New Implementation of High-Level Correlated Methods Using a General Block Tensor Library for High-Performance Electronic Structure Calculations

Evgeny Epifanovsky,[a,b] Michael Wormit,[c,d] Tomasz Kuś,[a] Arie Landau,[a] Dmitry Zuev,[a] Kirill Khistyaev,[a] Prashant Manohar,[a,e] Ilya Kaliman,[a,f] Andreas Dreuw,[d] and Anna I. Krylov*[a]

This article presents an open-source object-oriented C++ library of classes and routines to perform tensor algebra. The primary purpose of the library is to enable post-Hartree–Fock electronic structure methods; however, the code is general enough to be applicable in other areas of physical and computational sciences. The library supports tensors of arbitrary order (dimensionality), size, and symmetry. Implemented data structures and algorithms operate on large tensors by splitting them into smaller blocks, storing them both in core memory and in files on disk, and applying divide-and-conquer-type parallel algorithms to perform tensor algebra. The library offers a set of general tensor symmetry algorithms and a full implementation of tensor symmetries typically found in electronic structure theory: permutational, spin, and molecular point group symmetry. The Q-Chem electronic structure software uses this library to drive coupled-cluster, equation-of-motion, and algebraic-diagrammatic construction methods. © 2013 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23377

Introduction

Many-body quantum theory is used to compute the energy and properties of multiparticle systems. Interactions within such systems are described by the Hamiltonian, a many-body operator that can be represented by a tensor in an appropriate tensor space. In electronic structure theory,[1] the electronic Hamiltonian in the Schrödinger equation contains one- and two-electron interactions described by the kinetic energy and Coulomb operators. Wave function-based methods treat these operators explicitly by representing them as multidimensional tensors. Thus, the programmable expressions for solving the Schrödinger equation are comprised mostly of tensor contractions (generalized matrix multiplications), which dominate the computational effort. For example, the correlation energy in configuration interaction (CI), coupled-cluster (CC), or second-order Møller–Plesset perturbation theory (MP2) methods may be given by $E_{\text{corr}} = \sum_{ia} t_i^a f_{ia} + \frac{1}{4}\sum_{ijab} t_{ij}^{ab} \langle ij||ab\rangle$, where t_i^a and t_{ij}^{ab} contain the amplitudes (expansion coefficients) of the single- and double-excited determinants in a many-electron wave function expansion, or their effective counterparts. f_{ia} are the elements of the Fock matrix and $\langle ij||ab\rangle$ are the antisymmetrized electron repulsion integrals in the molecular orbital basis. $\langle ij||ab\rangle$ and t_{ij}^{ab} are four-dimensional tensors with antisymmetric properties: $t_{ij}^{ab} = -t_{ji}^{ab} = t_{ji}^{ba} = -t_{ij}^{ba}$.

The primary challenge of implementing many-body methods arises from the complexity of the programmable expressions, which consist of numerous tensor products. For example, the CCSD (coupled-cluster with single and double substitutions) amplitude equations involve more than 30 contractions between the amplitudes (T_1 and T_2) and the integrals. All of them have to be programmed correctly and efficiently.

The requirements for a practical tensor library suitable for electronic structure calculations, as well as possible algorithmic solutions, were first outlined by Windus and Pople[2]: arbitrary tensor order and types of contractions should be possible, tensors should be stored on disk and processed by parts, and symmetries (permutational, point-group, and spin) should be

[a] E. Epifanovsky, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. I. Krylov
Department of Chemistry, University of Southern California, Los Angeles, California 90089
E-mail: krylov@usc.edu

[b] E. Epifanovsky
Department of Chemistry, University of California, Berkeley, California 94720

[c] M. Wormit
Centre of Theoretical Chemistry and Physics, Massey University, Auckland, New Zealand

[d] M. Wormit, A. Dreuw
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen Ruprecht-Karls Universität Heidelberg, Heidelberg 69120, Germany

[e] P. Manohar
Department of Chemistry, Birla Institute of Technology and Science, Pilani, Rajasthan 333031, India

[f] I. Kaliman
Department of Chemistry, Purdue University, West Lafayette, Indiana 47907

Contract/grant sponsor: National Science Foundation (CHE-0951634 and OCI-1216644, A. I. K); Department of Energy through the DE-FG02-05ER15685 grant and through Discovery through Advanced Computing (SciDAC) program, AIK. Humboldt Research Foundation (Bessel Award) (to AIK); Contract/grant sponsor: Humboldt Research Foundation (Feodor-Lynen program) (to MW); Contract/grant sponsor: DST, India (to PM)

© 2013 Wiley Periodicals, Inc.

taken into account. The first generation of a CC and equation-of-motion (EOM) program called ccman,[3–21] which is a part of Q-Chem,[22] has enabled numerous computational studies. The code is based on a C++ BlockTensor library (A. I. Krylov, et al., Efficient C^{++} tensor library for coupled-cluster calculations, unpublished.) developed in 1997–1998 following Windus and Pople's ideas. Similar libraries were developed by the NWCHEM[23] (including optimization of tensor expressions[24]), ACES III[25], and PSI4[26] teams.

The original BlockTensor library allows general types of pair-wise contractions between tensors of arbitrary dimensionality (it has been used for coding expressions involving up to six-dimensional tensors[12,13]) and takes full advantage of permutational (e.g., $f_{ij} = f_{ji}$ or $t_{ij}^{ab}=-t_{ji}^{ab}=-t_{ij}^{ba}=t_{ji}^{ba}$) and point group symmetry (Abelian subgroups). Spin symmetry (e.g., $f_{\alpha\alpha}=f_{\beta\beta}$) was not fully implemented. The BlockTensor library also includes a variety of other tensor operations, such as additions, scalar products, applying denominators (i.e., $\frac{\langle ij||ab\rangle}{\Delta_{ijab}}$), scattering lower-order tensors into higher-order ones (i.e., $\Gamma_{ijij} = \gamma_{ij}$). Support for frozen, restricted, and active orbital spaces is also available.[4,5,12] The library stores the tensors on disk and processes them by parts. It is parallelized using OpenMP technology; however, due to the limitations of the algorithms, the resulting scalability is moderate (up to four cores).

Advances in computer architectures and hardware, as well as software requirements for new electronic structure methods, have revealed deficiencies in the old BlockTensor library prompting an update of the library design. This article presents a new efficient C++ general-purpose tensor algebra library (libtensor). The new library features a straightforward programming interface, full tensor symmetry (point group including non-Abelian subgroups, permutational, and spin), flexible memory management via a separate virtual memory component, and shared-memory parallel algorithms. The library is designed to provide multiple points of extension making it possible to add support for new tensor structures and symmetries, algorithms, and computer architectures. Q-Chem 4 [27] features a new generation of CI, CC, and EOM-CC methods in the ccman2 program (to distinguish it from an older suite of codes called ccman), as well as algebraic-diagrammatic construction (ADC) methods available through adcman. Both are implemented using the new tensor library. The source code of libtensor is available for free download [28], its use is permitted under the terms of a Boost-like software license (no limitations on use, further modification, or redistribution).

The goal of this article is to provide an overview of the underlying algorithms and explain the library design and code structure. We also give an example of using the library to implement a many-body electronic structure code and guidelines on interfacing the library with other electronic structure platforms. Finally, we present the results of performance benchmarks and outline future developments.

Data Structures and Algorithms

This section presents the block tensor structure used in the library, the handling of tensor symmetry, and the algorithms of essential tensor algebra operations.

Overview

From a programmer's point of view, tensors are multidimensional arrays. Using examples relevant to electronic structure theory, the Fock matrix is a two-dimensional symmetric tensor, whereas coupled-cluster T_2 amplitudes t_{ij}^{ab} and electron repulsion integrals $\langle pq||rs\rangle$ are four-dimensional anti-symmetric tensors.

Vectorizing a tensor by specifying a certain order to the entries yields a trivial way of storing it as a one-dimensional array in computer memory. For example, in a two-index $N \times M$ tensor with row-major ordering, the linear position of (i, j) th element is $i \times M + j$. Likewise, $T_2 = t_{ijab}$ can be stored as a linear array t_I, where the linear position $I=b+aN_{\mathrm{vir}}+jN_{\mathrm{vir}}^2+iN_{\mathrm{vir}}^2N_{\mathrm{occ}}$. Note that this representation allows one to reinterpret the same linear array as different matrices: T_2 can be viewed as a $N_{\mathrm{occ}}\times N_{\mathrm{occ}}N_{\mathrm{virt}}N_{\mathrm{virt}}$, $N_{\mathrm{occ}}N_{\mathrm{occ}}\times N_{\mathrm{virt}}N_{\mathrm{virt}}$, or $N_{\mathrm{occ}}N_{\mathrm{occ}}N_{\mathrm{virt}}\times N_{\mathrm{virt}}$ matrix.* As a consequence, with the right order of tensor indices, it is possible to cast tensor contractions as matrix multiplies and, therefore, apply efficient matrix multiplication kernels, such as GEMM provided by BLAS,[29] without converting tensor data to another format.

Correlated methods in electronic structure theory, such as CC, CI, EOM, and ADC, are best formulated in terms of linear tensor algebra, where Fock, Coulomb, cluster, and excitation operators are represented as tensors in the basis of molecular orbitals. In the typical variants of CC and EOM theories, the CCSD and EOM-CCSD methods, the size of the dataset grows as the fourth power of the number of basis set functions. Thus, in many interesting applications it is impossible to fit the problem in a computer's RAM entirely. This problem can be addressed by partitioning the tensor into smaller pieces, swapping them between disk and RAM, and using divide-and-conquer-type algorithms to perform tensor algebra,[2] as illustrated in Figure 1. This approach generalized to tensors of arbitrary dimensions is the essence of libtensor's design.

Tensors: Definitions and connection to many-body formalisms

Let $\mathcal{T}$ be a tensor of order N: $\mathcal{T} \in \mathcal{V}^{(1)} \otimes \mathcal{V}^{(2)} \otimes \cdots \otimes \mathcal{V}^{(N)}$, where $\mathcal{V}^{(k)}$ are vector spaces. After choosing basis sets $\{v_i^{(k)}\}$ spanning each mode of the tensor space, $\mathcal{T}$ can be represented as a multidimensional table of scalars T with entries $t_{i_1 i_2 \cdots i_N}=\langle \mathcal{T}, v_{i_1}^{(1)} \otimes v_{i_2}^{(2)} \otimes \cdots \otimes v_{i_N}^{(N)}\rangle$. The dimensionality of this array is equal to the order of the tensor.

In this article, we use the term "tensor" for the multidimensional array that represents a tensor. Expressions "tensor dimensionality", "n-index tensor", and "order-n tensor" refer to the order of the tensor.

Many-body operators, wave function expansions, and other objects in electronic structure theory can be represented as tensors in some basis, typically a spin-orbit or spinless

*In t_{ijab} and other examples, indices i, j span occupied and a, b span virtual molecular orbital subspaces: $i,j \in 1\ldots N_{\mathrm{occ}}$, $a,b \in 1\ldots N_{\mathrm{virt}}$, N_{occ} and N_{vir} denote the number of occupied and virtual orbitals, respectively.

