see (2.28) | $q(l) = m(l)$ | see [5, 23] |
| $\mathcal{R}$**-Leja double 4:** *rule_rlejadouble4*, *rleja-double4* | | | |
| Centered $\mathcal{R}$-Leja | see (2.29) | $q(l) = m(l)$ | see [5, 23] |
| $\mathcal{R}$**-Leja shifted:** *rule_rlejashifted*, *rleja-shifted* | | | |
| See (2.30) | $m(l) = l + 1$ | $q(l) = m(l) - 1$ | see [6] |
| $\mathcal{R}$**-Leja shifted even:** *rule_rlejashiftedeven*, *rleja-shifted-even* | | | |
| See (2.30) | $m(l) = 2(l + 1)$ | $q(l) = 2l + 1$ | see [6] |

Table 1: Summary of the available Chebyshev rules with the names used by the C++, *tasgird* and MATLAB interfaces.

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 [20] 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 [8], where

$$x_1 = 0, \qquad \text{for } j > 1 \quad x_{j+1} = \operatorname*{argmax}_{x \in [-1,1]} \prod_{i=1}^{j} \left| x - x_i \right|. \tag{2.31}$$

| Canonical Integral | Generalized Integral | Notes |
|---|---|---|
| **Gauss-Patterson:** *rule_gausspatterson*, *gauss-patterson* | | |
| $\int_{-1}^{1} f(x)dx$ | $\int_a^b f(x)dx$ | The only nested rule, see paragraph above |
| **Gauss-Legendre:** *rule_gausslegendre*, *gauss-legendre*, *rule_gausslegendreodd*, *gauss-legendre-odd* | | |
| $\int_{-1}^{1} f(x)dx$ | $\int_a^b f(x)dx$ | |
| **Gauss-Chebyshev type 1:** *rule_gausschebyshev1*, *gauss-chebyshev1*, *rule_gausschebyshev1odd*, *gauss-chebyshev1-odd* | | |
| $\int_{-1}^{1} f(x)(1-x^2)^{-0.5}dx$ | $\int_a^b f(x)(b-x)^{-0.5}(x-a)^{-0.5}dx$ | |
| **Gauss-Chebyshev type 2:** *rule_gausschebyshev2*, *gauss-chebyshev2*, *rule_gausschebyshev2odd*, *gauss-chebyshev2-odd* | | |
| $\int_{-1}^{1} f(x)(1-x^2)^{0.5}dx$ | $\int_a^b f(x)(b-x)^{0.5}(x-a)^{0.5}dx$ | |
| **Gauss-Gegenbauer:** *rule_gaussgegenbauer*, *gauss-gegenbauer*, *rule_gaussgegenbauerodd*, *gauss-gegenbauer-odd* | | |
| $\int_{-1}^{1} f(x)(1-x^2)^{\alpha}dx$ | $\int_a^b f(x)(b-x)^{\alpha}(x-a)^{\alpha}dx$ | Must specify $\alpha$ |
| **Gauss-Jacobi:** *rule_gaussjacobi*, *rule_gaussjacobiodd*, *gauss-jacobi*, *gauss-jacobi-odd* | | |
| $\int_{-1}^{1} f(x)(1-x)^{\alpha}(1+x)^{\beta}dx$ | $\int_a^b f(x)(b-x)^{\alpha}(x-a)^{\beta}dx$ | Must specify $\alpha$, $\beta$ |
| **Gauss-Laguerre:** *rule_gausslaguerre*, *rule_gausslaguerreodd*, *gauss-laguerre*, *gauss-laguerre-odd* | | |
| $\int_0^{\infty} f(x)x^{\alpha}e^{-x}dx$ | $\int_a^{\infty} f(x)(x-a)^{\alpha}e^{-b(x-a)}dx$ | Must specify $\alpha$ |
| **Gauss-Hermite:** *rule_gausshermite*, *gauss-hermite*, *rule_gausshermiteodd*, *gauss-hermite-odd* | | |
| $\int_{-\infty}^{\infty} f(x)x^{\alpha}e^{-x^2}dx$ | $\int_{-\infty}^{\infty} f(x)(x-a)^{\alpha}e^{-b(x-a)^2}dx$ | Must specify $\alpha$ |

Table 2: Summary of the available Chebyshev rules with the names used by the C++, *tasgird* and MATLAB interfaces.

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

$$x_1 = 0, \qquad \text{for } j > 1 \quad x_{j+1} = \operatorname*{argmax}_{x\in[-1,1]} \sum_{j'=1}^{j} \prod_{i=1, i\neq j'}^{j} \left|\frac{x - x_i}{x_{j'} - x_i}\right|. \tag{2.32}$$

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

$$x_{j+1} = \operatorname*{argmin}_{x\in[-1,1]} \max_{y\in[-1,1]} \prod_{i=1}^{j} \left|\frac{y - x_i}{x - x_i}\right| + \sum_{j'=1}^{j} \left|\frac{y - x}{x_{j'} - x}\right| \prod_{i=1, i\neq j'}^{j} \left|\frac{y - x_i}{x_{j'} - x_i}\right| \tag{2.33}$$

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

$$x_{j+1} = \operatorname*{argmin}_{x\in[-1,1]} \max_{y\in[-1,1]} \left(1 + \sum_{i=1}^{j} \prod_{j'=1, j'\neq i}^{j} \left|\frac{x - x_{j'}}{x_i - x_{j'}}\right|\right) \prod_{j'=1}^{j} \left|\frac{y - x_{j'}}{x - x_{j'}}\right|. \tag{2.34}$$

