ORNL REPORT

Unlimited Release
Printed August 2013

User Manual: TASMANIAN Sparse Grids

M. Stoyanov

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

The Oak Ridge National Laboratory is operated by UT-Battelle, LLC,
for the United States Department of Energy under Contract DE-AC05-00OR22725.
Approved for public release; further dissemination unlimited.

Computer Science and Mathematics Division

USER MANUAL: TASMANIAN SPARSE GRIDS

M. Stoyanov *

Date Published: August 2013

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

Appendix

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 interface.

ACKNOWLEDGEMENTS

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

1 Introduction

Sparse Grids is a family of algorithms for constructing multidimensional quadrature and interpolation rules from tensor products of one dimensional such rules [8, 10, 18, 19, 22]. Given a function $f(x) : \mathbb{R}^d \to \mathbb{R}$, a quadrature rule that integrates $f(x)$ over the domain $\Gamma \subset \mathbb{R}^d$ with respect to the weight $\rho(x)$ is defined as by the abscissas $\{x_i\}_{i=1}^N \subset \Gamma$ and weights $\{w_i\}_{i=1}^N \subset \mathbb{R}$ so that

$$\int_\Gamma f(x)\rho(x)dx \approx \sum_{i=1}^N w_i f(x_i). \tag{1.1}$$

The integration weight $\rho(x)$ is a tensor of one dimensional weight $\rho(x) = \rho_1(x_1)\rho_1(x_2)\cdots\rho_1(x_d)$ and the domain Γ is the tensor of a one dimensional domain $\Gamma = \Gamma_1 \otimes \Gamma_1 \otimes \cdots \otimes \Gamma_1$. An interpolation rule is defined by abscissas $\{x_i\}_{i=1}^N \subset \Gamma$ and basis functions $\{\phi_i(x)\}_{i=1}^N$ as

$$f(x) \approx \sum_{i=1}^N c_i \phi_i(x), \tag{1.2}$$

where the coefficients c_i are chosen so that

$$\sum_{i=1}^N c_i \phi_i(x_j) = f(x_j).$$

The coefficients c_i can be found by solving a system of linear equations, i.e. $\{c_i\}_{i=1}^N$ are the result of a linear transformation applied to $\{f(x_i)\}_{i=1}^N$. In general, the conditioning of this linear map is of consideration, however, a suitable choice of abscissas x_i and functions $\phi_i(x)$ results in a well conditioned problem. An alternative definition of the interpolant is given by

$$f(x) \approx \sum_{i=1}^N h_i(x) f(x_i),$$

where the functions $\{h_i(x)\}_{i=1}^N$ are constructed by applying the dual of the linear transformation to the vector of basis functions $\{\phi_i(x)\}_{i=1}^N$. For a specific point $\tilde{x}$, the approximation becomes

$$f(\tilde{x}) \approx \sum_{i=1}^N h_i(\tilde{x}) f(x_i) = \sum_{i=1}^N g_i f(x_i), \tag{1.3}$$

where $g_i = h_i(\tilde{x})$ are the interpolation weights.

One dimensional integration and interpolation rules can be transformed from a domain Γ_1 to $\Gamma_1^{a,b}$, where $\Gamma_1^{a,b}$ is the image of Γ_1 under a linear map defined by a and b. For example, if $\Gamma_1 = [-1, 1]$ then an arbitrary interval $[a, b]$ is the image of Γ_1 under the linear transformation

$$y = \frac{b-a}{2}x + \frac{b+a}{2}.$$

Analogously the integration weight $\rho_1(x)$ can be transformed into $\rho_1^{a,b}(x)$. Extending the result in multiple dimensions, it is trivial to define a map for every dimension and have

$$\Gamma = \Gamma_1^{a_1,b_1} \otimes \Gamma_1^{a_2,b_2} \otimes \cdots \otimes \Gamma_1^{a_d,b_d},$$

and the integration weights

$$\rho(x) = \rho_1(x_1)\rho_1(x_2)\cdots\rho_1(x_d).$$

For a full list of the transformations supported by this code, see Appendix A.

The goal of Sparse Grids is to select a subset from all possible tensor product abscissas $\{x_i\}_{i=1}^N$ so that maximum accuracy is achieved for the smallest number of function evaluations $\{f(x_i)\}_{i=1}^N$. This is usually achieved by balancing the error in different dimensions. For more information on the properties of sparse grids see [2–4, 6, 8–10, 12, 13, 15, 16, 18, 19, 22, 26, 27].

The TASMANIAN Sparse Grid code implements a number of different quadrature rules and function basis. The rules are grouped in three categories:

- **Global Grids**: suitable for globally smooth functions. Quadrature is based on a number of available rules (see Appendix A) and interpolation is based on global Lagrange polynomials.
- **Local Polynomial Grids**: suitable for non-smooth functions with locally sharp behavior. Interpolation is based on hierarchical piece-wise polynomials with local support and user specified order. These grids are suitable for local refinement.
- **Wavelet Grids**: are similar to the local polynomials, however, when coupled with local refinement, often times wavelet grids provide the same accuracy with fewer abscissas.

For a given grid the code can perform three tasks:

- generate a set of abscissas $\{x_i\}_{i=1}^N$ and weights $\{w_i\}_{i=1}^N$ for a quadrature rule of type (1.1).
- generate a set of abscissas $\{x_i\}_{i=1}^N$ and for the user specified function values $\{f(x_i)\}_{i=1}^N$, the code can create an interpolant of type (1.2). The interpolant can be evaluated for any arbitrary x and it can also be integrated over the domain.
- generate a set of abscissas $\{x_i\}_{i=1}^N$ and for any arbitrary x it can also generate the interpolation weights $\{h_i(x)\}_{i=1}^N$ for an approximation of type (1.3).