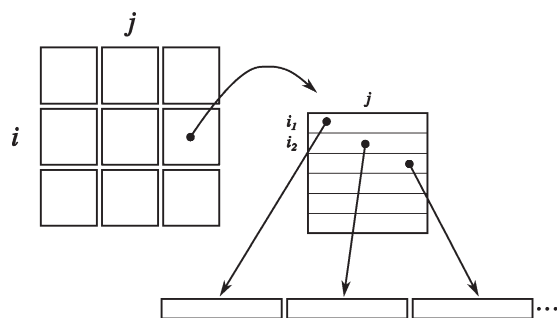


Figure 1. An example of block-tensor (or tiling) representation of a two-index tensor (e.g., Fock matrix). The tensor is represented as a tensor of small tensors (blocks). The data in each tensor is stored as a one-dimensional array. The blocking structure facilitates implementation of permutational and point-group symmetries (e.g., only unique nonzero *ij* blocks are stored and computed), parallelization (operations on different blocks can be performed by different processors), as well as multiple orbital spaces. Contractions and other operations on the actual data (i.e., individual blocks stored as linear arrays) can be performed by the GEMM subroutine.

molecular orbital basis, atomic orbital basis, or their combination. Permutational symmetry is only allowed between modes that correspond to the same vector space, and in general all modes of a tensor can correspond to different vector spaces. For example, the Fock matrix is a two-dimensional symmetric tensor in the atomic or full molecular orbital basis set, the coupled-cluster t_{ijab} amplitudes and antisymmetrized electron repulsion integrals are four-index partially antisymmetric tensors in the spin-orbit basis (here partially means symmetry only with respect to certain, not all, index permutations, that is, in t_{ijab} only i, j and a, b are related by permutational antisymmetry).

Dense tensors

The most basic data structure for a tensor is obtained through vectorization by specifying a certain order to the tensor entries. Any ordering can be used as long as it provides a unique bidirectional mapping between the multidimensional tensor indices and the linear indices. The tensor library uses the row-major order† extended to more than two dimensions as follows. Starting from a d-dimensional tensor $a_{i_1 i_2 \cdots i_d}$, vectorization is done for the first mode by slicing the tensor. The result is a vector that contains N_1 elements, each being $(d-1)$-dimensional tensor:

$$a_{i_1 i_2 \cdots i_d} = \left(a^{(1)}_{i_2 \cdots i_d} \quad a^{(2)}_{i_2 \cdots i_d} \quad \cdots \quad a^{(N_1)}_{i_2 \cdots i_d} \right)$$

†The row-major order follows the C/C++ convention, whereas Fortran employs the column-major order. A practical upshot is that when a linear array representing an $N \times M$ matrix in C/C++ is passed to a Fortran routine, it will be interpreted as a transposed $M \times N$ matrix. Therefore, to perform matrix multiplication $C = A \times B$ using GEMM, one needs to make a GEMM call requesting $C^T = B^T \times A^T$ and passing the pointers to the original (non-transposed) C, B, and A.

Then each vector entry is expanded along their own first index (which is the second index in the original tensor) so the resulting vector contains $(d-2)$-dimensional tensors as its elements. The length of this vector is $N_1 N_2$. The process continues until a vector of scalars is obtained. An $n \times m$ matrix (second-order tensor) a_{ij} is thus vectorized into $\bar{a_l} : a_{ij} = \bar{a_l} : l = im + j$.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \rightarrow \left(a_{11}\, a_{12} \cdots a_{1m}\ a_{21}\, a_{22} \cdots a_{2m} \ \cdots\ a_{n1}\, a_{n2} \cdots a_{nm} \right)$$

Similarly, tensor t_{ijab} containing coupled-cluster (or any two-body) amplitudes can be stored as a one-dimensional array $\bar{t_l}$, where the linear position l of each entry is computed as $l = b + aN_{vir} + jN_{vir}^2 + iN_{vir}^2 N_{occ}$.

The data structure that represents a tensor in this format is called the dense tensor because it stores all the entries without attempting to take advantage of symmetry or any other special properties that result in sparsity. The dense tensor consists of an array of data and an object that describes the shape of the tensor (the number of elements along each mode). Because this structure is so simple, most tensor algebra algorithms are trivial. Only the contraction algorithm requires special attention.

The most basic algorithm to contract two dense tensors can be implemented as a series of nested loops that run over all inner and outer indices as shown in the top panel of Figure 2. This straightforward algorithm is not cache efficient and as a result its performance is bound by memory bandwidth on modern computer architectures. The problem can be alleviated by finding and replacing innermost loops by a

```
for i = 1, ni
 | for j = 1, nj
 | | for k = 1, nk
 | | | for l = 1, nl
 | | | | for p = 1, np
 | | | | | for q = 1, nq
 | | | | | | c[i,j,k,l] +=
 | | | | | |    a[i,p,k,q] * b[j,p,l,q]

for i = 1, ni
 | for j = 1, nj
 | | for p = 1, np
 | | | C(i,j) += A(i,p) * B(j,p)^T
```

Figure 2. Simple algorithms for the contraction of two dense tensors $c_{ijkl} = \sum_{pq} a_{ipkq} b_{jplq}$. The top panel shows a naive nested-loop algorithm with four outer loops running over the outer indices i, j, k, l, and two inner loops running over p, q. In the bottom panel, the same effect is achieved by three nested loops over i, j, p; loops over k, l, and q are replaced with a single kernel that multiplies $k \times q$ and $q \times l$ matrices. A(i,p) extracts a $k \times q$ submatrix for given i and p. B(j,p) extracts a $l \times q$ submatrix, C(i,j) extracts a $k \times l$ submatrix.

high-performance matrix multiplication kernel acting on the slices of tensors as shown in the bottom panel of Figure 2. If the GEMM routine is used, matrix multiplications can be done in place avoiding the overhead of actually making slices $A(i, p)$,$B(j, p)$, and $C(i, j)$, which is achieved by passing pointers to input arrays with appropriate offsets and strides.

The performance of the contraction algorithm that uses matrix multiplications internally is critically dependent on whether the kernel can achieve peak efficiency, which means that the multiplied matrices have to be fairly large. Both the original BlockTensor library and the library described in this article have adopted an approach, in which the tensors are permuted prior to the multiplication so that all contraction indices are grouped together in order for the contraction to be cast as a matrix multiplication. Consider a contraction of a_{ipkq} and b_{jplq} over p and q:

$$c_{ijkl}=\sum_{pq} a_{ipkq} b_{jplq}$$

First the arguments are permuted to yield

$$\bar{a}_{ikpq}=a_{ipkq} \qquad \bar{b}_{jlpq}=b_{jplq}$$

Then the indices i and k can be combined into one index $I \in 1 \ldots N_i N_k$, and similarly indices j and l are combined into J, p and q become P. Going to the combined indices amounts to simply reshaping the data arrays without affecting the data itself. The contraction is thus written as a matrix multiplication

$$\bar{c}_{IJ}=\sum_{P} \bar{a}_{IP} \bar{b}_{JP} \qquad \bar{C}=\bar{A}\bar{B}^{\dagger}$$

To recover the result, matrix $\bar{c}_{IJ}$ is permuted back into tensor c_{ijkl}

$$c_{ijkl}=\bar{c}_{ikjl}$$

This permute-multiply approach is general and can be applied to any tensor contraction. The permutation step is the primary overhead, but it is possible to avoid it when the tensors can be readily reshaped as suitable matrices. For example, none of these contractions require permutation of the data:

$$\sum_{ab} t_{ij}^{ab} \langle kl||ab\rangle \quad \sum_{ij} t_{ij}^{ab} \langle ij||cd\rangle \quad \sum_{ija} t_{ij}^{ab} \langle ij||ad\rangle \quad \sum_{jab} t_{ij}^{ab} \langle kj||ab\rangle.$$

They can be performed by passing to GEMM the original nonpermuted linear arrays and specifying proper dimensions of the combined indices.

By operating on larger matrices, the permute-multiply algorithm reduces the multiplication runtime through better CPU utilization, but that is offset by having to permute the tensors so that they are properly matricized. The scaling of the permutation operation is more favorable than that of matrix multiplications: for $N \times N$ matrices, the permutation takes $O(N^2)$ memory operations, whereas matrix multiplication requires $O(N^3)$ floating point operations. Therefore, for sufficiently large matrices, the multiplication step always dominates. In addition, the number of actual data permutations can be kept to a minimum. For example, it is possible to build in a selector that will automatically choose an algorithm that has better expected performance based on the input tensors and type of contraction. Currently, in a typical CC calculation matrix multiplications account for 70–90% of CPU time with permutations taking an additional 5–10%.

Other tensor algebra operations, such as additions, element-wise products, generalized inner products, etc. usually require less performance tuning because of their low overall share of the total computational cost. As a rule, simple nested loop-based algorithms suffice. However, following performance profiling, certain optimizations can be done for these operations as well. The library combines vector and matrix algebra BLAS kernels to perform some additions and multiplications. The strategy of using BLAS here is different than in the case of contractions; because permuting tensor elements is unjustified, BLAS routines are applied directly to tensor data multiple times in a loop to achieve a cumulative effect, similar to the contraction method presented in Figure 2. As an example, consider the addition of two three-dimensional tensors: $c_{ijk}=a_{ijk}+b_{jik}$. It can be done by looping over indices i and j and using BLAS vector addition for index k, which is faster than preparing a permuted b_{ijk} or running three nested loops.

Block tensors

In order to be practical in larger calculations, a tensor format needs to be well suited for divide-and-conquer algorithms. The simple dense tensor is a poor fit in this respect. The lack of any built-in ability to handle symmetry or sparsity further reduces its applicability. A general strategy to address these problems is to break down large tensors into smaller and much lighter pieces. The details of how that is done distinguish different approaches.

One class of partitioning recipes is to split a large tensor into an array of reduced-dimensionality tensors, such as fibers (vectors) or slices (matrices), as illustrated in Figure 3. An auxiliary array is used to map the index of an entry to the appropriate slice where the entry can be located. The biggest shortcoming of this partitioning method is that it introduces an imbalance between two types of tensor modes: those that make up lower-dimensional objects and those that are used for indexing. This makes handling symmetry in tensors particularly difficult.

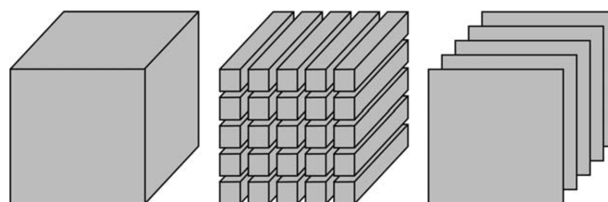


Figure 3. Partitioning of a cubic tensor (left) into reduced-dimensionality objects: fibers (center) or slices (right).

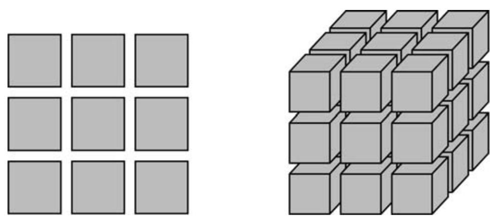


Figure 4. Two-dimensional (left) and three-dimensional (right) block tensors formed by partitioning tensors evenly along each mode. The blocks do not have to have equal sides as long as blocking is consistent. For example, the tiling patterns of the occupied and virtual orbital spaces may be naturally different, but they should be kept consistent among all tensors.

