TLib: A Flexible C++ Tensor Framework for Numerical Tensor Calculus

CEM BASSOY, Fraunhofer Institute of Optronics, System Technologies and Image Exploitation, Germany

Numerical tensor calculus comprise basic tensor operations such as the entrywise addition and contraction of higher-order tensors. We present, TLib, flexible tensor framework with generic tensor functions and tensor classes that assists users to implement generic and flexible tensor algorithms in C++. The number of dimensions, the extents of the dimensions of the tensors and the contraction modes of the tensor operations can be runtime variable. Our framework provides tensor classes that simplify the management of multidimensional data and utilization of tensor operations using object-oriented and generic programming techniques. Additional stream classes help the user to verify and compare of numerical results with MATLAB. Tensor operations are implemented with generic tensor functions and in terms of multidimensional iterator types only, decoupling data storage representation and computation. The user can combine tensor functions with different tensor types and extend the framework without further modification of the classes or functions. We discuss the design and implementation of the framework and demonstrate its usage with examples that have been discussed in the literature.

CCS Concepts: • **Mathematics of computing** → **Mathematical software**; • **Software and its engineering** → **Object oriented frameworks**; • **Theory of computation** → *Data structures design and analysis*; *Algorithm design techniques*;

1 INTRODUCTION

In modern mathematics a higher-order tensor is defined as an element of a tensor product space [da Silva and Machado 2017; Lim 2017]. Higher-order tensors are coordinate-free in an abstract fashion without choosing a basis of the tensor product space. In the realm of numerical tensor calculus, higher-order tensors with a coordinate representation are considered [Hackbusch 2014; Lim 2017]. The bases are chosen implicitly, and the values of some measurements are then recorded in the form of a multidimensional array. We define a multidimensional array as entity that holds a set of data all of the same type whose elements are arranged in a rectangular pattern. In some cases higher-order tensors are referred to as hypermatrices, N-way arrays or N-dimensional table of values [Cichocki et al. 2009; Lathauwer et al. 2000a; Lim 2017] where N is order, i.e. the number of dimensions.

Basic tensor operations are the tensor-tensor, tensor-matrix, tensor-vector multiplication, the inner and outer product of two tensors, the Kronecker, Hadamard and Khatri-Rao product [Cichocki et al. 2009; Lim 2017]. Common methods utilizing tensor operations are e.g. the higher order decompositions or to calculate the eigenvalues or singular values of a higher-order tensor [Cui et al. 2014; Kolda and Bader 2009; Lathauwer et al. 2000a; Ng et al. 2009]. Other types of tensor decomposition are the CP- (Canonical-Decomposition/Parallel-Factor-Analysis) [Faber et al. 2003; Harshman and Lundy 1994] and Tucker-Decomposition [Kim and Choi 2007; Tucker 1966] which are mainly used within the field of psychometrics and chemometrics. Other areas of application are signal processing [FitzGerald et al. 2005; Savas and Eldén 2007], computer graphics [Suter et al. 2013; Vasilescu and Terzopoulos 2002] and data mining [Kolda and Sun 2008; Rendle et al. 2009].

Author's address: Cem Bassoy, Fraunhofer Institute of Optronics, System Technologies and Image Exploitation, Ettlingen, Germany.

Many general-purpose programming languages such as `C`, `C++` or `Fortran` support multidimensional arrays as built-in data structures with which elements are accessed in a convenient manner. Yet, built-in data types might not meet the requirements of an application. For instance, the `C++` built-in multidimensional array is not a good fit if the application requires the number of dimensions to be runtime variable. A very common approach is to provide a library that extends the general-purpose language with user-defined data types and functions. In most cases, the interface of the libraries are designed to be close to the notation that is used within a field of application [Mernik et al. 2005]. Usually, `C++` is chosen to be the host language that is extended with user-defined data types. The key feature of `C++` is that it enables the programmer to apply object-oriented and generic programming techniques with a simple, direct mapping to hardware and zero-overhead abstraction mechanisms [Gregor et al. 2006; Stroustrup 2012a]. Functions and types can be parametrized in terms of types and/or values supporting parametric polymorphism and allow software to be general, flexible and efficient [Stroustrup 2012b].

We would first like to introduce `C++` libraries that are related to our framework and depict similar high-level interfaces. POOMA, described in [Reynders III and Cummings 1998], is perhaps one the first `C++` frameworks that have been designed to support arrays with multiple dimensions including tensor operations. Multidimensional arrays are generic data types where the number of dimensions are compile-time parameters. The framework supports high-level expressions for first-level tensor operations. The library described in [Landry 2003] offers a tensor class that is designed to support classical applications found in quantum mechanics. Tensor functions for high-level tensor operations are provided. However, the framework only support tensors up to four dimensions with four elements in each dimension limiting the application of the framework to classical applications of numerical tensor calculus. In [Garcia and Lumsdaine 2005], the design of generic data types for multidimensional arrays is discussed, including the addressing elements and subdomains (views) of multidimensional arrays with first- and last-order storage formats. The order and data type are compile-time (template) parameters. They offer iterators for their multidimensional arrays and suggest to parametrize their tensor algorithms in terms of multidimensional array types that generate stride-based iterators for the current dimension of the recursion. In [Andres et al. 2010] an implementation of a multidimensional array and iterators are presented where the order and the dimension extents of tensor types are runtime parameters. The paper also discusses addressing functions, yet for the first- and last-order storage format only. Please note that [Andres et al. 2010; Garcia and Lumsdaine 2005; Reynders III and Cummings 1998] do not support higher-order tensor operations for numerical tensor calculus.

Most `C++` frameworks supporting tensor contraction provide a convenient notation that is close to Einstein's summation convention. The Cyclops-Tensor-Framework (CT) described in [Solomonik et al. 2013] offers a library primarily targeted at quantum chemistry applications [Solomonik et al. 2013]. The order and the dimensions of their tensor data structures are dynamically configurable, while the interface omits the possibility to set a data layout or a user-defined data type. The tensor contractions are performed by index foldings with matrix operations where the indices are adjusted with respect to the tensor operation. The interface for specifying the contraction is similar to the one provided by the MiaArray library (LibMia) discussed in [Harrison and Joseph 2016] using either strings (CT) or objects (LibMia, Blitz). For instance, after having instantiated tensor objects, the 2-mode multiplication of a three dimensional with a matrix is given by `C["ijk"]=A["ilj"]*B["kl"]` in case of the CT framework, `C(i,j,k)=A(i,l,k)*B(k,l)` in case of the LibMia framework, or `C(i,j,k)=sum(A(i,l,j)*B(k,l),l)` in case of the Blitz framework [Veldhuizen 1998] where the type of the variables `i,j,k,l` of the example code snippets are user-defined. The notation necessitates the order of the arrays to be known before compile-time and does not allow the contraction mode to be runtime-variable. The implementation of tensor algorithms such as the higher-order singular value decomposition for instance, performs a sequence

of k-mode tensor-times-vector multiplications where the mode k depends on an induction variable [Bader and Kolda 2006; Lathauwer et al. 2000a].

Our framework fills this need with flexible and generic tensor classes and functions where the order, extents of the dimensions and the contraction mode(s) can be runtime variable. The interfaces of the generic tensor functions are similar to the ones provided by the `Matlab` library that is presented in [Bader and Kolda 2006]. The toolbox provides tensor classes and tensor operations for prototyping tensor algorithms in `Matlab`. The execution of some higher-order tensor operations such as tensor-tensor-multiplication requires a tensor to be converted or unfolded into matrix. The unfolding is performed with respect to the mode of tensor operation and requires additional memory space for the unfolded tensor. However, once the unfolding is accomplished, fast matrix multiplications can be used to perform the tensor contraction. This approach is also applied in frameworks such as in [Napoli et al. 2014]. The generic tensor functions of our framework execute tensor operations in-place for tensor types including domains of tensors without the process of unfolding. They also support a set of storage formats including the first- and last-order storage formats. Please note that we do not intend to replace any of the previously mentioned works, but to provide a flexible and extensible library that allows to easily validate numerical results with the Matlab toolbox provided by [Bader and Kolda 2006].

We would like to limit our discussion to the software design of a framework for basic tensor operations. The design and implementation of high-performance algorithms exploiting data locality and the parallel processing capabilities of multi-core processors lies beyond the scope of this paper. We would like to refer to [Springer and Bientinesi 2016] in which a method for fast tensor contractions with arbitrary order and dimensions is described. The optimization techniques are similar to the ones applied for matrix-matrix multiplication. The paper presents a TCCG, a C++ code generator that generates optimized C++ code for the tensor multiplication. The code generation requires an input file where the contracting dimensions and mode of the multiplication, order, dimensions of the multidimensional arrays are specified. The work presented in [Li et al. 2015] follows a similar approach and provides code generator that is called InTensLi.

Our framework consists of a software stack with two main components. The upper part of the software stack contains flexible tensor classes with runtime variable number of dimensions and dimension extents. The classes and their member functions simplify the management of multidimensional data and the selection of single elements or multidimensional domains using generic and object-oriented programming techniques. The data layout of the multidimensional data can be adjusted at runtime as well, supporting a class of storage layouts include the first- and last-order storage formats. Member functions of the tensor classes encapsulate generic tensor functions of the lower software stack components and help the programmer to write tensor algorithms independent of the storage layout of the tensors. We have used ad-hoc polymorphism, i.e. operator overloading, for lower-level (entrywise) tensor operations in order to enable the user to write tensor algorithms close to the mathematical notation. Member functions encapsulating higher-level tensor operations have an interface that is similar to the one provided by the toolbox discussed in [Bader and Kolda 2006]. The upper component also provides stream classes with which numerical results can be validated in `Matlab`. The lower part of the software stack includes generic tensor function that are defined in terms of multidimensional iterator types only. The decoupling of data storage representation and computation with multidimensional iterators allows different tensor types to be handled with same tensor function. Our framework provides its own multidimensional iterator that is one possible interface between the tensor classes and generic tensor functions. The user can extend the framework with tensor functions without modifying the tensor template classes and provide his own tensor and iterator types. In summary, the main contributions of our work are as follows:

for_each	transform	copy
fill	generate	count
min_element	max_element	find
equal	mismatch	mismatch
all_of	none_of	any_of
iota	accumulate	inner_product
tensor_times_vector	tensor_times_matrix	tensor_times_tensor
tensor_times_vectors	tensor_times_matrices	transpose
outer_product	inner_product	

Table 1. Summary of function templates that implement tensor operations for dense tensors and tensor references.

- Our implementation of higher-level tensor operations allows the mode of the tensor multiplication and the order, dimensions, storage format, index offsets of the tensor to be runtime parameters. The user can implement tensor algorithms with arguments that depend on runtime-variable parameters.
- Our implementations of tensor operations are parametrized in terms of multidimensional iterator types and do not rely on specific storage representation of data. The same tensor functions can be utilized for different tensor types including tensor references. Users can extend our framework and provide their own tensor or iterator types with a variety data layouts.
- We provide tensor template classes with member functions that encapsulate tensor template functions and enable the user to program tensor algorithms in an object-oriented fashion. Numerical results are conveniently verifiable with the `Matlab` toolbox described in [Bader and Kolda 2006] using overloaded stream operators. Output files can be directly used in the `Matlab` environment without further modification.

The remainder of the paper is organized as follows. The following Section 2 provides an overview of our tensor framework and discusses some general design decisions. Section 3 introduces the `C++` implementation of our tensor data structures with the focus on the layout and access of elements. We discuss the generality and limitations of our data structure with respect to the storage format. Multidimensional iterators for data structures supporting non-hierarchical data layouts are the topic of Section 4. Section 5 discusses the design and implementation of tensor operations using multidimensional iterators as template parameters. We complete the section by exemplifying the usage of higher-order tensor operations with tensor objects. The last Section 6 provides a conclusion of this work.

