A Manual for the Template Class Library `igpm_t_lib`

Siegfried Müller* and Alexander Voß†

August 25, 2000

Abstract

This is a manual for the C++ template library `igpm_t_lib` providing

i) a vector class (`tvector_n`),

ii) a multidimensional index class (`tmultiindex`) and several supporting classes for multidimensional access (`tlevelmultiindex`, `tpackedltmi`, `tmultirange`),

iii) various hash-table classes (`thashmap`, `thashmap_linked`, `thashmap_linked_one`) and

iv) a memory management class (`tmemheap`).

These classes are designed in view of applications to local multi-scale transformations. We outline the design criteria from algorithmic requirements by means of an example and explain how to realize it using the library classes.

Key Words: template classes, hash-maps, sparse data structures

AMS Subject Classification: 90C08, 90C29, 68Q65

*The work of this author is supported by the Deutsche Forschungsgemeinschaft (DFG) in the Collaborative Research Center SFB 401 "Modulation of Flow and Fluid-Structure Interaction at airplane wings".

†The work of this author is supported by the DFG in the Priority Program "Analysis and Numerics of Conservation Laws".

Contents

1 Introduction

¿From a theoretical point of view numerical algorithms are usually designed and analyzed with respect to efficiency and accuracy. Here the complexity of the algorithm at hand is usually measured in terms of floating point operations. In practice, however, the efficiency of the code will also depend on the selected *data structures* and the underlying *memory management*.
These ingredients are crucial because they have to preserve the theoretical complexity in order to provide a reasonable scaling of the algorithm. Otherwise the time used for data management dominates the overall time of the program. For instance, a mathematical algorithm of order $O(n)$, e.g. thresholding a set with cardinality n might behave like $O(n \log n)$ because internally the container chosen to represent the set sorts its data whenever an element is erased. Therefore one rarely can use a given data structure as a black box without knowing some internal details. Consequently the data structures should not be designed independently from their application.
This motivated the development of the specific template library `igpm_t_lib` which in particular can be used for applications containing local multi-scale transformations. These are used in the context of adaptive schemes based on multi-scale decompositions, see e.g. [BKV].

Due to the nature of adaptivity the most important data structure is an efficient index management for sparse data. Section 2 explains why in our context different types of *hash-tables* seem to be the most appropriate. Several supporting data structures are necessary to realize the adaptive schemes described in [M], [BKV]. Those with a close relationship to the hash-tables are contained in the class library `igpm_t_lib` as well. The main data structures are

- a *vector* class with an arbitrary but fixed dimension, see `tvector_n`,
- a multi-dimensional *index* class containing at least a multi-index or a multi-index and a level and additionally blocks of multi-indices representing supports, see `tmultiindex`, `tlevelmultiindex` and the class `tmultirange`,
- a compressed version of such an *index* class, see `tpackedlmi<2>`,
- various *hash-map* classes by which elements can be stored in an unordered way providing fast access, see `thashtab`, `thashtab_linked` and `thashtab_linked_one`, and
- an efficient *memory management* where memory space for variables of a given type is fast provided and freed, see `tmemheap`.

This paper is structured as follows: In Section 2 we briefly outline the frame for a local multi-scale transformation applied to a sequence of mean values. Additionally we present the resulting algorithms and derive design criteria for appropriate data structures. A complete description of all classes contained in `igpm_t_lib` can be found in Section 3. Section 4 describes the realization of the adaptive algorithm from Section 2 in terms of library classes. The design of the classes as well as the implementation of certain classes hide some special C++ features which are explained in Section 5. This section refers to the more advanced and interested C++ programmer and can be ignored if one only wants to use the library.

We like to thank Frank Bramkamp, Thomas Schlinkmann and Michael Konik for their fruitful ideas during the first design process. Last but not least we thank Arne Barinka for proof-reading and giving helpful comments on the different versions of this script.

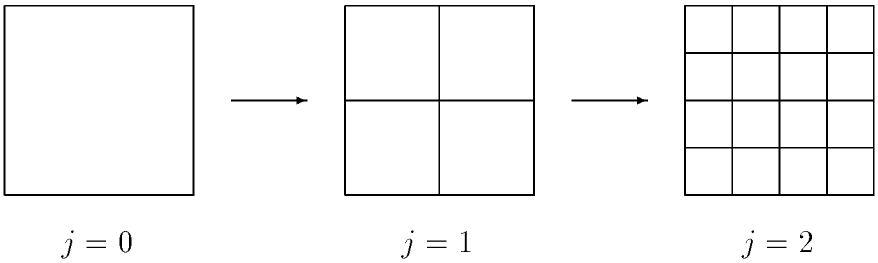


Figure 1: Sequence of nested grids

2 Example: Local Multiscale Transformation

In this section we summarize the ingredients by which a sequence of averages can be successively decomposed into a sequence of coarse scale averages and details corresponding to a nested grid hierarchy. By means of the details an adaptive grid, in particular, a locally refined grid with hanging nodes, can be determined. This tool can be incorporated to finite volume methods in order to accelerate the computation. The frame of the concept and its analytical investigation as well as its application is presented in [M].

First of all, we briefly outline the multi-scale analysis and present the algorithms for the local multi-scale transformation. From these algorithms we deduce the design criteria for the template classes to be described in the sequel.

2.1 Multi-scale Setting for Averages

Starting point is a sequence of *nested grids* $\mathcal{G}_j := \{V_{j,k}\}_{k \in I_j}$, $j = 0, \ldots, L$, by which a computational domain $\Omega \subset \mathbb{R}^d$ is decomposed into cells. Here the coarsest grid is denoted by 0 and the finest grid by L. Each grid $\mathcal{G}_j$ is assumed to be a partition of Ω, i.e., $\Omega = \bigcup_{k \in I_j} V_{j,k}$ and the cells of two neighboring levels are nested, i.e., $V_{j,k} = \bigcup_{r \in \mathcal{M}^0_{j,k}} V_{j+1,r}$, $k \in I_j$. The index sets $\mathcal{M}^0_{j,k} \subset I_{j+1}$ correspond to the new cells on level $j+1$ resulting from the refinement of the cell $V_{j,k}$. In particular, these refinement sets are assumed to be pairwise disjoint, i.e., $\mathcal{M}^0_{j,k} \cap \mathcal{M}^0_{j,k'} = \emptyset$ for $k \neq k'$, $k, k' \in I_j$, and they build a partition of the index set I_{j+1}, i.e., $\bigcup_{k \in I_j} \mathcal{M}^0_{j,k} = I_{j+1}$. A simple example for a nested grid hierarchy is shown in Figure 1 where a coarse grid is successively refined with increasing refinement level. In our applications we confine ourselves to structured grids and uniform dyadic refinements, i.e.,

$\# \mathcal{M}^0_{j,k} = 2^d$. Moreover, we only consider a finite domain Ω and, hence, $N_j := \# I_j < \infty$.

By means of the grids $\mathcal{G}_j$ we introduce the sequences of averages $\hat{\boldsymbol{u}}_j := \{\hat{u}_{j,k}\}_{k\in I_j}$ corresponding to a scalar, integrable function $u \in L^1(\Omega, \mathbb{R})$ as the inner product $u_{j,k} := \langle u, \varphi_{j,k}\rangle_{L^2(\Omega)}$ of u with the L^1–normalized box function $\varphi_{j,k}(\boldsymbol{x}) := |V_{j,k}|^{-1}\chi_{V_{j,k}}(\boldsymbol{x})$, $\boldsymbol{x} \in \Omega$, where $|V_{j,k}| := \int_{V_{j,k}} 1\, d\boldsymbol{x}$ denotes the cell volume.

The nesting of the grids as well as the linearity of the integration operator imply the two–scale relation

$$\hat{u}_{j,k} = \sum_{r\in\mathcal{M}^0_{j,k}} \frac{|V_{j+1,r}|}{|V_{j,k}|}\, \hat{u}_{j+1,r} = \sum_{r\in\mathcal{M}^0_{j,k}} m^{j,0}_{r,k}\, \hat{u}_{j+1,r}, \tag{2.1}$$

i.e., the coarse–grid average can be represented by a linear combination of the corresponding fine–grid averages. Consequently, the averages can be successively computed starting on the finest level. However it is not possible to reverse this process, i.e., determine the fine–grid averages by the coarse–grid averages only. Since information is destroyed by the averaging process (2.1), we have to store these information by means of additional coefficients. In analogy to the averages, these can be represented as inner products $d_{j,k,e} = \langle u, \psi_{j,k,e}\rangle_{L^2(\Omega)}$ of the function u with appropriate *wavelets* $\psi_{j,k,e}$, $e \in E^* := \{1, \ldots, 2^d-1\}$, $k \in I_j$, $j = 0, \ldots, L-1$. The wavelets exist provided that the linear spaces $S_j := \mathrm{span}\{\varphi_{j,k}\ ;\ k \in I_j\}$ are nested, i.e., $S_j \subset S_{j+1}$. From the nesting we conclude the existence of the complement spaces $W_j := S_{j+1}/S_j$. Hence, the wavelets can be interpreted as a basis for the complement spaces, i.e., $W_j := \mathrm{span}\{\psi_{j,k,e}\ ;\ k \in I_j,\ e \in E^*\}$. Since the box functions and the wavelets are assumed to be linearly independent, there exists a two–scale relation for the details, i.e.,

$$d_{j,k,e} = \sum_{r\in\mathcal{M}^e_{j,k}} m^{j,e}_{r,k}\, \hat{u}_{j+1,r} \tag{2.2}$$

with the index sets $\mathcal{M}^e_{j,k} \subset I_j$. On the other hand, we deduce from the change of bases the existence of a converse two–scale relation

$$\hat{u}_{j+1,k} = \sum_{r\in\mathcal{G}^0_{j,k}} g^{j,0}_{r,k}\, \hat{u}_{j,r} + \sum_{e\in E^*}\sum_{r\in\mathcal{G}^e_{j,k}} g^{j,e}_{r,k}\, d_{j,r,e} \tag{2.3}$$

with the index sets $\mathcal{G}^e_{j,k} \subset I_j$.

So far, the wavelets are not yet determined. Since they build any basis for the complement spaces, they are not unique. This degree of freedom is made use of in the construction of an appropriate wavelet basis which is adapted to the problem at hand. In principle, there are four design criteria, namely, (i) the two–scale transformation (2.1) and (2.2) can be reversed by (2.3) and vica versa, i.e., the relations are equivalent, (ii) the L^2–normalized bases satisfy a so–called Riesz property, i.e., they are l_2–stable, (iii) the basis functions are locally supported, i.e., the number of indices in the index sets $\mathcal{M}^e_{j,k}$, $\mathcal{G}^e_{j,k}$, $e \in E := E^* \cup \{0\}$, is uniformly bounded and much smaller than N_j and (iv) the wavelets provide good approximation properties, i.e., the inner products $\langle p, \psi_{j,k,e}\rangle_{L^2(\Omega)}$ vanish for all polynomials with degree less than M. Here we will not present the technical details for constructing an appropriate wavelet basis but refer to [M].

The main objective of the multi-scale decomposition determined by successively applying (2.1) and (2.2) is the construction of an adaptive grid. For this purpose, we introduce the set of significant details

$$\mathcal{D}_{L,\boldsymbol{\varepsilon}} := \{(j,k,e) \; ; \; |d_{j,k,e}| > \varepsilon_j, \; e \in E^*, \; k \in I_j, \; j \in \{0, \ldots, L-1\}\}$$

by which all details $d_{j,k,e}$ smaller than a prescribed threshold value ε_j are discarded. By means of this set we define the locally refined grid with hanging nodes, see e.g. Fig. 2, corresponding to the index set $\mathcal{G}_{L,\boldsymbol{\varepsilon}} \subset \bigcup_{j=0}^{L} I_j$ such that $\Omega = \bigcup_{(j,k)\in\mathcal{G}_\varepsilon} V_{j,k}$.

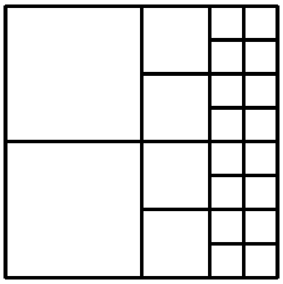


Figure 2: Locally refined grid with hanging nodes

To this end, we traverse through the levels starting on the coarsest grid level and refine a cell $V_{j,k}$ as long as there exists a significant detail, i.e., $(j,k,e) \in \mathcal{D}_{L,\boldsymbol{\varepsilon}}$ for at least one $e \in E^*$. This procedure works provided the set of significant details is *graded*, see [M] for details.

For later use we introduce the local index sets

$$I_{j,\boldsymbol{\varepsilon}} := \{k \; ; \; (j,k) \in \mathcal{G}_{L,\boldsymbol{\varepsilon}}\} \subset I_j, \quad J_{j,\boldsymbol{\varepsilon}} := \{k \; ; \; (j,k,e) \in \mathcal{D}_{L,\boldsymbol{\varepsilon}}, \; e \in E^*\} \subset I_j.$$

2.2 Algorithms for Local Multi-scale Transformations

We notice that the complexity of the index sets $\mathcal{G}_{L,\boldsymbol{\varepsilon}}$ and $\mathcal{D}_{L,\boldsymbol{\varepsilon}}$ corresponding to the adaptive grid and the set of significant details, respectively, can be much smaller than the number N_L of cells corresponding to the finest level. In the sequel, we will present algorithms by which the multi-scale transformation (2.1) and (2.2) as well as its inverse (2.3) can be locally realized, i.e., we perform the change of bases between the local bases $\{\varphi_{0,k}\ ;\ k \in I_j\} \cup \{\psi_{j,k,e}\ ;\ (j,k,e) \in \mathcal{D}_{L,\boldsymbol{\varepsilon}}\}$ and $\{\varphi_{j,k}\ ;\ (j,k) \in \mathcal{G}_{L,\boldsymbol{\varepsilon}}\}$. The objective is the construction of algorithms where the memory size and the number of floating point operations correspond to $\#\,\mathcal{G}_{L,\boldsymbol{\varepsilon}}$ and $\#\,\mathcal{D}_{L,\boldsymbol{\varepsilon}}$, respectively.