Another way of partitioning a tensor, particularly well suited for distributed storage, is the cyclic layout as implemented, for example, in Cyclops Tensor Framework (CTF).[30] The cyclic distribution defines a period m along each mode of the tensor, and each fragment is assigned a phase $p < m$. The fragment then contains entries p, $m + p$, $2m + p$ and so on. With appropriate padding, all the fragments have the same size and the permutational symmetry of the original tensor. This approach, while valuable in the cases of high-order permutational symmetry, by itself does not take advantage of tensor sparsity that arises from spin and point group symmetries. However, because the fragment tensors inherit the symmetry and sparsity of the full tensor, combining the cyclic layout with the blocked structure described below allows one to leverage the strength of both partitioning schemes.

This section presents the block tensor, an approach that partitions the modes of a tensor consistently‡ so that the subtensors ("blocks") have the same dimensionality as the original tensor. The approach, first introduced by Windus and Pople[2] and then used in BlockTensor library and SIAL,[25] is thus a multidimensional generalization of matrix tiling as illustrated in Figure 4.

Such implementation of block tensors enables a block sparse structure, in which only nonzero dense blocks are stored in memory. Zero blocks are not stored explicitly (marked as zero in auxiliary data structures) and are ignored appropriately when performing tensor algebra operations, reducing both storage requirements and computational cost.

Tensor symmetry is supported by the block tensor structure through relationships between blocks. For example, in a real symmetric matrix (two-index tensor) A the following holds: $a_{ij} = a_{ji}$; the blocks will be equal up to a permutation of entries: $A_{ij}=A_{ji}^T$, where A_{ij} is a submatrix of A. Section on symmetry below discusses this in more detail.

Structure of block tensors. Block tensor A combines a tensor space T, a set of symmetry relationships S, and a set of nonzero canonical blocks B that contain actual data (tensor entries): $A(T, S, B)$. The tensor space specifies the number of elements along each dimension of the tensor and the positions at which the tensor is to be split into blocks (subtensors).

Symmetry S is a set of mappings $i \mapsto \{j, U_{ij}\}$ such that for subtensors A_i and A_j the equality $A_i = U_{ij}(A_j)$ holds, where i and j are multi-indices and U_{ij} is a transformation that consists of a permutation and an elementwise operation on the tensor elements. The elementwise operation is usually scalar multiplication (often, a sign change), but may also involve complex conjugation in the case of tensors with complex entries.

B is a mapping $i \mapsto A_i$, where i is a multi-index and A_i is a subtensor of A. In practice, it is not necessary to store all A_i, but keeping only nonzero symmetry-unique blocks is sufficient. For example, consider a real square matrix $M = m_{ij}$ that consists of two blocks along each dimension so that M_{ij} are submatrices of M:

$$M=\begin{pmatrix} [M_{11}] & [M_{12}] \\ [M_{21}] & [M_{22}] \end{pmatrix} \tag{1}$$

If M is symmetric ($m_{ij} = m_{ji}$) and blocking is done equally along both dimensions, submatrix M_{21} can be obtained from $M_{12} : M_{21}=M_{12}^T$; the submatrices on the diagonal are themselves symmetric square matrices. In order to restore the entire matrix M, it is then sufficient to store blocks M_{11}, M_{12}, and M_{22}. This list can be shortened further if some of the blocks only contain zero entries.

The library does not impose any limitations on the type of tensors that are used as blocks in a block tensor. Typically, block tensors with dense blocks are used, but sparse, symmetric, and other kinds of blocks with intrinsic structure are possible. All block tensor structures and algorithms are implemented in the library in a templated form with an assumption that all blocks have the same type (i.e., no mixing of dense and sparse blocks). However, this limitation could be trivially removed by introducing a special type of tensor that wraps multiple other block types. The following example of a general block tensor contraction algorithm demonstrates how block tensor operations can be expressed in terms of individual blocks without knowing their type.

Contraction of block tensors. Consider the contraction of two block tensors A and B to form some block tensor C. Assume that all the related dimensions, inner and outer, agree on their blocking (there is no loss of generality because if some dimensions are split in any two block tensors differently, they can be brought into agreement by re-tiling using the union of split points). Let block tensors A, B, and C consist of smaller tensors A_{ik}, B_{kj}, and C_{ij}, respectively. Indexes i and j are used to collectively designate all outer indices in A and B; k designates all inner tensor indices. It can be shown that the contraction can be written as the sum of pairwise block contractions:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Symmetry and sparsity in block tensors A and B have two major implications. Noncanonical blocks (replicas) need to be

‡Consistent partitioning means that the modes representing the same vector spaces are tiled using the same spacings; however, the spacings themselves are often different in order to leverage spin and point group symmetries when the orbitals of the same spin and irreducible representation are blocked together.

obtained from canonical blocks by applying a necessary transformation. Terms that contain zero blocks can be excluded from the calculation. The next section discusses symmetry in block tensors in more detail.

The calculation of C starts by computing its symmetry group from the symmetries of A and B. Then, nonzero canonical blocks C_{ij} are calculated in parallel using the following procedure. For each value of k, canonical blocks $A_{i'k'}$ and $B_{k'j'}$ that correspond to A_{ik} and B_{kj}, respectively, are found together with their transformations U^a_{ik} and U^b_{kj}, so that $A_{ik}=U^a_{ik}A_{i'k'}$ and $B_{kj}=U^b_{kj}B_{k'j'}$. If any of $A_{i'k'}$ or $B_{k'j'}$ is found to be exactly zero, the contraction term with the current value of k is dropped. Otherwise, it is added to a list of contractions, which contains these quintuples:

$$L_{ij}=\{(k, A_{i'k'}, B_{k'j'}, U^a_{ik}, U^b_{kj})\}$$

List L_{ij} is then processed to combine any duplicating contractions and remove canceling terms, resulting in a shorter list of quintuples L'_{ij}. This step ensures that only the required minimum of block contractions is performed minimizing the number of floating point operations. L'_{ij} is used to assemble the block C_{ij}. If $L'_{ij}=\emptyset$, then C_{ij} is not calculated and marked as a zero block in C.

For illustration, consider the contraction $\sum_{cd} t^{cd}_{ij}\langle ab||cd\rangle$ from CC equations. There is antisymmetry in both T_2 amplitudes and the integrals with respect to the permutation of indices in pairs (i, j), (a, b), and (c, d) (for simplicity let us disregard that $\langle ab||cd\rangle=\langle cd||ab\rangle$). The contraction algorithm ensures that in this case only canonical blocks ($i <= j$) and ($a <= b$) are computed, and the summation is only done over ($c <= d$), resulting in the optimal number of block contractions. The resulting number of floating point operations is, however, slightly larger than the required minimum. It is due to extra work done for diagonal blocks, which have intrinsic symmetry that is not taken into account. In practice, this has not been a problem because the amount of this extra work is at least one order of magnitude less than the total work, and its relative contribution becomes less and less important as the size of the problem grows larger.

Symmetry

The intrinsic symmetry of a block tensor A is represented via relations or mappings between tensor blocks. These mappings partition the full set of tensor blocks of A into disjoint subsets, so that any pair of blocks for which a mapping exists belong to the same subset. For each subset only one block, called the canonical block, has to be stored, while all other blocks can be constructed from the canonical block and respective mappings. For example, in the case of T_2 amplitudes, it is sufficient to only store blocks T^{ab}_{ij} for which $i \le j$ and $a \le b$, all other blocks can be recovered through the permutational symmetry of T_2:

$$T^{ab}_{ij}=-T^{ab}_{ji}=-T^{ba}_{ij}=T^{ba}_{ji}$$

Consider again the symmetric matrix M [eq. (1)]. There is one mapping $M_{12} \to M_{21}$ due to permutational symmetry. The four blocks of M can be separated into three subsets:

$$s_1=\{M_{11}\} \quad s_2=\{M_{12}, M_{21}\} \quad s_3=\{M_{22}\}$$

Together, the subsets form the set of all blocks in M: $s=\cup_i s_i$, and have no common blocks among them. It is enough to store only one block per each subset to be able to restore the whole matrix M.

Symmetry mappings between blocks are defined by a collection of abstract symmetry elements which is assigned to a block tensor during initialization. Every symmetry element establishes a certain class of mappings. Currently, symmetry elements for three different symmetry types have been implemented: permutational symmetry, spin symmetry, and point group symmetry (Fig. 5).

Permutational symmetry. The symmetry element for permutational symmetry defines mappings by means of a permutation $\mathcal{P}$ and an element-wise transformation $\mathcal{S}$ (usually multiplication by ± 1). As shown in Figure 5 (left), for every block B_i, permutation $\mathcal{P}$ creates a mapping to another block B_j with $j=\mathcal{P}i$ and $B_j=\mathcal{SP}B_i$ (i and j being multi-indices). As a similar mapping may also exist from B_j to B_k with $k=\mathcal{P}j$, B_k can be constructed from B_i by combining the two mappings so that $B_k=\mathcal{S}^2\mathcal{P}^2B_i$. If the step is repeated several times, the block B_i will eventually map onto itself after n steps, that is, $B_i=\mathcal{S}^nB_i$. This results in a sequence of blocks than can be constructed in its entirety from one single member of the sequence. However, for the blocks in the sequence to be nonzero, the elementwise transformation has to fulfill the condition $\mathcal{S}^n=1$, which is trivially satisfied in the case of permutational symmetry.

Spin symmetry. The symmetry element for spin symmetry defines mappings between tensor blocks in terms of mappings between superblocks. A superblock is a set of tensor blocks obtained by partitioning the block tensor evenly along each dimension similar to the way a tensor is partitioned into tensor blocks. Figure 5 (middle) shows a two-dimensional block tensor partitioned into $\alpha\alpha$, $\alpha\beta$, and $\beta\beta$ superblocks. This creates a second layer of block structure upon which the mappings are defined by means of two multi-indices I and J of superblocks and an elementwise transformation $\mathcal{S}$. As a result, every tensor block in superblock I is mapped onto the equivalent tensor block in superblock J such that $A_{I,i}=\mathcal{S}A_{J,i}$ holds for all multi-indices i in the superblocks I and J. This already demonstrates that all superblocks defined by the symmetry element have to have the same block structure. In addition to the mappings between two superblocks, one superblock can also be mapped to zero, thereby defining all tensor blocks in the superblock as zero, such as $\alpha\beta$ blocks in Fock matrix or non-$M_s = 0$ blocks (e.g., $\alpha\beta\beta\beta$) in T_2. In nonrelativistic calculations, the superblock structure is such that every dimension is split into two parts, one for each spin value.