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 [23]. 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.

| Points | $m(l)$ | Points | $m(l)$ |
|---|---|---|---|
| **Leja:** *rule_leja*, *leja* | | **Leja odd:** *rule_lejaodd*, *leja-odd* | |
| See (2.31) | $m(l) = l + 1$ | See (2.31) | $m(l) = 2l + 1$ |
| **Max-Lebesgue:** *rule_maxlebesgue*, *max-lebesgue* | | **Max-Lebesgue odd rules:** *rule_maxlebesgueodd*, *max-lebesgue-odd* | |
| See (2.32) | $m(l) = l + 1$ | See (2.32) | $m(l) = 2l + 1$ |
| **Min-Lebesgue:** *rule_minlebesgue*, *min-lebesgue* | | **Min-Lebesgue odd rules:** *rule_minlebesgueodd*, *min-lebesgue-odd* | |
| See (2.33) | $m(l) = l + 1$ | See (2.33) | $m(l) = 2l + 1$ |
| **Min-Delta:** *rule_mindelta*, *min-delta* | | **Min-Delta odd rules:** *rule_deltaodd*, *min-delta-odd* | |
| See (2.34) | $m(l) = l + 1$ | See (2.34) | $m(l) = 2l + 1$ |

Table 3: Summary of the available greedy sequence rules with the names used by the C++, *tasgird* and MATLAB interfaces.

# 3 Local Polynomial Grids

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 [13, 16, 17, 22].

## 3.1 Hierarchical interpolation rule

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 3.3. 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

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

For linear functions

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

and functions of arbitrary order $p > 1$

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

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

We extend the one dimensional hierarchy to a $d$-dimensional context using multi-index notation[9]

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

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

$$P_j^{(k)} = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{i} \underset{k}{=} \boldsymbol{j}^{10} \text{ and } i_k \in P_{j_k}\} \qquad O_j^{(k)} = \{\boldsymbol{i} \in \mathbb{N}^d : \boldsymbol{i} \underset{k}{=} \boldsymbol{j} \text{ and } i_k \in O_{j_k}\}$$

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

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

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

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

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

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

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

$$X = \{\boldsymbol{j} \in \mathbb{N}^d : h(\boldsymbol{j}) \leq L\}, \tag{3.1}$$

for some use specified $L$.

## 3.2 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 $\Gamma$ 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

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

[9]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.

[10]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

where $\epsilon > 0$ is desired tolerance and $f_{\max} = \max_{\boldsymbol{i} \in X} |f(\boldsymbol{x_i})|$. 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 $\epsilon$. TASMANIAN implements $4$ different refinement strategies, where $\hat{X}$ is selected by including parents and/or children of $\boldsymbol{j} \in B$ in different directions. This is done based on consideration of "orphan" directions and directional surpluses.

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

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

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

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

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

$$\hat{X} = X \bigcup \left( \bigcup_{\boldsymbol{j} \in B} \bigcup_{k \in \{1,2,\ldots,d\}} O_{\boldsymbol{j}}^{(k)} \right). \tag{3.2}$$

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

$$\hat{X} = X \bigcup \left( \bigcup_{\boldsymbol{j} \in B} \left( \bigcup_{k \in T_{\boldsymbol{j}}} P_{\boldsymbol{j}}^{(k)} \right) \bigcup \left( \bigcup_{k \notin T_{\boldsymbol{j}}} O_{\boldsymbol{j}}^{(k)} \right) \right). \tag{3.3}$$

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.,

$$\hat{X} = X \bigcup \left( \bigcup_{\boldsymbol{j} \in B} \bigcup_{k \in C_{\boldsymbol{j}}} O_{\boldsymbol{j}}^{(k)} \right). \tag{3.4}$$

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

$$\hat{X} = X \bigcup \left( \bigcup_{\boldsymbol{j} \in B} \left( \bigcup_{k \in C_{\boldsymbol{j}} \cap T_{\boldsymbol{j}}} P_{\boldsymbol{j}}^{(k)} \right) \bigcup \left( \bigcup_{k \in C_{\boldsymbol{j}} \setminus T_{\boldsymbol{j}}} O_{\boldsymbol{j}}^{(k)} \right) \right). \tag{3.5}$$

For more details about the four refinement strategies see [22].

## 3.3 One dimensional rules

TASMANIAN implements three specific one dimensional hierarchical rules: standard rule with $\Delta x_j$ 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

$$x_0 = 0, \qquad x_1 = -1, \qquad x_2 = 1, \qquad \text{for } j > 2 \quad x_j = (2j-1) \times 2^{-\lfloor \log_2(j-1) \rfloor} - 3, \tag{3.6}$$

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 $\Delta x_j$ 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 (3.3) and (3.5) 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.

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

$$x_0 = 0, \qquad\qquad \text{for } j > 0 \quad x_j = (2j+3) \times 2^{-\lfloor \log_2(j+1) \rfloor} - 3, \tag{3.7}$$

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 $\Delta x_j$ is given by $\Delta x_0 = 2^{-h(j)}$. Figure 3 shows the first three levels of the linear, quadratic, and cubic functions.

