Data/Code | Editors' Suggestion

Tensor computing application programming interface: An application-oriented, lightweight API for portable high-performance tensor network computations

Rong-Yang Sun [1,2,3] Tomonori Shirakawa [3,4,2,5,6] Hidehiko Kohshiro [3,4,7] D. N. Sheng [1] and Seiji Yunoki [3,4,5,8]

[1] *Department of Physics and Astronomy, California State University Northridge, Northridge, California 91330, USA*

[2] *RIKEN Interdisciplinary Theoretical and Mathematical Sciences Program (iTHEMS), Wako, Saitama 351-0198, Japan*

[3] *Computational Materials Science Research Team, RIKEN Center for Computational Science (R-CCS), Kobe, Hyogo 650-0047, Japan*

[4] *Quantum Computational Science Research Team, RIKEN Center for Quantum Computing (RQC), Wako, Saitama 351-0198, Japan*

[5] *Computational Condensed Matter Physics Laboratory, RIKEN Cluster for Pioneering Research (CPR), Saitama 351-0198, Japan*

[6] *Quantum Computing Simulation Unit, Quantum-HPC Hybrid Platform Division, RIKEN Center for Computational Science (R-CCS), Kobe, Hyogo 650-0047, Japan*

[7] *Next-Generation HPC Application Development Unit, Next-Generation HPC Infrastructure Development Division, RIKEN Center for Computational Science (R-CCS), Kobe, Hyogo 650-0047, Japan*

[8] *Computational Quantum Matter Research Team, RIKEN Center for Emergent Matter Science (CEMS), Wako, Saitama 351-0198, Japan*

(Received 7 February 2026; published 13 November 2026)

Tensor networks (TNs) are a central computational tool in quantum science and artificial intelligence. However, the lack of a unified software interface across tensor-computing frameworks severely limits the portability of TN applications, coupling algorithmic development to specific hardware and software backends. To address this challenge, we introduce the tensor computing application programming interface (TCAPI)—an application-oriented, lightweight API designed to enable framework-independent, high-performance TN applications, currently specified for `C++` and `Python`. TCAPI provides well-defined type-system conventions that abstract tensor objects, together with a minimal yet expressive set of core functions for essential tensor manipulations and tensor linear-algebra operations, with operation semantics shared across language specifications. Through numerical demonstrations of representative tensor-network applications implemented in `C++` using TCAPI, we show that code written against TCAPI can be migrated seamlessly across heterogeneous hardware and software platforms while achieving performance comparable to native framework implementations. We further provide open-source implementations of TCAPI based on *Cytnx* and *ITensor* in `C++`, and on *NumPy* in `Python`, demonstrating its practicality and ease of integration with existing tensor-computing frameworks.

DOI: 10.1103/dmx5-95ny

Published by the American Physical Society under the terms of the Creative Commons Attribution 4.0 International license. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

I. INTRODUCTION

Tensor networks (TNs) have become a central computational paradigm for representing and manipulating high-dimensional classical and quantum data. They have been especially influential in quantum many-body physics, where tensor-network states (TNSs) provide compact variational representations of many-body wavefunctions whose efficiency is governed by entanglement structure. Prominent examples include matrix product states (MPSs) and projected entangled-pair states (PEPSs), which offer a unifying language for low-entanglement quantum states [1]. Tensor-network algorithms such as the density-matrix renormalization group [2], time-evolving block decimation (TEBD) [3], and time-dependent variational principle (TDVP) methods [4,5] have enabled highly accurate simulations of one-dimensional quantum lattice systems, while PEPS and related TNSs extend the framework to strongly correlated systems in two and higher spatial dimensions [6]. Tensor renormalization group methods provide a complementary approach to classical and quantum lattice models, evaluating partition functions and thermodynamic observables through successive coarse-graining of tensor networks [7,8]. These methods have become indispensable for studying quantum many-body systems in regimes that are otherwise computationally inaccessible, and they provide powerful tools for characterizing entanglement structure and topological properties. Examples include symmetry-protected topological states [9–12] and intrinsic topological orders [13–17].