In addition, local grids support iterative refinement, where additional abscissas are chosen based on the provided $\{f(x_i)\}_{i=1}^N$ to improve the approximation of the provided interpolant.

The code consists of three main components:

- *libtasmaniansparsegrids.a* (for short *libtsg*) which is a library written in C++ that implements the *TasmanianSparseGrid* class. The class provides an interface for manipulation of the grid. See Section 3.

- *tasgrid* which is an executable that provides a command line interface to *libtsg*. 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 Section 4.

- *MATLAB Interface* (for short *tsg.m*) which is a series of MATLAB functions that call the executable *tasgrid* and read the result into MATLAB matrices. See Section 5.

2 Compilation

Quick Build

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 4 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.

Known Problems

Mac users have reported problems with OpenMP and some versions of GCC. Mac users that want to use OpenMP should make sure to have the latest available version of GCC, versions 4.7 and newer tend to resolve most issues.

3 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 virtually no sanity check on the validity of input. Wrong input would most likely result in a crash.

3.1 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()*, *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 user can also call *getVersion()* and *getLicense()* functions at any time. Calling any other function will result in a *Segfault*.

3.2 Destructor TasmanianSparseGrid()

```
~TasmanianSparseGrid();
```

This is the destructor that releases any dynamical memory used by the class (this instance of the class can no longer be used).

3.3 function getVersion()

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

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

3.4 function getLicense()

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

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

3.5 function makeGlobalGrid()

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

This function creates a sparse grid induced by one of the global quadrature and interpolation rules. See Appendix A 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 abscissas associated with a sparse grid grows fast (i.e. 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 (1.1) and (1.3). There is no hard restriction on how many outputs can be handled, however, note that the code requires at least *outputs* $\times$ *number of abscissas* 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. For grids of *type_level* and *type_hyperbolic*, **depth** is strictly positive and it corresponds to the notion of sparse grid "level" (see [18, 19, 22]). For grids of *type_basis* the **depth** specifies the largest total degree polynomial that can be integrated or interpolated exactly. Gauss based rules imply integration, while non-Gauss rules imply interpolation (see Appendix A). There is no hard restriction on how big **depth** can be, however, it has direct effect on the number of abscissas and hence performance and memory requirements.

type is an enumerated type from *type_level,type_hyperbolic,type_basis* which guides the tensor selection to balance the precision in different directions.

- *type_level*: classical Smolyak Sperse Grid selection. The sum of the level indexes over all of the directions has to be less then *depth* [18, 19, 22].
- *type_hyberbolic*: hyperbolic cross-section. The product of the level indexes over all of the directions has to be less then *depth* [8, 11, 21].
- *type_basis*: takes into consideration the accuracy of the one dimensional rule. Any rule combined with *type_basis* is guaranteed to integrate or interpolate any polynomial of total degree no more than *depth*.

oned is an enumerated type from any of the global rules in Table A.1. Those are:

$rule\_clenshawcurtis$	$rule\_chebyshev$
$rule\_fejer2$	$rule\_gausslegendre$
$rule\_gausschebyshev1$	$rule\_gausschebyshev2$
$rule\_gaussgegenbauer$	$rule\_gaussjacobi$
$rule\_gausslaguerre$	$rule\_hermite$

anisotropic (**anisotropic_weights**) is either *NULL* or an array of integers of size *dimensions*. See [18] for the meaning of the weights. Note that in the literature, the weights are assumed to be real numbers. The code assumes that the weights are rational numbers and that they add up to 1, thus the **anisotropic_weights** array contains only the numerators of the rational numbers. This is done so that *type_level* and *type_basis* grids can be constructed using only integer based arithmetic; *type_hyperbolic* grids still use double precision arithmetic that may be unstable (i.e. the number of abscissa may be heavily influenced by rounding error). In addition, after introducing anisotropic weights, the value of *depth* still controls the number of abscissas, however, it no longer has the same relationship to "levels" or total degree polynomial order.

alpha_beta is either *NULL* or an array of one or two doubles. The first entry of the array is α and the second is β and those values are referenced only if the quadrature rule requires them. One dimensional rules of type *rule_gaussgegenbauer*, *rule_gaussjacobi*, *rule_gausslaguerre* and *rule_gausshermite* require an α and in addition *rule_gaussjacobi* requires β (see Table A.1).

3.6 function makeLocalPolynomialGrid()

```
void makeLocalPolynomialGrid( int dimensions,
                              int outputs,
                              int depth,
                              int order,
                              TypeOneDRule boundary );
```

Creates a grid based on local hierarchical piece-wise polynomial function basis. The main focus of hierarchical grids is the ability to do local iterative refinement to daptively obtain an interpolant that clusters abscissa in regions of sharp behavior and puts fewer abscissa in regions of smooth behavior. Local grids can be used for integration, however, in many cases, this would result in abscissas associated with zero weights.

dimensions same as *makeGlobalGrid()*

outputs same as *makeGlobalGrid()*, however, due to the non-trivial form of the coefficients c_i, large number of outputs comes with bigger computational cost in addition to the larger storage cost of $2 \times$ *outputs* $\times$ *number of abscissas*.

depth is a positive integer that specifies the initial number of levels for the grid.

order is an integer bigger than -2, which specifies the largest order of polynomial to be used. For a one dimensional interpolation rule, a polynomial of order l cannot be used before level $l-1$ (i.e. before $depth = l$). Thus, if *order* is larger than *depth* only lower degree local polynomials would be used. If *order* is set to -1, the largest possible order would be selected automatically "on the fly".

boundary is an enumerated type with value either *rule_pwpolynomial* or *rule_pwpolynomial0*. The difference is that the latter type assumes that the interpolated function is zero at the boundary.