For such commonly used two-index tensors as Fock matrix, the present scheme ensures that only the $\alpha\alpha$ block is stored and computed, which is exactly the same as in fully spin-adapted implementations. Thus, the size of the stored data equals approximately $(N_{\text{occ}}+N_{\text{virt}})^2/2$, where N_{occ} and N_{virt} denote the number of occupied and virtual molecular orbitals,

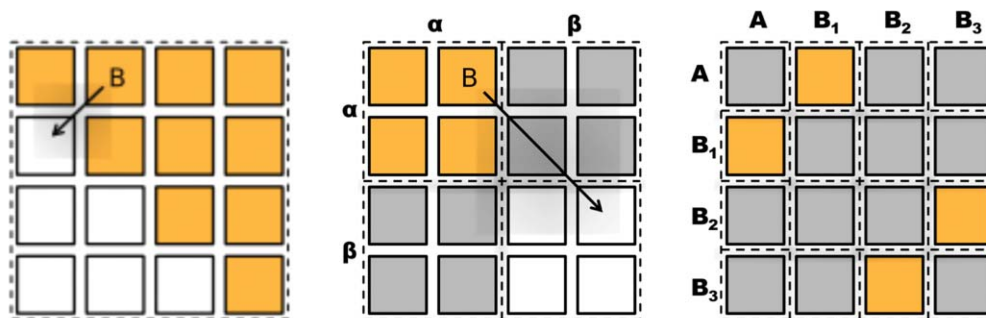


Figure 5. Illustration of the three symmetry element types available as part of the tensor library: permutational symmetry (left), spin symmetry (middle), point group symmetry (right). For a block tensor of rank 2, the block structure is shown indicating the simplifications due to each symmetry element type. Orange blocks are canonical blocks which are stored, white blocks can be constructed from the canonical blocks, and grey blocks are marked as zero by the symmetry element.

respectively, to be compared with $2\times(N_{\mathrm{occ}}+N_{\mathrm{virt}})^2/2=(N_{\mathrm{occ}}+N_{\mathrm{virt}})^2$ in spin-unrestricted calculations. However, in the case of four-index tensors such as T_2, this symmetry element creates less savings in storage and number of operations than a fully spin-adapted code. In the spin-unrestricted calculations, the size of T_2 is $\frac{6}{4}N_{\mathrm{occ}}^2N_{\mathrm{virt}}^2$ ($\alpha\alpha\alpha\alpha$, $\beta\beta\beta\beta$, and $\alpha\beta\alpha\beta$ spin blocks). In the spin-restricted case, we store $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ blocks, totaling in $\frac{3}{4}N_{\mathrm{occ}}^2N_{\mathrm{virt}}^2$. In a fully spin-adapted code, one needs to store only the $\alpha\beta\alpha\beta$ block, which is $\frac{1}{2}N_{\mathrm{occ}}^2N_{\mathrm{virt}}^2$ before anti-symmetrization. Thus, in the present implementation the size of the stored T_2 amplitudes is $1\frac{1}{2}$ times larger than should be possible in a fully spin-integrated code.

Molecular point group symmetry. The third symmetry element type has been implemented to mark tensor blocks that are zero due to molecular point group symmetry. It assigns sets of labels to every dimension of a block tensor so that the *i*th block along a given dimension is associated with the *i*th label of the dimension (see Figure 5, right panel). Thus, every block of an *N*-dimensional block tensor is associated with a set of *N* labels. In addition, the symmetry element contains an evaluation rule and a product table for labels. The evaluation rule defines a general rule how the *N* labels associated with a block determine whether the block is zero or nonzero. Usually, this entails forming a product of labels as specified by the product table and checking that the result belongs to a given set of labels. This allows one to target any irreducible representation (or a number of them) of the point group symmetry.

This way every block can be marked as zero or nonzero by evaluating the *N* associated labels using the evaluation rule and the product table. In order to relate the symmetry element to point group symmetry, a few assumptions have to be made concerning the structure of the block tensor. First of all, the block tensor has to describe a physical system that is invariant under the symmetry operations of a given point group (in other words, the tensor and the underlying system must agree on the point group symmetry). Second, the individual indices of a tensor element have to represent basis functions that can be associated with an irreducible representation of the point group. If the tensor indices along each dimension are ordered such that within each block the associated irreducible representations are identical, the block labels of the symmetry element can be directly identified with irreducible representations of the point group. Accordingly, the product table is the table of direct products of the irreducible representations.

Symmetry operations. In addition to the symmetry element types themselves, symmetry operations have been implemented for each symmetry element type in order to automatically determine the symmetry of a block tensor that results from a tensor operation. As the result symmetry has to be established for every possible tensor operation, in principle, one symmetry operation would have to be implemented for every symmetry element type and every tensor operation available. To avoid the resulting large number of possibilities, six basic symmetry operations have been identified from which more complex symmetry operations can be constructed: (1) permutation operation to change the order of tensor indices in the symmetry element types, (2) direct product, (3) direct sum operation to combine two symmetry elements of the same type, (4) fusion operation to merge two or more tensor dimensions into one, (5) trace operation to remove tensor dimensions by summing over them, and (6) symmetrization operation to introduce further permutational symmetry in a tensor. This set should be sufficient to construct the result symmetries of most, if not all, possible tensor operations. For example, the result symmetry of an addition of two tensors can be obtained by combining the direct sum and merge operations. Likewise, the symmetry of the contraction of two tensors is computed by building the symmetry of the direct product and tracing out the contracted indices.

Besides already available symmetry element types and respective symmetry operations, the tensor library also provides necessary interfaces to add custom symmetry element types without modifying existing structures. Naturally, a set of basic symmetry operations has to be implemented for every new symmetry element type.

Library Design and Code Structure

The primary purpose of the tensor algebra library is to provide a set of general computational routines with a programming interface that facilitates the creation of portable electronic structure codes. The library serves to reduce the effort

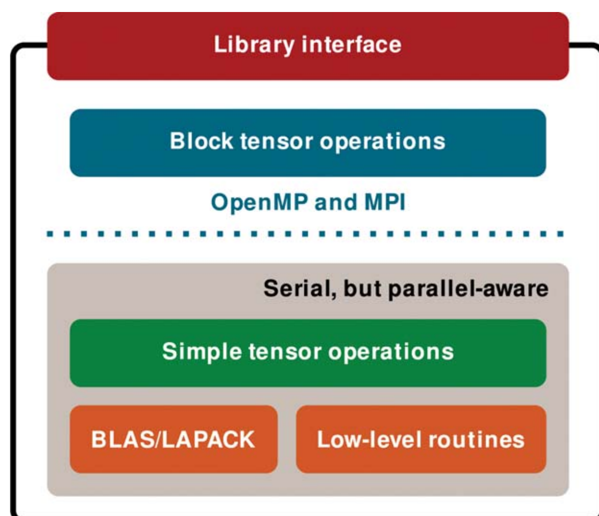


Figure 6. Overview of the libtensor library structure. A multilayer design allows for various extensions in terms of new algorithms and data types as well as new hardware architectures. The layers interact through well-defined interfaces; any layer can be substituted by an alternative implementation without the need to modify the code in the layers above or below.

```
//  Occupied space with 8 orbitals
bispace<1> o(8);
//  Split space at 2nd, 4th and 6th entry
//  to have two orbitals per block
o.split(2).split(4).split(6);

//  Virtual space with 12 orbitals
bispace<1> v(12);
//  Two and four orbitals per block
v.split(2).split(6).split(8);

//  Two-dimensional MO spaces
bispace<2> oo(o&o), ov(o|v), vv(v&v);

//  Fock matrix tensors
//  (symmetry is not yet specified)
btensor<2> f_oo(oo), f_ov(ov), f_vv(vv);
```

Figure 7. C++ code snippet that demonstrates the initialization of tensor objects. "&" indicates that the two spaces are of the same kind and may be related by permutational symmetry, whereas "|" indicates that the spaces are of different types.

required for prototyping, supporting the code, and migrating from one computational platform to another.

The library is organized in layers (Fig. 6). The top level provides a user-friendly C++ interface to program complex tensor contraction expressions, which are then translated into a sequence of block tensor operations that need to be invoked to compute the final result. Block tensor operations compute the structure and symmetry of the result, which is followed by a parallel computation of canonical data blocks. At a lower level, serial but thread-safe dense tensor operations perform work on individual blocks of a block tensor using a high-performance linear algebra back-end.

Library interface and examples of higher-level codes

The interface of the library is a set of C++ function and operator templates that allow the user to program tensor expressions and specify calculation intermediates. These functions do not perform computations themselves, but rather process the equations, prepare input and invoke lower-level block tensor operations, in the spirit of domain-specific programming languages. The interface allows one to write higher-level codes that are easy to read and maintain.

Before any tensor algebra can be done, initial tensors need to be prepared and filled with input data. Creating block tensors takes one parameter: tensor space that specifies the dimensions and blocking of the block tensor. Code fragment in Figure 7 shows the steps of creation of tensors for the Fock matrix. Tensor objects are initially empty and need to be populated with symmetry and data, which can be rather involved due to their complexity. There are programming tools in place for this process, but they are outside the scope of this article, and their description can be found in the programming manual (the code documentation can be generated by using doxygen software). Because tensor algorithms propagate symmetry, only input tensors need to be fully initialized. All explicit intermediates and results have to be created in proper tensor spaces,[§] but their symmetry and sparsity need not be manually resolved.

In an electronic structure calculation, one installs the symmetry and data into the Fock matrix, two-electron integrals, guess EOM amplitudes, etc. Occupied and virtual molecular orbital spaces are rearranged so that orbitals in any block belong to the same spin and irreducible representation. Intermediate tensors and placeholders for results are created empty. Their symmetry and sparsity is automatically determined by the library based on the input tensors and equations. For example, the symmetry properties and sparsity of T_2 can be determined from the amplitude update equation shown in Figure 8.

By the design of the interface, programmed expressions resemble the actual mathematical equations to be evaluated. Each tensor is assigned a number of letter indices that are used to identify the dimensions of the tensor for the purpose of multiplication, accumulation, etc. When programming, these indices are objects of a class called letter. Letters can be concatenated to form multidimensional tensor labels using the bitwise or operator (|). The length of a tensor label must be equal to the order of that tensor. The labels are used by the library to set up permutations of tensor elements for the underlying algorithms.

Some frequently used general tensor operations are shown in Table 1. For example, contraction of two tensors $c(i|j|k|l)=contract(m,a(i|j|m),b(m|l|k));$ represents the following operation: $C_{ijkl}=\sum_m a_{ijm}b_{mlk}$. This list is based on applying the library to program electronic structure methods; however, the interface can be expanded by introducing new templates both inside and outside the library.

[§]Although the tensor spaces of the computed tensors are determined by the equations used to compute them and the spaces of the input tensors, we choose to initialize their spaces explicitly to prevent programming errors.

```
void ccd_t2_update(...) {

  letter i, j, k, l, a, b, c, d;
  btensor<2> f1_oo(oo), f1_vv(vv);
  btensor<4> ii_oooo(oooo), ii_ovov(ovov);

  //  Compute intermediates
  f1_oo(i|j) =
      f_oo(i|j)
    + 0.5 * contract(k|a|b, i_oovv(j|k|a|b), t2(i|k|a|b));
  f1_vv(b|c) =
      f_vv(b|c)
    - 0.5 * contract(k|l|d, i_oovv(k|l|c|d), t2(k|l|b|d));
  ii_oooo(i|j|k|l) =
      i_oooo(i|j|k|l)
    + 0.5 * contract(a|b, i_oovv(k|l|a|b), t2(i|j|a|b));
  ii_ovov(i|a|j|b) =
      i_ovov(i|a|j|b)
    - 0.5 * contract(k|c, i_oovv(i|k|b|c), t2(k|j|c|a));

  //  Compute updated T2
  t2new(i|j|a|b) = div(
      i_oovv(i|j|a|b)
    + asymm(a, b, contract(c, t2(i|j|a|c), f1_vv(b|c)))
    - asymm(i, j, contract(k, t2(i|k|a|b), f1_oo(j|k)))
    + 0.5 * contract(k|l, ii_oooo(i|j|k|l), t2(k|l|a|b))
    + 0.5 * contract(c|d, i_vvvv(a|b|c|d), t2(i|j|c|d))
    - asymm(a, b, asymm(i, j,
        contract(k|c, ii_ovov(k|b|j|c), t2(i|k|a|c)))),

      d_oovv(i|j|a|b));
}
```

Figure 8. C++ code snippet that demonstrates the use of the library's interface to program equations that iteratively solve for CCD amplitudes.