Beyond quantum many-body physics, TNs have become important tools in quantum computing. They enable the classical simulation of many-qubit quantum circuits with moderate depth [18–25], support circuit synthesis [26–32], and facilitate quantum error correction and mitigation [33–37] (see Ref. [38] for a comprehensive review).

TNs and tensor decompositions are also increasingly used in machine learning. In this setting, tensorial neural networks (TNNs) [39] can provide efficient data representations [40–44] and replace dense model parameters with tensor-factored forms [44–49], enabling more resource-efficient machine-learning models. Recent studies have applied TNNs to reinforcement learning [50], prompt learning [51], statistical learning [52,53], and the training and fine-tuning of large language models [54–59]. More broadly, by connecting classical and quantum descriptions, TNs provide a natural framework for quantum machine learning [60–64].

Progress in TN algorithms and applications has, in turn, driven rapid advances in tensor computing—that is, the hardware and software stacks responsible for executing tensor operations underlying TN workloads. Here, we use the term tensor-computing framework (TCF) to denote a concrete combination of a hardware platform (e.g., a shared-memory CPU system) and the tensor-computing software stack designed to target it. In practice, TN applications are typically written directly against a specific TCF [see Fig. 1(a)]. As a result, application logic becomes tightly coupled to the chosen framework, since competing TCFs generally expose incompatible application programming interfaces (APIs). Portability across TCFs is therefore nontrivial. For example, an application implemented using *ITensor* [65] is not straightforward to migrate to *Cytnx* [66], despite the fact that both provide `C++` APIs. Adopting a different TCF thus commonly necessitates substantial code reimplementation, even when the underlying TN algorithm remains unchanged. This lack of portability increases development cost, complicates maintenance, and raises the risk of introducing implementation errors, as schematically illustrated in Fig. 1(b).

Accelerating progress in both TN algorithms and applications requires true application portability across TCFs, which in turn calls for a clear separation between application logic (i.e., TN algorithms) and the underlying execution backends through a standardized software interface. A well-specified interface would enable seamless migration of applications across heterogeneous TCFs while preserving a single codebase. As a result, development efforts could focus on algorithmic design, whereas advances in hardware and software backends would be inherited with little or no modification to application code. In the machine-learning community, where `Python` is the dominant programming language, the *de facto* standard interface for tensor computation is the array API introduced in *NumPy* [67]. Popular frameworks such as *JAX* [68] and *PyTorch* [69]

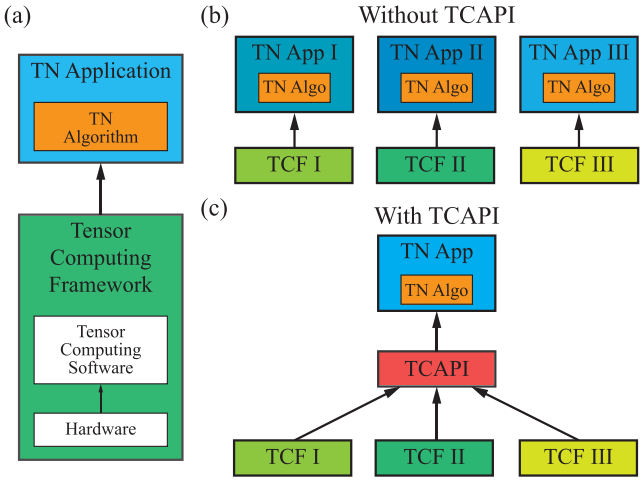


FIG. 1. System architectures and development workflows for TN applications. Arrows indicate dependency relations, with arrowheads pointing to the dependent components. (a) Typical software architecture of a TN application, in which the application logic and TN algorithms are directly tied to a specific TCF. (b) Development workflow without TCAPI: TN applications must be reimplemented separately for each TCF, leading to duplicated development effort and limited portability. (c) Development workflow with TCAPI: Application logic and TN algorithms are decoupled from backend hardware and software through a uniform interface, enabling portable development across heterogeneous TCFs.