2   OVERVIEW OF THE TENSOR FRAMEWORK

The primary scope of our framework is given by the following Table 1 that lists basic tensor template functions of the low-level interface of our framework. The first part of the table are first-level tensor operations and correspond semantically to the function templates provided in the algorithm and numeric package of the `C++` standard library. Typically, first-level tensor operations process data elements of multidimensional arrays with the same index tuple. The second part are referred to as higher-level tensor operations that are additionally required to implemented numerical multilinear algebra algorithms. Higher-level tensor operations typically manipulate data elements with different index tuples. All functions listed in Table 1 are able to combine multidimensional arrays and views and to process both equally efficiently with same time complexity.

2.1 Software Nomenclature

In the following we use the nomenclature developed in [Stroustrup 2013]. An object is defined in the standard as a region of storage that has a storage duration and a type. The term object type refers to to the type with which an object is created. The C++ language offers fundamental types that are built-in. There are five standard signed and unsigned integer types, a boolean type and three floating point types. We exclude the void type for our discussion. A class is a user-defined type that contains a set of objects of other types and functions that manipulate these objects. A template is a class or a function that we parametrize with a set of types or values. It defines a family of classes or functions or an alias for a family of types. We do not distinguish between the term template class and class template as well as template function and function template. We call the generation of a class or function from a template, template instantiation where the generated template instance is called specialization. Containers are referred to as data structures that manage memory. Algorithms denote template functions that process and manipulate container data with iterators.

2.2 Software Design

The software design of our framework is greatly influenced by the design principle of the Standard Template Library (STL). The STL provides five components algorithms, containers, iterators, function objects and adaptors for this purpose that allows programmers to program data structures of the STL with their own algorithms, and to use algorithms of the STL with their own data structures [Stepanov 1995]. Similarly, the C++ standard library provides multiple containers with different capabilities and runtime complexities, template classes such as the std::vector that help to organize data [Stroustrup 2013]. Allocation of memory is usually performed with additional predefined or used-defined allocator classes. The C++ standard library also provides free function templates, also known as algorithms, that operate on a one-dimensional range of container data. The range is specified by an iterator pair instantiated by the corresponding container. Iterators of the standard library are one of the five iterator types with different navigation and access capabilities that an algorithm requires for its execution. In other words, an algorithm can only process container data if the algorithm's iterator type requirement is fulfilled by the container's iterator.

We have used this approach to minimize the dependencies between tensor algorithms and the implementation of tensor data structures to yield a separation of concerns. Therefore, we did not use inheritance but mostly composition to establish a loose coupling of software elements. Our tensor framework consists of multiple software stack layers that are illustrated in Figure 1.

Fig. 1. UML diagram of the software stack of our tensor framework with four components. Black arrows denote associations and visualize the direction of the association. A line annotated with a diamond denotes a composition. Template parameters of the classes are presented with dashed rectangles.

We will start with the highest layer in the software stack. It consists of two template classes fhg::tensor and fhg::tensor_view. Both provide a high-level interface with which the user can conveniently implement tensor algorithms with member functions. Yet, the implementation of both template classes use software components and call functions of the lower layers. For instance, data organization and access is accomplished with the corresponding template classes fhg::multi_array and fhg::multi_array_view, respectively.

The template class fhg::multi_array can be regarded as a multidimensional container supporting random access with multiple and single indices. It is designed as a resource handle (similar to the std::vector) for storing a collection of elements that are ordered in a rectangular pattern. It abstracts from the storage layout in memory and provides a

convenient interface to access elements with multi-indices. Allocation and deallocation is handled with the help of the `std::vector` template that stores elements contiguously in memory. Its template parameters determine the type of the elements and allocator with which memory is acquired and released. Parameters such as order, dimensions, index offsets or the storage layout are member variables of the `fhg::multi_array` template. Choosing all of the parameters to be runtime-variable allowed us to verify all of tensor template functions and to provide flexible and runtime-adaptable containers.

The `fhg::multi_array_view` template class is similar to a container adaptor that references a selected region of an `fhg::multi_array` object offering the same functionality as the `fhg::multi_array` template class. It serves as a proxy for conveniently accessing and manipulating selected memory regions of an `fhg::multi_array` object. The template parameter is therefore restricted to a type that has the same properties as `fhg::multi_array`. We have applied the factory method design pattern without subtype polymorphism where an object of type `fhg::multi_array_view` can only be generated by calling the overloaded function operator of an `fhg::multi_array` object with a tuple of `fhg::range` class objects. An `fhg::multi_array` object therefore generates references to itself.

Both template classes `fhg::multi_array` and `fhg::multi_array_view` do not provide virtual member functions (especially a virtual destructor) and therefore promote aggregation instead of inheritance. The strategy is also used in the standard library for all containers in order to prevent indirect functions calls and runtime overhead. The template class `fhg::tensor` therefore contains the `fhg::multi_array` template class and provies additional member functions with which the user can perform arithmetic operations such as the tensor multiplications. The `fhg::tensor_view` template class is also designed to wrap and extend the functionality of the `fhg::multi_array_view` template. The element type of the both tensor templates must support scalar arithmetic operations for the unspecialized case.

The template class `fhg::multi_iterator` defines a multidimensional iterator with which the complete multi-index set of a multidimensional array is accessible while the array can have various data layouts, runtime variable dimensions and order. It provides member functions that create iterator pairs of type `iterator_t` and help to implement tensor functions without explicit specification of the data layout and dimensions. Our framework uses the `fhg::stride_iterator` class for this purpose. The class however does not depend on the `fhg::multi_iterator` template class. It can be instantiated and used separately to modify and access elements of a specific dimension for instance. Both iterator types can be conveniently instantiated with member functions of template classes.

The bottom layer contains tensor template functions that are listed in Table 1. The template parameters of the function templates denote multidimensional iterator types such as the `fhg::multi_iterator`. The user can utilize our or his own implementation to call the template functions as long as the multidimensional iterator type fulfills specified criteria. Using iterators, the template functions do make assumptions about the underlying data structure or management relieving the user to consider data layout. The tensor template functions are designed in a recursive fashion and implemented as non-member function templates where the maximum depth of the recursion mostly equals the number of dimensions of the corresponding multidimensional array. They do not flatten or rearrange tensors and perform tensor operations in-place using recursion. The framework offers two algorithm packages. The first one contains function templates that have the same semantic and similar function signature as the function templates offered by the algorithm package of the `C++` standard library. The second package contains function template that implement tensor multiplication operations such as the tensor-tensor multiplication.

3 MULTIDIMENSIONAL CONTAINERS AND VIEWS

A multidimensional array is a p-dimensional table of values that are arranged in a rectangular pattern and accessible via multiple indices. If it represents higher-order tensors for a given finite basis of the tensor product space, its elements are either real or complex numbers [Golub and Van Loan 2013; Lim 2017]. For the following discussion, we postulate multidimensional arrays to hold any type of values that can be arranged in a rectangular pattern. We use the following notation to denote a multidimensional array:

$$\underline{\mathbf{A}} = \left(a_{i_1, \ldots, i_p} \right)_{i_r \in I_r}, \tag{1}$$

where I_r is the r-th index set with

$$I_r := \{ i_r \in \mathbb{Z} \mid o_r \leq i_r < o_r + n_r \ \wedge \ o_r \in \mathbb{Z} \wedge n_r \in \mathbb{N} \}, \tag{2}$$

and $|I_r| = n_r$. The number p is a positive integer and will be referred to as the *order* or *rank* of a multidimensional array. The (dimension) extent n_r of the dimension r can be different but must be greater than one. The tuple $\mathbf{n}$ of length p is the *shape tuple* of a multidimensional array with $\mathbf{n} \in \mathbb{N}^p$. Each index $i_r \in I_r$ is biased with an index offset $o_r \in \mathbb{Z}$. We denote $\mathbf{o}$ with $\mathbf{o} \in \mathbb{Z}^p$ as the *index offset tuple* of a tensor. We can then derive the *multi-index set* as the Cartesian product of all index sets I_r such that

$$\mathcal{I} := I_1 \times I_2 \times \cdots \times I_p, \tag{3}$$

with $\mathcal{I} \subset \mathbb{Z}^p$. We denote an element $(i_1, \ldots, i_p)$ of a multi-index set as a *multi-index* with $i_r \in I_r$. Elements of a tensor $\underline{\mathbf{A}}$ are uniquely identifiable using round brackets with a multi-index and

$$\underline{\mathbf{A}}(i_1, i_2, \ldots, i_p) = a_{i_1 i_2 \ldots i_p}. \tag{4}$$

Example 3.1. Let $\underline{\mathbf{A}}$ be an array of order p with a shape tuple $\mathbf{n}$ where $\mathbf{n} = (4, 2, 3)$ and $I_1 = \{0, 1, 2, 3\}$, $I_2 = \{0, 1\}$, $I_3 = \{0, 1, 2\}$. Each element of a tensor can be identified with a multi-index (i_1, i_2, i_3) in $\mathcal{I}$. Using the notation in Eq. (1) the multidimensional array can be illustrated as follows:

$$\underline{\mathbf{A}} = \left(\begin{array}{cc|cc|cc} a_{0,0,0} & a_{0,1,0} & a_{0,0,1} & a_{0,1,1} & a_{0,0,2} & a_{0,1,2} \\ a_{1,0,0} & a_{1,1,0} & a_{1,0,1} & a_{1,1,1} & a_{1,0,2} & a_{1,1,2} \\ a_{2,0,0} & a_{2,1,0} & a_{2,0,1} & a_{2,1,1} & a_{2,0,2} & a_{2,1,2} \\ a_{3,0,0} & a_{3,1,0} & a_{3,0,1} & a_{3,1,1} & a_{3,0,2} & a_{3,1,2} \end{array} \right)$$

A (multidimensional) view $\underline{\mathbf{A}}'$ of a tensor $\underline{\mathbf{A}}$ is a reference to a specified region or domain of $\underline{\mathbf{A}}$ and has the same order p and data layout $\boldsymbol{\pi}$ as the referenced multidimensional array. It can be regarded as a lightweight handle with a shape tuple $\mathbf{n}'$ where the dimensions of the view and referenced multidimensional satisfy $n_r' \leq n_r$ for $1 \leq r \leq p$. We define a *section* or *view* of a multidimensional array of order p in terms index-triplets (f_r, t_r, l_r), pairs of indices (f_r, l_r) or scalars $i_r \in I_r$ for all p dimensions. The indices f_r, l_r define the lower and upper bound of an index range where t_r the step size for the r-th dimension satisfies $0 \geq f_r \geq l_r \geq n_r$ and $t_r \in \mathbb{N}$ for $1 \leq r \leq p$. The shape tuple $\mathbf{n}'$ of the view $\underline{\mathbf{A}}'$ is given by

$$n_r' = \left\lfloor \frac{l_r - f_r}{t_r} \right\rfloor + 1. \tag{5}$$

The r-th index set of a view $\underline{\mathbf{A}}'$ is then defined as

$$I_r' := \{ i_r' \in \mathbb{Z} \mid o_r \leq i_r' < n_r' + o_r \ \wedge \ o_r \in \mathbb{Z} \}, \tag{6}$$

where o_r is the r-th index offset of the referenced multidimensional array $\underline{\mathbf{A}}$. Using the notation in the previous section, a view $\underline{\mathbf{A}}'$ of a multidimensional array $\underline{\mathbf{A}}$ is denoted by

$$\underline{\mathbf{A}}' = \left(a'_{i'_1,\ldots,i'_p}\right)_{i'_r \in I'_r}. \tag{7}$$

Analogous to the multidimensional arrays, the multi-index set of a view is given by

$$\mathcal{I}' := I'_1 \times I'_2 \times \cdots \times I'_p, \tag{8}$$

where $\mathcal{I}' \subseteq \mathcal{I} \subseteq \mathbb{Z}^p$. A *slice* is a special kind of view with two dimensions that have the same extent as the referenced multidimensional array. Let k and l be the subscripts of those two dimensions. The resulting multi-index set $\mathcal{I}'$ of the slice is given by Eq. (8) with

$$I_r = \begin{cases} \{o_r, \ldots, o_r + n_r\} & \text{for } r = k \wedge r = l, \\ \{o_r\} & \text{otherwise.} \end{cases} \tag{9}$$

A *fiber* has exactly one dimension that is greater than one.