3.7 function makeWaveletGrid()

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

Creates a grid based on local hierarchical wavelet basis. It is very similar to the local polynomial rule, however, local refinement using wavelet functions would sometimes require fewer function evaluations.

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. The wavelet grids use the corresponding order of wavelet even for grid with *depth = 1*. However, a wavelet grid of a given *depth* would have more abscissas than a corresponding local polynomial grid.

3.8 function makeFullTensorGrid()

```
void makeFullTensorGrid( int dimensions,
                         int outputs,
                         int order[],
                         TypeOneDRule oned,
                         const double *alpha_beta = 0 );
```

Full tensor grids are not sparse grids, however, full tensors share many of the properties of global grids and hence the capability is included into the code (mostly for testing purposes). Note that in most situations, full tensor grids would require a much larger number of abscissas to achieve the same accuracy as than a sparse grid.

dimensions same as in *makeGlobalGrid()*

outputs same as in *makeGlobalGrid()*

order an array of non-negative integers of size *dimensions*. The array indicates the level of one dimensional rule to be used in every direction.

oned same as in *makeGlobalGrid()*

alpha_beta same as in *makeGlobalGrid()*

3.9 functions recycle***Grid()

```
void recycleGlobalGrid( int depth,
                        TypeDepth type,
                        const int *anisotropic_weights = 0 );
void recycleLocalPolynomialGrid( int depth,
                                 int order = 1 );
void recycleWaveletGrid( int depth,
                         int order = 1 );
void recycleFullTensorGrid( int order[] );
```

The recycle functions recreate a new grid with similar properties, but different number of abscissas (and hence different accuracy). The recycle functions modify some of the parameters of the grids, but keep all the ones that are not specified (i.e. *dimensions*, *outputs*, *oned*, *boundary*). Recycle will also try to use any values of $f(x_i)$ loaded in the old grid. The code discards values of $f(x_i)$ that are associated with abscissas in the old grid but are not part of the new grid.

3.10 function write()

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

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

3.11 function read()

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

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 the standard output stream.

3.12 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. At the end, the file is closed.

3.13 function read()

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

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

3.14 function setTransformAB()

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

By default integration and interpolation are performed on a canonical interval described in Table A.1. Optionally, the library can transform the canonical interval into a custom one defined by the a and b parameters for every direction. The transformation is applied as a post-processing step to the abscissas and weights.

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

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

3.15 function clearTransformAB()

```
const char* clearTransformAB() const;
```

Scales back all abscissas and weights to the canonical interval. Since the transformation is a post-processing step, the *clearTransformAB()* function simply removes the values set by *setTransformAB()*, i.e. the grid is not actually recomputed.

3.16 function getTransformAB()

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

Returns the transform parameters.

a on input it is either a *NULL* pointer or a non-*NULL* pointer that will be deleted.
on output returns a pointer to an array of size *getNumDimensions()* that contains the values of the a parameter for every direction.

b on input it is either a *NULL* pointer or a non-*NULL* pointer that will be deleted.
on output returns a pointer to an array of size *getNumDimensions()* that contains the values of the b parameter for every direction.

3.17 function getNumDimensions()

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

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

3.18 function getNumOutputs()

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

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

3.19 function getOneDRule()

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

For a global grid, returns the value of the *oned* parameter.

For a local polynomial grid, returns the value of the *boundary* parameter.

For a wavelet grid, it returns *rule_wavelet*.

3.20 function getOneDRuleDescription()

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

Returns a short string description of the one dimensional rule used.

3.21 function getNumPoints()

```
int getNumPoints() const;
```

Return the total number of abscissas associated with the grid.

3.22 function getPoints()

```
void getPoints( double* &pnts ) const;
```

Return the abscissas associated with the grid.

pnts on input it is either *NULL* or a non-*NULL* pointer that would be deleted.
on output returns an array of size *getNumDimensions()* $\times$ *getNumPoints()* of values that represent the abscissas. The first abscissa is located in the first *getNumDimensions()* number of entries, the second abscissa is located in the second *getNumDimensions()* number of entries, and so on.

3.23 function getWeights()

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

Return the quadrature weights associated with the abscissas, as in equation (1.1).

weights on input it is either *NULL* or a non-*NULL* pointer that would be deleted.
on output it is an array of size *getNumPoints()* of the quadrature weights associated with the abscissas. The first weight is associated with the first abscissa returned by *getPoints()*, the second weight is associated with the second abscissa and so on.

3.24 function getInterpolantWeights()

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

Returns the interpolantion weight associated with the abscissa and the point defined by x, as in equation (1.3).

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

weights on input it is either *NULL* or a non-*NULL* pointer that would be deleted.
on output returns an array of size *getNumPoints()* of the interpolation weights associated with the abscissas. The first weight is associated with the first abscissa returned by *getPoints()*, the second weight is associated with the second abscissa and so on.

3.25 function getNumNeededPoints()

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

Interpolation described in equation (1.2) requires the user to provide the values of the interpolated function at the abscissa points. This functions returns the number of abscissas that are still not associated with function values.

3.26 function getNeededPoints()

```
void getNeededPoints( double* &pnts ) const;
```

pnts on input it is either *NULL* or a non-*NULL* pointer that would be deleted.
on output returns an array of size *getNumDimensions()* $\times$ *getNumNeededPoints()* of entries that represent the abscissas that still need to be associated with function values. The first abscissa is located in the first *getNumDimensions()* number of entries, the second abscissa is located in the second *getNumDimensions()* number of entries, and so on. If *getNumNeeded-Points()* returns 0, then **pnts** is returned *NULL*.