provide NumPy-like interfaces, enabling a certain degree of portability at the source-code level. However, this portability remains limited in practice: Migrating nontrivial applications between frameworks typically requires substantial reimplementation effort. Moreover, although NumPy-like interfaces provide a useful convention within `Python`, they do not by themselves address portability between `Python`-based frameworks and performance-critical `C++` applications, particularly in HPC environments. Their portability within `Python` is further undermined by subtle but consequential API differences across frameworks. In addition, because many tensor operations are exposed as methods of the tensor object itself (e.g., slicing or reshaping), it is often difficult for application developers to customize or intercept framework behavior without modifying or extending the underlying tensor classes.

Motivated by these limitations, we introduce the tensor computing application programming interface (TCAPI), a lightweight software interface for generic tensor computations, currently specified for `C++` and `Python`. TCAPI is designed to be application oriented and to sit clearly between TN applications and TCFs, enabling developers to decouple application logic from execution backends without modifying the source code of the underlying TCFs. At its core, TCAPI provides carefully designed type-system conventions that abstract tensor objects and their associated metadata: The `C++` specification uses compile-time tensor types and traits, whereas the `Python` specification uses run-time tensor-kind descriptors and backend-native tensor values. Built on these conventions, TCAPI defines a concise yet expressive set of functions whose operation semantics are shared across language specifications and cover essential tensor manipulations and tensor linear-algebra operations. As illustrated in Fig. 1(c), this design allows TN applications to be written against a uniform interface, enabling portable development across heterogeneous hardware and software backends. As a result, application developers can focus on implementing TN algorithms rather than managing framework-specific details. The `C++` specification of TCAPI is formulated in modern `C++` based on the `C++17` standard. We provide open-source `C++` implementations based on *Cytnx* [70] and *ITensor* [71], together with a `Python` implementation based on *NumPy* [72]. An online version of the TCAPI specification is also available [73]. To evaluate the practicality and performance of TCAPI, we

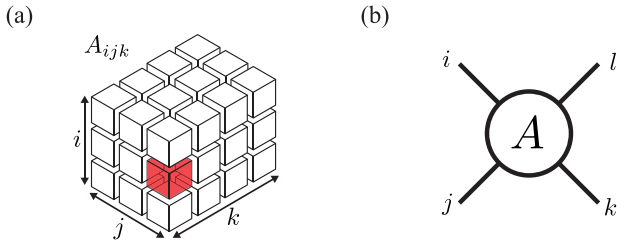


FIG. 2. Graphic representations of tensors. (a) Array representation of a third-order tensor A_{ijk}, where the entry at coordinates (i, j, k) corresponds to the tensor element $A_{i,j,k}$. (b) Penrose diagram of a fourth-order tensor A_{ijkl}, in which the node represents the tensor and each incident leg corresponds to a tensor mode (bond).

implement two representative TN applications—ground-state simulation and real-time dynamics of quantum many-body systems—using TCAPI in `C++` and benchmark them across multiple backends. Our results show that applications written against TCAPI can be migrated across TCFs without changes to their main code while achieving performance comparable to implementations based on native framework APIs. The application-oriented scope of TCAPI also clarifies its relation to lower-level tensor-algebra standardization efforts, such as tensor algebra processing primitives (TAPP). TAPP defines a `C`-based, BLAS-like interface for performance-critical tensor-algebra primitives, especially tensor contractions [74]. Thus, TAPP targets tensor-kernel portability, whereas TCAPI targets application-level portability of TN workloads across TCFs.