To this end, we have to specify the index sets. Here we only consider a *dyadic* grid hierarchy of *structured* grids where the cells on each level can be enumerated by a multi-index, i.e.,

$$I_j := \bigotimes_{i=1}^{d} \{0, N_{j,i}-1\} \subset \mathbb{N}_0^d, \qquad j = 0, \ldots, L$$

with $N_{j,i} = 2\,N_{j-1,i}$. ¿From the dyadic grid refinement we deduce the refinement sets

$$\mathcal{M}_{j,\boldsymbol{k}}^0 = \{2\boldsymbol{k} + \boldsymbol{i}\ ;\ \boldsymbol{i} \in E\}$$

where $E := \{0,1\}^d \simeq \{0,\ldots,2^{d-1}\}$, $E^* := E \backslash \{\boldsymbol{0}\}$. According to the wavelets constructed in [M] the remaining index sets are determined by

- $\mathcal{M}_{j,\boldsymbol{k}}^{\boldsymbol{e}} = \mathcal{M}_{j,\boldsymbol{k}}^1 = \bigotimes_{i=1}^d \{2l_{j,k_i}, \ldots, 2l_{j,k_i} + 4s + 1\},\ \boldsymbol{k} \in I_j,\ \boldsymbol{e} \in E^*,$
- $\mathcal{G}_{j,\boldsymbol{k}}^0 = \bigotimes_{i=1}^d \{l_{j,\lfloor k_i/2 \rfloor}, \ldots, l_{j,\lfloor k_i/2 \rfloor} + 2s + 1\},\ \boldsymbol{k} \in I_{j+1},$
- $\mathcal{G}_{j,\boldsymbol{k}}^{\boldsymbol{e}} = \mathcal{G}_{j,\boldsymbol{k}}^1 = \{\lfloor \boldsymbol{k}/2 \rfloor\},\ \boldsymbol{k} \in I_{j+1},\ \boldsymbol{e} \in E^*$

with the integer

$$l_{j,k} = \begin{cases} 0 & ,\ 0 \le k_i \le s-1 \\ k_i - s & ,\ s \le k_i \le N_{j,i} - 1 - s \\ N_{j,i} - 1 - 2s & ,\ N_{j,i} - s \le k_i \le N_{j,i} - 1 \end{cases}$$

where s is a parameter in the wavelet construction. In addition, we need the following index sets

- $\mathcal{M}_{j,\boldsymbol{k}}^{*,0} = \{\lfloor \boldsymbol{k}/2 \rfloor\},\ \boldsymbol{k} \in I_{j+1},$
- $\mathcal{M}_{j,\boldsymbol{k}}^{*,\boldsymbol{e}} = \mathcal{M}_{j,\boldsymbol{k}}^{*,1} = \bigotimes_{i=1}^d \{\bar{l}_{j,\lfloor k_i/2 \rfloor}^0, \ldots, \bar{l}_{j,\lfloor k_i/2 \rfloor}^1\},\ \boldsymbol{k} \in I_{j+1},\ \boldsymbol{e} \in E^*,$

- $\mathcal{G}^{*,0}_{j,\boldsymbol{k}} = \bigotimes_{i=1}^{d}\{2\bar{l}^0_{j,k_i},\ldots,2\bar{l}^1_{j,k_i}+1\},\ \boldsymbol{k}\in I_j,$

- $\mathcal{G}^{*,\boldsymbol{e}}_{j,\boldsymbol{k}} = \mathcal{G}^{*,1}_{j,\boldsymbol{k}} = \{2\boldsymbol{k}+\boldsymbol{i}\ ;\ \boldsymbol{i}\in E\},\ \boldsymbol{k}\in I_j,\ \boldsymbol{e}\in E^*$

with the integers

$$\bar{l}^0_{j,k_\mu} = \begin{cases} 0 & ,\ 0\le k_i\le 2s \\ k_i - s & ,\ 2s+1\le k_i\le N_{j,i}-1 \end{cases},$$

$$\bar{l}^1_{j,k_\mu} = \begin{cases} k_i + s & ,\ 0\le k_i\le N_{j,i}-2s-2 \\ N_{j,i}-1 & ,\ N_{j,i}-2s-1\le k_i\le N_{j,i}-1 \end{cases}.$$

These index sets can be interpreted as the supports determined by the non–vanishing elements of a column or a row corresponding to the mask matrices $\boldsymbol{M}_{j,e} := (m^{j,e}_{r,k})_{r\in I_{j+1},k\in I_j}$ and $\boldsymbol{G}_{j,e} := (g^{j,e}_{r,k})_{r\in I_j,k\in I_{j+1}}$.

By means of the index sets the algorithms for the local transformations are then determined by. First of all we consider the encoding algorithm. This algorithm needs the following input information (i) the number of refinement levels L, (ii) the local index sets $I_{j,\boldsymbol{\varepsilon}},\ j=0,\ldots,L$, representing the adaptive grid $\mathcal{G}_{L,\boldsymbol{\varepsilon}}$ and (iii) the corresponding sequences of local averages $(\hat{u}_{j,\boldsymbol{k}})_{\boldsymbol{k}\in I_{j,\boldsymbol{\varepsilon}}},\ j=0,\ldots,L$. Then the following output is provided by the encoding algorithm (i) the local index sets of significant details $J_{j,\boldsymbol{\varepsilon}},\ j=0,\ldots,L-1$, (ii) the corresponding sequences of significant details $(d_{j,\boldsymbol{k},\boldsymbol{e}})_{\boldsymbol{k}\in J_{j,\boldsymbol{\varepsilon}},\boldsymbol{e}\in E^*},\ j=0,\ldots,L-1$, and (iii) the averages $(\hat{u}_{0,\boldsymbol{k}})_{\boldsymbol{k}\in I_0}$ of the coarsest grid. Notice that the set of significant details $\mathcal{D}_{L,\boldsymbol{\varepsilon}}$ is not necessarily graded and it might include non–significant details.

Algorithm 2.1 *(Local multi-scale transformation)*

<u>for</u> $j=L-1$ <u>downto</u> *0* <u>do</u>

1. $U^0_j := \bigcup_{\boldsymbol{r}\in I_{j+1,\boldsymbol{\varepsilon}}}\mathcal{M}^{*,0}_{j,\boldsymbol{r}},\ U^1_j := \bigcup_{\boldsymbol{r}\in I_{j+1,\boldsymbol{\varepsilon}}}\mathcal{M}^{*,1}_{j,\boldsymbol{r}}$

2. $\hat{u}_{j,\boldsymbol{k}} := \sum_{\boldsymbol{r}\in\mathcal{M}^0_{j,\boldsymbol{k}}} m^{j,0}_{\boldsymbol{r},\boldsymbol{k}}\,\hat{u}_{j+1,\boldsymbol{r}},\ \boldsymbol{k}\in U^0_j$

3. $P_{j+1} := \bigcup_{\boldsymbol{k}\in U^1_j}\mathcal{M}^1_{j,\boldsymbol{k}}/I_{j+1,\boldsymbol{\varepsilon}}$

4. $\hat{u}_{j+1,\boldsymbol{k}} = \sum_{\boldsymbol{r}\in\mathcal{G}^0_{j,\boldsymbol{k}}} g^{j,0}_{\boldsymbol{r},\boldsymbol{k}}\,\hat{u}_{j,\boldsymbol{r}},\ \boldsymbol{k}\in P_{j+1}$

5. $d_{j,\boldsymbol{k},\boldsymbol{e}} := \sum_{\boldsymbol{r}\in\mathcal{M}^1_{j,k}} m^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}\hat{u}_{j+1,\boldsymbol{r}},\ \boldsymbol{k}\in U^1_j,\ \boldsymbol{e}\in E^*$

6. <u>if</u> $\boldsymbol{k}\notin U^0_j\cup J_{j,\boldsymbol{\varepsilon}}$ <u>then</u> *delete* $m^{j,0}_{\boldsymbol{r},\boldsymbol{k}},\ \boldsymbol{r}\in\mathcal{M}^0_{j,\boldsymbol{k}}$

7. <u>if</u> $\boldsymbol{k} \notin U_j^1 \cup \bigcup_{\boldsymbol{l} \in P_{j+1}} \mathcal{G}_{j,\boldsymbol{l}}^1$ <u>then</u> *delete* $m_{\boldsymbol{r},\boldsymbol{k}}^{j,\boldsymbol{e}},\ \boldsymbol{r} \in \mathcal{M}_{j,\boldsymbol{k}}^1,\ \boldsymbol{e} \in E^*$

Finally we consider the decoding algorithm. This algorithm needs the following input information (i) the number of refinement levels L, (ii) the local index sets of significant details $J_{j,\boldsymbol{\varepsilon}},\ j = 0, \ldots, L-1$, (iii) the corresponding sequences of significant details $(d_{j,\boldsymbol{k},\boldsymbol{e}})_{\boldsymbol{k} \in J_{j,\boldsymbol{\varepsilon}}, \boldsymbol{e} \in E^*},\ j = 0, \ldots, L-1$ and (iv) the averages $(\hat{u}_{0,\boldsymbol{k}})_{\boldsymbol{k} \in I_0}$ of the coarsest grid. Notice that the set of significant details $\mathcal{D}_{L,\boldsymbol{\varepsilon}}$ has to be graded, see [M]. Then the following output is provided by the decoding algorithm (i) the local index sets $I_{j,\boldsymbol{\varepsilon}},\ j = 0, \ldots, L$, representing the adaptive grid $\mathcal{G}_{L,\boldsymbol{\varepsilon}}$ and (ii) the corresponding sequences of local averages $(\hat{u}_{j,\boldsymbol{k}})_{\boldsymbol{k} \in I_{j,\boldsymbol{\varepsilon}}},\ j = 0, \ldots, L$.

Algorithm 2.2 *(Local inverse multi-scale transformation)*

$I_0^+ := I_0$

<u>for</u> $j = 0$ <u>to</u> *L-1* <u>do</u>

1. $I_{j+1}^+ := \bigcup_{\boldsymbol{l} \in J_{j,\boldsymbol{\varepsilon}}} \mathcal{G}_{j,\boldsymbol{l}}^{*,1}$

2. $\hat{u}_{j+1,\boldsymbol{k}} = \sum_{\boldsymbol{r} \in \mathcal{G}_{j,\boldsymbol{k}}^0} g_{\boldsymbol{r},\boldsymbol{k}}^{j,0}\, \hat{u}_{j,\boldsymbol{r}} + \sum_{\boldsymbol{e} \in E^*} \sum_{\boldsymbol{r} \in \mathcal{G}_{j,\boldsymbol{k}}^1} g_{\boldsymbol{r},\boldsymbol{k}}^{j,\boldsymbol{e}}\, d_{j,\boldsymbol{e},\boldsymbol{r}},\ \boldsymbol{k} \in I_{j+1}^+$

3. $I_j^- := \bigcup_{\boldsymbol{k} \in I_{j+1}^+} \mathcal{M}_{j,\boldsymbol{k}}^{*,0},\ I_{j,\boldsymbol{\varepsilon}} := I_j^+ / I_j^-$ *and delete* $\hat{u}_{j,\boldsymbol{k}},\ \boldsymbol{k} \in I_j^-$

4. <u>for</u> $\boldsymbol{k} \notin I_{j+1}^+ \cap \{\boldsymbol{k} \ :\ g_{\boldsymbol{r},\boldsymbol{k}}^{j,0}$ *exists*, $\boldsymbol{r} \in \mathcal{G}_{j,\boldsymbol{k}}^0\}$ <u>then</u> *delete* $g_{\boldsymbol{r},\boldsymbol{k}}^{j,0},\ \boldsymbol{r} \in \mathcal{G}_{j,\boldsymbol{k}}^0$

5. <u>for</u> $\boldsymbol{k} \notin I_{j+1}^+ \cap \{\boldsymbol{k} \ :\ g_{\boldsymbol{r},\boldsymbol{k}}^{j,\boldsymbol{e}}$ *exists*, $\boldsymbol{r} \in \mathcal{G}_{j,\boldsymbol{k}}^{\boldsymbol{e}},\ \boldsymbol{e} \in E^*\}$ <u>then</u> *delete* $g_{\boldsymbol{r},\boldsymbol{k}}^{j,\boldsymbol{e}},\ \boldsymbol{r} \in \mathcal{G}_{j,\boldsymbol{k}}^{\boldsymbol{e}}$

2.3 Algorithmic Requirements and Design Criteria

In the previous sections we outlined the core ingredients of a local multi-scale transformation and its inverse. The complexity of the corresponding algorithms is optimal in the sense that the number of operations is proportional to the number of unknowns, i.e., $\#\mathcal{D}_{L,\boldsymbol{\varepsilon}}$ and $\#\mathcal{G}_{L,\boldsymbol{\varepsilon}}$, respectively. We are now concerned with the design of an optimal code, i.e., the memory requirements and the CPU time are proportional to the same complexity of the local algorithms. To this end, we consider the Algorithms 2.1 and 2.2 by which the local multi-scale transformation and its inverse are performed. From these we deduce the following demands:

- The local transformations are performed level by level. Therefore data structures by which the local averages and the significant details are stored have to maintain the *level information*.

- For each level we have to collect the indices of cells to be refined or coarsened. Therefore we need a data structure for index sets.

- For the computation of the local (inverse) two–scale transformation (2.1) — (2.2) and (2.3), respectively, we have to determine the *supports* of the mask matrices $\boldsymbol{M}_{j,0}$, $\boldsymbol{G}_{j,0}$ and $\boldsymbol{M}_{j,\boldsymbol{e}}$, $\boldsymbol{G}_{j,\boldsymbol{e}}$, $\boldsymbol{e} \in E^*$. The summation is then performed on a column of these matrices. Obviously, we never access a single element of the matrices but a *column* corresponding to the non–vanishing entries indicated by the support. These data group has to be maintained by a data structure.

- Performing the local transformations we have to check whether a matrix column or a local average as well as a significant detail *exists*, i.e., this information has already been computed and has been stored.

- The data provided by the local transformations have to be *inserted* into the data structure or have to be *deleted* from a data structure.

- If we access details $d_{j,\boldsymbol{k},\boldsymbol{e}}$, we always consider *all* wavelet types indicated by $\boldsymbol{e} \in E^*$.

- The algorithm is independent of the spatial dimension d and the number of conservation laws m. Therefore the data structures have to be designed *flexible* with respect to this parameters.

¿From these algorithmic requirements we deduce the design criteria for an appropriate data structure for our sparse data distribution. This data structure should provide

- *fast random data access*, e.g. check whether an element already exists,

- *fast insert and delete* of elements, i.e., copying and sorting of elements within the data structure should be avoided,

- *fast dynamic memory allocation and extension*, since the memory requirement is not known before but can only be approximately predicted and

- *support group information*, i.e., connections of data corresponding to a common level, should be maintained.

Obviously, we are facing two main problems arising in the implementation. These are (i) *dynamic memory operations* and (ii) *fast data access* with respect to inserting, deleting and finding an element:

Dynamic memory operations. Due to refinement and coarsening operations in the algorithm, memory space for new elements or freeing unused elements are frequently performed and therefore should be very fast. Since the memory operations provided by the programming language are universally, they can be made much more efficient if we take into account that our data are of the same type. In particular, this can be realized by allocating a sufficiently large memory block and manage the memory requirements by the algorithm with a specific data structure. Since the overall memory demand can only be estimated, the data structure should provide dynamic extension of the memory.

Fast data access. For instance, lists might be considered as a container for fast inserting and deleting elements. However, finding an element requires the search in the whole list from its beginning until the position of the element is found. On the other hand a fast way of finding an element by means of an index is an array. But inserting and deleting elements in an array requires copy operations. However, if one does not need to maintain any ordering in the data, then other data structures, e.g. hash-maps, might be a good alternative.

Therefore the fundamental data structures we use are hash-tables. For the management of the local index sets we introduce an index class for multi-indices which naturally arises in the context of structured grids. This class is inherited from a special vector class efficiently supporting short vectors with only a few components.