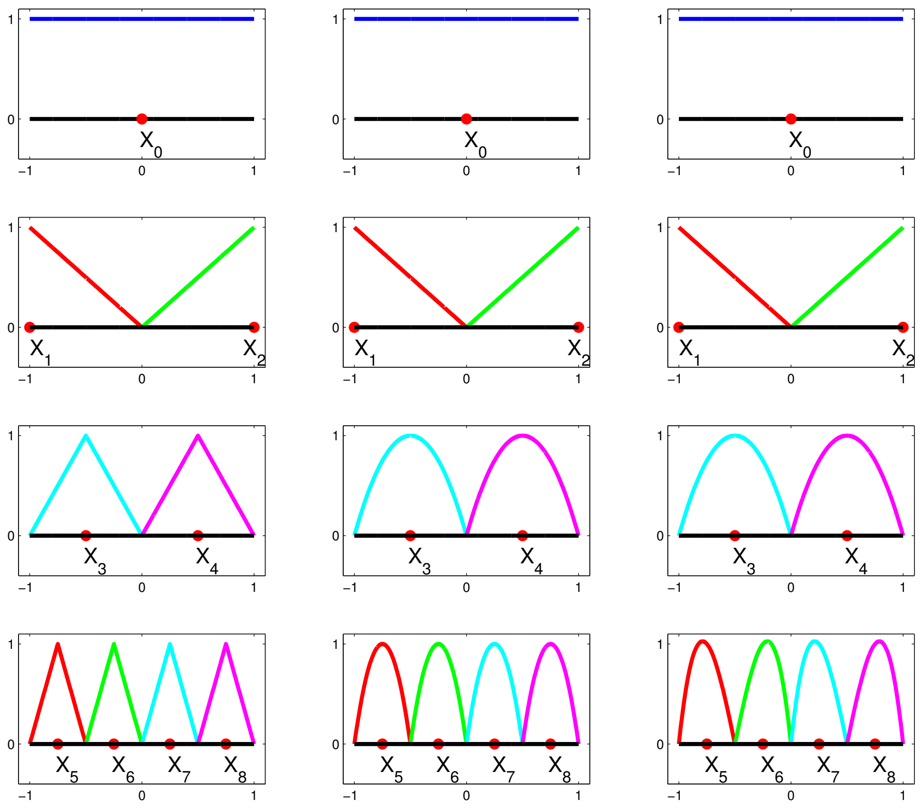

Figure 1: Local polynomial points (*rule_localp*) and functions, left to right: linear, quadratic, and cubic functions.

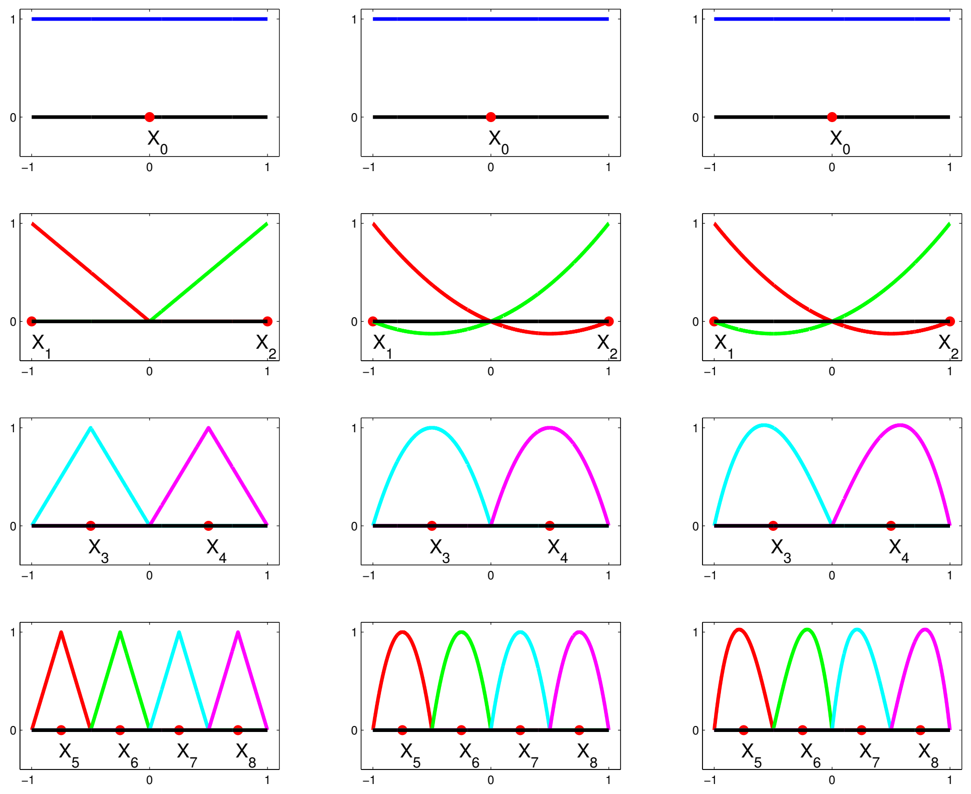

Figure 2: Semi-local polynomial points (*rule_semilocalp*) and functions, left to right: linear, quadratic, and cubic functions.