The remainder of this paper is organized as follows. Section II reviews the fundamentals of tensor computation, introducing tensors and the basic operations that underpin TN workloads. Section III illustrates the basic `C++` usage of TCAPI through a minimal working example, with step-by-step explanations of the core components. In Sec. IV, we present numerical benchmarks that demonstrate both the portability and the performance of two representative TN applications implemented using TCAPI in `C++`. Section V concludes the paper with a summary and a discussion of future directions. Implementation details of the `C++` TCAPI-based application for ground-state simulation are provided in Appendix A. Appendix B describes the TN algorithm employed for quantum dynamics simulation. A comprehensive specification of TCAPI for `C++` and `Python`, including type-system conventions, core functions, and supporting environment variables, is given in Appendix C.

II. TENSOR AND TENSOR OPERATIONS

We begin by reviewing the foundational elements of tensor computation, focusing on a computational definition of tensors and the core operations that underlie tensor workloads. The terminology and notation introduced in this section establish a common language for the subsequent presentation and form the basis for the TCAPI specification.

A. Tensor

From a computational perspective, we regard a *tensor* as an n-dimensional array $A_{ijk\ldots}$ that generalizes scalars A (zero dimensional), vectors A_i (one dimensional), and matrices A_{ij} (two dimensional). The number of dimensions—equivalently, the number of indices or modes—defines the *order* of the tensor. Accordingly, a scalar is a zeroth-order tensor, a vector is a first-order tensor, and a matrix is a second-order tensor. Low-order tensors (up to third order) can be conveniently depicted as multidimensional arrays [see Fig. 2(a) for a third-order example]. In contrast, higher-order tensors are more naturally represented using Penrose's graphical notation, in which a node denotes the tensor and each incident edge (or "leg") corresponds to a tensor dimension [75] [see Fig. 2(b) for a fourth-order example].

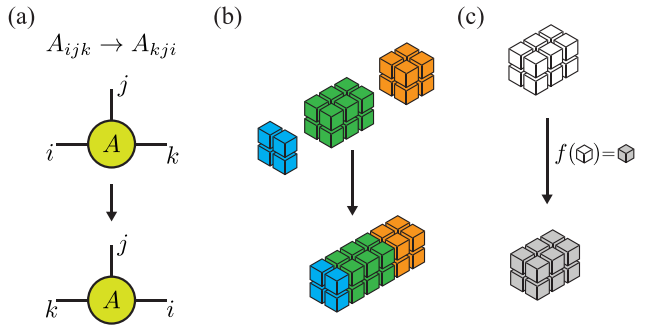


FIG. 3. Typical tensor-manipulation operations. (a) Transpose, which reorders tensor bonds. (b) Concatenation, which combines multiple tensors along a specific bond. (c) Broadcasting, which applies a scalar function elementwise while preserving tensor order and shape.

Each mode (dimension) of the tensor array is represented by a *bond*, which appears in diagrammatic notation as a "leg" attached to the tensor node [see Fig. 2(b)]. A bond is characterized by its *bond index* b, which specifies its position in an explicit ordering of the tensor modes, and its *bond dimension* d_b, which gives the size of that mode. For example, a 4×5 matrix can be represented as a tensor A_{ij}. Under zero-based indexing, the bond associated with index j has bond index $b = 1$ and bond dimension $d_1 = 5$. In an abstract TN diagram, bonds are specified by connectivity rather than by array-axis order. In software, however, each bond must be mapped to a definite tensor mode. This ordering fixes the bond indices used in TCAPI, determines the tensor shape, and affects the implementation of tensor operations. For tensor networks with nontrivial exchange structure, such as fermionic or braided tensor networks, the ordering may also be tied to convention-dependent data; incorporating such structures is left for future work. The ordered list of bond dimensions, arranged according to bond indices, defines the *shape* of the tensor. For a zeroth-order tensor, the shape is an empty list. In the above example, the shape of A is $(4, 5)$.