Example 3.2. Let $\underline{\mathbf{A}}$ be the 3-way array from the previous example and zero-based index offsets. Let $f_1 = 1$, $l_1 = 3$, $t_1 = 2$, and $f_2 = 0$, $l_2 = 1$ and $f_3 = l_3 = 2$ index triplets for choosing a section of $\underline{\mathbf{A}}$. With $\mathbf{n}' = (2, 2, 1)$ being the shape tuple of the view $\underline{\mathbf{A}}'$, elements of the view are given by

$$\underline{\mathbf{A}}' = \begin{pmatrix} a'_{0,0,0} & a'_{0,1,0} \\ a'_{1,0,0} & a'_{1,1,0} \end{pmatrix} = \begin{pmatrix} a_{1,0,2} & a_{1,1,2} \\ a_{3,0,2} & a_{3,1,2} \end{pmatrix}.$$

3.1 Data Organization and Layout

The above notation of a tensor is an abstract representation of a multidimensional array. Usually a multidimensional array stores its elements in memory where the latter can be addressed with a single index. We refer to that single index k as the (absolute) memory address. Our implementation of the tensor template class is stored contiguously in memory such that elements of array are addressable using a single index j as well. The single index shall denote a relative position within the multidimensional array. In that case, the j-th element of $\underline{\mathbf{A}}$ in memory is then given by the linear function

$$k = k_0 + j \cdot \delta, \tag{10}$$

where δ is the number of bytes to store an element of a multidimensional array $\underline{\mathbf{A}}$ and $k_0 \in \mathbb{N}_0$ is the (absolute) memory address of the first element. The domain of the function is given by

$$\mathcal{J} := \{0, 1, \ldots, \prod_{r=1}^{p} n_r - 1\} \tag{11}$$

which we denote as the *(relative) memory index set* of a multidimensional array of order p where n_r is the length of the r-th dimension. Therefore, we may enumerate elements of a multidimensional array $\underline{\mathbf{A}}$ with a single scalar index j of the index set $\mathcal{J}$. We denote elements of the memory index set as *memory indices* of the multidimensional array that correspond to displacements relative to the memory address of the first array element.

Given a contiguous region within a single indexed memory space, the *data layout* (storage format) of a contiguously stored dense multidimensional array defines the ordering of its elements within the specified memory region. A multidimensional array with the dimensions $n_1, \ldots, n_p$ has $\prod_r n_r!$ possible orderings. In practice however, only a subset of all possible orderings are considered. In case of two dimensions for instance, most programming languages

arrange elements of two-dimensional arrays either according to the row- or column-major storage format where adjacent elements of a row or column are successively stored in memory. More sophisticated non-linear layout functions have been investigated for instance in [Chatterjee et al. 1999; Elmroth et al. 2004] with the purpose to increase the data locality of dense matrix operations. These type of layout functions partition two-dimensional arrays into hierarchical ordered blocks where elements of a block are stored contiguously in memory.

Our data structure supports all non-hierarchical data layouts including the first- and last-order storage formats. We denote both formats as standard data layouts of dense multidimensional arrays, where the former format is defined in the Fortran, the latter in the C and C++ language specification, respectively. Non-hierarchical data layouts can be expressed in terms of permutation tuples $\boldsymbol{\pi}$ which we denote as the *(data) layout tuple*. The q-th element π_q of a permutation tuple $\boldsymbol{\pi}$ corresponds to an index subscript r of a multi-index i_r with the precedence q where $i_r \in I_r$ and $1 \leq q, r \leq p$. In case of the first-order format, the layout tuple is defined as

$$\boldsymbol{\pi}_F := (1, 2, \ldots, p), \tag{12}$$

where the precedence of the dimension ascends with increasing index subscript. The layout tuple of the last-order storage format is given by

$$\boldsymbol{\pi}_L := (p, p - 1, \ldots, 1). \tag{13}$$

We might therefore interpret the layout tuple also as a precedence tuple of the dimensions. The set of all possible layout tuples is denoted by Π_p. The number of all possible non-hierarchical element arrangements is $p!$ and equal to the number of elements in Π_p.

The q-th stride w_q is a positive integer and defined as the distance between two elements with identical multi-indices except the q-th indices differing by one. More specifally, given a layout tuple $\boldsymbol{\pi}$ and the shape tuple $\mathbf{n}$, elements of a stride tuple $\mathbf{w}$ are given by:

$$w_{\pi_r} = \begin{cases} 1 & \text{for } r = 1. \\ \prod_{q=1}^{r-1} n_{\pi_q} & \text{otherwise.} \end{cases} \tag{14}$$

Please note that for any layout tuple $\boldsymbol{\pi}$ the corresponding stride tuple $\mathbf{w}$ satisfies $1 \leq w_{\pi_q} \leq w_{\pi_r}$ with $1 \leq q < r \leq p$. For instance, the Fortran language specification stores elements of a multidimensional array according to the first-order storage format such that the corresponding stride tuple is defined as

$$\mathbf{w}_F = (1, \ n_1, \ n_1 \cdot n_2, \ \ldots, \ \prod_{r=1}^{p-1} n_r). \tag{15}$$

In case of the first-order storage format, the q-th stride increases with consecutive index subscripts by the factor of n_{q-1}. The last-order storage format is given by $\boldsymbol{\pi}_L = (p, p-1, \ldots, 1)$ Given the layout tuple in Eq. (13), the stride tuple is then

$$\mathbf{w}_L = (\prod_{r=2}^{p} n_r, \prod_{r=3}^{p} n_r, \ldots, n_p, 1), \tag{16}$$

which corresponds to the definition in the 2011 C-language specification. We denote the set of all stride tuples by $\mathcal{W}$ with $\mathcal{W} \subseteq \mathbb{N}^p$.

3.2 Tensor Template Class

The parameterized class `fhg::tensor` is a sequence container which is responsible for the data management. It resembles the vector template class `std::vector` of the `C++` standard library. It has two template parameters, `value_type` and `allocator`. The former one determines the data type of the elements, the latter the type of the memory allocator. The type `value_type` must be a numeric type with which addition, subtraction, multiplication and division can be performed.

```
template <class value_type, class allocator>
class tensor;
```

This restriction arises due to member functions such as overloaded arithmetic operators implementing pointwise tensor operations for convenience. In order to support a wider range of applications we offer the template class `fhg::multi_array` that does not include any arithmetic operations and therefore has lesser restrictions on the data type of its elements. It has the same template parameters and member variables as the `fhg::tensor` template class. The user is for instance free to set the first template parameter to boolean data type and equip the `fhg::multi_array` with bitwise operations.

Similar to the `std::vector` template class, `fhg::tensor` contains public member types such as `value_type`, `size_type`, `difference_type`, `pointer`, `const_pointer`, `reference`, `const_reference`, `iterator`, `const_iterator`. The last two types denote multidimensional iterators which will be explained in the following subsection. The template class also defines the member type `tensor_view_t` that is `fhg::tensor_view<tensor>` where tensor is the data structure itself. Consider the following listing with template class instances of the template class `fhg::tensor`.

```
using cftensor = fhg::tensor<std:complex<float>>;
using itensor  = fhg::tensor<int>;
using dtensor  = fhg::tensor<double>;
```

Instances of the first specialization contain complex numbers in single precision, objects of the second type integer numbers and the third numbers in double precision. The template class `tensor` provides public member functions in order to

- instantiate and copy tensor instances, e.g. `tensor()`,
- assign data elements, e.g. `operator=(..)`,
- access elements, e.g. `at(..)`, `operator[..]`,
- perform tensor multiplication operations, e.g. `times_tensor(..)`, `inner(..)`,
- perform tensor pointwise operations, e.g. `operator+=(..)`, `operator-()`,
- generate views, e.g. `operator()(..)`,
- generate iterators, e.g. `begin()`, `mbegin()`,

including all size and capacity functions of the `std::vector` such as `size()`, `empty()`, `clear()`, `data()`. However, we do not support stack and list operations such as `push_back`, `pop_back`, `insert` or `erase`.

The type `fhg::tensor` shall denote the type `fhg::tensor<value_type,allocator>` for given template parameters `value_type` and `allocator`. The `fhg::tensor` template class provides the following constructor declarations.

```
tensor();
tensor(shape const&);
```

```
tensor(shape const&, offset const&);
tensor(shape const&, layout const&);
tensor(shape const&, offset const&, layout const&);
```

The default constructor of the `fhg::tensor` class calls the default constructor of all member variables. The `fhg::tensor` template class contains the following `std::vector` private member variables in order to store

- data elements of type `value_type`,
- extents of type `size_type`,
- layout elements of type `size_type`,
- strides of type `size_type`,
- index offsets of type `difference_type`.

The above mentioned public member types are derived from the `std::vector` template class with the specified template parameters `value_type` and `allocator`. The member variable of type `std::vector<value_type,allocator>` helps `fhg::tensor` class template to allocate and free memory space where the data elements are contiguously stored. The position of the elements corresponds to indices of the plain index set given by Eq. (11). The second constructor specifies the extents with the help of `fhg::shape` class instances. The order is automatically derived and the length of the data vector is determined. The default layout tuple is computed with Eq. (12). The constructor can then compute the strides with Eq. (14) initializing the index offsets with zero. The remaining constructors allow to specify different offset and/or layout tuples using the auxiliary `fhg::layout` and `fhg::offset` classes, respectively. The specification of the layout using the `fhg::layout` class is constructed with the a permutation tuple with one-based indices. All three auxiliary classes provide an initialization with the `std::initializer_list` for a convenient initialization of their elements. The elements are stored with a specialization of the `std::vector` template class. Consider the following example where two statements instantiate objects of the `fhg::tensor<double>` class.

Example 3.3. The first constructor generates an object A where the offsets are all zero and the elements are stored according to the first-order storage format and therefore initializing the layout tuple with {1,2,3} as given by Eq. (12). The second object B has the same shape but initializes the index offsets with 1,-1, 0 using the last-order format for the element ordering.

```
tensor<double> A(shape {4,2,3});
tensor<double> B(shape {4,2,3}, offset {1,-1,0}, layout {3,2,1});
```

We can therefore assert that the strides of objects A and B are {1,4,8} and {6,2,1}, respectively.

The copy assignment operators `operator=()` of the `fhg::tensor` class template are responsible for copyin data and protecting against self-assignment. Consider the following overloaded assignment operators.

```
tensor& operator=(tensor        const&);
tensor& operator=(tensor_view_t const&);
```

The first method copies all private member variables of the source instance such that source and destination have the same shape tuple and the same order after the assignment. The layout, stride and index offset tuples of the destination tensor object are only copied if the order of the source and destination tensor objects differ. In either cases, the user can expect the source and destination `fhg::tensor` class instances to be equal and independent after the copy operation,

see [Stroustrup 2013] for a detailed discussion. We defined two tensor to be equal if they have the shape tuple, order and elements with the same multi-index independent of their layout, index offset tuple and allocator. The second assignment operator copies all elements of the source `fhg::tensor_view` class template instance including the shape tuple. The layout and index tuples are only copied if the order to the source and destination tensor objects differ.

Example 3.4. Let `A` and `B` be `fhg::tensor` objects of the previous Example 3.3. The following code snippet initializes all elements of `A` with the value one using the assignment operator and copies the content of `A` to `B`.

```
B = A = 1;
```

As both objects do have equal extents, only data values of `A` are copied to `B` without modifying the offset and layout tuple of `B`.