3 Template Classes

In this section one will find specifications of all classes from `igpm_t_lib`. Each class description has the following structure:

- Introduction
- Example
- Data Representation
- Class Description
  - Template Arguments, Public Types and Constants
  - Constructors and Destructor
  - Data Access
  - Supporting Functions
  - Operators, Input and Output
  - Iterators and Iterator Class Description
  - `#DEFINE`'s and Errors
- Comments

The topics *Template Arguments* and *Public Types and Constants* are discussed together because each template *type* is memorized by the class with a `typedef` and each template *integer* is saved in an `enum`. So the descriptions of the template arguments and the types are the same. For a reason to memorize these arguments in a class refer to Section 5.

If in *Constructors and Destructor* nothing is said about a *Destructor*, simply none exists.

Sometimes we refer to an *index type*. By this we mean an *integer value* but not necessarily an `unsigned int` or `signed int`. It could even be a larger or shorter integer type, with respect to the used bits, or even a more complex data type which behaves like an integer.

3.1 Vector: `tvector_n`

The class `tvector_n` is designed to store a fixed amount of elements of the same type. The most common vector operations like addition, scalar multiplication, and others are provided.
Unlike other vector classes known to the author, `tvector_n` possesses a special unroll mechanism based on recursive template instantiation. This allows fast access to the data stored in `tvector_n` especially if the number of elements is small. For details including a comparison with expression template based classes like e.g. the *Blitz library* see Section 4 and 5.
The class is declared in `igpm_tvector_n.h` and the associated test program is `igpm_tvector_n.test.cpp`.

3.1.1 Example

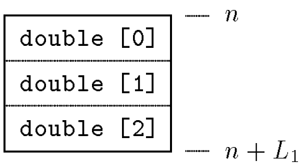
```
[01]      #include "igpm_tvector_n.h"
[02]      // ...
[03]      typedef tvector_n<double,3> euler;
[04]      euler q1,q2,q3;
[05]      // ...
[06]      q1 = q2 + 2*q3;
[07]      // ...
[08]      for (euler::iterator it=q1.begin();it!=q1.end();++it)
[09]         q1[it] = 1;    // equivalent: q1[*it] = 1;
```

- `[04]` declares three vectors $q1, q2, q3$ without initialization.
- `[06]` stores the sum of $q2$ and twice $q3$ into $q1$.
- `[08],[09]` traverses q1 and stores 1 in every coefficient of the vector.

3.1.2 Data Representation

Each vector of type `tvector_n` holds n coefficients of one given element type. In the example above n equals 3 and the used type is the floating point type `double`. So the size in bytes of a single vector is n times the size of the used element type. There is no room needed for internal memory or size management because the vector cannot grow or shrink and its length is handled by the class declaration itself. This means in particular, that different lengths and/or different types instantiate different classes. A representation of the memory used by one of the vectors of the example looks as follows:

A tvector_n<double,3>:



n memory address

$L_1 = 3*$sizeof(double)

3.1.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <typename DBL, int DIM>
[02]      class tvector_n
[03]      // ...
[04]      typedef DBL dbl;
[05]      enum { dim = DIM };
```

- DBL and dbl are the types of the vector elements.

- DIM and dim are the dimension of the vector. dim acts like a constant integer class member.

Constructors and Destructor:

```
[01]      tvector_n();
[02]      tvector_n(const tvector_n& v);
[03]      tvector_n(const dbl p[dim]);
```

- [01] is the standard constructor, it is empty.

- [02] is the copy constructor.

- [03] is implemented to initialize a vector from a given field of at least dim entries.

Data Access:

```
[01]      const dbl& operator[](unsigned int i) const;
[02]      dbl& operator[](unsigned int i);
[03]
[04]      dbl& operator[](const iterator& it);
[05]      const dbl& operator[](const iterator& it) const;
```

- `[01]`,`[02]` realize read and write access by means of an integer index.
- `[04]`,`[05]` realize read and write access by means of multi-index index.

Supporting Functions:

```
[01]      unsigned int length();
[02]      unsigned int size();
[03]      unsigned int max_size();
[04]      unsigned int memoryuse();
[05]
[06]      dbl Norm2() const;
[07]      dbl Norm2sqr() const;
[08]      dbl euclidean_norm() const;
[09]      dbl Norminf() const;
[10]
[11]      void DoNorm2();
[12]      tvector_n DoNorm (const tvector_n& v);
[13]
[14]      static const char* version();
[15]      static const char* date();
```

- `[01]`-`[03]` return the length of the vector, i.e.,, `dim`.
- `[04]` returns the used bytes by the vector, i.e.,, length times size of the used element type.
- `[06]`,`[08]` calculate the euclidean norm $\|v\|_2$ for a vector v.
- `[07]` calculates the square of the euclidean norm $\|v\|_2^2$ for a vector v.
- `[09]` calculates the infinity norm $\|v\|_\infty$ for a vector v.
- `[11]` normalizes a vector v with its 2-norm, i.e.,, $v := 1/\|v\|_2 * v$.

- [12] calculates the normalized vector with respect to its 2-norm without changing the original vector.
- [14] returns current class version.
- [15] returns last modification date.

Operators, Input and Output:

```
[01]      bool operator==(const dbl& d) const;
[02]      bool operator==(const tvector_n& v) const;
[03]      bool operator!=(const dbl& d) const;
[04]      bool operator!=(const tvector_n& v) const;
[05]
[06]      tvector_n& operator=(const dbl& d);
[07]      tvector_n& operator=(const tvector_n& v);
[08]
[09]      tvector_n& operator+=(const dbl& d);
[10]      tvector_n& operator-=(const dbl& d);
[11]      tvector_n& operator*=(const dbl& d);
[12]      tvector_n& operator/=(const dbl& d);
[13]      tvector_n& operator+=(const tvector_n& v);
[14]      tvector_n& operator-=(const tvector_n& v);
[15]
[16]      friend tvector_n operator+(const tvector_n& v1,
                                     const tvector_n& v2);
[17]      friend tvector_n operator+(const tvector_n& v1,
                                     const dbl& d);
[18]      friend tvector_n operator+(const dbl& d,
                                     const tvector_n& v1);
[19]      friend tvector_n operator-(const tvector_n& v1,
                                     const tvector_n& v2);
[20]      friend tvector_n operator-(const tvector_n& v1,
                                     const dbl& d);
[21]      friend tvector_n operator-(const dbl& d,
                                     const tvector_n& v1);
[22]      friend tvector_n operator-(const tvector_n& v1);
[23]      friend dbl operator*(const tvector_n& v1,
                               const tvector_n& v2);
[24]      friend tvector_n operator*(const dbl& d,
                                     const tvector_n& v1);
[25]      friend tvector_n operator*(const tvector_n& v1,
                                     const dbl& d);
```

```
[26]      friend tvector_n operator/(const tvector_n& v1,
                                     const dbl& d);
[27]      friend tvector_n operator/(const dbl& d,
                                     const tvector_n& v1);
[28]
[29]      friend std::istream& operator>>(std::istream& is,
                                          tvector_n& v);
[30]      friend std::ostream& operator<<(std::ostream& os,
                                          const tvector_n& v);
```

- [01]-[04] check vectors for equality or inequality with other vectors respectively compare all coefficients of a vector with one value.
  This operations make only sense for elements where equality is defined, for instance all kinds of integers. They are senseless for floating point types because in this case you have to check equality by distances or norms.

- [06],[07] copy another vector or set all coefficients to a given value.

- [09]-[14] are the defined unary operations.

- [16]-[27] are the defined binary operations; in particular [23] returns Norm2sqr().

- [29],[30] are the usual input and output operators. There are no other delimiters between the coefficients of a vector than a space, so one is able to read a sequence of values as a vector.

Iterators and Iterator Class Description:

```
[01]      const iterator begin() const;
[02]      const iterator end() const;
```

- [01] returns an iterator pointing to the first element of a vector.

- [02] returns an iterator pointing behind the last element of a vector.

```
[01]      class iterator
[02]
[03]      iterator();
[04]      iterator(const unsigned int n);
[05]
```

```
[06]      bool operator==(const iterator& it) const;
[07]      bool operator!=(const iterator& it) const;
[08]      const iterator& operator++();
[09]
[10]      const unsigned int operator*() const
```

- [03] is the default constructor, the iterator points to the first element.
- [04] is a constructor for an iterator pointing to the n'th element.
- [06],[07] check iterators for equality respectively inequality.
- [08] is the prefix increment operator so this is a forward only iterator.
- [10] returns an internal index counter, so that using an iterator as an index is possible, cf. the Example. 3.1.1.

3.1.4 #DEFINE's and Errors

#DEFINE's:

- IGPM_TVECTOR_N_DEBUG (off). If set to *on*, op[](index) performs an index check.
- IGPM_TVECTOR_N_TEMPLATEUNROLL (on) unrolls all loops per template. If set to *off*, one has to write additional code for all operators.

Errors:

- If IGPM_TVECTOR_N_DEBUG is defined, a wrong index in operator[] gives:
  "ERROR (tvector_n): wrong index" and exit(-1).

3.1.5 Comments

- To avoid confusion there are no further constructors defined. One could for instance think of

  ```
  [01]      tvector_n(const dbl& d) { (*this)[0]=d; }
  ```

  Such constructors are often used in variable length vector classes to define a start length or to initialize all of the coefficients of a vector with a value. Because of the fact that the default constructor is empty, a construct like

```
[01]      tvector_n<type,len> v; v=0;
```

is equivalent.

- The function `Norminf()` internally uses the `fabs()` function to determine whether an element is maximal or not. If you like to use other types for the vector class than `double` or `float` *and* the function `Norminf()`, you have to modify the vector class and add a template `fabs()` or `abs()` function.[1]

- There is an additional internal vector class `tvector_d` without the common operators and without template unrolling. It can be used for larger fixed size vectors instead of a simple C array.

[1] There will be a helping class called `tnum` in one of the next versions.

3.2 Stack: tstack_o

The class tstack_o is designed to work simultaneously like a stack and like a vector with respect to access, i.e., it is possible to push and pop elements to and from the end and also to access elements by means of an index. Hence it behaves like an increasing and decreasing vector.
The class is declared in igpm_tstack_o.h and the associated test program is igpm_tstack_o.test.cpp.

3.2.1 Example

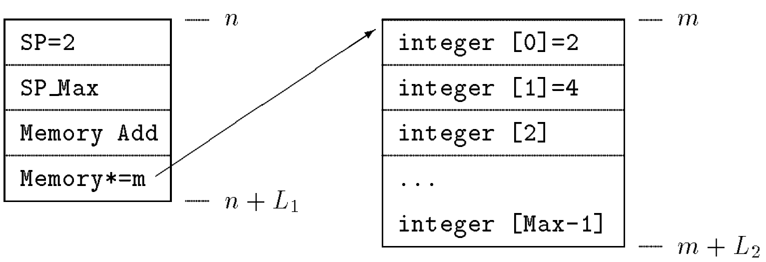
```
[01]      #include "igpm_tstack_o.h"
[02]      // ...
[03]      tstack_o<integer> s;
[04]      // ...
[05]      s.push_back(2);   // (s[0]==2)
[06]      s.push_back(4);   // (s[1]==4)
[07]      // ...
[08]      s.pop_back(n);    // (n==4)
[09]      s.pop_back(n);    // (n==2)
[10]      // ...
[11]      s.clear();
```

- [03] declares one stack variable, without initialization.
- [05],[06] push two values on top of the stack.
- [08],[09] pop two values from top of the stack.
- [11] clears the stack, i.e., the stack is empty.

3.2.2 Data Representation

Each stack class is instantiated with a given element type. A stack variable has to reserve space for a number of elements which will be pushed on top of the stack. This block of memory can be enlarged so there is no a priori size limit. All in all a stack variable has three internal counters and one pointer to the needed memory. In memory the variable s from the example above looks as follows:

A `tstack_o<integer>`:



n, m memory addresses

$L_1 = 3*$`sizeof(integer)+sizeof(element*)`

$L_2 =$`SP_Max`$*$`sizeof(element)`

3.2.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <typename ELEM>
[02]      class tstack_o
[03]      // ...
[04]      typedef ELEM element_type;
```

- `ELEM` and `element_type` are the type of the stack elements.

Constructors and Destructor:

```
[01]      tstack_o(size_type nStartSz = default);
[02]      ~tstack_o();
```

- `[01]` is the standard constructor. Note that, because of the default size the stack is never in an undefined state.

- `[02]` is the standard destructor. It frees the used memory blocks.

Data Access:

```
[01]      element_type& operator[](unsigned int i);
[02]      const element_type& operator[](unsigned int i) const;
[03]
[04]      void push_back(const element_type& e);
[05]      void pop_back(element_type& e);
```

- [01],[02] realize read and write access by means of an integer index.
- [04] pushes an element on top of the stack. After this operation the element is also accessible with `s[s.length()-1]`, if `s` denotes the stack variable.
- [05] pops an element from top of the stack.

Supporting Functions:

```
[01]      unsigned int length() const;
[02]      unsigned int size() const;
[03]      unsigned int max_size() const;
[04]      unsigned int memoryuse() const;
[05]
[06]      void clear();
[07]      void reserve(size_type nNew);
[08]
[09]      static const char* version();
[10]      static const char* date();
```

- [01],[02] return the actual extent of the stack, that is the stack pointer.
- [03] returns the maximal size of the stack. If the stack grows further more memory will be allocated.
- [04] returns the used bytes by the stack, i.e.,, length times size of the used element type.
- [06] sets the stack pointer to 0, i.e., empties the stack.
- [07] makes sure that there is room for at least `nNew` elements on the stack without forcing an internal re-growth. For instance if one plans to push 3 elements into the stack and calls `s.reserve(s.length()+3)` before, one can be sure that there is space for 3 elements and there will be no error generated between the push operations. If there is no free memory left one will get the error in `reserve`.
- [09] returns current class version.
- [10] returns last modification date.

Operators, Input and Output:

Iterators and Iterator Class Description:

- none;

3.2.4 #DEFINE's and Errors

#DEFINE's:

- `IGPM_TSTACK_O_DEBUG` (off). If set to *on*, `op[](index)` performs an index check.

Errors:

- If `IGPM_TSTACK_O_DEBUG` is defined, a wrong index in `operator[]` gives:
  "`ERROR (tstack_o): wrong index`" and `exit(-1)`.

- No available memory in `tstack_o(...)` or `push_back(elem)` gives:
  "`ERROR (tstack_o): out of memory`" and `exit(-1)`.

3.2.5 Comments

- Be aware of the following fact: while the push operation is a copy of the element by means of the *copy operator* of the elements class, the copy operation that will be performed on an element in the stack during the enlargement process of the stack is a *binary copy*. The enlargement takes place if the stack grows beyond `max_size` and is implemented by a call to `realloc`.

- Because the stack is not designed for vector operations there are no operators defined. Up to now there was no need to write or read a stack in or from any stream so this operations are still undefined. The same is true for iterators.

3.3 Multi-d: `tmultiindex`

The class `tmultiindex` is designed in order to store a d dimensional multi-index. It is inherited from `tvector_n` and specialized for $d = 1, 2, 3$. In contrast to `tvector_n` there are constructors with exactly d arguments and there is an additional operator which maps a multi-index to an unsigend int, which is important for using this class as a key in `thashmap`.
The class is declared in `igpm_tmulti.h` and the associated test program is `igpm_tmulti.test.cpp`.