The features of the interface are best demonstrated using examples. A code snippet in Figure 9 shows the calculation of MP2 electronic correlation energy, which is given by

$$E_{MP2}=\sum_{ia} t_i^a f_{ia}+\frac{1}{4}\sum_{ijab} t_{ij}^{ab}\langle ij||ab\rangle,\ \text{where}\ t_i^a=\frac{f_{ia}}{\varepsilon_i-\varepsilon_a}\ \text{and}\ t_{ij}^{ab}=\frac{\langle ij||ab\rangle}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b} \tag{2}$$

Figure 8 is an example the code used to update T_2 amplitudes in the CCD method, the core of a procedure that iteratively solves for the amplitudes. In this example, it is assumed for simplicity that the denominator $\Delta_{ijab}=\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b$ is formed prior to invoking T_2 update, but in a practical implementation Δ_{ijab} is formed and applied to T_2 on the fly to avoid the storage of the extra four-index intermediate.

Table 1. Examples of frequently used tensor operations available in the library. Both individual tensors and subexpressions can be used as arguments to these functions.

Operation	Example
Multiplication by a scalar	`c(i|j|k) = 2.0 * a(i|j|k);`
Addition of two tensors	`c(i|j|k) = a(i|j|k) + b(k|j|i);`
Dot (inner) product of two tensors	`c = dot_product(a(i|j|k), b(k|j|i));`
Direct (outer) product of two tensors	`c(i|j|k|l) = a(i|k) * b(j|l);`
Contraction of two tensors	`c(i|j|k|l) = contract(m, a(i|j|m), b(m|l|k));`
Elementwise product of two tensors	`c(i|j|k|l) = ewmult(k|l, a(i|k|l), b(j|k|l));`
Elementwise division of two tensors	`c(i|j|k|l) = div(a(i|j|k|l), b(i|j|k|l));`
Direct summation of two tensors	`c(i|j|k|l) = dirsum(a(i|k), b(j|l));`
General diagonal of a tensor	`c(i|j) = diag(j|k, j, a(i|j|k));`
Symmetrization	`c(i|j) = symm(i, j, a(i|j));`
Antisymmetrization	`c(i|j) = asymm(i, j, a(i|j));`

```
double mp2_energy(
  btensor<2> &t1, btensor<4> &t2,
  btensor<2> &f_ov, btensor<4> &i_oovv) {

  letter i, j, a, b;

  return dot_product(t1(i|a), f_ov(i|a)) +
    0.25 * dot_product(t2(i|j|a|b), i_oovv(i|j|a|b));
}
```

Figure 9. C++ code snippet that demonstrates the use of the library's interface to program the MP2 energy equation.

The expression templates are designed so that most common programming mistakes are detected at the time of compilation rather than runtime thus improving the robustness of the code.

Functions listed in Table 1 do not perform computations themselves, but rather help the user to build tensor expressions. The evaluation of the expressions is triggered by an overloaded C++ assignment operator. At that stage, the expressions are translated into a series of block tensor operations. Because the conversion takes place at runtime, it is possible for the library to automatically tune its performance and take full advantage of the available computational resources: memory and disk availability, type and number of processors, etc.

The level of abstraction offered by the expression templates allows us to introduce changes in the low-level library code without modifying the expressions. Once coded using this interface, all methods immediately take advantage of improvements in the core library. Examples of such improvements are new tensor representations (e.g. decomposed or sparse tensors), migration to new computational platforms, introduction of new types of parallelism.

Block tensor classes

The layer of block tensor structures and algorithms (Fig. 10) provides a capability to operate on large tensors using a divide-and-conquer approach.

At the level of block tensor operations, the primary task is to produce the structure of the result given the block structure of the operation arguments. This involves the calculation of the block dimensions, symmetry, and sparsity. Once the structure of the output is known, the computation of nonzero canonical blocks is delegated to the level of simple dense tensors.

The block tensor class template implements a container that allows one to access and modify the symmetry and data. The order of the tensor and type of tensor elements (single or double precision, complex, etc.) are template parameters. The space of a block tensor, which specifies the dimensions and blocking pattern, is set upon the creation of the tensor and cannot be changed later on.

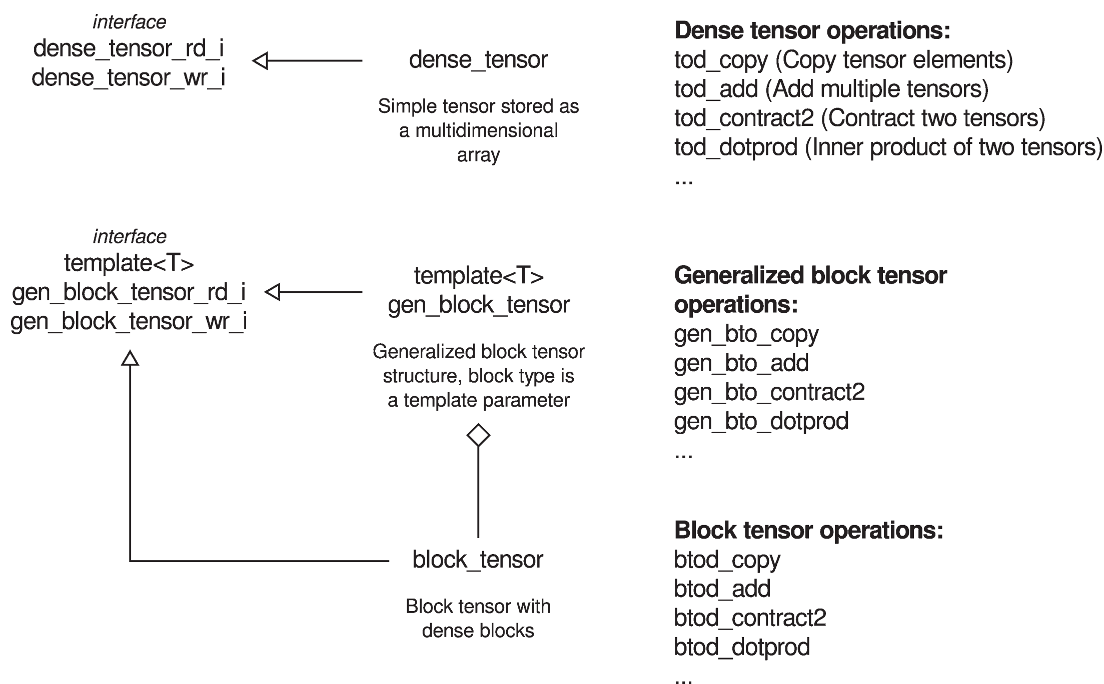


Figure 10. Diagram of classes in libtensor. For each type of tensors there are specialized tensor operations. General block tensors and operations on them are generic implementations. Concrete block tensors use the generic structures and algorithms within, but provide a simplified interface. The template argument in gen_block_tensor allows using different types of tensors, for example, real or complex dense tensors, sparse tensors, or tensors with special properties.

Block tensor operations are separate template classes, which implement algebra algorithms. This approach, as opposed to having tensor operations as the methods of the block tensor class, allows one to expand the functionality without modifying the block tensor class. It makes it possible to add general operations to the library, and keep more specific operations outside.

Dense tensor classes

The dense tensor represents a simple structure containing the dimensions of the tensor and a pointer to the data array. It is implemented in the library as a template with two parameters: order of the tensor and element type. Upon creation, dense tensors are initialized with the dimensions. At the same time, an array is allocated in memory to store the data. Dense tensors provide two interfaces: read-only (in which case multiple simultaneous readers are allowed in a parallel calculation) and read–write interface (only one writer is allowed at a time).

Dense tensor algorithms are implemented as separate classes that interact with the tensor objects through one of the two interfaces. As with block tensor operations, most commonly used dense tensor algebra operations are provided by the library, and there is a capability to add new dense tensor operations outside of the library.

Virtual memory mechanism

To allow applications of the tensor library in high-level electronic structure methods, it is necessary for it to be able to handle tensors that do not fit entirely in the core computer memory (RAM). In this case, external storage is utilized along with a mechanism to swap data in and out of core memory. Usually, computer operating systems allow a program to use more memory than physically available RAM by extending to virtual memory, a combination of hardware and low-level software components used to swap memory pages on the fly. However, because blocks of data required to compute tensor algebra and their order are predetermined, it is absolutely possible (and likely more efficient) to build a pipeline in which the next batch of blocks is loaded from disk in the background while the current batch is used for a computation.

The tensor library relies on an external virtual memory tool that provides virtual memory functionality via a predefined interface. There are no limitations on how the virtual memory library operates internally as long as it implements the interface.

Figure 11 shows the state diagram of a virtual memory block and functions that the tensor library uses to operate on memory blocks. Similar to the standard dynamic memory allocation/deallocation pairs of routines (malloc/free in C, new/delete in C++), the virtual memory defines allocate and deallocate functions, which use and return virtual pointers. Unlike the usual pointers, these cannot be directly dereferenced, but rather serve as handles for data blocks. When the user needs to use the data, the virtual memory block is first mapped onto the physical memory space (“locked”). Once the job is done, the user indicates that by “unlocking” the memory. While a block is unlocked, the memory manager is allowed to move it to a different location, including an external disk, to reuse physical memory. As a consequence, each time virtual blocks are locked, they may appear with

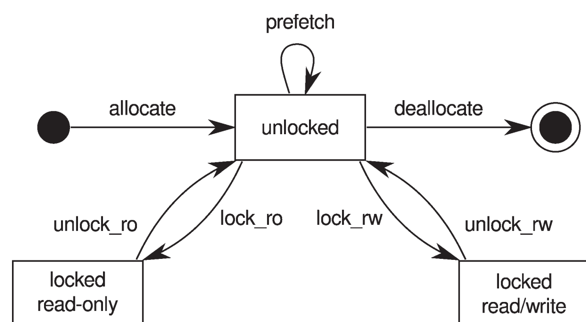


Figure 11. State diagram of a virtual memory block. The tensor library is able to use any compatible virtual memory implementation.

different physical memory addresses. There are two modes of locking virtual memory blocks: read-only mode in which the user is only allowed to read the data, but not modify it, and read–write mode, which allows both. The lock routine is blocking, that is, it may not return immediately, but it guarantees to return a proper physical pointer or an error. Finally, the user can indicate that a memory block will soon be required by calling a nonblocking prefetch routine.

This virtual memory protocol is used throughout the tensor library; however, the virtual memory code is not freely available at the moment. Instead, the distribution includes a simple implementation based on C++ new/delete operators. Following the formal description of the virtual memory interface and this example, one can develop their own virtual memory manager and use it together with the tensor library.

Parallel capabilities

The model of parallelism employed in the tensor library assumes multiple processors working in a shared-memory environment. It is implemented through a thread pool in which a limited number of threads are active and the others are either suspended or waiting for input or output. This allows performing computations and I/O simultaneously.

Block tensor operations produce a stream of tasks that is consumed by the thread pool with dynamic scheduling. Because the time required to complete each task is relatively long, this approach does not cause high contention on the task queue and scales well.

At the moment, work is ongoing to develop the capability to delegate computations to external accelerators such as general purpose GPUs. The strategy to enable the support for accelerators is to maintain a separate memory manager and a task queue for each external device. Block tensor algorithms need to be optimized for communication in order to minimize data transfer between main memory and device memory.