Besides the type of the data elements, the user can change the content and the size of all member variables at runtime. The framework offers `reshape` and `relayout` member functions that allow to dynamically adjust the shape and layout tuple, respectively. The container automatically adjusts the strides and reorders the in memory if necessary. The `fhg::tensor` template class provides member functions for accessing elements with multi-indices and scalar memory indices. The following listing depicts two member functions that return a `reference` of the specified element.

```
template<class ... MultiIndex>
reference at(MultiIndex&& ...);
reference operator[](size_type index);
```

The first access function is a template function with a variadic template that allows to conveniently access elements with multi-indicies. It transforms multi-indices onto memory indices according to the stride tuple of the template class. Given the order p and a stride tuple $\mathbf{w}$ with respect to a layout tuple $\boldsymbol{\pi}$ and stride tuple $\mathbf{w}$ given by Eq. (14), the *non-hierarchical layout function* λ is defined by

$$\lambda : \mathbb{N}^p \times \mathbb{Z}^p \times \mathbb{Z}^p \to \mathbb{N}, \quad \lambda(\mathbf{w}, \mathbf{i}, \mathbf{o}) = \sum_{r=1}^{p} w_r(i_r - o_r). \tag{17}$$

For fixed stride tuples $\mathbf{w}_F$ and $\mathbf{w}_L$, the layout functions $\lambda_{\mathbf{w}_F}$ and $\lambda_{\mathbf{w}_L}$ coincide with the definitions provided in [Andres et al. 2010; Chatterjee et al. 1999; Garcia and Lumsdaine 2005].

Example 3.5. Let `A` be a multidimensional array of order 3 and type `fhg::tensor` with equal extents n and elements all set to zero. We can create an identity tensor with ones in the superdiagonals by the following statement.

```
for(auto i = 0u; i < A.extents().at(0); ++i)
  A.at(i,i,i) = 1.0;
```

Object `A` is processed independently of its layout tuple.

The argument size of the variadic template however must be specified at compile time. Note that, the multidimensional array `A` of Example 3.5 must be of order three. The class offers additional member functions that allow to specifiy multi-indices with the runtime-variable length using the `std::vector`. Consider the following code snippet in the next example.

Example 3.6. Given a multidimensional array `A` of order p where p is initialized at runtime. We can create an identity tensor with a vector of size p that is initialized with the value i.

```
for(auto i = 0u; i < A.extents().at(0); ++i){
  A.at(std::vector<std::size_t>{p,i}) = 1.0;
}
```

Using multi-indices abstracts from the underlying data layout and enables the user to write layout invariant programs as all elements have a unique multi-index independent of the data layout, see Eq. (4). However, each element access with multi-indices involves a multi-index to memory index transformation that is given by Eq. (17).

The second member function of the previous listing provides memory access with a single scalar parameter corresponding to memory indices of the memory index set of a multidimensional array, see Eq. (11). Consider the following example.

Example 3.7. Let A be a multidimensional array specified in the previous example. The statement in the listing sets all elements of the object using a single induction variable j that takes values of the complete memory index set of the array.

```
for(auto j = 0u; i < A.size(); ++j)
  A[j] = 0.0;
```

In contrast to the access with multi-indices, referencing elements with memory indices does not include index transformations. It might be seen as a low-level access where the user is responsible for referencing the correct elements. We might say that scalar indexing is convenient whenever the complete memory index set is accessed and the multi-index or the ordering of the data elements are not relevant for the implementation of the tensor operation. Higher-level tensor operations such as the tensor transposition require some type of multi-index access.

3.3 View Template Class

An instance of the parameterized class `fhg::tensor_view` references a container instance that has been selected using `fhg::range` class instances.

```
template <class tensor_t>
class tensor_view;
```

The template parameter `tensor_t` can be of type `fhg::tensor<value_type, allocator>`. However, the user can also set a different `tensor_t` type. In that case `tensor_t` type requires the type `tensor_t::value_type` to be an arithmetic type and the referenced container to store its data contiguously with access to all previously mentioned public member types and variables. A template class instance `fhg::tensor_view` contains therefore the same public member types and member methods as the referenced template class instance `fhg::tensor` with only minor differences. This has the advantage that both template class instances `fhg::tensor` and `fhg::tensor_view` can be treated almost identically.

The `fhg::tensor_view` template class contains the following private member variables:

- a pointer to the selected tensor of type `tensor_t*`,
- a pointer to the first element of type `value_type*`,
- ranges of type `fhg::range`,
- extents of type `size_type`,
- strides of type `size_type`.

The layout and offset tuple can be obtained by calling the appropriate functions of the referenced `fhg::tensor` object. The constructor only requires the specification of the referenced `fhg::tensor` and p `fhg::range` objects that are used to initialize the remaining private member are initialized. The r-th `fhg::range` object defines an index set I'_r that must be a subset of the index set I_r of the `fhg::tensor` as we have already previously defined, see Eq. (6). Moreover, the constructor of the `fhg::range` class takes either two or three indices where `first`, `last` and `step` corresponds to f_r, l_r and t_r indices of Eq. (6).

```
range(size_type first, size_type last);
range(size_type first, size_type step, size_type last);
```

The constructor also tests if the specified ranges are valid and do not violate any bounds of the selected `tensor_t` instance. The extents n'_r of the view are therefore computed according to Eq. (5). The pointer to the first element of the array is given by adding an offset γ to the pointer to first element of the selected `fhg::tensor`. The offset is computed by combining the p lower bounds `first` using the layout function λ in Eq. (17) such that

$$\gamma := \lambda(\mathbf{w}, \mathbf{f}, \mathbf{o}), \quad \text{with } \mathbf{f} = (f_1, \ldots, f_p) \tag{18}$$

where $\mathbf{w}$ and $\mathbf{o}$ are the stride and offset tuple of the `fhg::tensor` object and f_r is the lower bound `first` of the r-th `fhg::range` instance. The stride tuple $\mathbf{w}'$ of a view can be computed with Eq. (14).

The tensor template class `fhg::tensor` provides two methods that overload the function call operator and generate class instances of type `fhg::tensor_view<tensor>`. The latter type is equivalent to `tensor_view_t` where `tensor` is the type of the calling template class instance. Please consider the following listing which is an excerpt of the `fhg::tensor` template class.

```
template<class ... domain>
tensor_view_t operator()(domain&& ...);
tensor_view_t operator()(std::vector<range> const&);
```

The first function is a template function with a template parameter pack `domain` that is recursively unpacked for either integers or objects of type `fhg::range`, i.e. index triplet or pairs. The unpacking however is performed with protected template functions that are called from the constructor of the corresponding `tensor_view_t` class.

```
tensor_view(tensor_t*);
template<class ... domain>
tensor_view(tensor_t*, domain&& ...);
```

Both methods are declared as friend in the `tensor_view` template class and only callable from the referenced `fhg::tensor` objects. The first constructor generates a reference to an unselected `tensor_t` object where no domain is specified. This means that the reference covers the complete domain of the tensor object that has been selected. The second constructor uses a variadic template that processes either integers or objects of type `fhg::range`. Both constructors are called from the tensor template class from the overloaded function call operator. Please note that `fhg::tensor_view` the class instance cannot be instantiated without a `tensor_t` object as the default constructor of `fhg::tensor_view` the template class is deleted. Consider the following example that illustrates the creation of an view object `Av` of type `tensor_view<tensor<float>>`. Please note that only a `tensor_t` object is able to call the protected `fhg::tensor_view` constructor with a function call operator of `tensor_t`.

Example 3.8. Let A be an object tensor of type tensor<float> from the previous Example 3.3 with n = {4,2,3} and zero-based index offsets. Let also r1 = range(1,2,3), r2 = range(0,1) and r3 = range(2) be index ranges with which the 3-way array A is selected. According to Eq. (5) the shape tuple of the view Av is then given by nv = {2,2,1}.

```
tensor_view<tensor<float>> Av(A.data(), range{1,2,3}, range{0,1}, 2);
```

The 3-way array A can also be seen as a composition of three slices Av0, Av1 and Av2 where the i-th slice is given by

```
tensor_view<tensor<float>> Avi(A.data(), range{}, range{}, i);
```

where an empty range object contains all indices of the corresponding index set.

The instantiation of the view Av is only exemplified in the following listing in order to demonstrate the instantiation procedure. Please note that we did not provide a mechanism for counting the number of fhg::tensor_view objects associated to one tensor_t object. The implementation of such a mechanism requires tensor objects to know about their references and to signal its views when the destructor is called. An existing fhg::tensor_view object can therefore become invalid when its referenced fhg::tensor instance does not exist any more. This situation is similar to a dangling pointer. The user is therefore responsible for avoiding the situation where the referenced fhg::tensor object falls out of scope or is deleted. An alternative way to implement views of tensors is to only allow temporary fhg::tensor_view instances that is an rvalue. In order to enable this restriction, the copy constructor and copy assignment operator needs to be deleted or hidden from the user. In that case, only functions that allow only rvalue references of fhg::tensor_view instances using the && declarator operator. Please consider following listing that demonstrates the instantiation of tensor views.

Example 3.9. Let A be an instance of fhg::tensor<float> template class with the shape tuple (3,4,2). We can then create a view Av of A by calling the overloaded call operator operator() of A with fhg::range instances.

```
tensor_view<tensor<float>> Av = A ( range {1,2}, range {1,2}, 1 );
```

The operator calls the constructor of the tensor_view_t class with the specified ranges. Please note that instead of using the auto specifier, we explicitly defined the type of the view for demonstration purposes. The number of arguments in the parameter pack of the template function must be known at compile time. If the number of fhg::range instances are runtime-variable, the user can call the second member function that creates a view of tensor using a std::vector. Please note that statement in Example 3.9 is very similar to the MATLAB syntax where a section of a multidimensional array A is created with A(1:2,1:2,1) using the call operator in conjunction with the colon operator.

Access functions of the fhg::tensor_view template class with multi-indices depict the same interface as for the fhg::tensor template class.

```
template<class ... MultiIndex>
reference at(MultiIndex&& ...);
reference operator[](size_type);
```

However, the access function cannot use the layout function λ in Eq. (17) in order to generate memory indices. Each index in I'_r needs to be transformed into an index of the set I_r before an element can be accessed. Memory indices of a view's elements are then given by the function λ' with

$$\lambda' : \mathbb{N}^p \times \mathbb{Z}^p \times \mathbb{Z}^p \to \mathbb{N}, \quad \lambda'(\mathbf{w}', \mathbf{i}', \mathbf{o}) = \gamma + \lambda(\mathbf{w}'', \mathbf{i}', \mathbf{o}) \tag{19}$$

where γ is the offset given by Eq. (18) and $\mathbf{w}''$ is a modified stride tuple with $w''_r = w'_r t_r$.

Example 3.10. Let A be the multidimensional array where all the extent of the dimensions are 4. Similar to the Example 3.5, we can instantiate all diagonal elements of a slice Av to one using a single for-loop.

```
auto Av = A ( range {}, range {}, 1 );
for(auto i = 0; i < 4; ++i)
  Av.at(i,i,0) = 1;
```

Internally, the at() operation computes the relative memory indices from the multi-indices (i,i,0) according to Eq. (19).

The instantiation of multidimensional iterators, views and the interface will be postponed to the following chapters.

3.4 Output Stream Class

In order to compare and verify our numerical results with MATLAB we provided our own output stream class fhg::matlab_ostream for a formatted output of the fhg::tensor and fhg::tensor_view instances. The class overloads stream operators << for the formatted output. The constructor of the class expects an instance of the std::basic_ios class template from the standard template library such as std::cout. The user can directly utilize the instance fhg::mcout of the fhg::matlab_ostream for convenience. The stream function relayouts the multidimensional array for the first-order storage format and outputs the elements of the tensor using the MATLAB notation. One can also input additional MATLAB commands into the stream such as plotting statements. In this way, MATLAB scripts can be generated with our tensor library. Consider the following listing where the complete tensor A is inserted into the standard output stream.

Example 3.11. The elements of A are stored according to the last-order storage format and their values are initialized with indices of the plain index set.

```
fhg::mcout << "A = " << A << std::endl;
fhg::mcout << "plot(A(:) - Aref(:));" << std::endl;
// output:
// A = cat(3, [ 0 2 4 6 ; 8 10 12 14 ; 16 18 20 22 ],...
//            [ 1 3 5 7 ; 9 11 13 15 ; 17 19 21 23 ]);
// plot(A(:) - Aref(:));
```

The output can be directly copied into MATLAB's command window and executed if the reference tensor Aref is already defined. Please note that the matlab_stream object could have been also instantiated with an output file stream std::ofstream. We have used this approach to verify the results of our tensor algorithms with the tensor toolbox described in [Bader and Kolda 2006]. The framework also provides overloaded input and output stream operators for the tensor class template in conjunction with standard output streams.