3.27 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()* $\times$ *getNumNeededPoints()*. The first *getNumOutputs()* entries correspond to the outputs of the interpolated function at the first abscissa point. The second set of *getNumOutputs()* entries correspond to the second abscissa and so on.

3.28 function evaluate()

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

Finds the value of the interpolant at the provided point x as defined by equation (1.2). 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.

3.29 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.

3.30 function printStats()

```
void printStats();
```

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

3.31 function setRefinement()

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

Improves the accuracy of the sparse grid based on the loaded values of the interpolant. After calling *setRefinement()*, the needed points are updated and *evaluate* and *integrate* will work with the old interpolant until the new needed values are loaded. If *setRefinement()* is called twice in a row without *loadNeededValues()*, then any data from the first call will be cleared and only the second refinement would persist.

tolerance is a positive number with the desired tolerance.

criteria is an enumerated type from *type_classic,type_parent_first,type_direction_selective,type_fds*. For the three types of local refinement, see [23].

Note that the inputs have different meaning depending on the grid.

- *Global Grid* will ignore both parameters and will only increase the accuracy isotropically in every direction by adding the next levels of the one dimensional rule.
- *Local Polynomial Grid* will compare the relative magnitude of the surpluses c_i divided by the largest provided value $f(x_i)$ to the *tolerance*. The algorithm will refine only in the neighborhood of the abscissas where the ratio is large for at least one of the outputs. The *criteria* defines whether or not all direction should be refined and whether or not the "parents" should be added before the "children", where the family is described by the hierarchy [23].
- *Wavelet Grid* will compare the surpluses c_i to the *tolerance* and will refine only in the neighborhood of the abscissas associated with large surpluses. The *criteria* is ignored (i.e. wavelet grids currently only implement *type_classic* refinement).

3.32 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.

4 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.

4.1 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 and error in the input, *tasgrid* will print an error message as well as some help for the input.

4.2 Command: -h –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 it will also list the available one dimensional quadrature and interpolation rules.

4.3 Command: -version

Prints the version of the library and executable.

4.4 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 wavelet tests take a long time and hence they are performed last. Unless one is interested in using wavelet grids, the wavelet tests can be skipped.

4.5 Command: -makegrid

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

This command creates a new sparse grid. It uses the following options that can be given in any order:

-onedim specifies whether to use global, local polynomial or wavelet grids, as well as the underlying one dimensional quadrature and interpolation rule. The available values for the rules are summarized in Table A.1.

clenshaw-curtis creates a global grid with Clenshaw-Curtis rule.

chebyshev creates a global grid with Chebyshev rule.

fejer-2 creates a global grid with Fejer type 2 rule.

gauss-legendre creates a global grid with Gauss-Legendre rule.

gauss-chebyshev-1 creates a global grid with Gauss-Chebyshev rule of type 1.

gauss-chebyshev-2 creates a global grid with Gauss-Chebyshev rule of type 2.

gauss-gegenbauer creates a global grid with Gauss-Gegenbauer rule.

gauss-jacobi creates a global grid with Gauss-Jacobi rule.

gauss-laguerre creates a global grid with Gauss-Laguerre rule.

gauss-hermite creates a global grid with Gauss-Hermite rule.

local-polynomial creates a local polynomial grid.

local-polynomial-zero creates a local polynomial grid with zero boundary.

local-wavelet creates a wavelet grid.

-dimensions specifies the dimension of the problem. The value should be a positive integer and it gets passed directly to *make***Grid()* (see the previous section).

-outputs specifies the number of outputs of the function that needs to be interpolated. The values should be a positive integer that gets passed directly to *make***Grid()* (see the previous section). To set zero outputs, use the *-makequadrature* command.

-depth specifies the *depth* parameter of the *make***Grid()* function (see the previous section). In case of a *tensor* grid and if no *anisotropyfile* is give, *makeFullTensor()* will be called with *order* array having all entries equal to *-depth*.

-type specifies the selection criteria of the sparse grid and it is used only by global and tensor grids. Available values are

level calls *makeGlobalGrid()* and it specifies the *type_level* enumerated type.

basis calls *makeGlobalGrid()* and it specifies the *type_basis* enumerated type.

hyperbolic calls *makeGlobalGrid()* and it specifies the *type_hyperbolic* enumerated type.

tensor calls *makeFullTensorGrid()*.

-order specifies the order of the local polynomials or wavelets and is used only with local polynomial and wavelet grids. For polynomials, the value is -1 for automatically using the maximum possible order or a non-negative integer that restricts the maximum order. Wavelet grids accept only orders 1 and 3.

-alpha specifies the α parameter of the one dimensional quadrature rule. The value is a real number and it is used by *gauss-gegenbauer*, *gauss-jacobi*, *gauss-hermite* and *gauss-laguerre* rules.

-beta specifies the β parameter of the one dimensional quadrature rule. The value is a real number and it is used by the *gauss-jacobi* rule.

-inputfile 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 parameter and the second column is the b parameter and each row corresponds to one dimension. For detail on the matrix file format see subsection 4.17.

-outputfile is an optional matrix file. At the end of the program, *tasgird* will write in the file the abscissas associated with the grid. The matrix file will have *getNumPoints()* number of rows and *dimensions* number of columns. The first abscissa 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 *dimensions* number of rows and only one column. If a global grid is being created, then the file will specify the *anisotropy_weights* array given to *makeGlobalGrid()*. If a *tensor* grid is being created, then this file will specify the *order* array given to *makeFullTensorGrid()*.

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

4.6 Command: -makequadrature

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

This command works the same as *-makegrid* except for the *-outputfile* command.

-outputfile is an optional matrix file. At the end of the program, *tasgird* will write in the file the quadrature weights and abscissas 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 associated abscissa.