Quality control

The primary quality control instrument used in the tensor library is an extensive collection of unit tests. Each unit test is a small subroutine that checks that the correct result is produced in a particular case scenario. Overall, there are a few hundred unit tests ranging from small low-level class tests to larger cases such as the computation of a complex tensor expression.

Simple tensor operations are tested against reference data generated using clear and straightforward (albeit inefficient) procedures. Block tensor algorithms are checked against the tested simple dense tensor results.

Higher-level modules that use the tensor library for computations also employ unit tests. At that level, the primary concern is the validity of the equations. Each expression, including all intermediates, is computed using multiple data sets extracted from real calculations. The results are compared with independently precomputed reference data.

Unit tests provide a facility to verify correctness during initial code development through use cases, as well as create an environment to replicate and trace defects and prevent their reappearance by converting problematic scenarios into new test cases.

Structure of High-Level Codes

Figure 12 shows the components that are used in CC, EOM, and ADC codes in Q-Chem. Each component performs its specific task:

- Top-level drivers are located in adcman (ADC methods) and ccman2 (CC, CI, and EOM). These functions are responsible for setting up and coordinating solvers.
- Ccman2 uses a libcc library that contains all programmable expressions used in CC and EOM calculations.
- Libsolve contains generic solvers (DIIS and Davidson procedures).
- Libctx provides utilities to access the context of a calculation, which is a simple key-value map for data objects. Integrals, amplitudes, and other similar tensors, as well as energies, are saved in the context.
- Libmo contains routines that help set up tensor spaces and symmetry in libtensor format using information imported from Q-Chem.
- Liblegacy is a bridge connecting Q-Chem with the CC/EOM and ADC codes. It contains routines to import data from Q-Chem, for example, the integrals. It also provides a way to export data back to Q-Chem, for example, the density matrices used in property and analytic gradient calculations. To interface ccman2 with another quantum chemistry code, this part should be replaced by a package-specific counterpart.
- Libvmm is a library that provides a set of routines to work with virtual memory.

Use Case: Coupled-Cluster Doubles (CCD) Method

In this section, we consider the implementation of the CCD method in Q-Chem to illustrate the use of the tensor library and its collaboration with other components. This example assumes interfacing with some Q-Chem modules that are not part of the tensor library and may not be freely available.

Top-level electronic structure routines	**adcman** Suite of ADC methods	**ccman2** Suite of CC and EOM methods	
Electronic structure utility libraries	**libmo** Library of tools for MO orbital spaces	**libcc** Library of CC and EOM equations	**liblegacy** Interface with other Q-Chem codes
Numerical tools	**libtensor** Tensor algebra library	**libsolve** Library of generic solvers (DIIS, Davidson, etc.)	
Low-level support	**libvmm** Virtual memory library	**libutil** Library of machine-dependent routines	**libctx** Calculation context tools

Figure 12. Overview of ADC and CC/EOM codes in Q-Chem. In this hierarchy, only the upper two layers are specific to electronic structure. Lower-level libraries are designed to be reusable in other areas of scientific computations. The codes are interfaced with Q-Chem via liblegacy, which is responsible for reading and setting up initial data from the integrals engine and other modules of Q-Chem.

However, this sample CC component is designed to minimize interaction with the core of Q-Chem and, therefore, should be a useful guide to creating a CC code outside of Q-Chem. If the library is to be used to implement a CC solver interfaced with another software package, it will be necessary to create or reuse the providers of initial data, such as Hartree–Fock solution and electron repulsion integrals, as well as a solver for CC equations (e.g., DIIS algorithm).

CCD theory uses an exponential ansatz for the ground state wave function $|\Psi\rangle$ that involves a reference determinant $|\Phi_0\rangle$ and cluster amplitudes T_2:

$$|\Psi\rangle = e^{T_2}|\Phi_0\rangle$$

The cluster amplitudes T_2 can be written in the second quantization form using the traditional notation in which i, j designate occupied orbitals in the reference determinant and a, b designate virtual orbitals. $a_p^\dagger$ and a_p are the creation and annihilation operators on orbital p, respectively:

$$T_2 = \frac{1}{4}\sum_{ijab} t_{ij}^{ab} a_a^\dagger a_b^\dagger a_j a_i$$

The CCD energy E is given by:

$$E = \langle\Phi_0|H|\Psi\rangle = \langle\Phi_0|He^{T_2}|\Phi_0\rangle$$

The cluster amplitudes required to compute the ground state wave function and energy are found by solving a set of linear equations:

$$\langle\Phi_{ij}^{ab}|e^{-T_2}He^{T_2}|\Phi_0\rangle = 0$$

In practice, these equations are not solved directly by writing them in a matrix form and finding the inverse of the matrix. Instead, the diagonal of the matrix is separated from the nondiagonal part and an iterative linear solver is used—an approach that works well with diagonally dominant matrices. The convergence can be improved by using the DIIS extrapolation.

The CCD equations are solved by extracting the diagonal of the Hamiltonian $D_{ij}^{ab} = \frac{1}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b}$ and using the following update step in the iterative procedure:

$$\begin{aligned} T_{ij}^{ab} D_{ij}^{ab} = {} & \langle ij||ab\rangle + \mathcal{P}_-(ab)\left(\sum_c f_{bc} t_{ij}^{ac} - \frac{1}{2}\sum_{klcd}\langle kl||cd\rangle t_{kl}^{bd} t_{ij}^{ac}\right) \\ & - \mathcal{P}_-(ij)\left(\sum_k f_{jk} t_{ik}^{ab} + \frac{1}{2}\sum_{klcd}\langle kl||cd\rangle t_{jl}^{cd} t_{ik}^{ab}\right) + \frac{1}{2}\sum_{kl}\langle ij||kl\rangle t_{kl}^{ab} \\ & + \frac{1}{4}\sum_{klcd}\langle kl||cd\rangle t_{ij}^{cd} t_{kl}^{ab} + \frac{1}{2}\sum_{cd}\langle ab||cd\rangle t_{ij}^{cd} \\ & - \mathcal{P}_-(ij)\mathcal{P}_-(ab)\left(\sum_{kc}\langle kb||jc\rangle t_{ik}^{ac} - \frac{1}{2}\sum_{klcd}\langle kl||cd\rangle t_{lj}^{db} t_{ik}^{ac}\right) \end{aligned} \tag{3}$$

where $\mathcal{P}_-(ij)A_{ij} = A_{ij} - A_{ji}$, capital T denotes the updated amplitudes, and t denotes the amplitudes from the previous iteration.

As shown in Figure 8, it is straightforward to implement this update step efficiently using the tensor library. Four intermediates are introduced to minimize the number of required contractions.

The implementation of CCD in Q-Chem involves multiple components (Figure 12), each having a specific role. Libsolve, a library of generic solvers, provides a templated linear solver with DIIS acceleration. Libcc contains the programmable equations for CCD amplitudes and energy written using the libtensor programming interface, the equations are similar to the ones shown in Figure 8. By putting together the generic DIIS solver, the CCD update step, and a few other solver policies,

```
//  Given current T2 amplitudes,
//  computes updated T2 for next iteration
class ccd_update_policy {
public:
  void update(
    ccd_amplitudes &told, ccd_amplitudes &tnew) {

    ccd_t2_update(tnew.t2, told.t2, ...);
  }
};

//  Checks termination criteria:
//  convergence or too many iterations
class ccd_conv_policy {
public:
  bool converged(int iter, ccd_amplitudes &err) {
    return (iter >= maxiter || err.norm() < thresh)
  }
};

//  Installs policies into a generic DIIS solver
class ccd_diis_solver :
  public libsolve::generic_diis_solver<
    ccd_amplitudes, ccd_amplitudes_algebra,
    ccd_update_policy, ccd_conv_policy> {

};

//  Main CCD function
void ccd() {
  ccd_amplitudes t2_guess, t2;
  import_integrals();
  make_ccd_guess(t2_guess);
  ccd_diis_solver s;
  s.set_guess(t2_guess);
  s.solve();
  if(s.get_niter() > maxiter) {
    cout << "CCD did not converge" << endl;
    return;
  }
  s.get_result(t2);
  print_ccd_result(t2);
}
```

Figure 13. C++ code snippet that sketches the implementation of the CCD method. While greatly simplified, it shows the essential ingredients that go into the solver. The update procedure is shown earlier in Figure 9.

such as a convergence test function, one obtains a procedure capable of solving CCD equations. This CCD solver is a part of the ccman2 component in Q-Chem.

Overall, ccman2 performs the following steps when finding the CCD energy of a molecule (Figure 13):

1. Import information about the molecule and computation settings from Q-Chem: basis set, molecular orbitals, point group symmetry.
2. Set up block tensor spaces required for the calculation.
3. Use Q-Chem's integral code to compute antisymmetrized electron repulsion integrals in the basis of molecular orbitals (Q-Chem returns transformed integrals, but it is also possible to start with the integrals in the atomic orbital basis and transform them using a sequence of four contractions).
4. Initialize the CCD solver with calculation settings (such as maximum number of iterations, convergence criteria) and initial data (Fock matrix, two-electron integrals).
5. During each CCD iteration, compute current energy and print it for the user along with the norm of the error vector.
6. Once CCD equations have converged, compute and print the final CCD energy. Save the CCD amplitudes for further use or analysis in the context of the calculation.

Steps 1–3 of this procedure are preparatory and are done outside of the coupled cluster code. In this example, ccman2 imports initial data from Q-Chem, but a similar interface can be established with any other program that is capable of delivering the same initial data.

Steps 4–6 are the essence of the implementation of CCD and deserve some attention.

The CCD solver is derived from a generic DIIS solver from libsolve by customizing its behavior through a programming technique called policies. The templated solver makes no assumptions with regard to the type of vectors or operators, it is implemented in a generalized way. The generic solver must be provided instructions on how to work with a particular vector type in order to be useful for a specific problem. For example, the CCD solver's vector type is the tensor of T_2 amplitudes. The solver needs a policy that tells it how to compute basic vector algebra: vector norm, overlap, linear combination, and other operations. The second important policy the solver requires is the update, that is, the result of acting of the linear operator on a vector. Usually, that translates into a matrix–vector multiplication, but in the case of CCD the update is a computation shown in eq. (3). Finally, the generic solver requires a termination policy, which stops it upon either convergence or exceeding a limit on the number of iterations.

Once the final T_2 is found, the CCD procedure saves the results in the context of the calculation. Context is an associative array, in which paths to objects are the keys, and objects themselves are the corresponding values. The paths are derived from a tree structure in which the objects are organized, similar to paths to locations in a file system. Routines have access to all the nodes within their context, but upper levels remain inaccessible. By passing the context from one to the next, the routines are able to build upon previous results preserving data encapsulation.

Benchmarks

We illustrate the performance of the library by using the following examples:

1. CCSD calculation of a methylated uracil–water dimer (mU–H_2O), 6-311+G(d,p) basis (302 basis functions), C_s symmetry.
2. CCSD calculation of mU–H_2O^+ (doublet radical), 6-311+G(d,p) basis (302 basis functions), C_s symmetry; unrestricted CCSD calculation using a ROHF reference.
3. CCSD calculation of a methylated uracil–water cluster, $(mU)_2$–H_2O, 6-31+G(d,p) basis (489 basis functions), C_1 symmetry.

Core electrons are frozen in all calculations. The thresholds and number of iterations for CCSD and EOM calculations are given in Table 2. The Cartesian geometries and relevant

Table 2. Converged CCSD energies and convergence[a] in the three test cases.