3.3.1 Example

```
[01]      #include "igpm_tmulti.h"
[02]      // 2 dimensional implementation
[03]      typedef tmultiindex<2,unsigned int>  multiindex;
[04]      multiindex      mi1(1,2), mi2(2,3), mi3(mi1+mi2);
```

- `[04]` declares three multiindex variables. Note that such a variable is still a vector, so vector addition in the initialization of `mi3` is valid.

3.3.2 Data Representation

Because `tmultiindex` *is* a `tvector_n` the data representation is the same, cf. 3.1.2.

3.3.3 Class Description

Template Arguments, Public Types and Constants:

```
template <int DIM, typename INDEX=unsigned int>
struct tmultiindex { };  // empty!

// specialization of tmultiindex for d=1,2,3
template <typename INDEX>
struct tmultiindex<d,INDEX>
   : public tvector_n<INDEX,d>
```

- `DIM` is the dimension of the vector.
- `INDEX` is a index type.
- For public types and constants see `tvector_n`.

Constructors and Destructor:

```
[01]      tmultiindex();
[02a]     tmultiindex(INDEX n0);                        // d=1
[02b]     tmultiindex(INDEX n0, INDEX n1);              // d=2
[02c]     tmultiindex(INDEX n0, INDEX n1, INDEX n2); // d=3
[03]      tmultiindex(const tvector_n& v);
```

- `[01]` is the standard constructor, it is empty.
- `[02a]-[02c]` are the constructors corresponding to the dimension d with d arguments.
- `[03]` initializes a multi-index from `tvector_n`, so all operations resulting in a `tvector_n` are available for `tmultiindex`.

Data Access:

Supporting Functions:

```
[01]      static const char* version();
[02]      static const char* date();
```

- `[01]` returns current class version.
- `[02]` returns last modification date.
- For further functions see `tvector_n`.

Operators, Input and Output:

```
[01]      tmultiindex& operator=(INDEX n);
[02]      operator unsigned int() const;
```

- `[01]` sets all coefficient to `n`.
- `[02]` converts the `tmultiindex` to an `unsigned int`.
- For further operators see `tvector_n`.

Iterators and Iterator Class Description:

- See `tvector_n`.

3.3.4 `#DEFINE`'s and Errors

`#DEFINE`'s:

- `IGPM_TMULTIINDEX_DEBUG` (off). If set to *on*, `operator unsigned int` performs an index check.

Errors:

- If `IGPM_TMULTIINDEX_DEBUG` is defined, a wrong index in `operator unsigned int` gives:
  "`ERROR (tmultiindex): argument greater than maximum`".

3.3.5 Comments

- If you use more complex types for `INDEX` than an `unsigned int` or other generic data types it makes sense to change concerned parameters to `const INDEX&`.

3.4 Multi-d: `tlevelmultiindex`

The class `tlevelmultiindex` is a compound of a level type and a multi-index. It is to store a level and a multi-index simultaneously in what will be called *levelmultiindex*. Like the `tmultiindex` class there is a mapping to an `unsigned int` and a so called *link* function, so this type can be used as a key in `thashmap_linked`.
The class is declared in `igpm_tmulti.h` and the associated test program is `igpm_tmulti.test.cpp`.

3.4.1 Example

```
[01]      #include "igpm_tmulti.h"
[02]      // 2 dimensional implementation
[03]      typedef tmultiindex<2,unsigned int>  multiindex;
[04]      typedef tlevelmultiindex<multiindex> levelmultiindex;
[05]      levelmultiindex lmi(1,multiindex(2,3));
```

- `[05]` declares a `levelmultiindex` variable and it is initialized with a level and a multi-index.

3.4.2 Data Representation

The variable `lmi` from the example looks as follows:
A `tlevelmultiindex<multiindex>`:

`uns.int/level=1`	— n
`uns.int [0]=2`	
`uns.int [1]=3`	— $n+3*$`sizeof(uns.int)`

n memory address

3.4.3 Class Description

Template Arguments, Public Types and Constants:

```
template <typename MULTIINDEX,
          typename INDEX=unsigned int>
struct tlevelmultiindex

typedef INDEX      size_type;
typedef MULTIINDEX multiindex;
```

- `INDEX` and `size_type` are the index types.
- `MULTIINDEX` and `multiindex` are the multi-index types.

Constructors and Destructor:

```
[01]      tlevelmultiindex() { }
[02]      tlevelmultiindex(size_type l, const multiindex& mi);
```

- `[01]` is the standard constructor, it is empty.
- `[02]` initializes a levelmultiindex with a level and a multi-index.

Data Access:

```
[01]      size_type   level;
[02]      multiindex  index;
```

- `level` is the public level variable.
- `index` is the public multi-index variable.

Supporting Functions:

```
[01]      size_type link() const;
[02]
[03]      static const char* version();
[04]      static const char* date();
```

- `[01]` returns `level` and is used for `thashmap_linked`.
- `[03]` returns current class version.
- `[04]` returns last modification date.

Operators, Input and Output:

```
[01]      operator unsigned int() const;
[02]
[03]      friend ostream& operator<<(ostream& os,
                          const tlevelmultiindex& l);
[04]      friend istream& operator>>(istream& is,
                          tlevelmultiindex& l);
```

- `[01]` converts a variable of type `tlevelmultiindex` into an `unsigned int` and this function is also used for `thashmap`.
- `[03]`,`[04]` are the usual input and output operators. There are no other delimiters between the level and the coefficient of the multi-index vector than a space, so one is able to read a sequence of elements as a level and a multi-index.

3.4.4 Comments

- For a compressed version of this class see 3.6, class `tpackedltmi`.

3.5 Multi-d: `tmultirange`

A multi-range is a d dimensional interval of length r. It can be used for instance to represent a difference stamp in a numerical scheme. The class `tmultirange` provides a possibility to get all necessary multi-indices relative to a given starting point. Therefore the class only needs memory for this starting point, the calculation of the indices is determined by the class itself. The advantage in using this multi-range class is the independence of the given dimension d. If you code it without multi-range the *number* of loops depends on d, e.g.

```
if (1==d)          // or #if ...
   for (i=0; i<r; ++i) {
     cout <<"index :" << i << endl;
      // ...
   }
else if (2==d)     // or #if ...
   for (i=0; i<r; ++i)
      for (j=0; i<r; ++i) {
         cout <<"index :" << i << "," << j << endl;
         // ...
      }
// else ...
```

If one uses multi-range iterators instead to traverses all indices there is only one loop and the calculation of the next index is hidden in the iterators `operator++`.
There are cases where it is necessary to store data to every multi-index in this multi-range, i.e., there is the starting point and a field of r^d data elements behind every possible index, cf. the figure in 3.5.2 and Section 4. The multi-range class and its iterator class provide easy access to such an array.
The class is declared in `igpm_tmulti.h` and the associated test program is `igpm_tmulti.test.cpp`.

3.5.1 Example

```
#include "igpm_tmulti.h"
// 2 dimensional implementation
typedef tmultiindex<2,unsigned int>    multiindex;
typedef tmultirange<3,multiindex>      multirange;

typedef tvector_n<unsigned int,2>      data;
```

```
[07]      typedef tmultirange<3,multiindex,data> multirangedata;
[08]
[09]      multiindex     mi(1,2);
[10]      multirange     mr(mi);
[11]      multirangedata mrd(mi);
[12]      // ...
[13]      for (multirange::iterator it=mr.begin();
                                    it!=mr.end(); ++it)
[14]         cout << (*it) << "," << it.loopindex()
                           << "," << it.linear() << endl;
[15]      // ...
[16]      for (multirangedata::iterator it=mrd.begin();
                                        it!=mrd.end(); ++it)
[17]         cout << (*it) << "," << mrd[it]==mrd[it.linear()]
                  << endl;
```

- [09] declares a 2 dimensional multi-index initialized with $(1, 2)$.

- [10],[11] declare a multirange and a multirangedata, which are initialized with the given multi-index.
  The difference between a multirange without or with data is only the second argument in the template instantiation. For the internal coding refer to 5.

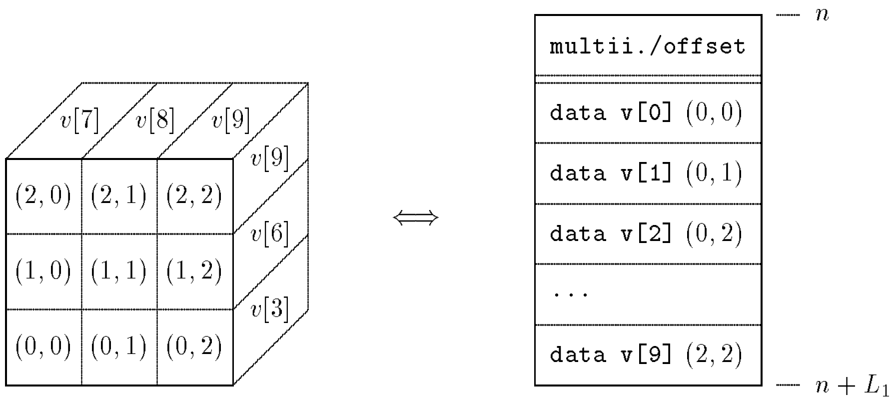
- [13],[14] and [16],[17] are loops for the types `multirange` and `multirangedata` to traverse all multi-indices. The output from [14], the first loop, is:

  ```
  [01]      2 1, 0 0, 0
  [02]      2 2, 0 1, 1
  [03]      2 3, 0 2, 2
  [04]      3 1, 1 0, 3
  [05]-[08] ...
  [09]      4 3, 2 2, 8
  ```

  The first multi-index is the starting multi-index plus the second multi-index, the `loopindex()`. The last row is just a counter and can be used to access the data in the multirangedata directly, i.e., in the second loop in [17] the boolean expression is always true.

3.5.2 Data Representation

The memory representation of the variable `mr` from the example above is like a multi-index, cf. 3.1.2 and 3.3.2.
The data from the variable `mrd` looks like the right cube in following figure. The numbers on top are the relative multi-indices. The left box is the representation of the complete multirange with data.
A `tmultirange<3,multiindex,data>`:



n memory address

L_1 =`sizeof(multiindex)`$+r^d*$`sizeof(data)`

d =dim `multiindex`, r =dim `multirange`

3.5.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <unsigned int DIM, typename MULTIINDEX,
                   typename ARRAYTYPE=_tmultirange_null,
                   unsigned int LEN=
                     _tmultirange<DIM,MULTIINDEX::dim>::nPot>
[02]      class tmultirange
[03]
[04]      enum { range = DIM };
[05]      enum { array_size = LEN };
[06]
[07]      typedef MULTIINDEX multiindex;
[08]      typedef ARRAYTYPE  array_type;
```

- `DIM` and `range` are the dimension r of the multirange.
- `MULTIINDEX` and `multiindex` are the multi-index types.

- ARRAYTYPE and array_type are the data types. If one does not use any, it is empty per default, or to be precise, it is the dummy type _tmultirange_null.
- LEN and array_size give the number of elements to store in a vector of data type elements, normally it is r^d and is calculated directly. One can change this, but one has to make sure then that all accesses by means of an iterator or index are in a valid range.

Constructors and Destructor:

```
[01]      tmultirange();
[02]      tmultirange(const multiindex& mi);
```

- [01] is the standard constructor, it is empty.
- [02] initializes the multi-range with the given multi-index.

Data Access:

```
[01]      const array_type& operator[](unsigned int n) const;
[02]      array_type& operator[](unsigned int n);
[03]
[04]      const array_type& operator[](const iterator& it) const;
[05]      array_type& operator[](const iterator& it);
[06]
[07]      multiindex& start():
[08]      const multiindex& start() const;
```

- [01],[02] realize read and write access to the data in a multirange with data by means of an integer index.
- [04],[05] realize read and write access to the data in a multirange with data by means of an iterator index.
- [07],[08] returns the starting point for read and write access.

Supporting Functions:

```
[01]      static const char* version();
[02]      static const char* date();
```

- [01] returns current class version.
- [02] returns last modification date.

Operators, Input and Output:

```
[01]      tmultirange& operator=(const multiindex& mi);
```

- `[01]` sets the starting point of a multi-range to a given multi-index. `start()` does the same.

Iterators and Iterator Class Description:

```
[01]      const iterator begin() const;
[02]      const iterator end() const;
```

- `[01]` returns an iterator pointing to the first multi-index.
- `[02]` returns an iterator pointing behind the last multi-index.

```
[01]      class iterator
[02]
[03]      iterator();
[04]
[05]      bool operator==(const iterator& it) const;
[06]      bool operator!=(const iterator& it) const;
[07]      const iterator& operator++()               // prefix
[08]
[09]      const multiindex& operator*() const;
[10]      const multiindex& loopindex() const;
[11]      const size_type linear() const;
```

- `[03]` is the default constructor. The iterator points to the first multi-index.
- `[05]`,`[06]` check iterators for equality respectively inequality.
- `[07]` is the prefix increment operator so this is a forward only iterator.
- `[09]` returns the current multi-index, this is the starting point plus the `loopindex()`, cf. the example.
- `[10]` returns the current multi-index starting from 0, cf. the example.
- `[11]` returns a counter from 0 to `array_size`-1 corresponding to the current multi-index. One can use it to access the data in a multirange directly by means of `operator[]`, cf. the example.

3.5.4 Comments

- At present the class `tmultirange` is restricted to intervals of the same length r.

3.6 Multi-d: `tpackedltmi`

Using the classes `tmultiindex` or `tlevelmultiindex` is fast, but inside a hash-map or another container variables of this kind waste a lot of memory. This is because from a realistic point of view a maximal level L rarely exceeds 15 and therefore the single indices are in $[0, \ldots, 2^{L+s} - 1]$, where s is an additional offset. Hence it is possible to encode all used information on level and indices in a smaller data type. This is realized by `tpackedltmi`.
In this class an additional information is provided, the so called *typ*. This is an integer in the range $[0, \ldots, 2^d - 1]$, when d denotes the used dimension.
The current implementation stores the information in one integer. This shall be sufficient for $d = 1, 2$. For larger dimensions however one might be forced to use more than one integer to store the required information.
The classes are declared in `igpm_tmulti.h` and the associated test program is `igpm_tmulti.test.cpp`.

3.6.1 Example

```
[01]      #include "igpm_tmulti.h"
[02]      // ...
[03]      tpackedltmi<2> p(1,0,2,3);
[04]      // ...
[05]      nLevel = p.level();  // read level
[06]      p.level()=4;         // set level to 4
[07]      // ...
[08]      nIndex = p.index(1); // read index 1
[09]      p.index(1)=5;        // set index 1 to 5
[10]      p[1]=5;              // does the same
```

- `[03]` declares a variable initialized with: level$= 2$, typ$= 0$ and multi-index $= (2, 3)$.
- `[05],[06],[08][10]` read and write the level and one index. Read access is also possible with `p[n]`.

3.6.2 Data Representation

For the case $d = 2$ the representation in memory for the example above looks as follows:
A `tpackedlmi<2>`:

	index[0]	index[1]	typ	level
Bits	$0 \ldots 12$	$13 \ldots 25$	$26, 27$	$28 \ldots 31$

3.6.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <int DIM>
[02]      class tpackedltmi
[03]
[04]      enum { dim = DIM };
```

- `DIM` and `dim` are the used dimension. At present the class is designed for $d = 1, 2, 3$.