4.7 Command: -recycle

```
./tasgrid –recycle <option1> <value1> <option2> <value2> ....
```

This command creates a new grid with most of the parameters taken from an existing grid specified by the *-gridfile* option. Furthermore, the command will try to use existing values of the interpolated function, if they have already been loaded. This command calls one of the *recycle***Grid()* functions. The recycled grid created will use the same number of dimensions and outputs as well as the same base one dimensional quadrature or interpolation rule (including the same values for the α and β parameters).

-gridfile on input this is the file with an already created grid. On exit the file will be overwritten with the new grid.

-depth same as in *-makegrid*

-type if the loaded grid is a global grid, then the value is one of the strings *level*, *basis* or *hyperbolic* which has the same meaning as in *-makegrid*. If the grid file contains a tensor grid, then *-type* is ignored.

-order same as in *-makegrid*

-inputfile same as in *-makegrid*

-outputfile is an optional matrix file. At the end of the program, *tasgird* will write in the file the needed abscissas, 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 abscissa will be on the first row, the second on the second row ...

-anisotropyfile same as in *-makegrid*

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

4.8 Command: -getquadrature

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

Reads a grid from a file, computes the quadrature associated with the grid and writes it to a matrix file.

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

-outputfile is an optional matrix file. The program will write in the file the quadrature weights and abscissas associated with the grid. The matrix file will have *getNumPoints()* number of rows and *getNumDimensions()+1* 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 associated abscissa.

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

4.9 Command: -getpoints

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

Reads a grid from a file, extracts the abscissas associated with the grid and writes them out in a matrix file.

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

-outputfile is an optional matrix file. The program will write in the file the abscissas associated with the grid. The matrix file will have *getNumPoints()* number of rows and *getNumDimensions()* number of columns. The first abscissa 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.

4.10 Command: -getinterweights

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

Reads a grid from a file and a list of points of interest form a matrix file. For each point in the matrix file, *tasgrid* computes the corresponding interpolation weights as in equation (1.3). The result is written to an output matrix file

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

-inputfile 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 *-inputfile*. 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.

4.11 Command: -getneededpoints

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

Reads a grid from a file, extracts the abscissas associated with the grid that are not yet associated with values of the interpolated function. Those abscissas are written to a matrix file.

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

-outputfile is an optional matrix file. The program will write in the file the abscissas 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 abscissa 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.

4.12 Command: -loadvalues

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

Reads a grid from a file and the values of the interpolated function is read from a matrix file. The function values are given to the grid via the *loadValues()* function and the modified grid is written back to the same input file.

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

-inputfile 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.

4.13 Command: -evaluate

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

Reads a grid from a file and a list of points of interest form a matrix file. The interpolant is evaluated at all the points and the result is written to a matrix file.

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

-inputfile 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 *-inputfile*. 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.

4.14 Command: -integrate

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

Reads a grid from a file and integrates the interpolant over the domain. The result is written to a matrix file.

-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.

4.15 Command: -refine

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

Reads a grid from a file and improves the interpolant by adding a new set of abscissas. Refer to *setRefinement()* for details.

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

-tolerance is a positive real number that is given to the *setRefinement()* command.

-refinement is a string specifying the refinement criteria.

classic corresponds to *type_classic*

parents corresponds to *type_parent_first*

direction corresponds to *type_direction_selective*

fds corresponds to *type_fds*

-outputfile is an optional matrix file. At the end of the program, *tasgird* will write in the file the needed abscissas, 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 abscissa 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.

4.16 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.

4.17 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 the exception of the *-gridfile* that contains saved sparse grids and the *-anisotropyfile* that is a matrix with one column and it should contain only integers.

5 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.

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 end of the file name.
- The *tsgDeleteGrid()*, *tsgDeleteGridByName()* and *tsgListGridsByName()* allow for cleaning the files in the temporary folder.
- 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.

5.1 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.

5.2 functions tsgReadMatrix() and tsgWriteMatrix()

Those functions are used internally to read from or write to matrix files. The user shouldn't call those functions directly.

5.3 function tsgMakeGrid()

```
[lGrid, points]=tsgMakeGrid( sGridName, dim, out, oned, depth, order, type,
                                anisotropy, alpha_beta, transformAB )
```

Calls *tasgrid* with the *-makegrid* command. This function creates an *lGrid* object that can be used to refer to the grid by other functions.

INPUTS

sGridName a user provided name that will be appended to all the file names associated with this grid. If a grid with this name already exists, it will be overwritten and the *lGrid* object associated with the old grid will be invalid.

dim the dimension of the grid, same as *-dimensions*.

out the number of outputs of the grid, same as *-outputs*.

oned the one dimensional integration and interpolation rule, same as *-onedim*.

depth same as *-depth*.

order is used by local polynomial and local wavelet grids only. Specifies the order of the polynomial, same as *-order*.

type is used by global grids only. Specifies the *-type* option.

anisotropy is either an empty MATLAB matrix *[]* or a matrix of integers with size *dim* $\times 1$ that describe the anisotropic weights (see *-anisotropicfile* option for *tasgrid* and *const int *anisotropic_weights* parameter for *libtsg*).

alpha_beta is either an empty MATLAB matrix *[]* or a matrix of size 2×1 with the α and β parameters. See *-alpha* and *-beta* option for *tasgrid* and *const double* alpha_beta* parameter for *libtsg*.

transformAB is either an empty MATLAB matrix *[]* or a matrix of *dim* rows and 2 columns that contains the a and b parameters associated with the transformation of the domain. This is the same as *-inputfile* given to *tasgrid* with the *-makegrid* command. See Table A.1 for the various types of transformation.