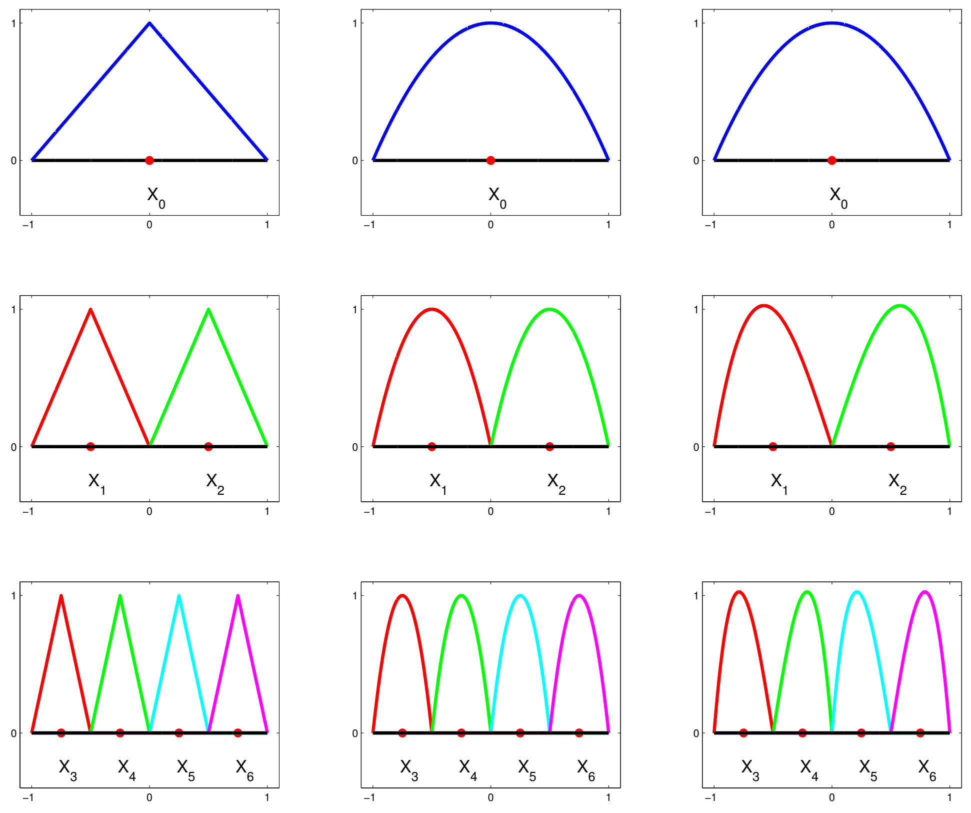

Figure 3: Semi-local polynomial points (*rule_localp0*) and functions, left to right: linear, quadratic, and cubic functions.

## 3.4 Wavelets

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

- The zeroth levels of wavelet rules of order $1$ and $3$, have $3$ and $5$ points respectively. This is a sharp contrast to the single point of of the polynomial rules, since level $0$ wavelet grid has $3^d$ (or $5^d$) points in $d$-dimensions (as opposed to a single point). See Figure 4.
- Wavelet rules have larger Lebesgue constant, which is due to the large magnitude of the boundary wavelet functions. This can lead to instability of the wavelet interpolant around the boundary of the domain.
- The linear system of equations associated with the wavelet surpluses is not triangular, hence a sparse matrix has to be inverted every time values are loaded into the interpolant. This leads to a significantly higher computational cost in manipulating the wavelet grids, especially in loading values and performing direction selective refinement.
- Wavelets form a Riesz basis, which over-simplistically means that the wavelet surpluses are much sharper indicators of the local error and hence wavelet based refinement strategy “could” generate a grid that is more accurate and has fewer points. The quotations around the word “could” relate to the point about the Lebesgue constant.
- For more details about wavelets, see [13, 15, 24].

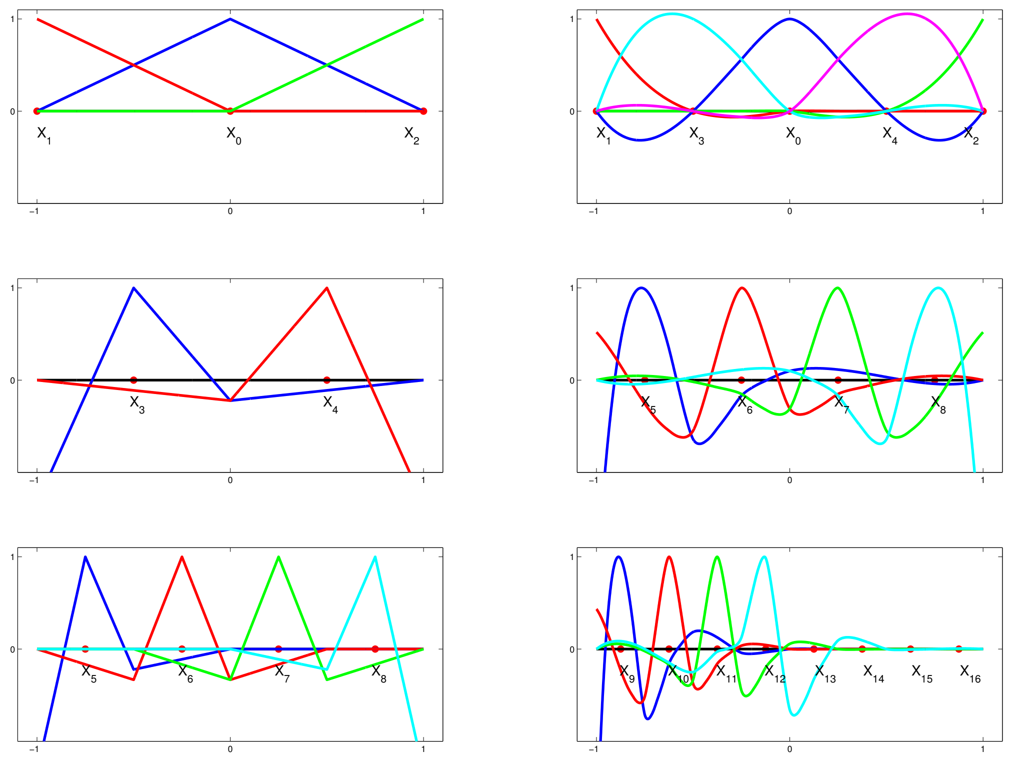

Figure 4: The first three levels for wavelets of order 1 (left) and 3 (right). The functions associated with $x_{13}$, $x_{14}$, $x_{15}$, and $x_{16}$ are purposely omitted to reduce the clutter on the plot, since the funcitons are mirror images of the those associated with $x_{12}$, $x_{11}$, $x_{10}$,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