Constructors and Destructor:

```
[01]      tpackedltmi();
[02]      tpackedltmi(unsigned int nLevel, unsigned int nTyp,
                      unsigned int nI0);
[03]      tpackedltmi(unsigned int nLevel, unsigned int nTyp,
                      unsigned int nI0, unsigned int nI1);
[04]      tpackedltmi(unsigned int nLevel, unsigned int nTyp,
                      unsigned int nI0, unsigned int nI1,
                      unsigned int nI2);
```

- `[01]` is the standard constructor, it is empty.
- `[02]`-`[04]` are the constructors for $d = 1, 2, 3$, they are contained in one class for simplicity.

Data Access:

```
[01]      unsigned int level() const;
[02]      _level_typ level();
[03]
[04]      unsigned int typ() const;
[05]      void _typ_typ typ();
[06]
[07]      unsigned int index(unsigned int n) const;
[08]      _index_typ index(unsigned int n);
[09]
[10]      unsigned int operator[](unsigned int n) const;
[11]      _index_typ operator[](unsigned int n);
```

- the internal types _level_typ, _typ_typ and _index_typ only trigger the right operator=, so you can write p.level()=12 with an integer on the right side.
- [01],[02] provide read and write access for the level.
- [04],[05] provide read and write access for the typ.
- [07],[08] provide read and write access for the indices. The parameter n is the index which has to be contained in $[0, \ldots, d-1]$.
- [10],[11] read and write index n and just call index(n).

Supporting Functions:

```
[01]      unsigned int link() const;
[02]
[03]      static const char* version();
[04]      static const char* date();
```

- [01] returns level and is used for thashmap_linked.
- [03] returns current class version.
- [04] returns last modification date.

Operators, Input and Output:

```
[01]      bool operator==(const tpackedltmi& i);
[02]      operator unsigned int() const;
```

- [01] compares two packed variables. Because they have the form of an integers this operation is an ordinary integer comparison.
- [02] returns the packed variable as an integer.

Iterators and Iterator Class Description:

- None.

3.6.4 Comments

- Currently for every d the last four bits are reserved for the level. That implies that the level cannot exceed 15. If you have more levels you have to change the internal distribution of bits or take more than one integer for the packed data type.

- It is not possible to declare an `unsigned int&` as return type to write level, typ or indices because this variables are bit fields. Such a reference is actually a pointer to an integer location and the information of being bit fields is lost. This is the reason for the mentioned help classes.

3.7 Memory Management: `tmemheap`

The class `tmemheap` offers a fast way to provide memory space for a variable of a given type. It is useful to allocating and freeing a lot of variables of the same type. This works by first allocating a chunk of memory, called *heap*, sufficient for a given number of variables of the given type at once. The heap is able to increase so it is not required to know the exact number of variables a priori. If one then needs space for a concrete variable, `tmemheap` provides an appropriate slice of the heap. The first is done by calling a function similar to the standard C++ `new`.
If the type, the `tmemheap` is instantiated with, needs a constructor, one has to call it explicitly by means of *placement new*. This is because the request for such a new variable returns only a *pointer* to unused memory in the heap. The same holds for a destructor, i.e., if one has a class which needs a destructor one has to make sure that an analogous function is called before freeing the variable in the memory heap.
The class is declared in `igpm_tmemheap.h` and the associated test program is `igpm_tmemheap.test.cpp`.

3.7.1 Example

```
[01]      #include "igpm_tmemheap.h"
[02]
[03]      tmemheap<unsigned int> h(10,0.5);
[04]      unsigned int           *p1,*p2;
[05]      // ...
[06]      p1 = h.newElem(); *p1 = 1;
[07]      p2 = h.newElem(); *p2 = 2;
[08]      // ...
[09]      h.delElem(p2);
```

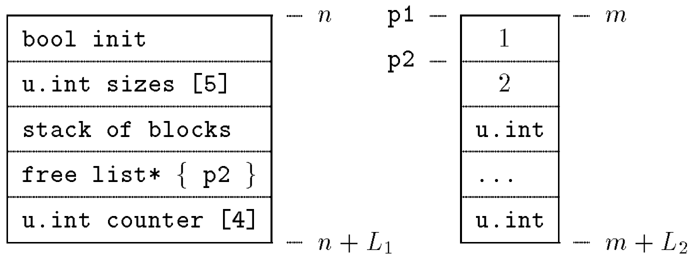
- `[03]` declares a memory heap of 10 `unsigned int` objects. Every time the heap is completely filled it allocates space for $10 * 0.5 = 5$ more elements.

- `[06]`,`[07]` fill the pointers `p1`,`p2` with the allocated positions and store a value there.

- `[09]` frees the space from pointer `p2`.

3.7.2 Data Representation

The representation of the memory heap `h` from the above example looks as follows:
A `tmemheap<unsigned int>`:



n, m memory addresses

$L_1 =$`sizeof(bool)`$+(5+4)*$`sizeof(unsigned int)`

$+$`sizeof(tstack_o)`$+$`sizeof(unsigned int*)`

$L_2 = 10*$`sizeof(unsigned int)`

3.7.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <typename E>
[02]      class tmemheap
[03]
[04]      typedef E             element_type;
[05]      typedef unsigned int  size_type;
```

- `E` and `element_type` are the data types.
- `size_type` is `unsigned int`.

Constructors and Destructor:

```
[01]      tmemheap(size_type nMinBlockSz = default,
                   double dAddBlockSz = default,
                   size_type nMaxBytes = default );
[02]      ~tmemheap();
```

- `[01]` is the standard constructor, where `nMinBlockSz` is the starting size for the heap, given in element units. `dAddBlockSz` times `nMinBlockSz` determines the number of elements for a new memory

block if the heap is completely filled. The idea is to guess the maximal size of the heap a priorly. If that initial size is to small, the size of the newly allocated memory block is a certain percentage of the initial size. `nMaxBytes` is in experimental phase.
Every parameter has an internal default value, so the heap is never in an undefined state. If you cannot guess the parameters at construction time use `init()` at a later time.

- `[02]` is the destructor, it frees all allocated memory. After calling it every pointer stemming from a tmemheap variable is invalid.

Data Access:

```
[01]      element_type* newElem();
[02]      void delElem(element_type* pE);
```

- `[01]` returns a pointer to a new element.
- `[02]` frees the element `pE`. One has to make sure that `pE` is a pointer stemming from `newElem`.

Supporting Functions:

```
[01]      size_type size() const;
[02]      size_type size_local() const;
[03]      size_type max_size() const;
[04]      size_type max_size_local() const;
[05]      size_type capacity() const;
[06]
[07]      void init(size_type nMinBlockSz = default,
                    double dAddBlockSz = default,
                    size_type nMaxBytes = default );
[08]      void clear();
[09]      void reset();
[10]
[11]      static const char* version();
[12]      static const char* date();
```

- `[01]` returns the number of stored elements.
- `[02]` returns the number of stored elements in the last allocated memory block.

- [03],[05] returns the maximal number of elements which can be stored without allocating a new memory block.
- [04]] returns the maximal number of elements in the last allocated block.
- [07] initializes the heap; for parameter description cf. the constructor.
- [08] frees all allocated memory blocks except the first.
  Calling `clear()` is equivalent to initialize the heap with the same parameters as before.
- [09] is equivalent to construct or initialize the tmemheap variable without parameters.
- [11] returns current class version.
- [12] returns last modification date.

Operators, Input and Output:

```
[01]      friend ostream& operator<<(ostream& os,
                                     const tmemheap& h);
```

- [01] creates an output with information about the tmemheap variable like the number of allocated elements, free elements and other.

Iterators and Iterator Class Description:

- None.

3.7.4 #DEFINE's and Errors

`#DEFINE`'s:

- None.

Errors:

- A size parameter equals 0 gives:
  "`ERROR (tmemheap): size of new memory is 0`".
- No more free memory gives:
  "`ERROR (tmemheap): out of memory`".

3.7.5 Comments

- The copy constructor and the assignment operator are private, i.e., it is not allowed to copy a variable of type tmemheap.

3.8 Hashing: `thashtab`

A hash-map is a data structure to store elements in an unordered way providing fast access. One can insert and delete elements or ask for existence of a specific element. The elements of a hash-map can be complex data structures. The hash-map uses a specified part of this data structure for storage and access management, called *key*. The part of its data structure not being *key* is called *value*. However it is possible that the entire data structure is key, i.e. there is *no* value. In this case the hash-map works like a mathematical set. Otherwise it works like a mapping between key and value. Such a hash-map is called *an associative container*.

As basis for fast access a hash-map has a very fast function, the *hash* function, which maps the range of the keys into an integer range between 0 and some integer, say $n-1$. This integer in principle encodes the position of the key. Usually this operation is not injective, e.g. a modulo function, i.e., there are keys which are mapped to the same integer value. This is called a *collision*. For choosing a good integer n refer to the Comments 3.8.5.
One design property of a hash-table is the collision strategy. There are various strategies like choosing a position near the original one, take a second hash function or link all keys with the same hash value in a list. For our purpose the latter is sufficient, easy to implement and works very well.

While using a hash-map the fill rate is an important variable. The probability of collisions depends strongly on n. The larger n, the smaller is the chance of one value for two keys. To decide whether a key exists already in a hash-map it has to store the key itself. This can be done, e.g. in a vector with n elements. If you take into account that a large n makes sense to minimize collisions, storing information in a vector of length n wastes most of the space. So the solution of this problem is to store only pointers in a vector to different positions in another container. That implies, together with the collision strategy, that every position in this vector is the starting point of a connected list. This vector is called the *hash-table*. The container to store the keys and values is for our case the structure **`tmemheap`**.

To work efficient with this class it is important to have an initial guess of the amount of data one wants to store. In this case controlling the fill rate is easy and the memory usage is more or less optimal. It is often possible to guess the number of new elements or deleted elements, so this is not difficult to achieve. Even with a bad guess or no idea of a realistic number the hash-map works, but the access and all other operations like insert and delete are noticeable slower.
The class is declared in `igpm_thashmap.h` and the associated test program is `igpm_thashmap.test.cpp`.

3.8.1 Example

```
[01]      #include "igpm_thashmap.h"
[02]
[03]      typedef thashmap<unsigned int>              set;
[04]      typedef thashmap<unsigned int,unsigned int> map;
[05]      // ...
[06]      set s(7,0.8);
[07]      map m(7,0.8);
[08]      // ...
[09]      s.on(1); s.on(3); s.on(7); s.on(10);
[10]      s.off(0); s.off(3); s.off(7);
[11]      // ...
[12]      m.insert(1,11); m.insert(3,33); m.insert(7,77);
          m.insert(10,101);
[13]      m.erase(0); m.erase(3); m.erase(7);
[14]      // ...
[15]      for (set::iterator it=s.begin(); it!=s.end(); ++it)
[16]         cout << (*it).key << endl;
```

- `[06]` declares a hash-map with only keys, a set. There are 7 free positions in the hash-table and room for 5 elements in the internal memory heap.
- `[07]` declares a hash-map with keys and values. There are also 7 free positions in the hash-table and room for 5 elements in the internal heap.
- `[09]` put the numbers $1, 3, 7$ and 10 into the set `s`.
- `[10]` delete the numbers $0, 3$ and 7; for the case 0 nothing happens.
- `[12]` put the listed pairs into the map `m`.
- `[13]` deletes as in `[10]`.
- `[16]` traverses the hash set. There are only the elements 1 and 10 left.

3.8.2 Data Representation

The hash-table and the heap of the hash-map `m` from the example above looks as follows:

A `thashmap<unsigned int,unsigned int>`:

[0]
[1] -> 1
[2]
[3] -> 10
[4]
[5]
[6]

Hast table, 7 pointers

1	3	7	10	
11	33	77	101	

MemHeap, 5 pairs<key,value>

3.8.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <typename KEY, typename VALUE=_thashmap_null>
[02]      class thashmap
[03]
[04]      typedef typename pair_type::key_type   key_type;
[05]      typedef typename pair_type::value_type value_type;
[06]      typedef unsigned int                   size_type;
```

- Because the hash-table needs a data structure for both, key and value, these types are based on pair type. For a key only hash-table, a set, the value is empty. As in C++ a class is never empty the implementation of the type pair makes sure that there is only a key and not even one byte for a value.
- `KEY` and `key_type` are the key types.
- `VALUE` and `value_type`, if given, are the value types.
- `[06]` defines the size type for all indices. It is always `unsigned int` but can be changed to another integer typ.

Constructors and Destructor:

```
[01]      thashmap(size_type nLen = default,
                   double dFill = default);
[02]      ~thashmap();
```

- [01] is the default constructor. The parameter `nLen` gives the length of the internal hash-table, i.e., the length of the pointer vector and the number for the modulo function. `dFill` is a factor in $[0, \ldots, 1.0]$ to determine the amount of required memory for the keys and values. The number of free elements in the internal memory heap structure is `nLen` times `dFill`.
  To store for instance 10 elements in a hash-table with a fill rate of 0.5, you have to call `thashmap(10*(1/0.5),0.5)`.
  Because of the default arguments a hash-table is never in an undefined state.

- [02] is the destructor, it frees all used memory.

Data Access:

```
[01]      bool find (const key_type& key) const;
[02]      bool find (const key_type& key,
                     value_type& value) const;
[03]      bool find (const key_type& key,
                     value_type*& pvalue) const;
[04]      bool erase (const key_type& key);
[05]      void insert(const key_type& key);
[06]      void insert(const key_type& key,
                      const value_type& value);
[07]      void insert(const key_type& key,
                      value_type*& pvalue);
[08]
[09]      bool exist (const key_type& key) const;
[10]      void off(const key_type& key);
[11]      void on(const key_type& key);
```

- [01] checks if an element with key `key` exists.

- [02] checks if an element with key `key` exists and copies the value, if the hash-table is not a set, into `value`.

- [03] checks if an element with key `key` exists and copies a pointer onto the value into `pvalue`. This is useful if one only needs parts of the stored value and wants to avoid the copy process.

- [04] deletes an element with `key` from the hash-map. If there is not such an element nothing happens.

- [05] inserts a new key `key` into the hash-map. If there already exists an entry with this key nothing happens.
- [06] inserts a new key `key` into the hash-map and copies the value from `value`. If there already exists an entry with this key the new value given in `value` is copied and the value before is lost.
- [06] inserts a new key `key` into the hash-map and copies the pointer to the value into `pvalue`.If there already exists an entry with this key nothing else happens. This is useful if for instance only parts of your value are calculated and you want to store them directly into the right position. Hence a copy process is avoided.
- [09]-[11] are for convenience only. They call `find`, `off` and `insert` for a set.

Supporting Functions:

```
[01]      size_type size() const;
[02]      size_type max_size() const;
[03]
[04]      void init(size_type nLen = default, dFill = default);
[05]      void clear();
[06]      void reset();
[07]
[08]      static const char* version();
[09]      static const char* date();
```

- [01] returns the number of stored elements.
- [02] returns the length of the hash-table, i.e., the maximal number of elements without having a collision.
- [04] initialized a hash-table with the new parameters, cf. the description of the constructor.
- [05] clears the hash-table and the internal memory heap but keeps all given parameters.
- [06] initialized a hash-map with its default values.
- [08] returns current class version.
- [09] returns last modification date.