4 MULTIDIMENSIONAL ITERATOR

An iterator is data structure that represents positions of elements in a container and allows to traverse between successive elements of the container. The iterator concept allows to decouple or minimize the dependency between algorithms that use iterators and data structures that create iterators. Pointers for instance are akin to iterators and may

be regarded as an instance of the iterator concept. A pair of iterators $[a, b)$ define a (half-open) range of the referenced container. Containers of the standard template library provide member functions `begin()` and `end()` in order to generate a half-open range for the corresponding container where the iterator a points to the first element and b to the position after the last element.

The standard template library divides an iterator into five iterator categories with different capabilities where random-access iterators provide the largest number of access and iteration methods with iterator arithmetic similar to that of a pointer. Moreover, it accesses any valid memory locations in constant time. Template functions provided by the standard template library operate with iterators and determine the required category of the iterators. The `std::for_each()` template function for instance works with input iterators while `std::sort()` requires iterators with a random-access iterator tag. Hence, not every container can instantiate a random-access iterator such as associative container `std::map` such that `std::sort()` is not compatible with iterators generated by `std::map` objects. However, iterator instances generated by `std::vector`, `std::array` and `std::deque` objects support random-access and can use the function `std::sort()`. In case of the containers `std::vector`, `std::array` and `std::deque` that implement one-dimensional array with a contiguous memory region, the half-open range covers the complete (memory) index set I. Moreover, the memory locations of the iterators generated by `begin()` and `end()` are k_0 and $k_0 + (|I|+1)\cdot\delta$, respectively, where δ is the number of bytes to store an element as previously discussed. We have implemented two template classes `fhg::iterator` and `fhg::multi_iterator` that allow to iterate over the multi-index set of a multidimensional array or a view. Objects of `fhg::multi_iterator` instantiate `fhg::iterator` objects in order to define stride-based ranges.

4.1 Stride-based Iterator

The iterator `fhg::stride_iterator` has the same traits as the iterator type of the `std::vector` and therefore exhibits the same template class signature.

```
template<class iterator_t>
class stride_iterator;
```

It provides all access properties of a random-access iterator and is therefore tagged as such. The template parameter `iterator_t` is an iterator type and must be accepted by the template class `std::iterator_traits` or by any of its pointer specializations for accessing the public member types such as `iterator_category`. We might think of the `fhg::stride_iterator` template class as an iterator adaptor for the standard random-access iterator with the same member functions and same interface. However, a stride-based iterator also features an additional member variable in order to store a stride of type `std::size_t`. The constructor has therefore the following signature.

```
stride_iterator(iterator_t location, std::size_t stride);
```

An object of type `iterator_t` points to a valid memory location which we will denote as `k`. The second parameter of the constructor determines the stride `w` with which the iteration is performed.

```
stride_iterator& operator=(stride_iterator const& other);
```

The copy-assignment operator of the `fhg::iterator` copies the current position `k` and the stride `w` of the `other` argument. We therefore consider two dimension-based iterators `i1` and `i2` equal if the current positions `i1.k`, `i2.k` and the strides `i1.w`, `i2.w` of the iterators are equal. The expression `(i1=i2) == i2` therefore returns true as both iterators have equal

position and stride after the assignment (i1=i2). The following example illustrates how two stride-based iterators define a range for a given dimension.

Example 4.1. Let A be a three dimensional array with elements of type float stored according to the first-order storage format and let {4,3,2} and {1,2,3} be the shape and data layout tuple, respectively. The resulting stride tuple is then given by {1,4,12} according to Eq. (16). We can then use the following two statements where the first line instantiates iterators of std::vector and the second to iterators of the template class fhg::tensor.

```
std::vector<float>::iterator f1 {A.data()      }, l1 {A.data()+w[2]     };
fhg::tensor<float>::iterator f2 {A.data(),w[1]}, l2 {A.data()+w[2],w[1]};
```

The expressions A.data() returns a pointer to first element. The array w is the stride tuple of A. The first half open range [f1,l1) covers all elements with memory indices from 0 and to 12. The second range [f2,l2) only covers elements with the multi-indices $(0, i, 0)$ for $0 \leq i < 2$ that corresponds to a fiber of the multidimensional array with elements that have the memory indices 0, 4 and 8, see Eq. (17).

In order to iterate over the second dimension using the iterator pair [f1,l1), one has to explicitly provide the corresponding stride w[1]. In case of the second iterator pair, no stride has to be specified within the loop.

Example 4.2. Let f1,l1 and f2,l2 be the iterator pair that have been defined in Example 4.1. Both of the following statements can be used to initialize the first row of the object A.

```
for(; f1 != l1;  f1+=w[1]) { *f1 = 5.0; }
for(; f2 != l2;  f2+=1   ) { *f2 = 5.0; }
```

Please note that the standard iterator f1 needs to be explicitly incremented with the stride of the array A. The multidimensional iterator encapsulates the stride from the algorithm such that a normal increment with the operator++() or operator+=(1) suffices. Consider for instance the following function call of the std::fill function that corresponds to the previous statements.

```
std::fill(f2, l2, 5.0);
```

The iterator can therefore be used with all algorithms in the algorithm and numeric headers of the C++ standard library. For convenience, the user can also call member functions of the container classes to instantiate stride-based iterators and to define a range.

```
iterator begin(std::size_t dim);
iterator end  (std::size_t dim);
iterator begin(std::size_t dim, std::vector<std::size_t> const&);
iterator end  (std::size_t dim, std::vector<std::size_t> const&);
```

The first two functions define a range for a given dimension that must be smaller the order where the first element of the range always corresponds the first element of the corresponding container. The second argument of the last two functions correspond to a multi-index position that define the initial displacement within the multi-index space except the dimension that has been specified by dim. The same initialization of the multidimensional container along the second dimension can be stated in one line.

```
std::fill(A.begin(2),A.end(2),5.0);
```

The user does not have to keep track of the layout of the container A and can read or write elements with the help of ranges that can be conveniently instantiated with appropriate function calls. Moreover, the user can combine fibers of multidimensional containers tuples using template functions of the C++ standard library. For instance, the inner product of two fibers of two multidimensional arrays with the same length can be simply expressed with the std::inner_product template function.

```
std::inner_product(A.begin(3),A.end(3),B.begin(2),0.0);
```

Please note that the objects A and B can be instances of type tensor or fhg::tensor_view with different layout tuples. We also provide begin() and end() member functions for the fhg::tensor template class. The iterators have unit strides and define a range that covers the complete memory index set of the container class. They can be categorized as random-access iterators and behave just as iterators provided by std::vector. The user can initialize all elements an fhg::tensor object A with the help of the std::fill template function or the range-based for-loop and the range defined by A.begin() and A.end().

4.2 Multidimensional Iterator

The template class fhg::multi_iterator defines a multidimensional iterator that allows to define the multi-index set of a multidimensional array or view. It is not used as an iterator per se, but functions as a factory class that instantiates stride-based iterators such as the previously described fhg::iterator class.

```
template<class stride_iterator_t>
class multi_iterator;
```

The template parameter stride_iterator_t is an iterator type that must allow to instantiate stride-based iterators with a current position and a stride. The template class contains four private member variables:

- the current pointer type pointer,
- the order of the container of type std::size_t and
- two pointers to stride and extent tuples of type std::size_t*.

The multidimensional iterator provides a single constructor with which all member variables are initialized.

```
multi_iterator(pointer location,size_t rank,size_t* strides,size_t* extents);
```

The fhg::tensor and fhg::tensor_view template classes offer the member functions mbegin() and mend() to conveniently instantiate fhg::multi_iterator classes. Next to the default assignments, the template class fhg::multi_iterator overloads the assignment operator for stride-based iterators.

```
multi_iterator& operator=(stride_iterator_t const& it);
```

The operator assigns the current pointer with the pointer of the argument. Next to the constructor, assignment and comparison operator, the fhg::multi_iterator provides two factory methods that instantiate stride-based iterators.

```
stride_iterator_t begin(std::size_t r);
stride_iterator_t end  (std::size_t r);
```

Listing 1. Baseline algorithm for higher-order tensor functions using multidimensional iterators

The instantiated stride-based iterators are initialized with the dimension r and the current position of the calling fhg::multi_iterator. In this way, we can iterate over the complete multi-index sets of multidimensional arrays and views with non-hierarchical data layouts. The following example illustrates how a three-dimensional array A is iterated with the multidimensional and stride-based iterators fhg::multi_iterator and fhg::stride_iterator. Listing 1 can be regarded as the baseline implementation for first-order template functions. Let A be an object of type fhg::tensor or fhg::tensor_view of order three storing its elements according to for instance the first-order storage format. Let I be a multidimensional iterator that points to the first element of A. We can then use the template function base in Listing 1 to initialize all elements with the value v. The function generates a range with the stride-based iterator j2 and j2End for the outermost dimension. Each increment operation ++j2 adds its stride to its internal pointer until the end of the range is reached. Using the assignment operator of I, the internal pointer of the stride-based iterator j2 is copied to I with which the stride-based iterator j1 is initialized. The stride-based iterator j0 is initialized in the same manner. Finally, the elements of A are set to the value v. We can also replace the innermost loop with

```
std::fill(I.begin(0), I.end(0), v);
```

where std::fill is the template function provided by the C++ standard library.

5 TENSOR FUNCTION TEMPLATES

Our tensor algorithms are implemented with multidimensional iterators and support a combination of multidimensional arrays and views with different data layouts, arbitrary order and extents. All of following tensor algorithms are parametrized by multidimensional iterator types that need

- to provide two member functions begin() and end() in order to select and iterate over the corresponding index range and
- to have the same functionality as a standard iterator with at least the input iterator capabilities.

Additional requirements of a dimension specific iteration depends on the algorithm that may require different iterator capabilities. An example of such an iterator is the multidimensional iterator template class fhg::multi_iterator that has been introduced in the previous section. Users can specify their own iterator type and use our tensor template functions. Please note that the member functions of fhg::tensor and fhg::tensor_view call the following first- and if applicable higher-order functions. Both data structures, for instance, overload the relational and numeric operators. They provide a convenient interface that allows to express numerical algorithms close to the mathematical notation without specifying multidimensional iterators. The user may decide whether to call the higher-order tensor functions using member functions of tensor template class instances or to call them with a different multidimensional container or multidimensional iterator.

5.1 First-Level Tensor Operations

First-level tensor template functions such as fhg::for_each implement algorithms of the C++ standard library for dense multidimensional arrays and their views. Table 2 lists some of the implemented template functions of our tensor framework that make use of multidimensional iterators. Similar to the C++ standard library, the user can pass function objects to some of the first-level template functions that apply one or more function objects in each iteration. Functions objects are function-like objects that provide a function call operator allowing the function to have a state. The user can define its own class with a function call operator to instantiate a function object or construct a function object using lambda-expressions. Most of our first-level template functions require unary or binary predicates and unary or binary function objects just as it is the case for the C++ standard library.

In contrast to the C++ standard algorithms, our first-level tensor functions are designed for multidimensional arrays and views. They are implemented in a recursive fashion and iterate over multiple half-open ranges. The iteration is accomplished with a multidimensional iterator where one or more elements of multidimensional arrays and views with different data layouts but the same multi-index are combined. Please note that dense multidimensional arrays with a contiguous data layout can also be manipulated

Function	Example (MATLAB)	Description
for_each()	C = C+3	Performs a unary operation for each element
copy()	C = A	Copies elements starting with the first element
copy_if()	C(A>3) = A(A>3)	Copies elements that match a criterion
transform()	C = A+3	Modifies, copies elements accord. to a unary op.
transform()	C = A+B	Combines elements accord. to a binary op.
fill()	C = 3	Replaces each element with a given value
generate()	C(i) = i.*i	Replaces each element with the result of an op.
count()	sum(A(:)==3)	Returns the number of elements
count_if()	sum(A(:)<=2)	Returns the number of elements matching a crit.
min_element()	[,i] = min(A(:))	Returns the element with the smallest value
max_element()	[,i] = max(A(:))	Returns the element with the largest value
find()	find(C(:)==3)	Searches for the first element matching the value
find_if()	find(C(:)<=2)	Searches for the first element matching a crit.
equal()	all(C(:)==A(:)	Returns whether two ranges are equal
mismatch()	find(C(:)!=A(:)	Returns the first elements that differ
all_of()	all(C(:)==3)	Returns whether all elements match a crit.
any_of()	sum(C(:)==3)>0	Returns whether at least one element matches a crit.
none_of()	sum(C(:)==3)==0	Returns whether none of the elements matches a crit.
iota()	C(:) = [1:n]	Replaces each element with incremented values
accumulate()	sum(C(:))	Combines all element values (accord. to a binary op.)
inner_product()	dot(C(:),A(:))	Combines all elements (accord. to a binary op.)