# 4 Compilation

## 4.1 Build system: `cmake`

Starting with version 4.0, TASMANIAN supports cmake build engine, which is the new preferred way of compiling the library. The old GNU-Make will be supported for the foreseeable future.

Example of the cmake command would be:

```
cmake \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=<prefix-path> \
  -DENABLE_PYTHON=ON \
  -DENABLE_MATLAB=ON \
  -DMATLAB_WORK_FOLDER=<path-to-work-folder> \
  -DENABLE_OPENMP=ON \
  <path-to-source-code>
```

The accepted build commands are the standard:

```
make
make test
make install
```

The installer creates sub-folders in the install prefix using standard convention:

|  |  |
|---|---|
| `bin` | contains the tasgrid executable |
| `lib` | contains *libtasmaniansparsegrids.a* and *libtasmaniansparsegrids.so* corresponding to the shared and static versions of the library. |
| `include` | contains the C++ headers, ending in .hpp, and the C header TasmanianSparseGrid.h |
| `example` | contains the C++ example with the corresponding Make script. In addition, if Python is enabled, this also contains Python example. |
| `matlab` | contains the files needed by the MATLAB interface, a number of .m files |
| `python` | contains the files needed by the Python interface, a single Python module file |

## 4.2 Old build system: Unix based systems (Linux/MacOSX)

### Quick Build

In a shell inside the folder with the source files, type

```
make
```

The code doesn't require any external libraries and uses the simple GNU-Make engine. Hence, it will most likely compile just fine.

To verify the build you should run

```
./tasgrid -test
```

and make sure all the test pass. See Section 6 for more details.

### Advanced Build Options

Open the *Makefile* in an editor and adjust the options.

**CC** specifies the compiler command. The code was written for the GNU C++ compiler (GCC). The default command is *g++*, however, that can be changed to force a specific version of the compiler or even a different compiler.

**OpenMP** is used throughout the code for multicore parallelism. It can be optionally enabled by specifying the *COMPILE_OPTIONS = -fopenmp* or alternatively disabled by removing the options.

**OPTC** specifies standard GCC compiler options, refer to the GCC manual for details.

Note: since version 3.1 the Makefile contains example directives for Clang and Intel ICC compilers.

### Known Problems

OpenMP is not fully supported by all compilers on Mac OSX and OpenMP doesn't scale well in performance. By default, OpenMP is disabled on non-Linux platforms, this can be changed by editing *COMPILE_OPTIONS* in the *Makefile*.

## 4.3 Old build system: Windows using Mircosoft Visual C++ 2015

Starting with version 3.1, TASMANIAN can be compiled using MS Visual C++. The *tasgrid* executable can be compiled from the development console using the commands:

```
cl -c *.cpp /Ox /EHsc /openmp
cl *.obj /Fe:tasgrid.exe
```

The executable and a static library can also be compiled using the included WindowsMake.bat script. See WindowsREADME.txt for more details.

# 5 LIBTASMANIANSPARSEGRIDS (libtsg)

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

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

## 5.1 Constructor TasmanianSparseGrid()

```
TasmanianSparseGrid();
```

This is the only class constructor (called by default), makes an empty grid. Before any operations can be performed, a grid has to be made with one of the *makeGlobalGrid()*, *makeSequenceGrid()*, *makeLocalPolynomialGrid()* or *makeWaveletGrid()* functions or alternatively the grid can be read from a stream/file using the *read()* functions (in order to read a grid, it must first be written to the file with the *write()* function). The *getVersion()* and *getLicense()* functions can be called at any time. Calling any other function will result in a *Segfault*.

## 5.2 Destructor TasmanianSparseGrid()

```
~TasmanianSparseGrid();
```

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

## 5.3 function getVersion()

```
const char* getVersion() const;
```

Returns the version of the library, which is a simple hard-coded string.

## 5.4 function getLicense()

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

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

## 5.5 function makeGlobalGrid()

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

This function creates a sparse grid induced by one of the global quadrature and interpolation rules. See Section 2 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 (1.3) and (1.2). There is no hard restriction on how many outputs can be handled, however, note that the code requires at least *outputs* $\times$ *number of points* in storage and hence for large number of **outputs** memory management may have adverse effect on performance.

**depth** is a non-negative (or strictly positive) integer that controls the density of abscissa points. This is the $L$ parameter in tensor selection (2.15) - (2.23). 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 (2.15)
- *type_curved*: see (2.16)
- *type_hyperbolic*: see (2.17)
- *type_iptotal*: see (2.18)
- *type_ipcurved*: see (2.19)
- *type_iphyperbolic*: see (2.20)
- *type_qptotal*: see (2.21)
- *type_qpcurved*: see (2.22)
- *type_qphyperbolic*: see (2.23)

- *type_tensor*: creates a full (not sparse) tensor grid in the notation of §2, $G = \bigotimes_{k=1}^{d} \mathcal{U}^{m(L\cdot\xi_k)}$.
- *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 A.