OUTPUTS

lGrid is an object that saves the grid name and some additional parameters, the object is used by other functions to access the files associated with the grid.

ponits is an optional output MATLAB matrix that contains the abscissas associated with the sparse grid. Each abscissa is stored on one row of the matrix.

5.4 function tsgMakeQuadrature()

```
[ weights, points ] = tsgMakeQuadrature( dim, oned, depth, order, type,
                                         anisotropy, alpha_beta, transformAB )
```

Calls *tasgrid* with the *-makequadrature* command. The grid is not written to a file and only the abscissas and weights are returned. The inputs are the same as *tsgMakeGrid()*, however, no *sGridName* and *out* are needed.

INPUTS

Same as *tsgMakeGrid()* except no *sGridName* is needed as the grid isn't stored permanently and *out* is assumed to be zero.

OUTPUTS

weights is a MATLAB matrix containing the quadrature weights associated with the *points*.

ponits is a MATLAB matrix that contains the abscissas associated with the sparse grid. Each abscissa is stored on one row of the matrix.

5.5 function tsgRecycleGrid()

```
[ newp ] = tsgRecycleGrid( lGrid, depth, order, type, anisotropy )
```

Calls *tasgrid* with the *-recycle* command.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

depth same as in *tsgMakeGrid()* and *-depth*

order same as in *tsgMakeGrid()* and *-order*

type same as in *tsgMakeGrid()* and *-type*

anisotropy same as in *tsgMakeGrid()* and *-anisotropy*

transformAB same as in *tsgMakeGrid()* and *-inputfile*

OUTPUTS

newp is an optional output MATLAB matrix containing the set of abscissas that are not yet associated with values from the interpolated function.

5.6 function tsgGetQuadrature()

```
[ weights, points ] = tsgGetQuadrature( lGrid )
```

Calls *tasgrid* with the *-getquadrature* command.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

OUTPUTS

weights is a MATLAB matrix containing the quadrature weights associated with the *points*.

ponits is a MATLAB matrix that contains the abscissas associated with the sparse grid. Each abscissa is stored on one row of the matrix.

5.7 function tsgGetInterpolationWeights()

```
[ weights ] = tsgGetInterpolationWeights( lGrid, points )
```

Calls *tasgrid* with the *-getinterweights* command.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

points is a MATLAB matrix with *dim* number of columns and arbitrary number of rows. Each row represents one point of interest.

OUTPUTS

weights is a MATLAB matrix of *getNumPoitns()* number of columns and the same number of rows as *points*. The **weights** contain the interpolation weights associated with each point of interest in *points*.

5.8 function tsgGetNeededPoints()

```
[ newp ] = tsgGetNeededPoints( lGrid )
```

Calls *tasgrid* with the *-getneededpoints* command.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

OUTPUTS

newp is a MATLAB matrix of *getNumNeededPoitns()* number of rows and *out* number of columns. Each rows is an abscissa that is not yet associated with a value of the interpolated function.

5.9 function tsgLoadValues()

```
tsgLoadValues( lGrid, values )
```

Calls *tasgrid* with the *-loadvalues* command.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

values is a MATLAB matrix with *getNumNeededPoitns()* number of rows and *out* number of columns. Each row represents the values of the function at one point.

OUTPUTS

there are no output variables, however, the files associated with the grid are modified.

5.10 function tsgEvaluate()

```
[ result ] = tsgEvaluate( lGrid, points )
```

Calls *tasgrid* with the *-evaluate* option.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

points is a MATLAB matrix with rows representing points of interest.

OUTPUTS

result is a MATLAB matrix with rows representing the values of the interpolant at the point of interest.

5.11 function tsgIntegrate()

```
[ result ] = tsgIntegrate( lGrid )
```

Calls *tasgrid* with the *-integrate* option.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

OUTPUTS

result is a MATLAB matrix with one row that represents the integral of the interpolant over the canonical domain associated with the quadrature rule (i.e. *oned*).

5.12 function tsgRefineGrid()

```
[ newp ] = tsgRefineGrid( lGrid, tolerance, criteria )
```

Calls *tasgrid* with the *-refine* option.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

tolerance the tolerance for the refinement, same as *-tolerance*

criteria the type of refinement, same as *-refinement*

OUTPUTS

newp is a MATLAB matrix of *getNumNeededPoitns()* number of rows and *out* number of columns. Each rows is an abscissa that is not yet associated with a value of the interpolated function.

5.13 function tsgPrintStats()

```
tsgPrintStats( lGrid )
```

Calls *tasgrid* with the *-summary* option.

5.14 function tsgDeleteGrid()

```
tsgDeleteGrid( lGrid )
```

Delete all files associated with the grid. The *lGrid* object is no longer valid for further reference.

INPUTS

lGrid is an object created by *tsgMakeGrid()*

OUTPUTS

deletes all the files associated with the grid and the *lGrid* object is no longer valid.

5.15 function tsgDeleteGridByName()

```
tsgDeleteGrid( sGridName )
```

Delete all files associated with the grid with the name *sGridName*. This function should be called whenever the *lGrid* object has been lost (similar to a memory leak).

INPUTS

sGridName the name of the grid to be deleted.

OUTPUTS

deletes all the files associated with the grid.

5.16 function tsgListGridsByName()

```
tsgListGridsByName()
```

Reads the temp folder and finds all the grids regardless whether or not they are associated with grid objects.

5.17 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.

5.18 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 *tsgExample2* or *tsgExample5*. 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.

5.19 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, which is a potential security issue.
- 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.