Table 2. Overview of the implemented higher-order tensor functions where the algorithms can combine multidimensional arrays with different storage layouts. The symbol ⊙ denotes a binary operator for real numbers and implements e.g. a multiplication or addition. The transform, compare as well as the tensor-multiplication algorithms allow to transpose the multidimensional arrays according to the permutation tuples before applying the corresponding operation.

Listing 2. Template function `fhg::for_each` based on multidimensional iterators

---

with template functions of the C++ standard library. However, they are designed for containers with one dimension and therefore cannot manipulate arrays with different data layouts or with non-contiguously stored elements.

Please consider the `fhg::for_each` template function in Listing 2. The template function is recursively iterates over the complete memory index set of a multidimensional array using multidimensional iterators of type `InputIt` and applies the unary function of type `UnaryFn` to the elements that are referenced by the iterator. The implementation accepts any order greater zero with which `fhg::for_each` is called and therefore is the generalized version of the base function that has been provided in the previous section in Listing 1. The iteration of the memory index set is accomplished by recursively adding memory indices in each function call with a newly defined index range given by `in.begin(r)` and `in.end(r)`. More specifically, if p is the order of the multidimensional array, the function `for_each` needs to be initially called with `p-1`. In this case, the outermost loop iterates over the half-open range that is defined by `it.begin(p-1)` and `it.end(p-1)` covering the index range of the r-th dimension. The copy-constructed iterator f is then copied to the next function instance where $r = p - 2$. This is repeated until r equals zero where the first dimension of the data structure is accessed and the `std::for_each` template function of the C++ standard library is called. The multidimensional iterators f and l define a range and act like a standard iterator with input iterator capabilities. The maximum depth of the recursion is therefore `p-1` and the recursive function is called $n_2 \cdots n_p$ times where n_r is the r-th extent of the multidimensional array or view.

Incrementing the r-th stride-based iterator in the r-th loop corresponds to a shift of its internal pointer by the r-th stride w_r. The i-th iteration therefore sets the internal pointer of the r-th stride-based iterator to k with $k = k' + \delta \cdot i_r \cdot w_r$, where k' is the memory address computed by the previous stride-based iterator and δ is the number of bytes to store an element of A. Given zero-based indices i_r for all r, the address j of an element is given by

$$j = k_0 + \delta \cdot \sum_{r=1}^{p} i_r \cdot w_r = k_0 + \delta \cdot \lambda(\mathbf{w}, \mathbf{i}, \mathbf{0}), \tag{20}$$

```
```

where λ is the layout function already defined in Eq. (10). In case of a multidimensional array view, the r-th stride of the multidimensional iterator is given by $w_r \cdot t_r$ where t_r is step of the r-th range and w_r is the stride of the referenced multidimensional array as discussed in Section 3.3. Additionally, the index of the first element is initialized with $w_1 \cdot f_1 + \cdots + w_p \cdot f_p$ where f_r is the r-th lower bound of the view that has been used in Eq. (19).

Example 5.1. Let $\underline{A}$ be a tensor or view of order p with the shape tuple **n** and any non-hierarchical layout. The following equation denotes the initialization of $\underline{A}$ with a scalar.

$$\underline{A}(i_1, \ldots, i_p) = \alpha, \quad \text{for all } i_r \in I_r. \tag{21}$$

Given a tensor object A with elements of type float, the next listing implements Eq. (21) and initializes elements of an array A with the value alpha using the template function fhg::for_each.

fhg::for_each(A.mbegin(), A.mend(), [alpha](**float**& a){a=alpha;});

Note that the object A can be a view and may have any arbitrary non-hierarchical layout.

Please note the similarity of the function signature of fhg::for_each with the one of std::for_each. In case of fhg::for_each, the first argument must be a multidimensional iterator that is capable of creating multiple half-open index ranges. The second argument must be a unary function that is accepted, i.e. applicable by the std::for_each function. The first template parameter of the function indicates that the multidimensional iterator needs only to have input iterator traits. This might be different for other template functions such as fhg::sort postulating a random iterator.

Consider the fhg::transform template function in Listing 3. The implementation recursively applies the unary function fn to elements of a multidimensional array that is referenced by the multidimensional input iterator in. The results are stored in an array with the same order and extents using the multidimensional output iterator out. The template functions returns an iterator to the element past the last transformed element. The first iterator in points to and iterates over a multidimensional array or view with a data layout that may be different from the multidimensional array or view referenced by the second iterator. The control flow is similar to the one of the fhg::for_each function with the difference that each function call returns the current output pointer. The innermost loop calls the C++ standard library std::transform template function. The dereferenced input and output iterators correspond to elements with the same multi-index regardless of the data layout. All template functions listed in Table 2 are implemented in the same manner.

Example 5.2. Let $\underline{A}$ and $\underline{C}$ be multidimensional arrays of order p with the same shape tuple **n**. The multiplication of a scalar α with elements of a tensor $\underline{A}$ is then given by the equation

$$\underline{C}(i_1, \ldots, i_p) = \alpha \cdot \underline{A}(i_1, \ldots, i_p), \tag{22}$$

where $i_r \in I_r$ and $r = 1, \ldots, p$. Let A and C be objects of type fhg::tensor or fhg::tensor_view with the same shape tuple and let alpha be a scalar. Eq. (22) can then be implemented with help of the fhg::transform function.

fhg::transform(A.mbegin(), C.mbegin(), [alpha](**float const**& a){return a*alpha;});

Example 5.3. Let $\underline{A}$, $\underline{B}$ and $\underline{C}$ be multidimensional arrays with same shape tuples **n** and $\odot$ a binary operation that is either an addition, subtraction, multiplication or division. The elements of $\underline{C}$ are then given by the equation

$$\underline{C}(i_1, \ldots, i_p) = \underline{A}(i_1, \ldots, i_p) \odot \underline{B}(i_1, \ldots, i_p), \tag{23}$$

where $i_r \in I_r$ and $r = 1, \ldots, p$. Let A, B and C be objects of type fhg::tensor or fhg::tensor_view with the same shape tuple. Similar to the previous example, Eq. (23) can be implemented using the fhg::transform function.

Function	Notation	Example
transpose()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}}^{\tau}$	C = permute(A,[2,1,3])
tensor_times_vector()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}} \bar{\bullet}_m \mathbf{b}$	C = ttv(A,b,1)
tensor_times_matrix()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}} \bullet_m \mathbf{B}$	C = ttm(A,B,1)
tensor_times_tensor()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}} \bullet_q^{\varphi,\psi} \underline{\mathbf{B}}$	C = ttt(A,B,[1 3],[2 3])
outer_product()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}} \circ \underline{\mathbf{B}}$	C = ttt(A,B)
inner_product()	$\alpha \leftarrow \langle \underline{\mathbf{A}}, \underline{\mathbf{B}} \rangle$	C = ttt(A,B,[1:3])
norm()	$\alpha \leftarrow |\underline{\mathbf{A}}|_F$	C = norm(A)
tensor_times_vectors()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}} \bar{\bullet}_r \mathbf{v} \bar{\bullet}_q \mathbf{u}$	C = ttv(A,u,v,[1,3])
tensor_times_matrices()	$\underline{\mathbf{C}} \leftarrow \underline{\mathbf{A}} \bullet_r \mathbf{U} \bullet_q \mathbf{V}$	C = ttm(A,U,V,[1,3])

Table 3. Overview of the implemented higher-order tensor functions. The first column lists template function that implement the corresponding operations listed in the second column. The third column provides example code that can be run in Matlab using the toolbox presented in [Bader and Kolda 2006].

Listing 4. Template Function `ttv` using Multidimensional Iterators implementing Eq. (24)

```
fhg::transform(A.mbegin(), B.mbegin(), C.mbegin(), std::plus<>());
```

Please note that we did not specify the storage formats and index offsets of the tensor data types. User of the framework is free to implement different types operations with appropriate binary operators using their own template classes or multidimensional iterators. For convenience, the `fhg::tensor` and `fhg::tensor_view` template classes provide member functions that perform entrywise operations with overloaded operators for different argument types. The overloaded `operator+=()` member function of both template classes calls the `fhg::transform` template function if its arguments are of type `fhg::tensor` and `fhg::tensor_view`. If the argument is a scalar of type `const_reference` or `value_type`, the template function `fhg::for_each` is called, see Example 21.

5.2 Higher-Level Tensor Operations

Higher-level tensor operations exhibit a higher arithmetic intensity ratio than first-level tensor operations and perform one or more inner products over specified dimensions. Table 3 provides an overview of the implemented higher-order tensor operations. We will discuss the implementation of the three higher-order tensor operations, that are the tensor-times-vector and tensor-times matrix and tensor-times-tensor multiplication. Please note that the transposition is included in this subsection as it can be expressed with a copy operation and a modified stride tuple.

Let $\underline{\mathbf{A}}$ be an p-way array with the shape tuple $\mathbf{n}$ and let $\mathbf{b}$ be a 1-way array with the shape tuple (n_m). Let also $\underline{\mathbf{C}}$ be a $p-1$-way array with its shape tuple $\mathbf{n} = (n_1, \ldots, n_{m-1}, n_{m+1}, \ldots, n_p)$. The m-mode tensor-vector multiplication, denoted by $\underline{\mathbf{C}} = \underline{\mathbf{A}} \bar{\bullet}_m \mathbf{b}$, multiplies $\underline{\mathbf{A}}$ with $\mathbf{b}$ according to

$$\underline{\mathbf{C}}(i_1, \ldots, i_{m-1}, i_m, \ldots, i_p) = \sum_{i_m=0}^{n_m-1} \underline{\mathbf{A}}(i_1, \ldots, i_p) \cdot \mathbf{b}(i_m), \tag{24}$$

where $1 \leq m \leq p$ and $2 \leq p$. Please note that the tensor-vector product equals to a common vector-matrix multiplication if $m = 1$ and $p = 2$. For $p > 2$, the vector $\mathbf{b}$ is multiplied with each frontal slice of the array $\underline{\mathbf{A}}$. Listing 4 implements the tensor-times-vector multiplication according to Eq. (24) where the contracting dimension m must be greater than zero. Our framework provides also an implementation for this case. The second and third arguments, r and q, track the recursion depth where $q = r$ for $m < r < p$ and $q - 1 = r$ for $0 < r \leq m$ and are initially set to $r = p - 1$ and $q = p - 1$.

The `fhg::ttv` template function has a similar control flow compared to the previously discussed implementation of the `fhg::transform` template function. One difference is the recursive call in line 5 where the recursion depth r equals the mode m. After $p - 1$ recursive

```
Listing 5.  Most inner Recursion Level of the Template Function ttm
```

function calls with $r = 0$, the inner product of two fibers is performed in line 10 using the `std::inner_product` of the standard library. The range of the first fiber is given by the range that is defined by the iterators `A.begin(m)` and `A.end(m)`. The start of the second range is given by `B.begin(0)`. The result is stored into the location pointed by `fc`. Interpreting lines 8 to 11 as slice-vector multiplications, we might say that the tensor-vector multiplication is composed of multiple slice-times-vector multiplications.