Code	Test case	SCF energy	CCSD energy	Ediff[b]	Tdiff[b]	Iterations
ccman	1	−566.7112775	−568.5609278	2.6×10^{-9}	1.5×10^{-5}	12
ccman2	1	−566.7112775	−568.5609272	1.1×10^{-8}	5.9×10^{-5}	10
Molpro	1	−566.7112774	−568.5609268	2.4×10^{-7}	6.5×10^{-9}[c]	9
ccman2	2	−566.395034	−568.231747	7.3×10^{-7}	6.8×10^{-5}	18
Molpro	2	−566.395806	−568.252101	5.3×10^{-7}	1.2×10^{-10}[c]	22
ccman2	3	−1057.143160	−1060.477186	1.6×10^{-7}	3.3×10^{-5}	12
Molpro	3	−1057.143160	−1060.477186	2.4×10^{-6}	1.1×10^{-9}[c]	11

[a] Molpro thresholds: THRVAR = 1.00D-08; THRDEN = 1.68D-06. Q-Chem thresholds: CC_T_CONV=4 ($\sqrt{\sum_{ijab}(t_{ijab}^{n}-t_{ijab}^{n-1})^2}$); CC_E_CONV=6 ($|E^n-E^{n-1}|$). [b] Differences at the last iteration. [c] VAR(S).

energies are given in Supporting Information. The benchmarks were performed on two designated Xeon machines referred to as Xeon-USC and Xeon-Stanford. The Xeon-USC configuration is Intel Xeon X5675 (2 × 6 cores, 3.0 GHz, 12 Mb cache), 128 Gb RAM, SCSI RAID 0 4 × 600 Gb = 2.2 Tb. The Xeon-Stanford setup is Intel Xeon X5690 (2 × 6 cores, 3.47 GHz, 12 Mb cache), 128 Gb RAM, SCSI RAID 0 700 Gb.

We begin by comparing the serial performance of our old and new codes (Table 3). As spin adaptation is not available in ccman, spin-unrestricted results provide the most direct comparison. On a single core, CCSD is about 20% faster with ccman2 than ccman. Using RHF symmetry in ccman2 cuts the total time further almost in half, which is expected as the T_2 tensor shrinks by 50% in the restricted case. Our implementation of spin symmetry eliminates the need to compute the block of T_2 thus reducing the calculation to the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ blocks only. Improving the algorithm such that only the $\alpha\beta\alpha\beta$ block is stored and computed would yield another 33% speedup. Full spin adaptation,[31] however, can yield a 2.5-fold improvement by reducing the operation count for the most time-consuming contraction, $\sum_{cd} t_{ij}^{cd}\langle ab||cd\rangle$, to $\frac{1}{4}N_{\text{occ}}^2N_{\text{virt}}^4$ relative to $\frac{10}{8}N_{\text{occ}}^2N_{\text{virt}}^4$ in a spin-unrestricted calculation. In our current implementation, this contraction involves $\frac{5}{8}N_{\text{occ}}^2N_{\text{virt}}^4$ floating point operations.

Table 3. Test case 1 (mU–H_2O), 302 basis functions, C_s symmetry. Wall times in seconds per CCSD iteration for Xeon-USC.

	ccman		ccman2	
Cores	Time	Speedup	Time	Speedup
UHF reference (data size 50 Gb, limit 80 Gb)				
1	1595		1232	
4	565	2.8 ×	325	3.8 ×
8	390	4.1 ×	192	6.4 ×
12	342	4.7 ×	153	8.0 ×
UHF reference (data size 50 Gb, limit 16 Gb)				
1	1600		1652	
4	799	2.0 ×	784	2.1 ×
8	711	2.3 ×	674	2.5 ×
12	688	2.3 ×	640	2.6 ×
RHF reference (data size 25 Gb, limit 80 Gb)				
1			665	
4			180	3.7 ×
8			110	6.1 ×
12			92	7.2 ×

The effect of different memory management algorithms is evident from the results in Table 3. Ccman exploits the operating system's caching mechanism to make the best use of memory. The memory limit specified is only used for memory allocation routines (i.e., new and alloc) and can be considered as advisory by operating system; through the file cache, the program can take as much as all available memory. Ccman2, on the other hand, strictly enforces the memory limit and disables the OS cache on the files used for tensor storage. This has a mild negative impact if the user sets a lower memory limit on a computation that could in principle fit in memory completely. At the same time, this strategy has shown to have a positive effect for larger calculations where significant amounts of I/O are inevitable.

Table 4 shows how CCSD timings depend on the memory limit. In a constrained regime with memory limits of 10 and 20 Gb, the smaller test 1 (total data size is 25 Gb) incurs over 50% of wasted idling time.[¶] However, as the memory limit in test 1 is above the data set size, idling drops to 10%. It is, therefore, reasonable to believe that this 10% accounts for load imbalance and contention as no I/O is expected in this regime. While the performance degradation in the case of test 1 is the subject of further investigation, the larger test 3 (data size is 235 Gb) shows more promising results: the fraction of

Table 4. Total wall and CPU clock timings for CCSD using ccman2 for test 1 and test 3 using various memory limits. Calculations are performed using Xeon-USC with 12 processors. Idling ratio is computed as $\boldsymbol{I}=\left(1-\frac{T_{\text{CPU}}}{12T_{\text{wall}}}\right)\times 100\%$.

	Test 1			Test 3		
Memory (Gb)	Wall (s)	CPU (s)	Idle (%)	Wall (h)	CPU (h)	Idle (%)
10	3652	10,934	75			
20	2044	10,219	58			
40	934	10,145	9	26.2	178.3	43
60	893	9676	10	24.1	173.5	40
80	834	8991	10	23.1	167.9	39
100				22.6	165.3	39

[¶]Idling is computed as the fraction of CPU cycles not spent doing useful work. Idling arises from load imbalance, resource contention, and waiting for I/O.

Table 5. Ccman2 profiling results on Xeon-USC. CCSD wall time (s) per iteration and the fraction of CPU time taken by dominating computational kernels are shown.

Cores	1	4	8	12
Test 1 (data size 25 Gb, limit 80 Gb)				
CCSD iteration	665	180	110	92
Scaling		3.7 ×	6.1 ×	7.2 ×
Matmul	86%	82%	74%	68%
Permute	7%	11%	18%	24%
Symmetry	4%	4%	4%	3%
Test 3 (data size 235 Gb, Limit 100 Gb)				
CCSD iteration	47,680	14,472	9468	8136
Scaling		3.3 ×	5.0 ×	5.9 ×
Matmul	94%	92%	90%	88%
Permute	2%	3%	4%	5%
Symmetry	2%	2%	2%	2%

Table 6. Q-Chem versus Molpro timings (full CCSD and average per-iteration wall times) for Xeon-Stanford.

	Q-Chem			Molpro		
Cores	CCSD time	Per iteration	Speedup	CCSD time	Per iteration	Speedup
Test 1 (closed shell)						
1	6392 s	639 s		5446 s	605 s	
2	3179 s	318 s	2.4 ×	2866 s	318 s	1.9 ×
4	1675 s	167 s	3.8 ×	1532 s	170 s	3.6 ×
8	973 s	97 s	6.6 ×	909 s	101 s	6.0 ×
12	778 s	78 s	8.2 ×			
Test 2 (open shell)						
1	21,033 s	1168 s		48495 s	2204 s	
2	10,706 s	595 s	2.0 ×			
4	5751 s	320 s	3.7 ×			
8	3593 s	200 s	5.9 ×			
Test 3 (closed shell)						
1		12.80 h			12.78 h	
2	78.9 h	6.58 h		73.5 h	6.68 h	
4	42.5 h	3.55 h		38.7 h	3.52 h	
8	25.5 h	2.12 h		23.8 h	2.17 h	
12	20.6 h	1.72 h				

wasted CPU cycles is stable at about 40%. To address some of the inefficiencies, we plan future algorithmic improvements such as optimization for data locality and automatic tuning of contraction batching. Currently, the best strategy for the user is to allow ccman2 to use as much memory as possible by specifying a memory limit of 75–90% of total available RAM.

Let us now compare the parallel performance of ccman and ccman2. Table 3 shows that ccman does not attain higher than four- to five-fold speedup even with 12 cores, which is due to the poor scalability of many used algorithms. With ccman2, it is possible to achieve a more than seven-fold speedup by using 12 cores.

To gain more insight into the scalability, we consider the breakdown of the total CPU time into matrix multiplications (GEMM BLAS calls), permutation of tensor blocks for optimal index alignment, and symmetry handling overhead. Table 5 shows these results for test 1 and test 3. In both of these low-symmetry cases (C_s and C_1, respectively), the symmetry handling overhead remains stable and low, not exceeding 4%. In test 1, permutations take 7% of the CPU time on a singe core. However, the memory bandwidth gets quickly saturated in the parallel mode, and on 12 cores the fraction of the time used for permutation grows to 24%. At the same time, matrix multiplications take 86% on a single core and 68% on 12 cores. As expected, in a larger case (test 3), the calculation spends less times in tensor permutations (permutation of an $N \times N$ matrix scales as N^2, whereas matrix multiplication of two such matrices scales as N^3). With 12 cores, this job spends 88% on matrix multiplication and only 5% on permutations.

As the next step, we compare the performance of our coupled-cluster codes with Molpro[32] (Molpro 2012.1 release binary was used). The timings are collected in Table 6.

We are interested in both total times for CCSD calculation and performance of our tensor library. For the test case 1 on a single core, ccman2 is only 6% slower than Molpro. This is very encouraging, as due to complete spin-adaptation in Molpro, coupled-cluster calculations involve considerably fewer floating point operations than in ccman2. We note that the parallel scaling of Molpro is slightly less efficient, for example, on eight cores, the speedup of Molpro is 6.0, whereas for ccman2 it is 6.6. Consequently, the difference in timings between the two codes shrinks and on four and eight processors ccman2's time per CCSD iteration is 2–4% faster than in Molpro.

To make a more adequate comparison between the two codes, we turn to a spin-unrestricted CCSD calculation of the same size (test 2). On a single core, we observe that Q-Chem is approximately twice faster than Molpro. As far as scaling is concerned, we observe a slight decline, possibly because parallel scaling is affected by I/O. Unfortunately, we were not able to obtain multicore spin-unrestricted timings for Molpro.

In sum, the coupled-cluster codes implemented in the ccman2 module of Q-Chem are about two times faster than Molpro for the spin-unrestricted case; however, for the spin-restricted case the gap shrinks and ccman2 is slightly slower due to incomplete spin adaptation. Thus, improving spin symmetry handling algorithms within the library is expected to lead to significant performance gains. As far as general CCSD algorithms are concerned, the two codes perform similarly when consistent convergence thresholds are used.

Due to technical issues (hardware failure), we were not able to obtain a full set of benchmarks for test 3. However, for this larger test case, we observe the same trend as for smaller examples: Q-Chem and Molpro show similar scaling and per-iteration times.

The range of available CC and EOM methods is much wider in ccman2 than in Molpro, in particular for open-shell systems. In addition, our code also features the capability to compute CC and EOM analytic gradients and state and transition properties.