**anisotropic** (**anisotropic_weights**) is either *NULL* or an array of integers of size *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 $\alpha$ parameter of $\rho(x)$, this is used only if **rule** requires the $\alpha$ parameter. See Table 2.

**beta** specifies the $\beta$ parameter of $\rho(x)$, this is used only if **rule** requires the $\beta$ 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 A for more information about the file format of the custom file.

## 5.6 function makeSequenceGrid()

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

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

| | | |
|---|---|---|
| *rule_rleja* | *rule_leja* | *rule_minlebesgue* |
| *rule_rlejashifted* | *rule_maxlebesgue* | *rule_mindelta* |

## 5.7 function makeLocalPolynomialGrid()

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

Creates a grid based on one of the local hierarchical piece-wise polynomial rules described in section 3. 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 (3.1).

**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*.

## 5.8 function makeWaveletGrid()

```
void makeWaveletGrid( int dimensions,
                      int outputs,
                      int depth,
                      int order = 1 );
```

Creates a grid based on local hierarchical wavelet basis, see 3.4.

**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.

## 5.9 function makeFullTensorGrid()

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

## 5.10 functions recycle***Grid()

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

## 5.11 functions update***Grid()

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

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

### 5.12 functions update***Grid()

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

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

### 5.13 function write()

```
void write( std::ofstream &ofs ) const;
```

Writes out the grid in text format to the *ofstream*.

### 5.14 function read()

```
bool read( std::ifstream &ifs );
```

Reads a grid that has already been written to the stream. The function returns *True* if the reading was successful or *False* if errors with the file format were encountered. The function will write error information to *std::cerr* stream.

### 5.15 function write()

```
void write( const char* filename ) const;
```

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

### 5.16 function read()

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

Opens a file with *filename* and calls *bool read( std::ifstream &ifs ) const;* with the associated stream. In the end, the file is closed.

## 5.17 function setTransformAB()

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

Since TASMANIAN version 3.0, this function is renamed to *setDomainTransform()*.

## 5.18 function setDomainTransform()

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

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

**a** is an array of real numbers of size *getNumDimensions()* that defines the $a_k$ parameter associated with every direction.

**b** is an array of real numbers of size *getNumDimensions()* that defines the $b_k$ parameter associated with every direction.

## 5.19 function isSetDomainTransform()

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

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

## 5.20 function clearTransformAB()

```
void clearTransformAB();
```

Since TASMANIAN version 3.0, this function is renamed to *clearDomainTransform()*.

## 5.21 function clearTransformAB()

```
void clearDomainTransform();
```

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

### 5.22 function getTransformAB()

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

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

### 5.23 function getDomainTransform()

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

Returns the transform parameters.

**a** the first *getNumDimensions()* entries of **a** are overwritten with the $a_k$ parameters of the transform.

**b** the first *getNumDimensions()* entries of **b** are overwritten with the $b_k$ parameters of the transform.

### 5.24 function getNumDimensions()

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

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

### 5.25 function getNumOutputs()

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

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

### 5.26 function getOneDRule()

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

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

## 5.27 function getOneDRuleDescription()

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

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

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

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

## 5.28 function getCustomRuleDescription()

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

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

## 5.29 function 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.

## 5.30 function getOrder()

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

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

## 5.31 function getNum***()

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

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

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

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

## 5.32 function get***Points()

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

Returns an array of length *getNumDimensions()* $\times$ *getNum***()* of values that represent the points of the grid. The number of points corresponds to the above *getNum***()* functions. The first point is located in the first *getNumDimensions()* number of entries, the second point is located in the second *getNumDimensions()* number of entries, and so on.

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

## 5.33 function getWeights()

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

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

## 5.34 function getQuadratureWeights()

```
double* getQuadratureWeights() const;
```

Returns an array of size *getNumPoints()* of the quadrature weights associated with the points. The first weight is associated with the first point returned by *getPoints()*, the second weight is associated with the second point and so on.

## 5.35 function getInterpolantWeights()

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

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

## 5.36 function getInterpolationWeights()

```
double* getInterpolationWeights( const double x[] ) const;
```

Returns the interpolantion weights associated with the point **x**, as in equation (1.2). For global 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 other grids, computing the *getInterpolationWeights()* is very expensive and should be avoided (if possible).

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

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

### 5.37 function getNumNeededPoints()

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

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

### 5.38 function loadNeededPoints()

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

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

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

### 5.39 function evaluate()

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

Finds the value of the interpolant (or point-wise approximation) at the provided point $x$ as defined by equation (1.1). The result is written into **y**.

**x** an array of size *getNumDimensions()* that indicate the point where the interpolant should be evaluated.

**y** an already allocated array of size *getNumOutputs()*. On exit, the entries of **y** are overwritten with the values of the interpolant at the point **x**.

### 5.40 function integrate()

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

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

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

### 5.41 function is***()

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

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

### 5.42 function setRefinement()

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

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

### 5.43 function setAnisotropicRefinement()

```
void setAnisotropicRefinement( TypeDepth type,
                               int min_growth,
                               int output );
```

Implements the anisotropic refinement strategy described briefly in section 2.4 and in more details in [23]. This function can only be called for Global and Sequence grids. Note that refinement cannot be used if the grid has no outputs or before values have been loaded, i.e., *loadNeededPoints()* has been called.