Example 5.4. Let `A`, `b` and `C` are tensor objects of the same element type and layout and let the shape tuples of `A`, `b` and `C` be given by the tuples (3, 4, 2), (2, 1) and (3, 4), respectively. We can then use following statement that performs the 2-mode tensor-times-vector multiplication.

```
fhg::ttv(1, 2, 2, A.mbegin(), b.mbegin(), C.mbegin());
```

A second implementation with only two branches is given when $m = 0$. The first branch contains the expressions of the `else`-branch that are executed for $r > 1$. The second branch contains the computation of a loop over the second dimension (instead of the first) and computes the inner product of two fibers of the first dimension. We might therefore say that for $m = 0$, the tensor-times-vector multiplication consists of multiple vector-times-matrix multiplication where the matrix is the frontal slice of the first input array pointed by `A`.

Let $\underline{\mathbf{A}}$ be an p-way array with the shape tuple $\mathbf{n}$ and let $\mathbf{B}$ be a 2-way array with the shape tuple (n', n_m). Let also $\underline{\mathbf{A}}$ be an p-way array with the shape tuple $(n_1, \ldots, n_{m-1}, n', n_{m+1}, \ldots, n_p)$. The *m-mode tensor-matrix multiplication*, denoted by $\underline{\mathbf{C}} = \underline{\mathbf{A}} \bullet_m \mathbf{B}$, multiplies $\underline{\mathbf{A}}$ with $\mathbf{B}$ according to

$$\underline{\mathbf{C}}(i_1, \ldots, i_{m-1}, j, i_{m+1}, \ldots, i_p) = \sum_{i_m=0}^{n_m-1} \underline{\mathbf{A}}(i_1, \ldots, i_p) \cdot \mathbf{B}(j, i_m), \qquad (25)$$

with $1 \leq m \leq p$ and $2 \leq p$. The tensor-matrix multiplication corresponds to a tensor-vector product when $n' = 1$. If the order p of the arrays $\underline{\mathbf{A}}$ and $\underline{\mathbf{C}}$ equals 2, the tensor-matrix product corresponds to a common matrix-matrix multiplication.

The implementation of tensor-matrix multiplication is almost identical to the recursive template function `ttv` where only the control flow of the most inner recursion differs. Moreover, the tensor-times-matrix multiplication can be expressed in terms of multiple slice-matrix multiplications where one parameter suffices in order to generate ranges for the input and output iterators. Consider Listing 5 that illustrates the implementation of the innermost part of the template function `fhg::ttm`. It is the base case of the recursion and corresponds to a slice-times-matrix multiplication. Parameter `m` is the contraction dimension and `r` denotes the recursion depth that initially is set to $p - 1$. Please note that m needs to be greater than zero in this case. Our framework provides a second implementation for $m = 0$ with a control and data flow that is similar to the tensor-times-vector implementation.

The initial addresses of the input and output slices are generated in line 5 and 7 by calling `A.begin(0)` and `C.begin(0)`, respectively. The initial address of the input matrix is generated in line 6 by calling `B.begin(0)`. The instantiated stride-based iterators `fa0`, `fb0` and `fc0` iterate in lines 7 and 9 along the first dimension of the corresponding data structures and point to fibers and rows of the input and output slices, respectively. The range of the output fiber is determined in line 8 by the instantiation of the stride based iterators `fcm` and `lcm`. The value for each element of the output fiber is given by the inner product of the input fiber and input matrix row in lines 10 and 11. The range of the input fiber and matrix row are given by the ranges (`A.begin(m)`,`A.end(m)`) and (`B.begin(1)`, `B.end(1)`), respectively. Please note that iterator `B` needs to be initialized with the initial address in line 7.

The tensor-tensor product is the general form of the tensor-matrix and tensor-vector multiplication. Let $\underline{\mathbf{A}}$ be $(q + r)$-way array with $\mathbf{n}_a = (n_1^a, \ldots, n_{q+r}^a)$ and let $\underline{\mathbf{B}}$ be a $(q + s)$-way array with $\mathbf{n}_b = (n_1^b, \ldots, n_{q+s}^b)$ where $r, s, q \in \mathbb{N}_0$ and $r + q > 0, s + q > 0$. The (q, φ, ψ)-*mode tensor-tensor multiplication*, depicted by $\underline{\mathbf{A}} \bullet_q^{\varphi, \psi} \underline{\mathbf{B}}$, computes elements of the $(r + s)$-way array $\underline{\mathbf{C}}$ according to

$$\underline{\mathbf{C}}(i_1^c, \ldots, i_{r+s}^c) = \sum_{j_1=0}^{n_1'-1} \cdots \sum_{j_q=0}^{n_q'-1} \underline{\mathbf{A}}(i_1^a, \ldots, i_{q+r}^a) \cdot \underline{\mathbf{B}}(i_1^b, \ldots, i_{q+s}^b), \qquad (26)$$

Listing 6. Template Function `ttt` using multidimensional iterators implementing Eq. (26)

where the shape tuples satisfy the following equations:

$$
\begin{aligned}
n_k^c &= n_{\varphi_k}^a & \text{for } 1 \le k \le r, \\
n_{k+r}^c &= n_{\psi_k}^b & \text{for } 1 \le k \le s, \\
n_k' &= n_{\varphi_{k+r}}^a = n_{\psi_{k+s}}^b & \text{for } 1 \le k \le q.
\end{aligned}
\tag{27}
$$

The first r,s elements of the permutation tuples φ_a and φ_b, respectively, determine the dimension ordering for $\underline{C}$. The last q elements determine the dimensions that are contracted.

Equations (26) and (27) can be used to define the tensor-matrix and tensor-vector multiplication. Let $\mathbf{n}_a = (n_1^a, \ldots, n_p^a)$ and $\mathbf{n}_b = (n_m^a)$ be the shape tuples of the first and second operand. A tensor-tensor multiplication is the m-mode tensor-vector multiplication, if $q = 1$, $r = p - 1$, $s = 0$ and $\varphi = (1, \ldots, m-1, m+1, \ldots, p, m)$, $\psi = (1)$ such that $\mathbf{n}_c = (n_1^a, \ldots, n_{m-1}^a, n_{m+1}^a, \ldots, n_p^a)$. The m-mode tensor-matrix multiplication is given if $m = p$ and $n_2^b = n_p^a$, where $q = 1$, $r = p - 1$, $s = 1$ and $\varphi = (1, \ldots, p - 1, p)$, $\psi = (1, 2)$ such that $\mathbf{n}_c = (n_1^a, \ldots, n_{p-1}^a, n_1^b)$. The inner product is given when $r = 0$ and $s = 0$ for some $q > 0$, while the outer product is given when $q = 0$ for some $r > 0$ and $s > 0$.

Listing 6 illustrates the recursive implementation of the tensor-times-tensor multiplication as defined in Eq. (26). The implementation is recursive and performs the contraction without unfolding the tensors using multidimensional iterators only. Let us first consider the function signature of `fhg::ttt`. The first parameter k is the recursion depth of the template function. The parameter q is the number of dimensions of the input arrays that are contracted. The parameters r and s are the remaining number of dimensions of the input arrays that sum up to the order of the output array. The next two variables phi and psi are or point to permutation tuples that are needed to specify the dimension ordering of the output array and the contraction. The template function expects all integer variables to be zero-based. The control flow of the recursive function `fhg::ttt` contains four branches using `if-else` statements. In each branch multidimensional iterators are adjusted and stride-based iterators are instantiated and incremented with respect to the equations (26) and (27). The `else` statement performs the innermost loop of the tensor multiplication which computes the inner product of two fibers.

The recursive template function `fhg::ttt` is called initially with $k = 0$. For $k < r$, the control flow branches into the loop in line 7 that calls the function `fhg::ttt` with modified iterators A, C and with k+1. The iteration length of the k-th loop where k is smaller than r is determined by the stride-based iterators fa, la and fc which are instantiated with the multidimensional input and output iterators A and C, respectively. The iterator pair fa and la must be generated with the permutation tuple phi in order to satisfy Eq. (27). Similarly, for $r \le k < r + s$, stride-based iterators fb, lb and fc are instantiated with the multidimensional input and output iterators B and C, respectively. In this case, the permutation tuple psi is used, in order to generate the correct iterator pair for fb and lb with the indices psi[k-r]. For $k < r + s + q - 1$, stride-based iterators are instantiated with the iterators A, B and indices phi[k-s], psi[k-r] of the dimensions to be contracted. Finally, when the last branch is reached with $k = r + s + q - 1$, the inner product of two fibers of the input arrays are computed. The dimension indices for the instantiation of the stride-based iterators with the multidimensional iterators A and B are phi[r+q-1] and psi[s+q-1].

Example 5.5. Let A, B and C be tensor objects (such as `fhg::tensor` or `fhg::tensor_view`) of the same type and layout and let the shape tuples of the tensor objects A, B and C are {3,4,2}, {5,4,3,6} and {3,1,2}, respectively. We can then write the following statement in order to multiply A with B.

```
unsigned phi[] = {2,0,1}, psi[] = {0,3,1,2};
fhg::ttt(0,3,1, phi, psi, A.mbegin(), B.mbegin(), C.mbegin());
```

The first statement initializes C++ arrays that represents the permutation tuples phi and psi. The second statement calls the function `fhg::ttt` in Listing 6 and performs the 2-mode tensor-times-tensor multiplication.

Higher-order tensor operations are also provided as member functions of the fhg::tensor class template and may be combined with pointwise operations. The following code snippet shows two template methods of the fhg::tensor template class for the tensor-matrix and tensor-vector multiplication.

```
tensor times_vector(tensor const& rhs,std::size_t mode) const;
tensor times_matrix(tensor const& rhs,std::size_t mode) const;
```

The tensor instance in function times_matrix needs to have at most two dimensions. In function times_vector one the two extents of tensor must be one. The second parameter mode determines the dimension over which the contraction is performed, see equations (24) and (25). Please note that the template class fhg::tensor_view provides member functions with the same signature. As we will show with examples, the function signatures for higher-order tensor operations are similar to the function signatures that are provided in [Bader and Kolda 2006]. The user can therefore easily verify and compare the results of our tensor functions with those provided in [Bader and Kolda 2006].

Example 5.6. Consider the following listing, where A, b, B and C are tensor objects of the same element type and layout. Let the shape tuples of A, b, B and C be {3,4,2,6}, {3,1}, {4,5,6} and {2,5}, respectively. We can then express the multiplication of the tensor A with the vector b along the first dimension and the multiplication of the tensor B with the matrix C along the second dimension with the following statement.

```
auto D = A.times_vector(b,1) + B.times_matrix(C,2); //D = ttv(A,b,1)+ttm(A,B,2)
```

The resulting temporary array object D is of type fhg::tensor<float> and has the shape tuple fhg::shape {4,2,6}.

The first expression A.times_vector(b,1) internally calls the template function ttv where the mode is zero. The second expression B.times_matrix(C,2) calls the template function ttm which is shown in Listing 5. Please consider the following member function signature of times_tensor.

```
tensor times_tensor(tensor const& rhs,std::size_t q,
                                 std::vector<std::size_t>const& phi,
                                 std::vector<std::size_t>const& psi)const;
```

Function times_tensor encapsulates the template function fhg::ttt presented in Listing 6 and provides a user-friendly interface. The parameters phi and psi can have different sizes and determine the permutation of the dimension indices of the input and output arrays. The second parameter q determines the number of contraction as discussed for the template function fhg::ttt.

The possibility to permute the resulting dimension indices of the output array can be regarded as a minor extension of the equivalent functions that are provided in [Bader and Kolda 2006]. The tensor classes provide two member function that exhibit a simpler interface with the minor restriction of not being able to permute the dimension indices of the output array.

```
tensor times_tensor(tensor const& rhs,std::vector<std::size_t>const& phi)const;
tensor times_tensor(tensor const& rhs,std::vector<std::size_t>const& phi,
                                 std::vector<std::size_t>const& psi)const;
```

The first function allows to specify one permutation tuple which specifies the permutation for both tensors. In case of the second function, parameter phi and psi with equal size specify the permutation of the dimension indices of the left and right hand side tensors. In both cases, the permutation tuples specify the dimension indices to be contracted and can be smaller than the order of the tensors.

The following listing illustrates a multiplication of two tensors A and B with unequal shape and layout tuples where the multiplication is performed according to Eq. (26).