A Types of One Dimensional Rules

A.1 Global Grids

All global grids use Lagrange polynomial for interpolation. Gauss rules are most suitable for optimal integration with respect to a given weight. The non-Gauss rules are more suitable for interpolation. However, depending on the structure of $f(x)$, it is perfectly possible for a non-Gauss rule to produce a more accurate integral or for a Gauss rule to produce a more accurate interpolant. For more details on the various quadrature rules and their properties see [1, 5, 7, 14, 17, 24].

Clenshaw-Curtis

Also known as nested Chebyshev. It uses nested Chebyshev points and integration weights on $\Gamma_1 = [-1, 1]$. The rule is known to produce heavy bias on the main axis. When used with the *basis* type, the axis growth is slowed, which is equivalent to [20]. A *basis* type grid is guaranteed to interpolate exactly any polynomial of degree no more than the *depth* property, the rule will also interpolate some higher order polynomials depending on the *depth* and dimension.

Chebyshev

The Chebyshev rule on $\Gamma_1 = [-1, 1]$ with non-nested points that grow one point at a level. This is the only rule where *level* and *basis* types grids are identical. A *basis* type grid is guaranteed to interpolate exactly any polynomial of degree no more than the *depth* property.

Chebyshev two point growth

The Clenshaw-Curtis rule on $\Gamma_1 = [-1, 1]$ is split into more levels so that every level adds exactly two points to the previous one. A *basis* type grid is guaranteed to interpolate exactly any polynomial of degree no more than the *depth* property.

Note: this is not a stable rule and it should be used with great caution.

Fejer type 2

The Fejer type 2 rule on $\Gamma_1 = [-1, 1]$ that is nested and uses Chebyshev points and weights. Unlike the Clenshaw-Curtis rule, Fejer type 2 assumes that $f(x)$ vanishes at the two end points. A *basis* type grid is guaranteed to interpolate exactly any polynomial of degree no more than the *depth* property, so long as the polynomial vanishes at the boundary of the domain.

Gauss-Legendre

Optimal integration rule for functions on a bounded domain and constant weight:

$$\int_{-1}^{1} f(x)dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Gauss-Chebyshev Type 1

Optimal integration rule for functions on a bounded domain and weight $\rho_1(x) = (1 - x^2)^{-0.5}$:

$$\int_{-1}^{1} f(x)\rho_1(x)dx = \int_{-1}^{1} f(x)(1 - x^2)^{-0.5}dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Gauss-Chebyshev Type 2

Optimal integration rule for functions on a bounded domain and weight $\rho_1(x) = (1 - x^2)^{+0.5}$:

$$\int_{-1}^{1} f(x)\rho_1(x)dx = \int_{-1}^{1} f(x)(1 - x^2)^{+0.5}dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Gauss-Gegenbauer

Generalized Gauss-Chebyshev rule. Optimal integration rule for functions on a bounded domain and weight $\rho_1(x) = (1 - x^2)^{\alpha}$:

$$\int_{-1}^{1} f(x)\rho_1(x)dx = \int_{-1}^{1} f(x)(1 - x^2)^{\alpha}dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Gauss-Jacobi

Generalized Gauss-Gegenbauer rule. Optimal integration rule for functions on a bounded domain and weight $\rho_1(x) = (1-x)^\alpha(1+x)^\beta$:

$$\int_{-1}^{1} f(x)\rho_1(x)dx = \int_{-1}^{1} f(x)(1-x)^\alpha(1+x)^\beta dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Gauss-Laguerre

This is the generalized Gauss-Laguerre quadrature. Optimal integration rule for functions on a bounded domain and weight $\rho_1(x) = x^\alpha e^{-x}$:

$$\int_{0}^{\infty} f(x)\rho_1(x)dx = \int_{0}^{\infty} f(x)x^\alpha e^{-x} dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Note: Interpolation with this rule is extremely unstable due to the infinite range of the domain.

Gauss-Hermite

This is the generalized Gauss-Hermite quadrature. Optimal integration rule for functions on a bounded domain and weight $\rho_1(x) = |x|^\alpha e^{-x^2}$:

$$\int_{-\infty}^{\infty} f(x)\rho_1(x)dx = \int_{-\infty}^{\infty} f(x)|x|^\alpha e^{-x^2} dx.$$

A *basis* type grid is guaranteed to integrate exactly any polynomial of degree no more than the value of the *depth* property.

Note: Interpolation with this rule is extremely unstable due to the infinite range of the domain.

A.2 Local Polynomials

The local polynomial rules are best for interpolation of a function with locally sharp gradients. The domain of interpolation is $\Gamma_1 = [-1, 1]$. Local polynomial grids allow for different types of local adaptivity. The maximum degree of the polynomials is specified by the *order* property. Note