Operators, Input and Output:

```
[01]      friend ostream& operator<<(ostream& os, const self& h);
[02]      friend ostream& operator<<(ostream& os,
                                  const self::_info_hashstats& ih);
```

- `[01]` creates an output with information about the hash-map like the number of stored elements and much more.
- `[02]` creates an output with internal information about the number of linked collision lists. `3->2 (6)` means: there are two lists of length three, hence six elements.
  If `h` denotes a hash-map, call `cout << h.info_hashelem();` to get this information.

Iterators and Iterator Class Description:

The following functions and the class `iterator` itself are defined only if `IGPM_THASHMAP_ENABLE_ITERATOR` is set.

```
[01]      const iterator begin() const;
[02]      const iterator end() const;
```

- `[01]` returns an iterator pointing to the first element in a hash-map.
- `[02]` returns an iterator pointing behind the last element in a hash-map.

```
[01]      class iterator
[02]
[03]      iterator();
[04]
[05]      bool operator==(const iterator& it) const;
[06]      bool operator!=(const iterator& it) const;
[07]      const iterator& operator++();     //  prefix
[08]
[09]      const element_type& operator*() const;
```

- `[03]` is the default constructor, the iterator is undefined.
- `[05]`,`[06]` check iterators for equality respectively inequality.
- `[07]` is the prefix increment operator so this is a forward only iterator.
- `[09]` returns a reference to the internal pair type. One can access the key and value with `(*it).key` and `(*it).value`.

3.8.4 #DEFINE's and Errors

#DEFINE's:

- IGPM_THASHMAP_ENABLE_ITERATOR (off). If set to *on*, the iterator class is defined which is usually not necessary. Refer to 3.8.5.

Errors:

- A wrong initial size gives:
  "ERROR (thashmap): size of new memory is 0"
- No more memory for the heap or for the hash-table gives:
  "ERROR (thashmap): out of memory"

3.8.5 Comments

- An iterator which traverses a hash-map has no list to walk along. It has to search for the next element in the hash-table, walk the potential collision list and again search the next element in the hash-table. Because a hash-table should not be filled up to 100%, the iterator touches a lot of empty positions to find the next pointer. To traverse a hash-map often and fast, use a thashmap_linked or thashmap_linked_one, as described next.

- Choosing the right initial size, i.e., the modulo parameter n for the hash function, is a little bit tricky.
  On one hand a prime number is optimal with respect to the modulo function but on the other hand calculating an appropriate prime number for an unknown number of elements is expensive. One good strategy might be an a priori calculation of a prime table and choosing the right slot at run time.
  Surprisingly, even if the prime number seems to be crucial, is has not to be. For our project we took the very simple but fast choice

  ```
  [01]      nPrime = nAmount*nFactor;   // "Prime"
  [02]      dFill = 1/nFactor;
  ```

  which is still sufficient. Here nPrime and dFill denote the parameters for the init function respectively the constructor and nAmount the number of elements we wanted to store in the hash-map. nFactor usually equals 3 which implies that the fill rate is about 30%.
  One can use this strategy if the time consumption caused by additionally collisions due to a bad "prime" number is less than that caused by spotting the right prime number.

3.9 Hashing: thashtab_linked

The class `thashtab_linked` is an expansion of `thashtab`. The main improvement is to classify keys into groups, for instance all variables of type `tlevelmultiindex` with one level. The new iterator class `iterator` is able to traverse all elements with this specific property.
To know which element of the key is the important one to link them together, the key type needs a function called `link()`. For the classes `tpackedltmi` and `tlevelmultiindex` this function is implemented and returns the level.
The class is declared in `igpm_thashmap.h` and the associated test program is `igpm_thashmap.test.cpp`.

3.9.1 Example

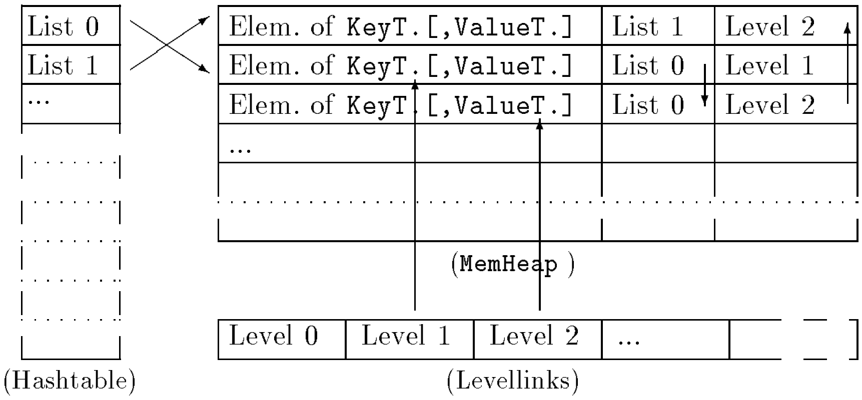
```
[01]      #include "igpm_thashmap.h"
[02]
[03]      typedef tmultiindex<1,unsigned int>      mi;
[04]      typedef tlevelmultiindex<unsigned int,mi> lmi;
[05]      typedef thashmap_linked<lmi,unsigned int> map;
[06]      // ...
[07]      map m(5,7,1.0);
[08]      lmi lm_1_1(1,1),lm_1_3(1,3),lm_2_7(2,7);
[09]
[10]      m.insert(lm_1_1,11); m.insert(lm_1_3,33);
          m.insert(lm_2_7,77);
[11]
[12]      for (map::iterator it=m.begin(1); it!=m.end(1); ++it)
[13]         n += (*it).value;
```

- `[07]` declares a hash-map with 5 link positions, 7 spaces in the hash-table and room for 7 elements in the heap.
- `[08]` declares `tlevelmultiindex` variables and initializes it with the level and the 1 dimensional index.
- `[10]` inserts the keys and values into the map.
- `[12]`,`[13]` traverses all pairs where the level of the key is 1, i.e., the values 11 and 33.

3.9.2 Data Representation

A linked hash-map with some elements may look as follows. There are at least two elements in the collision list starting at position 0 in the hash-table, indicated with `List 0`. On the other hand there are the link lists, indicated with `Level` n.

A `thashmap_linked<keyT,valueT>`:



3.9.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <typename KEY, typename VALUE=_thashmap_null>
[02]      class thashmap_linked
             : public thashmap< KEY,_thashmap_link<VALUE> >
[03]
[04]      typedef typename basic::size_type  size_type;
[05]      typedef typename basic::key_type   key_type;
[06]      typedef typename basic::value_type value_type;
```

- The template parameters `KEY` and `VALUE` have the same meaning as in the class `thashmap`, the internal wrapping in `_thashmap_link` only triggers a special key value pair.
- The same is true for the types.

Constructors and Destructor:

```
[01]      thashmap_linked(size_type nMaxLinks = default,
                          size_type nLen = default,
                          double dFill = default);
[02]      ~thashmap_linked();
```

- [01] is the standard constructor. The first parameter `nMaxLinks` gives the maximal number of links, for a description of `nLen` and `dFill` cf. the constructor of `thashmap`.

- [02] is the destructor. It frees memory from the link management and calls the base class destructor.

Data Access:

```
[01]      bool erase (const key_type& key);
[02]      bool erase (iterator& it);
[03]      void erase (const iterator& beg,
                      const iterator& end);
[04]      void insert(const key_type& key,
                      const value_type& value);
[05]      void insert(const key_type& key,
                      value_type*& pvalue);
[06]      void insert(const key_type& key);
[07]
[08]      void on(const key_type& key);
[09]      void off(const key_type& key);
```

- The meanings of `[01]`,`[04]`-`[09]` is the same as in `thashmap`. There is nothing new for the user of the hash-map, only internally the class has to follow all operations to update the links.

- `[02]` erases the element the iterator refers to. Because the iterator is invalid afterwards it refers to the element behind the deleted one. One has to make sure that in a loop the iterator is not incremented twice, i.e., typical loops looks like

```
[01]      for(iterator it=m.begin(n); it!=m.end(n); erase(it))
[02]         ;
[03]
[04]      for(iterator it=m.begin(n); it!=m.end(n); )
[05]         if (condition) erase(it);
             else           ++it;
```

- `[03]` erases all elements from `beg` (inclusive) to `end` (exclusive), cf. the first example above.

Supporting Functions:

```
[01]      size_type size() const;
[02]      size_type size(size_type n) const;
[03]      unsigned int minLevel() const;
[04]      unsigned int maxLevel() const;
[05]
[06]      void init(size_type nMaxLinks = default,
                    size_type nLen = default,
                    double dFill = default);
[07]      void clear();
[08]      void reset();
[09]
[10]      static const char* version();
[11]      static const char* date();
```

- `[01]` returns the number of stored elements.
- `[02]` returns the number of stored elements of the link `n`.
- `[03]` returns the smallest link number, it is always 0.
- `[04]` returns the maximal number of links, this is the number from the initialization.
- The meanings of `[06]-[08]` are the same as in `thashmap`, except the first parameter `nMaxLinks` gives the maximal number of links.
- `[10]` returns current class version.
- `[11]` returns last modification date.

Operators, Input and Output:

- Cf. `thashmap`.

Iterators and Iterator Class Description:

```
[01]      const iterator begin() const;
[02]      const iterator end() const;
[03]
[04]      const iterator begin(size_type n) const;
[05]      const iterator end(size_type n) const;
```

- [01] returns an iterator pointing to the first element in a hash-map.
- [02] returns an iterator pointing behind the last element in a hash-map.
- [03] returns an iterator pointing to the first element with link n in a hash-map.
- [04] returns an iterator pointing behind the last element with link n in a hash-map.

The old type iterator_link is replaced by this iterator class and the functions begin_link and end_link with [04],[05].

```
[01]      class iterator
[02]
[03]      iterator();
[04]
[05]      bool operator==(const iterator& it) const;
[06]      bool operator!=(const iterator& it) const;
[07]      const iterator& operator++();    //  prefix
[08]
[09]      inline const element_type& operator*() const;
```

- [03] is the default constructor, the iterator is undefined.
- [05],[06] check iterators for equality respectively inequality.
- [07] is the prefix increment operator so this is a forward only iterator.
- [09] returns a reference to the internal pair type. One can access the key and value with (*it).key and (*it).value.

3.9.4 Comments

- It is not allowed to use thashmap as a base class for function calls and work with thashmap_linked or thashmap_linked_one instead. This is because no functions are defined virtual.
- In this version it is not allowed to use begin(link) and end(link) for different link values, i.e. to use code like

```
[01]      for (iterator it=h.begin(1); it!=h.end(2); ++it)
[02]         ; // do something
```

Please write it this way

```
for (unsigned int l=1; l<=2; ++l)
   for (iterator it=h.begin(l); it!=h.end(l); ++it)
      ; // do something
```

Even if it is possible (and easy) to implement such a behavior, such an iterator class needs more internal information and therefore it is a little bit slower. For the frequently case of equal link values this is not necessary.

3.10 Hashing: thashtab_linked_one

The class thashtab_linked_one is a thashtab_linked with only one group of elements linked together. This is advantageous if one wants to traverse to the whole hash-map fast, cf. 3.8.5.
The class is declared in igpm_thashmap.h and the associated test program is igpm_thashmap.test.cpp.

3.10.1 Example

```
[01]      #include "igpm_thashmap.h"
[02]
[03]      typedef tmultiindex<1,unsigned int>          mi;
[04]      typedef thashmap_linked_one<mi,unsigned int> map;
[05]      // ...
[06]      mi  mi_1(1), mi_3(3);
[07]      map m(7,1.0);
[08]
[09]      m.insert(mi_1,11); m.insert(mi_3,33);
[10]      // ...
[11]      for (map::iterator it=m.begin(); it!=m.end(); ++it)
[12]         cout << (*it).value << endl;
```

- [07] declares a hash-map with 7 spaces in the hash-table and room for 7 elements in the heap.
- [06] declares tmultiindex variables and initializes it with the 1 dimensional index.
- [09] inserts the keys and values into the map.
- [11],[12] traverses all elements. Note, the argument of begin_link and end_link, the link number, is missing.
- In this example the key type has no *link* function unlike the key type in the example from thashmap_linked. This is not necessary because there is only one link and the linkage is handled internally.

3.10.2 Data Representation

For a data representation see thashtab_linked.

3.10.3 Class Description

Template Arguments, Public Types and Constants:

```
[01]      template <typename KEY, typename VALUE=_thashmap_null>
[02]      class thashmap_linked_one
             : public thashmap_linked<_thashmap_link_one<KEY>,
                                      VALUE>
```

- The template parameters `KEY` and `VALUE` have the same meaning as in the class `thashmap_linked`.
- For types cf. `thashmap` and `thashmap_linked`.

Constructors and Destructor:

```
[01]      thashmap_linked_one(size_type nLen = default,
                              double dFill = default);
```

- `[01]` is the standard constructor, cf. the constructors for `thashmap` and `thashmap_linked`. Please have in mind the maximal link number is 1.

Data Access:

- cf. `thashmap` and `thashmap_linked`;

Supporting Functions:

```
[01]      void init(size_type nLen = default,
                    double dFill = default);
[02]
[03]      static const char* version();
[04]      static const char* date();
```

- `[01]` initializes the hash-map, cf. `thashmap` and `thashmap_linked` and have in mind the maximal link number is 1.
- `[03]` returns current class version.
- `[04]` returns last modification date.

Operators, Input and Output:

- Cf. `thashmap`.

Iterators and Iterator Class Description:

```
[01]      const iterator begin() const;
[02]      const iterator end() const;
```

- `[01]` returns an iterator pointing to the first element in a hash-map.
- `[02]` returns an iterator pointing behind the last element in a hash-map.

3.10.4 Comments

- See the comments 3.9.4 in `thashmap_linked` concerning inheritance.

4 Implementation of Local Multiscale Transformation

In the following we describe the data types that are needed for the implementation of the Algorithms 2.1 and 2.2 of the local transformations. By means of these types we declare the data structures and their initialization. Finally we present some characteristic examples how to work with these data structures in the context of the local transformations.

4.1 Data Types

Since the multi-scale setting can be applied component-wise to a vector of functions, i.e., $u \in L^1(\Omega, \mathbb{R}^m)$, we consider in the following vectors of averages $\hat{\boldsymbol{u}}_{j,\boldsymbol{k}} \in \mathbb{R}^m$ and details $\boldsymbol{d}_{j,\boldsymbol{k},\boldsymbol{e}} \in \mathbb{R}^m$. First of all we derive the array types

```
[01]      typedef tvector_n<double,  d>        gvector
[02]      typedef tvector_n<double,  m>        uvector
[03]      typedef tvector_n<double,  2^d-1>    evector
[04]      typedef tvector_n<uvector, 2^d-1>    dvector
```

from the template class `tvector_n` for the storing of vectors $\boldsymbol{x} \in \mathbb{R}^d$, averages, details and vectors of details corresponding to all wavelet types $(\boldsymbol{d}_{j,\boldsymbol{k},\boldsymbol{e}})_{\boldsymbol{e}\in E^*}$.

Since the averages and the details are enumerated by multi-indices $\boldsymbol{k} = (k_1, \ldots, k_d) \in \mathbb{N}_0^d$ and level–multi-indices $(j, \boldsymbol{k})$, we derive the integer arrays