Listing 7. Higher-Order Power Method

Example 5.7. Consider the following listing, where A, B and C are tensor objects of the same type and layout. Let the shape tuples of A, B and C be {3,4,2,6}, {4,3,2} and {2,3,4}, respectively.

```
auto D = A.times_tensor(B, {2,3}); // C = ttt(A,B,[2:3])
auto E = A.times_tensor(C, {2,3}, {3,1}); // C = ttt(A,B,[2:3],[3,1])
```

The first statement multiplies A with B with the permutation tuple {2,3} which contracts the dimension 2 and 3 of the input tensors. The second statement performs a multiplication of the 2-nd, 3-rd and 3-rd, 1-st dimensions of the first and second input tensors, respectively.

The comments denote `Matlab` statements that contain user-defined functions of the toolbox discussed in [Bader and Kolda 2006]. In some cases, a tensor is multiplied with a sequence of vectors and matrices such as in case of the higher-order singular value decomposition [Bader and Kolda 2006; Lathauwer et al. 2000b]. The following listing exemplifies the function call where the tensor A from the previous example is multiplied with multiple vectors.

Example 5.8. Let A be the tensor from the previous example and let a, b and c be tensor tensor objects of type `fhg::tensor<float>` representing vectors. The multiplication of a list of vectors denoted by {a,b,c} with the dimensions given by {1,2,4} is then given by the following statements.

```
fhg::tensor<float> a{3,1}, b{4,1}, c{2,1};
fhg::tensor<float> d = A.times_vectors( {a,b,c}, {1,2,4});
//fhg::tensor<float> d = A.times_vectors( {a,b,c}, 3);
```

The commented statement performs the same operation where the last parameter denotes the excluded mode.

Function `times_vectors` internally calls the `times_vector` function starting with the vector c to the vector a. The implementation of the higher-order power method in Listing 7 uses the `times_vectors`.

The method is discussed in [Lathauwer et al. 2000b] and can be regarded as generalization of the best rank-one approximation for matrices [Bader and Kolda 2006]. The method estimates the best rank-one approximation of a real valued tensor $\underline{\mathbf{A}}$ of order p, by finding a rank-1 tensor $\underline{\mathbf{B}}$ composed of a scalar λ and unit-norm vectors $\mathbf{u}_1, \ldots, \mathbf{u}_p$ with which the least square cost function

$$f(\underline{\mathbf{B}}) = ||\underline{\mathbf{A}} - \underline{\mathbf{B}}||_2 \quad \text{with } \underline{\mathbf{B}} = \lambda \, \mathbf{u}_1 \circ \cdots \circ \mathbf{u}_p \tag{28}$$

is minimized. The first parameter K is maximum number of iterations of the outer loop. We could additionally insert a convergence criteria such as difference of the previous and current value of $lambda$ as stated in [Lathauwer et al. 2000b]. The second parameter is the tensor of type `fhg::tensor<T>` that shall be approximated. The third input parameter u are starting values of the unit-norm vectors. In [Lathauwer et al. 2000b] the starting values are set as the most dominant left singular vectors of the matrix unfolding of $\underline{\mathbf{A}}$. The last input parameter l denote scaling factors λ.

6 CONCLUSIONS

We have presented a flexible C++ tensor framework that allows users to conveniently implement tensor algorithms with C++ and to easily verify numerical results with the toolbox presented in [Bader and Kolda 2006]. The framework contains a multi-layered software stack and applies different types of abstractions at each layer.

Users can choose the high-level layers of the framework to instantiate tensor classes with arbitrary non-hierarchical data layout, order and dimension extents. Member functions of the tensor classes help to generate views and access multidimensional data supporting all types of non-hierarchical data layouts including the first- and last-order storage formats. The transparency of the data access functions with respect to the data layout can be identified as a unique feature of our framework.

The lower layers of the software stack provide tensor template functions that are implemented only in terms of multidimensional iterator types separating algorithms and data structures following the design principle of the Standard Template Library. Users of our framework are able to include their own tensor types and extend the functionality of our library without modification of the tensor template classes. We have presented our own multidimensional and stride-based iterator classes and exemplified their usage together with implementations of first- and higher-level tensor operations.

Member functions of the tensor classes encapsulate tensor template functions and allow to program tensor algorithms in a convenient fashion. We have exemplified their usage and implemented a method for the best rank-one approximation of real valued tensors that has been described in [Lathauwer et al. 2000b]. All of the tensor template functions have been implemented in a recursive fashion and execute in-place with no restriction of the contraction mode, order or dimension extents.

In future, we plan to provide and incorporate in-place tensor functions into our framework for a faster computation. We would like to offer or integrate parallelized and cache-efficient versions of the template functions for different data layouts and investigate their runtime behavior.

A SOFTWARE

A.1 Organization and Usage

TLib is a header-only library that only depends on the C++ standard library. The main folder of our framework has two subdirectories. The `tensor/code` directory contains all library code contents with class declarations and implementations. The `tensor/examples` directory contains all examples which demonstrate the instantiation and usage of the tensor class template and tensor functions. The declaration of the class templates and function templates are found in the `tensor/code/inc` folder where as the definitions are placed in the folder `tensor/code/src`. When working with our tensor library, the inclusion of the `code/inc/tensor.h` is sufficient. However, one could also use the tensor function templates without our tensor classes but a different multidimensional array. In such a case, only the corresponding header files can be included.

A.2 Examples

TLib is implemented in `C++` using features of the `C++14` standard. Users of our framework need to compile the examples in the `tensor/examples` with a `C++` compiler that supports the `C++14` standard. We have tested our framework using gcc version 6.3.0 and `clang` version 4.0.0 compilers. We have provided a `Makefile` that generates all executables in the `tensor/examples/build` directory. Examples 7 to 11 also generate `MATLAB` script files for verification that are placed in the `tensor/examples/out` directory. The script files can be directly executed in `MATLAB`. Examples 8 to 11 also require the `MATLAB` toolbox described in [Bader and Kolda 2006] with which the numerical results are validated.

ACKNOWLEDGMENTS

REFERENCES

Björn Andres, Ullrich Köthe, Thorben Kröger, and Fred A. Hamprecht. 2010. Runtime-Flexible Multi-dimensional Arrays and Views for C++98 and C++0x. *CoRR* abs/1008.2909 (2010).

Brett W. Bader and Tamara G. Kolda. 2006. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Trans. Math. Softw.* 32 (December 2006), 635–653. Issue 4.

Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. 1999. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures (SPAA '99)*. ACM, New York, NY, USA, 222–231.

A. Cichocki, R. Zdunek, Phan A.H., and S. Amari. 2009. *Nonnegative Matrix and Tensor Factorizations* (1 ed.). John Wiley & Sons.

Chun-Feng Cui, Yu-Hong Dai, and Jiawang Nie. 2014. All real eigenvalues of symmetric tensors. *SIAM J. Matrix Anal. Appl.* 35, 4 (2014), 1582–1601.

Jose Dias da Silva and Armando Machado. 2017. Multilinear Algebra. In *Handbook of Linear Algebra* (2 ed.), Leslie Hogben (Ed.). Chapman and Hall.

Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kågström. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM review* 46, 1 (2004), 3–45.

Nicolaas Klaas M Faber, Rasmus Bro, and Philip K Hopke. 2003. Recent developments in CANDECOMP/PARAFAC algorithms: a critical review. *Chemometrics and Intelligent Laboratory Systems* 65, 1 (2003), 119–137.

Derry FitzGerald, Matt Cranitch, and Eugene Coyle. 2005. Non-negative tensor factorisation for sound source separation. In *Proceedings of Irish Signals and Systems Conference*. Dublin Institute of Technology.

Ronald Garcia and Andrew Lumsdaine. 2005. MultiArray: a C++ library for generic programming with arrays. *Softw., Pract. Exper.* 35, 2 (2005), 159–188.

Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations* (4 ed.). JHU Press.

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: Linguistic Support for Generic Programming in C++. *ACM SIGPLAN Notices* 41, 10 (2006), 291–310.

Wolfgang Hackbusch. 2014. Numerical tensor calculus. *Acta numerica* 23 (2014), 651–742.

Adam P. Harrison and Dileepan Joseph. 2016. Numeric tensor framework: Exploiting and extending Einstein notation. *Journal of Computational Science* 16 (2016), 128 – 139.

Richard A. Harshman and Margaret E. Lundy. 1994. PARAFAC: Parallel factor analysis. *Computational Statistics & Data Analysis* 18, 1 (1994), 39 – 72.

Yong-Deok Kim and Seungjin Choi. 2007. Nonnegative tucker decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1–8.

Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.

Tamara G. Kolda and Jimeng Sun. 2008. Scalable Tensor Decompositions for Multi-aspect Data Mining. In *Proceedings of the 8th IEEE International Conference on Data Mining*. IEEE, Washington, DC, USA, 363–372.

Walter Landry. 2003. Implementing a High Performance Tensor Library. *Sci. Program.* 11, 4 (Dec. 2003), 273–290.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000a. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl* 21 (2000), 1253–1278.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000b. On the Best Rank-1 and Rank-(R1,R2,. . .,RN) Approximation of Higher-Order Tensors. *SIAM J. Matrix Anal. Appl.* 21, 4 (2000), 1324–1342.

Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. 2015. An Input-adaptive and In-place Approach to Dense Tensor-times-matrix Multiply. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 76, 12 pages.

Lek-Heng Lim. 2017. Tensors and Hypermatrices. In *Handbook of Linear Algebra* (2 ed.), Leslie Hogben (Ed.). Chapman and Hall.

M. Mernik, J. Heering, and A. M. Sloane. 2005. When and how to develop domain-specific languages. *Comput. Surveys* 37, 4 (2005), 316–344.

Edoardo Di Napoli, Diego Fabregat-Traver, Gregorio Quintana-Ortí, and Paolo Bientinesi. 2014. Towards an efficient use of the BLAS library for multilinear tensor contractions. *Appl. Math. Comput.* 235 (2014), 454 – 468.

Michael Ng, Liqun Qi, and Guanglu Zhou. 2009. Finding the largest eigenvalue of a nonnegative tensor. *SIAM J. Matrix Anal. Appl.* 31, 3 (2009), 1090–1099.

Steffen Rendle, Leandro Balby Marinho, Alexandros Nanopoulos, and Lars Schmidt-Thieme. 2009. Learning Optimal Ranking with Tensor Factorization for Tag Recommendation. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 727–736.

John VW Reynders III and Julian C Cummings. 1998. The POOMA framework. *Computers in Physics* 12, 5 (1998), 453–459.

Berkant Savas and Lars Eldén. 2007. Handwritten digit classification using higher order singular value decomposition. *Pattern recognition* 40, 3 (2007), 993–1003.

Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 813–824.

Paul Springer and Paolo Bientinesi. 2016. Design of a high-performance GEMM-like Tensor-Tensor Multiplication. *CoRR* abs/1607.00145 (2016). http://arxiv.org/abs/1607.00145

Alexander Stepanov. 1995. The Standard Template Library. *Byte* 20, 10 (1995), 177–178.

Bjarne Stroustrup. 2012a. Foundations of C++. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP (Lecture Notes in Computer Science)*, Vol. 7211. Springer, 1–25.

Bjarne Stroustrup. 2012b. Software Development for Infrastructure. *Computer* 45, 1 (2012), 47–58.

Bjarne Stroustrup. 2013. *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.

S. K. Suter, M. Makhynia, and R. Pajarola. 2013. TAMRESH - Tensor Approximation Multiresolution Hierarchy for Interactive Volume Visualization. In *Proceedings of the 15th Eurographics Conference on Visualization (EuroVis '13)*. Eurographics Association, 151–160.

Ledyard Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

M.A.O. Vasilescu and D. Terzopoulos. 2002. Multilinear image analysis for facial recognition. In *Proceedings of the 16th International Conference on Pattern Recognition*, Vol. 2. 511–514.

Todd L. Veldhuizen. 1998. Arrays in Blitz++.. In *ISCOPE (Lecture Notes in Computer Science)*, Denis Caromel, R. R. Oldehoeft, and Marydell Tholburn (Eds.), Vol. 1505. Springer, 223–230.