Canonical Integral	Generalized Integral	Domain of Canonical Interpolation	Additional Assumptions	Nested
Clenshaw-Curtis: *rule_clenshawcurtis*, *clenshaw-curtis*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	N/A	Yes
Chebyshev: *rule_chebyshev*, *chebyshev*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	N/A	No
Fejer type 2: *rule_fejer2*, *fejer-2*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	$f(-1) = f(1) = 0$	Yes
Gauss-Legendre: *rule_gausslegendre*, *gauss-legendre*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	N/A	No
Gauss-Chebyshev type 1: *rule_gausschebyshev1*, *gauss-chebyshev-1*				
$\int_{-1}^{1} f(x)(1-x^2)^{-0.5}dx$	$\int_{a}^{b} f(x)(b-x)^{-0.5}(x-a)^{-0.5}dx$	$[-1, 1]$	N/A	No
Gauss-Chebyshev type 2: *rule_gausschebyshev2*, *gauss-chebyshev-2*				
$\int_{-1}^{1} f(x)(1-x^2)^{0.5}dx$	$\int_{a}^{b} f(x)(b-x)^{0.5}(x-a)^{0.5}dx$	$[-1, 1]$	N/A	No
Gauss-Gegenbauer: *rule_gausssgegenbauer*, *gauss-gegenbauer*				
$\int_{-1}^{1} f(x)(1-x^2)^{\alpha}dx$	$\int_{a}^{b} f(x)(b-x)^{\alpha}(x-a)^{\alpha}dx$	$[-1, 1]$	Must specify α	No
Gauss-Jacobi: *rule_gaussjacobi*, *gauss-jacobi*				
$\int_{-1}^{1} f(x)(1-x)^{\alpha}(1+x)^{\beta}dx$	$\int_{a}^{b} f(x)(b-x)^{\alpha}(x-a)^{\beta}dx$	$[-1, 1]$	Must specify α, β	No
Gauss-Laguerre: *rule_gausslaguerre*, *gauss-laguerre*				
$\int_{0}^{\infty} f(x)x^{\alpha}e^{-x}dx$	$\int_{a}^{\infty} f(x)(x-a)^{\alpha}e^{-b(x-a)}dx$	$[0, \infty]$	Must specify α	No
Gauss-Hermite: *rule_gausshermite*, *gauss-hermite*				
$\int_{-\infty}^{\infty} f(x)x^{\alpha}e^{-x^2}dx$	$\int_{-\infty}^{\infty} f(x)(x-a)^{\alpha}e^{-b(x-a)^2}dx$	$[-\infty, \infty]$	Must specify α	No
Local Polynomials: *rule_pwpolynomial*, *local-polynomial*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	N/A	Yes
Local Polynomials with Zero Boundary: *rule_pwpolynomial0*, *local-polynomial-zero*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	$f(-1) = f(1) = 0$	Yes
Local Wavelets: *rule_wavelet*, *local-wavelet*				
$\int_{-1}^{1} f(x)dx$	$\int_{a}^{b} f(x)dx$	$[-1, 1]$	N/A	Yes

Table A.1: Summary of the available quadrature rules. For each rule, we have the enumerated type as *rule_**** followed by the string given to the *tasgrid* and MATLAB interfaces.

that at a given *depth* we can only use a polynomial of the same order. There are two variations of the local polynomial rule that assume zero and non-zero boundary. Also note that quadrature rules based on local polynomials may have abscissas associated with zero weights. For more details on the local polynomials see [10, 15, 16]

A.3 Wavelet Grid

Same as local polynomials, however, it assumes that *order* is either 1 or 3. The boundary is not assumed to be zero and only one type of local refinement is possible. Functionality would be expanded in the future. For more details on the wavelet grid see [10, 13, 25]

REFERENCES

[1] M. ABRAMOWITZ AND I. A. STEGUN, *Handbook of Mathematical Functions: With Formulars, Graphs, and Mathematical Tables*, vol. 55, Dover Publications, 1964. 35

[2] 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. 3

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

[4] V. BARTHELMANN, E. NOVAK, AND K. RITTER, *High dimensional polynomial interpolation on sparse grids*, Advances in Computational Mathematics, 12 (2000), pp. 273–288. 3

[5] P. J. DAVIS AND P. RABINOWITZ, *Methods of numerical integration*, Courier Dover Publications, 2007. 35

[6] 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. 3

[7] S. ELHAY AND J. KAUTSKY, *Algorithm 655: Iqpack: Fortran subroutines for the weights of interpolatory quadratures*, ACM Transactions on Mathematical Software (TOMS), 13 (1987), pp. 399–415. 35

[8] T. GERSTNER AND M. GRIEBEL, *Numerical integration using sparse grids*, Numerical algorithms, 18 (1998), pp. 209–232. 2, 3, 7

[9] ——, *Dimension–adaptive tensor–product quadrature*, Computing, 71 (2003), pp. 65–87. 3

[10] M. GRIEBEL, *Adaptive sparse grid multilevel methods for elliptic pdes based on finite differences*, Computing, 61 (1998), pp. 151–179. 2, 3, 38

[11] M. GRIEBEL AND J. HAMAEKERS, *Sparse grids for the schrödinger equation*, M2AN Math. Model. Numer. Anal, 41 (2007), pp. 215–247. 7

[12] 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. 3

[13] 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. 3, 38

[14] J. KAUTSKY AND S. ELHAY, *Calculation of the weights of interpolatory quadratures*, Numerische Mathematik, 40 (1982), pp. 407–422. 35

[15] 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. 3, 38

[16] 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. 3, 38

[17] R. S. MARTIN AND J. WILKINSON, *The implicitql algorithm*, Numerische Mathematik, 12 (1968), pp. 377–383. 35

[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, 3, 7, 8

[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, 3, 7

[20] K. PETRAS, *Smolyak cubature of given polynomial degree with few nodes for increasing dimension*, Numerische Mathematik, 93 (2003), pp. 729–753. 35

[21] J. SHEN AND L.-L. WANG, *Sparse spectral approximations of high-dimensional problems based on hyperbolic cross*, SIAM Journal on Numerical Analysis, 48 (2010), pp. 1087–1109. 7

[22] 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, 3, 7

[23] M. STOYANOV, *Hierarchy-direction selective approach for locally adaptive sparse grids*, Tech. Rep. ORNL/TM-2013/384, Oak Ridge National Laboratory, 2013. 17

[24] A. H. STROUD AND D. SECREST, *Gaussian quadrature formulas*, vol. 374, Prentice-Hall Englewood Cliffs, NJ, 1966. 35

[25] W. SWELDENS AND P. SCHRÖDER, *Building your own wavelets at home*, in Wavelets in the Geosciences, Springer, 2000, pp. 72–107. 38

[26] 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. 3

[27] 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. 3


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