Finally, to illustrate the capabilities of libtensor, we report timings for two large examples. The first one is the oligoporphyrin dimer (D_{2h} symmetry, 942 basis functions) used in NWChem benchmarks.[33] The size of the data in this calculation is 660 Gb. On 12 cores, one CCSD iteration takes 13.2 h

(which is equivalent to 160 CPU-hours). One CCSD iteration in a massively parallel calculation using NWChem on 1024 cores takes 810 s, which is equivalent to 230 CPU-hours. Although the absolute time required for the ccman2 calculation is longer, the job can be completed using ~100 times less computational resources and 40% less CPU-hours. The second example is a nucleobase tetramer, AATT[34] (C_1 symmetry, 966 basis functions). We use frozen core and frozen natural orbital approximations, which reduce the correlated space to 98 occupied and 551 virtual orbitals. The dataset size for this computation exceeds 1 Tb, and each CCSD iteration takes 60 h on 12 cores. Finally, using resoluion-of-identity and Cholesky decomposition approaches reduces these timings by 10–60% (E. Epifanovsky, et al., submitted to J. Chem. Phys.).

Conclusions and Outlook

We present a new general purpose tensor library that enables efficient production-level implementation of many-body electronic structure theories. The library features a straightforward expression programming interface and can be trivially combined with symbolic algebra equation generators. The library features all general tensor operations, takes advantage of tensor symmetry (permutational, point group, and spin symmetry) and sparsity in block tensors. It can be combined with different virtual memory management tools to adjust to a particular architecture. The library is multicore parallel showing reasonable scaling up to 12 cores.

The library represents an excellent development environment and enables fast implementation of efficient production-level codes for advanced correlated methods. The Q-Chem 4 release features a broad family of CC and EOM-CC methods (CCSD, EOM-EE/SF/IP/EA/DIP-CCSD), including analytic gradients and property calculations implemented using libtensor. The library (and the respective codes) can easily be ported to another quantum chemistry package.

By comparing the library performance against state-of-the-art codes, we note that for the spin-unrestricted case, ccman2 is about two times faster (on a single core) than Molpro. For spin-restricted calculations, the gap shrinks due to more complete spin-adaptation in Molpro. For a smaller example (302 basis functions), the two codes show almost identical performance. Molpro is 5% faster on a single core, whereas Q-Chem is 4% faster on eight cores.

Current developments include the implementation of CC, EOM/CI, and ADC methods that take advantage of resolution-of-the-identity (RI) and reduced-rank Cholesky representation of the two-electron integrals; improvements in the virtual memory mechanism and generic block tensor algorithms to enable offloading of computations to accelerators; an extension to distributed tensor storage and operations. In addition, improvements in spin adaptation and extensions to complex-valued tensors are planned.

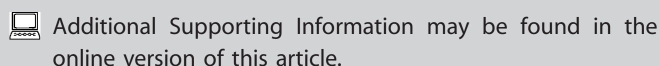
Because the library is not specific to electronic structure theory, we hope that it will find applicability in other areas of computational physics and chemistry that use tensor algebra.

Acknowledgments

We are grateful to Dr. Jiahao Chen and Prof. Todd Martinez for their help with benchmark calculations. A. I. K. is grateful to Prof. Juergen Gauss for stimulating discussions. The library development was supported by the National Science Foundation (CHE-0951634 and OCI-1216644, A. I. K). Implementation of higher-level methods was supported by the Department of Energy through the DE-FG02-05ER15685 grant. Benchmarking and extension of the library to handle RI and Cholesky-type contractions was supported by Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences. We also acknowledge support from the Humboldt foundation (A.I.K. and M.W.).

Keywords: tensor algebra · electronic structure · coupled-cluster theory · quantum chemistry software

How to cite this article: E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, P. Manohar, I. Kaliman, A. Dreuw, A. I. Krylov. *J. Comput. Chem.* **2013**, *34*, 2293–2309. DOI: 10.1002/jcc.23377


Additional Supporting Information may be found in the online version of this article.

[1] T. Helgaker, P. Jørgensen, J. Olsen, Molecular Electronic Structure Theory. Wiley & Sons, Chichester, West Sussex, England, **2000**.
[2] T. L. Windus, J. A. Pople, *Int. J. Quant. Chem.* **1995**, *56*, 485.
[3] C. D. Sherrill, A. I. Krylov, E.F.C. Byrd, M. Head-Gordon, *J. Chem. Phys.* **1998**, *109*, 4171.
[4] A. I. Krylov, C. D. Sherrill, E. F. C. Byrd, M. Head-Gordon, *J. Chem. Phys.* **1998**, *109*, 10669.
[5] A. I. Krylov, C. D. Sherrill, M. Head-Gordon, *J. Chem. Phys.* **2000**, *113*, 6509.
[6] S. R. Gwaltney, C. D. Sherrill, M. Head-Gordon, A. I. Krylov, *J. Chem. Phys.* **2000**, *113*, 3548.
[7] A. I. Krylov, *Chem. Phys. Lett.* **2001**, *338*, 375.
[8] A. I. Krylov, C. D. Sherrill, *J. Chem. Phys.* **2002**, *116*, 3194.
[9] A. I. Krylov, *Chem. Phys. Lett.* **2001**, *350*, 522.
[10] S. V. Levchenko, A. I. Krylov, *J. Chem. Phys.* **2004**, *120*, 175.
[11] S. V. Levchenko, T. Wang, A. I. Krylov, *J. Chem. Phys.* **2005**, *122*, 224106.
[12] L. V. Slipchenko, A. I. Krylov, *J. Chem. Phys.* **2005**, *123*, 084107.
[13] P. U. Manohar, A. I. Krylov, *J. Chem. Phys.* **2008**, *129*, 194105.
[14] P. U. Manohar, J. F. Stanton, A. I. Krylov, *J. Chem. Phys.* **2009**, *131*, 114112.
[15] D. Casanova, L. V. Slipchenko, A. I. Krylov, M. Head-Gordon, *J. Chem. Phys.* **2009**, *130*, 044103.
[16] A. A. Golubeva, P. A. Pieniazek, A. I. Krylov, *J. Chem. Phys.* **2009**, *130*, 124113.
[17] P. A. Pieniazek, S. E. Bradforth, A. I. Krylov, *J. Chem. Phys.* **2008**, *129*, 074104.
[18] C. M. Oana, A. I. Krylov, *J. Chem. Phys.* **2007**, *127*, 234106.
[19] A. Landau, K. Khistyaev, S. Dolgikh, A. I. Krylov, *J. Chem. Phys.* **2010**, *132*, 014109.
[20] A. I. Krylov, *Acc. Chem. Res.* **2006**, *39*, 83.
[21] A. I. Krylov, *Annu. Rev. Phys. Chem.* **2008**, *59*, 433.
[22] Y. Shao, L. Fusti-Molnar, Y. Jung, J. Kussman, C. Ochsenfeld, S. Brown, A. T. B. Gilbert, L. V. Slipchenko, S. V. Levchenko, D. P. O'Neill, R. A. Distasio Jr., R. C. Lochan, T. Wang, G. J. O. Beran, N. A. Besley, J. M.

Herbert, C. Y. Lin, T. Van Voorhis, S. H. Chien, A. Sodt, R. P. Steele, V. A. Rassolov, P. Maslen, P. P. Korambath, R. D. Adamson, B. Austin, J. Baker, E. F. C. Bird, H. Daschel, R. J. Doerksen, A. Drew, B. D. Dunietz, A. D. Dutoi, T. R. Furlani, S. R. Gwaltney, A. Heyden, S. Hirata, C.-P. Hsu, G. S. Kedziora, R. Z. Khalliulin, P. Klunziger, A. M. Lee, W. Z. Liang, I. Lotan, N. Nair, B. Peters, E. I. Proynov, P. A. Pieniazek, Y. M. Rhee, J. Ritchie, E. Rosta, C. D. Sherrill, A. C. Simmonett, J. E. Subotnik, H. L. Woodcock III, W. Zhang, A. T. Bell, A. K. Chakraborty, D. M. Chipman, F. J. Keil, A. Warshel, W. J. Herhe, H. F. Schaefer III, J. Kong, A. I. Krylov, P. M. W. Gill, and M. Head-Gordon, *Phys. Chem. Chem. Phys.* **2006**, *8*, 3172.

[23] S. Hirata, *J. Phys. Chem. A* **2003**, *107*, 9887.

[24] P.-W. Lai, H. Zhang, S. Rajbhandari, E. Valeev, K. Kowalski, P. Sadayappan, *Proc. Comp. Sci.* **2012**, *9*, 412.

[25] B. Sanders, R. Bartlett, E. Deumens, V. Lotrich, M. Ponton, A block-oriented language and runtime system for tensor algebra with very large arrays, In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. Washington, DC: IEEE Computer Society, **2010**, November 13-19; Washington DC.

[26] J. M. Turney, A. C. Simmonett, R. M. Parrish, E. G. Hohenstein, F. A. Evangelista, J. T. Fermann, B. J. Mintz, L. A. Burns, J. J. Wilke, M. L. Abrams, N. J. Russ, M. L. Leininger, C. L. Janssen, E. T. Seidl, W. D. Allen, H. F. Schaefer, R. A. King, E. F. Valeev, C. D. Sherrill, T. D. Crawford, *Wiley Interdiscip. Rev.: Comput. Mol. Sci.* **2012**, *2*, 556.

[27] A. I. Krylov, P. M. W. Gill, *WIREs Comput. Mol. Sci.* **2013**, *3*, 317.

[28] E. Epifanovsky, M. Wormit, T. Kuś, A. Landau, D. Zuev, K. Khistyaev, I. Kaliman, P. Manohar, A. Dreuw, A. I. Krylov, *New implementation of high-level correlated methods using a general block-tensor library for high-performance electronic structure calculations*, Available at: http://iopenshell.usc.edu/downloads/tensor/, **2011**. Accessed July 1, 2013.

[29] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, *ACM Trans. Math. Soft.* **2002**, *28*, 135.

[30] E. Solomonik, D. Matthews, J. Hammond, J. Demmel, Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions, In IEEE International Parallel and Distributed Processing Symposium (IPDPS). Boston, MA, **2013**.

[31] J. F. Stanton, J. Gauss, J. D. Watts, R. J. Bartlett, *J. Chem. Phys.* **1990**, *94*, 4334.

[32] H.-J. Werner, P. J. Knowles, G. Knizia, F. R. Manby, M. Schütz, P. Celani, T. Korona, R. Lindh, A. Mitrushenkov, G. Rauhut, K. R. Shamasundar, T. B. Adler, R. D. Amos, A. Bernhardsson, A. Berning, D. L. Cooper, M. J. O. Deegan, A. J. Dobbyn, F. Eckert, E. Goll, C. Hampel, A. Hesselmann, G. Hetzer, T. Hrenar, G. Jansen, C. Köppl, Y. Liu, A. W. Lloyd, R. A. Mata, A. J. May, S. J. McNicholas, W. Meyer, M. E. Mura, A. Nicklaß, D. P. O'Neill, P. Palmieri, D. Peng, K. Pflüger, R. Pitzer, M. Reiher, T. Shiozaki, H. Stoll, A. J. Stone, R. Tarroni, T. Thorsteinsson, M. Wang. Molpro version 2012.1; www.molpro.net; Accessed July 01, 2013.

[33] NWChem: High-performance computational chemistry software, Available at: http://www.nwchem-sw.org/index.php/Benchmarks [accessed on 13 March 2013].

[34] K. B. Bravaya, E. Epifanovsky, A. I. Krylov, *J. Phys. Chem. Lett.* **2012**, *3*, 2726.

Received: 19 March 2013
Revised: 13 June 2013
Accepted: 18 June 2013
Published online on 10 July 2013