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

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

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

### 5.44 function setSurplusRefinement() - global version

```
void setSurplusRefinement( double tolerance, int output );
```

Implements the surplus refinement strategy described in equation (2.26) 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 $-1$.

## 5.45 function setSurplusRefinement() - local version

```
void setSurplusRefinement( double tolerance,
                           TypeRefinement criteria,
                           int output );
```

Implements the surplus refinement strategy described briefly in section 3.2 and in more details in [22]. 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 $\epsilon$ parameter in equations (3.2), (3.3), (3.4), (3.5).

**criteria** specifies the refinement strategy

| | |
|---|---|
| *refine_classic*, see (3.2) | *refine_direction_selective*, see (3.4) |
| *refine_parents_first*, see (3.3) | *refine_fds*, see (3.5) |

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

## 5.46 function clearRefinement()

```
void clearRefinement();
```

Every *set***Refinement()* function expands the grid to include a new set of points (unless all surpluses are smaller than **tolerance**). The *clearRefinement()* function removes the needed points and all internal data structures associated with the last call to *set***Refinement()*. Note that once *loadNeededPoints()* is called for the new set of points, then the refinement cannot be undone. The purpose of this function is to reduce the memory footprint of the grid in case the user decides not to load the new points from the last refinement.

## 5.47 function getPolynomialIndexes()

Since TASMANIAN version 3.0, this function is renamed to *getPolynomialSpace()*.

## 5.48 function getPolynomialSpace()

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

Computes the polynomial associated with the grid, see $\Lambda^m$ and $\Lambda^q$ in equations (2.11) and (2.12). 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., (2.11) and (2.12).

**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.

## 5.49 function printStats()

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

Prints short description of the sparse grid. The output is written to standard output (i.e., *cout*).

## 5.50 functions getSurpluses() and getPointIndexes()

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

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

## 5.51 Examples

The file *example.cpp* in the *Examples/* folder has sample code that demonstrates proper use of the *TasmanianSparseGrid* class. In addition, there is also a *Makefile* that compiles the example.

# 6 TASGRID

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

## 6.1 Basic Usage

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

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

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

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

## 6.2 Command: -h, help, -help, –help

Prints information about the usage of *tasgrid*. Note that many commands and options have a long and short name and the help command will list both.

In addition, writing *help* after any command will print information specific to that command. Thus, *help* is a universal option.

## 6.3 Command: -listtypes

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

## 6.4 Command: -version or -info

Prints the version of the library.

## 6.5 Command: -test

```
./tasgrid -test
```

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

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

On a Intel 3.2Ghz 6-core Sandy Bridge-E CPU all tests take about 5 seconds (with OpenMP enabled), if the test take much longer on your machine this is an indication of either a slow CPU or problem with the build or code.

## 6.6 Command: -makegrid

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

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

## 6.7 Command: -makeglobal

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

| | |
|---|---|
| **-dimensions** | the dimensions parameter |
| **-outputs** | the outputs parameter |
| **-depth** | the depth parameter |
| **-onedim** | is a string specifying the **rule** parameter of *makeGlobalGrid()*. See *./tasgrid -listtypes* for the list of accepted strings, it's pretty self-explanatory. |
| **-type** | is a string specifying the **type** parameter of *makeGlobalGrid()*. See *./tasgrid -listtypes* for the list of accepted strings, it's pretty self-explanatory. |
| **-alpha** | the alpha parameter |
| **-beta** | the beta parameter |
| **-outputfile** | is an optional matrix file. At the end of the program, *tasgird* will write in the file the points associated with the grid. The matrix file will have *getNumPoints()* number of rows and *-dimensions* number of columns. The first points will be on the first row, the second on the second row and so on. |

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

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

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

**-transformfile** is an optional matrix file that specifies the transformation from the canonical domain to a custom domain. The matrix file should have *dimensions* number of rows and 2 columns. The first column is the $a_k$ parameter and the second column is the $b_k$ parameter and each row corresponds to one dimension. For detail on the matrix file format see subsection 6.29. Note: this option used to be called *-inputfile*.

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

## 6.8 Command: -makesequence

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

**-dimensions** the dimensions parameter

**-outputs** the outputs parameter

**-depth** the depth parameter

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

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

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

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

**-anisotropyfile** same as in *-makeglobal*

**-transformfile** same as in *-makeglobal*

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

## 6.9 Command: -makelocalpoly

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

**-dimensions** the dimensions parameter

**-outputs** the outputs parameter

**-depth** the depth parameter

**-order** the order parameter

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

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

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

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

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

## 6.10 Command: -makewavelet

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

**-dimensions** the dimensions parameter

**-outputs** the outputs parameter

**-depth** the depth parameter

**-order** the order parameter

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

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

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

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

## 6.11 Command: -makequadrature

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

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

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

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

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

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

## 6.12 Command: -recycle

```
./tasgrid -recycle
```

Removed in version 3.0.

## 6.13 Command: -makeupdate

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

Calls *updateGlobalGird()* or *updateSequenceGrid()*.

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

**-depth** the depth parameter

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

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

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

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

### 6.14 Command: -getquadrature

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

Calls *getQuadratureWeights()*.

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

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

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

### 6.15 Command: -getpoints

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

Calls *getPoints()*.

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

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

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

### 6.16 Command: -getinterweights

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

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

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

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

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

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

### 6.17 Command: -getneededpoints

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

Calls *getNeeded()*.