```
[01]      typedef tmultiindex<d>          mi
[02]      typedef tlevelmultiindex<mi>   lmi
```

from the template classes `tmultiindex` and `tlevelmultiindex`, respectively.

For the local transformations we always access simultaneously all non–vanishing elements of a column of the mask matrices. Therefore it is convenient to agglomerate the indices of the corresponding supports in one data structure. However, we have not to store index by index, since in the curvilinear case the supports can be represented as a multidimensional integer interval $\boldsymbol{k} + [0, i]^d \subset \mathbb{N}_0^d$ characterized by the multi-index $\boldsymbol{k}$ and the interval length i. By this knowledge we designed the special template class `tmultirange`. According to the supports of the box function and the wavelets we need four different types of multi-ranges

```
typedef tmultirange<1,     mi>   mr1
typedef tmultirange<2,     mi>   mr2
typedef tmultirange<4*s+2, mi>   mr4s
typedef tmultirange<2*s+1, mi>   mr2s
```

for the supports $\mathcal{M}^{*,0}_{j,\boldsymbol{k}}(\mathcal{G}^1_{j,\boldsymbol{k}})$, $\mathcal{M}^0_{j,\boldsymbol{k}}(\mathcal{G}^{*,1}_{j,\boldsymbol{k}})$, $\mathcal{M}^1_{j,\boldsymbol{k}}$ and $\mathcal{G}^0_{j,\boldsymbol{k}}$, respectively. Here the first argument specifies the length i and the second argument the initial index $\boldsymbol{k}$.

Analogously, we can agglomerate the non–vanishing matrix elements corresponding to the columns of the mask matrices by multi-ranges where in addition to the blocked multi-indices the corresponding matrix elements are stored whose type is specified by the third argument. Here we distinguish between the types

```
typedef tmultirange<1, mi, double>  mrA_G0
typedef tmultirange<2, mi, double>  mrA_M0
```

for the matrix columns of $\boldsymbol{G}_{j,0}$ and $\boldsymbol{M}_{j,0}$, respectively. In case of the matrices $\boldsymbol{M}_{j,\boldsymbol{e}}$, $\boldsymbol{G}_{j,\boldsymbol{e}}$, $\boldsymbol{e} \in E^*$, we proceed differently. By construction the supports $\mathcal{M}^{\boldsymbol{e}}_{j,\boldsymbol{k}}$ and $\mathcal{G}^{\boldsymbol{e}}_{j,\boldsymbol{k}}$, respectively, are the same for all wavelet types $\boldsymbol{e} \in E^*$, only the matrix elements $m^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$ and $g^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$ depend on the wavelet type. Moreover, in the local transformations we always have to access all types corresponding to an index pair $(j,\boldsymbol{k})$. Therefore it is more convenient to agglomerate all matrix elements $m^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$, $\boldsymbol{e} \in E^*$ and $g^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$, $\boldsymbol{e} \in E^*$, respectively, in one multirange array

```
typedef tmultirange<2*s+1, mi, evector>     mrA_M1
typedef tmultirange<4*s+2, mi, evector>     mrA_G1
```

By this we avoid the multiple storing of the index $\boldsymbol{k}$ and the length i of the integer interval.

The mask matrices $\boldsymbol{M}_{j,0}$, $\boldsymbol{G}_{j,0}$ and $\boldsymbol{M}_{j,1} = (\boldsymbol{M}_{j,\boldsymbol{e}})_{\boldsymbol{e}\in E^*}$, $\boldsymbol{G}_{j,1} = (\boldsymbol{G}_{j,\boldsymbol{e}})_{\boldsymbol{e}\in E^*}$ are then stored in hash-maps where each element is a multirange representing the non–vanishing matrix elements corresponding to one matrix column

```
typedef thashmap_linked<lmi, mrA_M0>  liMap_M0
typedef thashmap_linked<lmi, mrA_G0>  liMap_G0
typedef thashmap_linked<lmi, mrA_M1>  liMap_M1
typedef thashmap_linked<lmi, mrA_G1>  liMap_G1
```

Here the first argument of the hash-map represents the type of the key, e.g. a level–multi-index, and the second argument the type of the value, e.g. a multirange.

Analogously the local averages $\boldsymbol{u}_{j,\boldsymbol{k}}$, $(j,\boldsymbol{k}) \in \mathcal{G}_{L,\boldsymbol{\varepsilon}}$, and the significant details $\boldsymbol{d}_{j,\boldsymbol{k},\boldsymbol{e}}$, $(j,\boldsymbol{k},\boldsymbol{e}) \in \mathcal{D}_{L,\boldsymbol{\varepsilon}}$, are stored in linked hash-maps

```
typedef thashmap_linked<lmi, uvector>    liMap_u
typedef thashmap_linked<lmi, dvector>    liMap_d
```

where each element is either a vector of averages or an array of vectors representing the details of all components and all wavelet types. We notice that the index sets $\mathcal{I}_{j,\boldsymbol{\varepsilon}}$ and $\mathcal{J}_{j,\boldsymbol{\varepsilon}}$ are implicitly determined by the linked lists corresponding to the different levels.

The local transformations are performed level by level. To this end, we need data structures where temporary data corresponding to one level can be stored. Again we use hash-maps. Since only *one* level is involved, we can simplify this type of hash-map. Therefore the data type

```
typedef thashmap_linked_one<mi>   iSet
```

uses the template class `thashmap_linked_one`.

Notice that the hash-map has to be linked, since we have to traverse through all elements. In particular, the data type `iSet` is a hash-map where only keys are stored but no values. This can be interpreted as a set of keys, e.g. multi-indices.

4.2 Data Structures and their Initialization

By means of the above data structures we now introduce the hash-maps

```
liMap_M0   M0
liMap_M1   M1
liMap_G0   G0
liMap_G1   G1
```

for the management of the mask matrices $\boldsymbol{M}_{j,0}$, $\boldsymbol{G}_{j,0}$ and $\boldsymbol{M}_{j,1}$, $\boldsymbol{G}_{j,1}$, $j = 0, \ldots, L-1$, as well as the local averages and the significant details

```
liMap_u   u_map
liMap_d   d_map
```

corresponding to the adaptive grid and the set of significant details.

The initialization of these hash-maps crucially influences the performance of the computation. We therefore describe the choice of `nMax` which determines the length of the hash-table and the size of the memory needed for storing `nMax` elements. First of all, we consider the mask matrices. We notice that the number of elements corresponds to the number of columns of these matrices, since each element in the hash-map represents the non–vanishing matrix coefficients corresponding to one column. For the matrices $\boldsymbol{M}_{j,0}$ and $\boldsymbol{M}_{j,1}$ the total number of columns for all levels is determined by

$$\sum_{j=0}^{L-1} N_j = N_{L-1} \frac{1-q^L}{1-q} \quad \text{with} \quad q = 2^{-d}$$

where we make use of the relation $N_j = q\, N_{j+1}$. In case of the matrix $\boldsymbol{G}_{j,0}$ we obtain the total number

$$\sum_{j=0}^{L} N_j = N_L \frac{1-q^{L+1}}{1-q}.$$

For the averages and the details the total number of elements is restricted by N_L for the full grid of the finest level and accordingly all details are significant, i.e., $N_{L-1}\,(1-q^L)/(1-q)$.

¿From the total number of the elements that have to be stored in the worst case of a uniform refinement over all levels, we determine the number `nMax = nTotal * rFill` of predicted elements where `rFill` denotes the fill rate. Notice that `rFill` differs from the fill rate `dFill` of the hash-table. In [M] the influence of `rFill` and `nFactor` for the performance of the computation id investigated.

Finally we like to remark that the hash-maps for the mask matrices as well as the adaptive grid are initialized once and then the length of the hash-table as well as the memory heap size remain unchanged throughout the computation. Therefore the choice of `rFill` is more significant for the mask matrices and the adaptive grid. In particular, the corresponding hash-maps require the bulk of memory. This is different for the hash-maps by which the local averages and the details are stored. Here the length of the hash-table has to be reinitialized in each time step and the memory eventually has to be dynamically extended. From the refinement strategy we conclude that the complexity of the sets $\mathcal{G}_{L,\boldsymbol{\varepsilon}}$ and $\mathcal{D}_{L,\boldsymbol{\varepsilon}}$ are related by $\#\,\mathcal{G}_{L,\boldsymbol{\varepsilon}} = q^{-1}\,\#\,\mathcal{D}_{L,\boldsymbol{\varepsilon}}$, since a cell is refined as long as there exists a significant details. Then we can reinitialize the hash maps `u_map` and `d_map` by the actual number of elements determined by one of the hash-maps with `nMax =` $q^{-1}\,\#\,\mathcal{D}_{L,\boldsymbol{\varepsilon}}$ for `u_map` and `nMax =` $q\#\,\mathcal{G}_{L,\boldsymbol{\varepsilon}}$ for `d_map`, respectively.

4.3 Examples

We now present some examples how to use the data types and data structures derived from the template classes in the context of the Algorithms 2.1 and 2.2. Here we will not outline the whole implementation of these algorithms but focus on some typical situations:

- storing of a support by means of multi-ranges,
- computation of a local index set by means of multi-ranges,
- computation and storing of non–vanishing matrix elements corresponding to a row of a mask matrix by means of multirange arrays ,
- storing of a mask matrix with respect to the level by means of linked hash-maps,
- performing the local two–scale relations and
- deleting of elements in a linked hash-map representing a mask matrix.

Supports. In order to locally perform the two–scale relations (2.1) and (2.2) as well as (2.3), see step 2, 4 and 5 (Alg. 2.1) and step 2 (Alg. 2.2), we have to determine the supports of indices corresponding to non–vanishing elements in a row or column, respectively, of the mask matrices. These are also needed for the computation of the index sets U_j^0, U_j^1, P_{j+1} as well as I_{j+1}^+, I_j^-, see step 1 and 3 (Alg. 2.1 and Alg. 2.2, respectively).

```
[01]      mr1   mr_1 = mi(k/2);
[02]      mr2   mr_2 = mi(2*k);
```

- `[01]` : $\mathcal{G}_{j,\boldsymbol{k}}^1$, $\mathcal{M}_{j,\boldsymbol{k}}^{*,0}$
- `[02]` : $\mathcal{M}_{j,\boldsymbol{k}}^0$, $\mathcal{G}_{j,\boldsymbol{k}}^{*,1}$

where `k` is a multi-index of type `mi`. In case of the supports $\mathcal{M}_{j,\boldsymbol{k}}^1$ and $\mathcal{G}_{j,\boldsymbol{k}}^0$ it is more convenient to realize the computation of these supports by functions.

```
[01]      void suppM1(mr_M1& mr4s, int j, const mi& l) const
[02]      {  mi  m;
[03]         for (int i=0; i<gvector::dim; ++i)
[04]            if (k[i]<s)               m[i] = 0;
[05]            else if (k[i]<N[j][i]-s)  m[i] = k[i]-s;
[06]            else                      m[i] = N[j][i]-1-2*s;
[07]         mr_4s = 2*m;
[08]      }
```

- `[03]` : `m[i] =` $l_{j,k_{i+1}}$
- `[05]` : `N[j][i] =` $N_{j,i}$
- `[07]` : $\mathcal{M}_{j,\boldsymbol{k}}^1$

```
[01]      void suppG1(mr_G1& mr2s, int j, const mi& l) const
[02]      {  mi    m;
[03]         for (int i=0; i<gvector::dim; ++i)
[04]            if (k[i]/2<s)                m[i] = 0;
[05]            else if (k[i]/2<N[j][i]-s)  m[i] = k[i]/2-s;
[06]            else                         m[i] = N[j][i]-1-2*s;
[07]         mr_2s = m;
[08]      }
```

- `[03]` : `m[i] =` $l_{j,k_{i+1}/2}$
- `[07]` : $\mathcal{G}^0_{j,\boldsymbol{k}}$

We emphasize that the supports $\mathcal{M}^{*,1}_{j,\boldsymbol{k}}$ and $\mathcal{G}^{*,0}_{j,\boldsymbol{k}}$ can not be stored in a multirange, since the interval lengths $\overline{l}^1_{j,k_i} - \overline{l}^0_{j,k_i}$ may differ for each $i = 1, \ldots, d$. **Index Sets**. For the performance of the two–scale relations (2.1), (2.2) and (2.3) we have to collect all indices for which the averages $\hat{\boldsymbol{u}}_{j,\boldsymbol{k}}$ and details $\boldsymbol{d}_{j,\boldsymbol{k},\boldsymbol{e}}$ as well as the averages $\hat{\boldsymbol{u}}_{j+1,\boldsymbol{k}}$ have to be computed, see the sets U^0_j, U^1_j, P_{j+1} and I^+_{j+1}, I^-_j in Alg. 2.1 and 2.2. These are stored in index sets of type `iSet`. Here we consider the computation of the set U^0_j as an example.

```
[01]      unsigned int nMax   = u_map.size(j+1)/(1<<gvector::dim);
[02]      unsigned int nPrime = nMax*nFactor;
[03]      double       dFill  = 1.0/nFactor;
[04]
[05]      iSet U0(nPrime,dFill);
[06]      mr1  mr_1;
[07]
[08]      for (liMap_u::iterator_link it=u_map.begin(j+1);
                                       it!=u_map.end(j+1); ++it) {
[09]         mr_1 = (*it).key.index/2;
[10]         for (mr_1::iterator it2=mr_1.begin();
                                  it2!=mr_1.end(); ++it2)
[11]            U0.on(*it2);
[12]      }
```

- `[05]` : $U^0_j = \emptyset$
- `[09]` : $\mathcal{M}^{*,0}_{j,\boldsymbol{k}}$
- `[11]` : $U^0_j = U^0_j \cup \mathcal{M}^{*,0}_{j,\boldsymbol{k}}$

Matrix columns. For the local transformations (2.1), (2.2) and (2.3) we have to provide the non–vanishing elements of the columns corresponding to the mask matrices $\boldsymbol{M}_{j,0}$, $\boldsymbol{G}_{j,0}$ and $\boldsymbol{M}_{j,1}$, $\boldsymbol{G}_{j,1}$, respectively. These are stored in multirange arrays by means of the data types `mrA_M0`, `mrA_G0` and `mrA_M1`, `mrA_G1`, respectively. In case of the matrices $\boldsymbol{M}_{j,0}$, $\boldsymbol{G}_{j,0}$ the proceeding is identical. Therefore we present only the realization for the columns of $\boldsymbol{M}_{j,0}$. Since the computation of the column entries has to be frequently performed, it is convenient to realize these by a function.