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

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

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

### 6.18 Command: -loadvalues

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

Calls *loadNeededPoints()*.

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

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

### 6.19 Command: -evaluate

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

Calls *evaluate()*.

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

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

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

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

### 6.20 Command: -integrate

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

Calls *integrate()*

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

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

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

### 6.21 Command: -getanisotropy

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

Calls *estimateAnisotropicCoefficients()*

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

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

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

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

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

## 6.22 Command: -refine

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

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

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

## 6.23 Command: -refineaniso

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

Calls *setAnisotropicRefinement()*

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

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

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

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

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

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

## 6.24 Command: -refinesurp

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

Calls *setSurplusRefinement()*

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

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

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

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

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

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

### 6.25 Command: -cancelrefine

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

Calls *clearRefinement()*

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

### 6.26 Command: -getpoly

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

Calls *getGlobalPolynomialSpace()*

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

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

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

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

### 6.27 Command: -summary

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

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

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

## 6.28 Commands: -getsurpluses, -getpointindexes

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

Calls *getSurpluses()* or *getPointIndexes()*.

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

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

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

## 6.29 Matrix File Format

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

The file containing

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

represents the matrix

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

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

```
1 2
13.0 14.0
```

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

# 7 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:

- **If not using *cmake***: Before using the interface you must manually edit the *tsgGetPath.m* file.
- 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.
- 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.
- 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 |

## 7.1 function tsgGetPaths()

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

You must edit the two strings in this file.

**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.

## 7.2 functions tsgReadMatrix() and tsgWriteMatrix()

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

## 7.3 functions tsgCleanTempFiles()

Those functions are used internally to clean the temporary files.

## 7.4 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()*.

## 7.5 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()*.

## 7.6 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 7.12. 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> )`.

## 7.7 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.

## 7.8 function tsgWriteCustomRuleFile()

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

## 7.9 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.

## 7.10 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.

## 7.11 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.

## 7.12 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.

# 8 Python Interface

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

Required Python modules:

- *c_types*
- *numpy*
- optional: *matplotlib.pyplot*

Each Python function 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.

In addition to the standard C++ interface, the Python class implements several *Batch* functions, specifically when dealing with evaluations. The batch functions allow the interpolant evaluations for multiple $x$ values to be computed with a single function call, thus reducing overhead. In addition, if OpenMP is enabled in the C++ library, the batch evaluations will take advantage of multiple available cores and execute the evaluations in parallel.

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 |

## A 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 *tsgHardcodedConstants.hpp* by modifying the `NUM_TOL` constant,
- Naturally, TASMANIAN cannot create a sparse grid that requires a one dimensional rule with level higher than what is provided in the file. Predicting the required number of levels can be hard in the case of anisotropic *iexact/qexact* grids, the code will print a warning message if the custom rule does not provide a sufficient number of points.
- The exactness constants are used only if *qexact* is used for *makeGlobalGrid* or the *getPolynomialIndexes* functionality. If *qexact* is not used, then the exactness integers can be set to $0$.
- The quadrature weights are used only if integration is performed. If no quadrature or integration is used, then the weights can all be set to $0$.
- If a custom rule is used together with *TransformAB*, then the transform will assume that the rule is defined on the canonical interval $[-1, 1]$. A custom rule can be defined on any arbitrary interval, however, for any interval different from $[-1, 1]$ the *TransformAB* functions should not be used.
- The code comes with an example custom rule file that defines $9$ levels of the Gauss-Legendre-Patterson rule, a.k.a., nested Gauss-Legendre rule.

## REFERENCES

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

[2] 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

[3] V. Barthelmann, E. Novak, and K. Ritter, *High dimensional polynomial interpolation on sparse grids*, Advances in Computational Mathematics, 12 (2000), pp. 273–288. 2

[4] 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. 8

[5] 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. 10, 11

[6] ——, *On the lebesgue constant of a new type of $\mathcal{R}$-leja sequences*, tech. rep., ORNL/TM-2015/657, Oak Ridge National Laboratory., 2015. 11

[7] C. W. Clenshaw and A. R. Curtis, *A method for numerical integration on an automatic computer*, Numerische Mathematik, 2 (1960), pp. 197–205. 10

[8] S. De Marchi, *On Leja sequences: some results and applications*, Applied Mathematics and Computation, 152 (2004), pp. 621–647. 11

[9] 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

[10] 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. 10

[11] T. Gerstner and M. Griebel, *Numerical integration using sparse grids*, Numerical algorithms, 18 (1998), pp. 209–232. 2

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

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

[14] 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

[15] M. Gunzburger, C. Webster, and G. Zhang, *An adaptive wavelet stochastic collocation method for irregular solutions of partial differential equations with random input data*, Tech. Rep. ORNL/TM-2012/186, Oak Ridge National Laboratory, 2012. 2, 21

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

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

[18] 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

[19] 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

[20] T. C. PATTERSON, *The optimum addition of points to quadrature formulae*, Mathematics of Computation, 22 (1968), pp. 847–856. 11

[21] 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

[22] M. STOYANOV, *Hierarchy-direction selective approach for locally adaptive sparse grids*, tech. rep., ORNL/TM-2013/384, Oak Ridge National Laboratory., 2013. 2, 3, 14, 16, 38

[23] 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, 7, 8, 10, 11, 12, 37

[24] W. SWELDENS AND P. SCHRÖDER, *Building your own wavelets at home*, in Wavelets in the Geosciences, Springer, 2000, pp. 72–107. 21

[25] 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

[26] 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