```
void calcColM0(mrA_M0& mra, const lmi& mi)
{
   mra = 2*mi.index;

   unsigned int j = mi.level;
   double       Vcoarse = volume(j,mi.index), V=0;

   mrA_M0::iterator it=mra.begin();
   ++it;
   for (; it!=mra.end(); ++it) {
      mra[it] = volume(j+1,(*it)) / Vcoarse;
      V += mra[it];
   }
   it=mra.begin();
   mra[it] = 1.0 - V;

   M0.insert(mi, mra);
}
```

Here the coefficients $m^{j,0}_{\boldsymbol{r},\boldsymbol{k}} = |V_{j+1,\boldsymbol{r}}|/|V_{j,\boldsymbol{k}}|, \boldsymbol{r} \in \mathcal{M}^0_{j,\boldsymbol{k}}$, are stored in the multirange array `mr` of type `mrA_M0` which is put into the linked hash-map `M0`. The volume of a cell is provided by the function `volume(...)` that is not specified here, since this depends on the explicit representation of the grid.

In case of the matrices $\boldsymbol{M}_{j,1}$ and $\boldsymbol{G}_{j,1}$ we proceed differently, since we store all elements corresponding to the different wavelet types $\boldsymbol{e} \in E^*$ and the same position $(\boldsymbol{r}, \boldsymbol{k})$ in one vector, i.e., $(m^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}})_{\boldsymbol{e}\in E^*}$ and $(g^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}})_{\boldsymbol{e}\in E^*}$ which is stored in a multirange array. As an example we consider the columns of the matrix $\boldsymbol{M}_{j,1}$.

```
void calcColM1(mrA_M1& mra, const lmi& mi)
{
   mr2s mr;
   suppM1(mr2s,mi.level,mi.index);

```

```
[06]         mra = mr2s.start();
[07]         for (mrA_M1::iterator it2=mra.begin();
                                   it2!=mra.end(); ++it2)
[08]            for (int l=0; l<dvector::dim; ++l)
[09]               mra[it2][l] = ... ;
[10]
[11]         M1.insert(mi, mra);
[12]      }
```

- [08] : loop on $\boldsymbol{e} \in E^*$
- [09] : matrix element $m^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$

Here we do not present the explicit computation of the matrix elements $m^{j,\boldsymbol{e}}_{\boldsymbol{r},\boldsymbol{k}}$, $\boldsymbol{e} \in E^*$, since this involves the construction of appropriate wavelets which is not subject of this paper. The multirange array `mra` of type `mrA_M1` is finally stored in the linked hash-map `M1`.

Two–scale relations. By means of the above structures the two–scale relations (2.1), (2.2) and (2.3) can be realized. Here we only present the implementation of (2.1) which can be interpreted as a coarsening of the grid, since cells are agglomerated.

```
void coarsen(const iSet &U0, int j)
{
   double  mkr;
   uvector ujk, uj1r;
   bool    bFlag;
   mrA_M0  mra;

   for (iSet::iterator_link it=U0.begin_link();
                              it!=U0.end_link(); ++it) {
      bFlag = M0.find(lmi(j,(*it).key), mra);
      if (!bFlag)
         calcColM0(mra,lmi(j,(*it).key));
      ujk = 0.0;

      for (mrA_M0::iterator it2=mra.begin();
                              it2!=mra.end(); ++it2) {
         bFlag = v.find(lmi(j+1,*it2), uj1r);
         mkr = mra[it2];
         ujk += mkr*uj1r;
      }
      u_map.insert(lmi(j,(*it).key), ujk);
```

```
        }
    }
```

Before we can perform the summation in step 2 of Algorithm 2.1, we first have to check whether all non–trivial mask coefficients $m^{j,0}_{\boldsymbol{r},\boldsymbol{k}}$ corresponding to the $\boldsymbol{k}$th column of $\boldsymbol{M}_{j,0}$ are provided by the linked hash-map `M0`. If they have been stored, then we have to get these elements stored in a multirange array from the hash-map. Otherwise, we first have to compute these information by the function `calcColM0(...)`. Then we can determine the averages $\hat{\boldsymbol{u}}_{j,\boldsymbol{k}} \in \mathbb{R}^m$ and store this vector in the linked hash-map `u_map`.

Matrix reduction. If we perform the multi-scale transformation and its inverse several times within one computation, e.g. in the context of a finite volume discretization these transformations have to be performed after each time step, the matrix columns which have not been accessed in the current application should be discarded in order to optimize the memory size. The discarding of mask matrix elements is realized by the steps 6, 7 and 4, 5 in the Algorithms 2.1 and 2.2, respectively. As an example we consider here $\boldsymbol{M}_{j,0}$.

```
    unsigned int nMax   = M0.size(j)/10;
    unsigned int nPrime = int(nMax*nFactor);
    double       dFill = 1.0/nFactor;

    iSet tmpI(nPrime,dFill);
    mi   k;

    for (liMap_M0::iterator_link it=M0.begin(j);
                                 it!=M0.end(j); ++it) {
       k = (*it).key.index;
       if ( !( U0.exist(k) || d_map.exist(lmi(j,k)) ) )
          tmpI.on(k);
    }

    for (indexSet::iterator it2=tmpI.begin();
                            it2!=tmpI.end(); ++it2)
       M0.erase(lmi(j,(*it2).key));
```

First we declare a temporary index set `tmpI`. This hash-map is initialized by `nMax` which is assumed to be 10 % of the existing columns on level j. Then we traverse through all existing columns and check whether the column index is an element of the index set `U0` or corresponds to a significant detail, i.e., $\mathcal{J}_{j,\boldsymbol{\varepsilon}}$. If this is not the case, then the index is added to the index set `tmpI`.

Finally we delete all multirange arrays in the linked hash-map `M0` associated to the indices in the index set `tmpI`.

5 Internal Structures and Remarks

In this chapter we want to discuss some difficulties that arose during the designing of appropriate data structures for the project *Adap2D*.

5.1 Why not Expression Templates?

After a motivating example we describe the idea behind Expression Templates and discuss situations where these technique is suitable. In the end we explain why we did not use such a library, e.g. the *Blitz library*, but the *non* Expression Template vector class `tvector_n` instead.

In an ordinary vector class an expression like the following

```
v1 = v2+c*v3;   // v1,v2,v3 vectors, c scalar
```

produces a lot of *temporary* variables during its evaluation. This is due to the fact that required operators like `operator+` or `operator*` which work on vectors return their results in a variable of type vector. This temporary vector is only needed as input for the next operator in the evaluation process. After delivering the value the variable is deleted.
Additionally most of the common vector operators have to traverse their arguments because vector operations are usually operations on every coefficient. Thus during the evaluation process a lot of loops are performed.

The solution to get rid of these two drawbacks is a technique called *Expression Templates*. The main idea is to build an expression tree with objects acting like an operator on scalars and to evaluate these tree in *one* loop at the end, e.g. in `operator=`. A simple analogue looks as follows

```
for (i=0; i<n; ++i)
   v1[i] = v2[i]+c*v3[i];
```

where the mathematical expression is scalar and only one loop has to be performed to calculate `v1`. In the above expression there are temporary objects as well but these are not of type *vector*. They are of type *scalar*, so there is a good chance for most machines to store them into processor registers and it is not necessary to call constructors and/or destructors. That speeds up the evaluation significantly. In principle this situation is mimicked by Expression Templates.
However, the situation is not as perfect as it sounds, because to be efficient Expression Templates depends heavily on good compiler optimization. To be precise, there are also temporary objects building the expression tree. Without going into details fewest of them are really necessary to evaluate

the expression. They exist only to generate function calls to scalar operators in the right order. It depends on the compiler how many of these temporary objects are created. Unfortunately not every compiler is capable of good optimization and for this case one will get overhead again.

If the number of *long* vector expressions is small in a program the temporary variable problem is not that relevant. But in any case working with vectors generates a lot of loops. Some compilers have an option to unroll small loops but that the compiler uses it on specific code can not be guaranteed.

The class `tvector_n` is a compromise. There are temporary objects stemming from an operator in a vector expression but there won't be even one loop. This is guaranteed by a recursive template technique which generates an unroll code sequence for sure. Moreover it works independently from the compiler optimization and one only needs a compiler capable to deal with recursive templates in general. More details are given in 5.2.

A priori it is hard to say whether it is necessary to use an Expression Template library or not. For our project *Adap2D* it was not, because only a few long vector expressions are computed. The loop factor was much more crucial so it was sufficient to use the simpler `tvector_n`. Be aware of the fact that an Expression Template library takes a lot of compile time so in any case it might be appropriate to use a fast compiling vector class during the development process. If the interfaces are fixed they can be replaced by Expression Templates later on.
In particular, for *Adap2D* we get a speedup of a factor 2–3 in time for exchanging a dynamical length vector class with a fixed length one, i.e., `tvector_n`. In contrast we get no further improvements for exchanging this vector class with the *Blitz library* but again we want to emphasize that this will be different for every project.

5.2 A Note on Template Specializations

A technique we mentioned before is *recursive template instantiation*. We used this to implement *unrolling* of our vector operations in `tvector_n`.
The basic idea is contained in the following simple example which calculates `B^E` for integer `B,E` at compile time:

```
template <unsigned int B, unsigned int E>
struct math {
   enum { pot = B*math1<B,E-1>::pot };
};

```

```
[06]      template <unsigned int B>
[07]      struct math<B,0> {
[08]         enum { pot = 1 };
[09]      };
[10]
[11]      // use:
[12]      const unsigned int n = math<5,2>::pot;  // 5^2 = 25
```

The starting point for calculating the `enum pot` is `[03]`. At this point the compiler generates the class `math<B,E-1>` and if `E` equals 0 it takes the specialization in `[08]` [2].

5.2.1 Template Specializations in `tvector_n`

In principle the specialization is analogous to the example above.

```
template <typename DBL, int nr>
struct _tvector_n_ops {
static void _set(DBL v[], const DBL& d)
   { v[nr]=d; _tvector_n_ops<DBL,nr-1>::_set(v,d); }
// ...
static void _add(DBL v[], const DBL v1[], const DBL v2[])
   { v[nr]=v1[nr]+v2[nr];
     _tvector_n_ops<DBL,nr-1>::_add(v,v1,v2); }
// ...
template <typename DBL>
struct _tvector_n_ops<DBL,0> {
static void _set(DBL v[], const DBL& d)
   { v[0]=d; }
// ...
static void _add(DBL v[], const DBL v1[], const DBL v2[])
   { v[0]=v1[0]+v2[0]; }

// use in
self& operator=(const dbl& d)
   { _tvector_n_ops<dbl,dim-1>::_set(m_d,d);
     return *this; }
// ...
friend tvector_n operator+(const tvector_n& v1,
                           const tvector_n& v2)
{ tvector_n v;
```

[2] A nice exercise is to program the calculation of the square root of an integer by means of template specialization.

```
    _tvector_n_ops<dbl,dim-1>::_add(v.m_d,v1.m_d,v2.m_d);
    return v; }
```

The helping class for `tvector_n` is `_tvector_n_ops`. [18] and [21] contain representatively the member operator `operator+` and the friend operator `operator=` which call the template helping class. `m_d` denotes the begin of the internal vector field.

5.2.2 Template Specializations in `tmultirange` and `thashmap`

Both classes `tmultirange` and `thashmap` have an optional template parameter. This parameter, an arbitrary data type, decides whether to store additional information to each element in the container or not. In the case of `tmultirange` it is an array of this data type associated to one multi-index and in the case of `thashmap` it is a value associated to a key.
The basic idea to implement this is to declare a helping template data type to store the coupled information and a specialization of this helping template data type without the additional information. The class `tmultirange` and `thashmap` contain a member of this helping data type instantiated with the used optional template parameter. To trigger the specialization of the helping data type one only has to add a default template argument. The following example illustrates this in principle.

```
struct _dummy_help { };

template <class DBL, class ADD>
struct help_class {
   DBL d;
   ADD a;
};

template <class DBL>
struct help_class<DBL,_dummy_help> {
   DBL d;
};

template <class DBL, class T=_dummy_help>
struct myclass {
   help_class<DBL,T> m_hc;
};

myclass<double>     mc1;  // only double
myclass<double,int> mc2;  // double and int
```

In [19] the default value _dummy_help is implicitly used and the specialization in [10] is instantiated. In [20] the additional information is of type `int` so the helping class is the one in [04] containing a `double` and an `int`.

The helping class in `tmultirange` is called _tmultirange_data. The slightly more complicated implementation for the class `thashmap` of the helping class _thash_pair contains a nested specialization needed for the *linked* hash-map case but the technique is still the same.

5.3 Triggering the Right Operator

Motivating problems:

- Have a look at the following example:

  ```
  [01]      struct S {
  [02]         unsigned int _x : 10;
  [03]         unsigned int& x() { return _x; }
  [04]
  [05]      S s;
  [06]      s.x() = 12;    // -> error, no address available
  ```

  One wants to store a value in a bit field, e.g. as in `tpackedltmi`, but there is no reference to a bit field and a write operator like the one in [03] does not work.

- One has a complex data structure like `tmemheap` or `thashmap` and writes an output operator `operator<<` for general information of the container. Additionally one wants to have another possibility for some internal output on a stream. A member function like

  ```
  void output(ostream& os) { ... }
  ```

  is not appropriate because one likes to write something like

  ```
  cout << "Map " << h << ", " << h.internal() << endl;
  ```

  but the standard output operator is occupied already.

The solution of both problems is very similar: One has to trigger the right operators.
Assume both `x()` and `internal()` return a variable of an internal data type instead of the desired type. Then another operators with different argument types are required. The following example deals with the second problem:

```
[01]      struct C {
[02]         int m_internal;
[03]
[04]         struct _int {
[05]           _int(const C& c) : m_c(c) { }
[06]           const C& m_c;
[07]         };
[08]
[09]         friend ostream& operator<<(ostream& os, const C& c)
[10]            { os << "general "; return os; }
[11]
[12]         _int internal() const { return _int(*this); }
[13]
[14]         friend ostream& operator<<(ostream& os, const _int& i)
[15]            { os << "data " << i.m_c.m_internal; return os; }
[16]
[17]      // use
[18]      C c;
[19]      cout << c << "," << c.internal() << endl;
```

The triggering function is `[12]`. It returns a temporary variable of the internal type and therefore `[14]` is called. Note that the internal typ `_int` has to store the data source, i.e., the variable of type `C` itself. As all references are `const` there is a good chance for a compiler to optimize the code.
Similar techniques can be found in the standard libraries `iostream` and `iomanip`.

5.4 Comments

- A name space `igpm` is planed but not realized at present.

- All library classes memorize their template type arguments or constants, if any, in `typedef`'s or `enum`'s. While using this classes as template arguments of other template classes one is able to refer to the saved information.
  For instance, `tmultiindex` is a possible template argument for the class `tmultirange` and the used vector dimension `dim` of `tmultiindex` can be addressed in `tmultirange` without explicitly given as a further template argument.

- No tests with respect to *exceptions* have been performed, i.e., per default none of the classes can be considered as *exception save*.

6 Sources and Applications

A current version and additional information are available from

`http://www.igpm.rwth-aachen.de/~voss`

References

[M] S. Müller. Adaptive Finite Volume Schemes for Conservation Laws employing Local Multiscale Transformations, RWTH Aachen, in preperation.

[BKV] A. Barinka, M. Konik, A. Voß. Software for Adaptive Methods based on Cardinal B–Splines, RWTH Aachen, in preparation.

[OW] T. Ottmann, P. Widmayer. Algorithmen und Datenstrukturen. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 1996