We refer to the entries of a tensor array as *elements*. Each element is uniquely identified by a zero-based *coordinate* tuple $\mathbf{c} = (c_0, \ldots, c_{n-1})$ and has an associated *value* $A_{\mathbf{c}} = A_{c_0,\ldots,c_{n-1}}$. For instance, the highlighted entry in Fig. 2(a) corresponds to the coordinates $\mathbf{c} = (1, 2, 0)$ and has value $A_{1,2,0}$. The *size* of a tensor is defined as the total number of its elements. For a zeroth-order tensor, the size is 1. Throughout this paper, we denote by $\mathbb{K}$ the scalar field from which tensor elements are drawn. In the context of TCAPI, we restrict $\mathbb{K}$ to the real field $\mathbb{R}$ or the complex field $\mathbb{C}$, while the concrete element type and precision are determined by the underlying TCF.

B. Tensor operations

Tensor computations are performed by applying well-defined operations to tensors. These operations generalize familiar vector- and matrix-level primitives and can be broadly classified into two categories: tensor-manipulation operations and tensor linear-algebra operations.

1. Tensor-manipulation operations

This class of operations comprises tensor manipulations that act either on the structure of a tensor or independently on its individual elements. They include nonarithmetic transformations—such as bond reordering, reshaping, the combining or splitting of bonds (tensor modes), and slicing—that permute the ordering of elements or modify the tensor shape without changing element values, together with elementwise operations that preserve tensor order and shape. Below, we illustrate three representative tensor-manipulation operations shown in Fig. 3. A complete list of tensor-manipulation functions supported by TCAPI is provided in Appendix C 2 d.

Transpose. Given a permutation π of the bond indices, the transpose operation produces a tensor B whose entries satisfy

$$B_{i_{\pi(0)} i_{\pi(1)} \cdots i_{\pi(n-1)}} = A_{i_0 i_1 \cdots i_{n-1}}. \tag{1}$$

This operation permutes bonds and therefore reorders tensor elements without changing their values. While the memory layout is typically altered, the underlying data remain unchanged. Figure 3(a) illustrates the specific transpose $A_{ijk} \to A_{kji}$.

Concatenation. Tensor concatenation combines multiple tensors of the same order along a specified bond. All bonds with the same bond index must have identical dimensions across the input tensors, except for the concatenated bond. For example, concatenating three third-order tensors along bond k yields a third-order tensor whose kth bond dimension equals the sum of the corresponding bond dimensions of the inputs. Figure 3(b) illustrates the concatenation of three tensors into a single tensor.

Broadcasting. Given a tensor $A \in \mathbb{K}^{d_0 \times \cdots \times d_{n-1}}$ and a unary function $f : \mathbb{K} \to \mathbb{K}$, broadcasting (or entrywise mapping) defines a tensor $B = f(A)$ by

$$B_{i_0 i_1 \cdots i_{n-1}} = f\big(A_{i_0 i_1 \cdots i_{n-1}}\big), \quad 0 \leqslant i_b < d_b. \tag{2}$$

Broadcasting preserves both tensor order and shape, yielding $B \in \mathbb{K}^{d_0 \times \cdots \times d_{n-1}}$. Figure 3(c) shows an example in which a scalar function $f(x)$ is applied elementwise to a third-order tensor.

2. Tensor linear-algebra operations

This class comprises tensor linear-algebra operations, which generalize matrix-level primitives to higher-order tensors. Such operations typically proceed by (1) matricizing (unfolding) a tensor with respect to a chosen partition of its bonds, (2) applying a matrix-level routine—such as multiplication, singular value decomposition (SVD), eigensolving, or linear solving—and (3) refolding the result into tensor(s) of appropriate shape. In contrast to tensor-manipulation operations, which act on tensor structure or on elements independently, tensor linear-algebra operations treat the tensor as a single coupled algebraic object. In the following, we present two representative examples: contraction and decomposition. A complete list of tensor linear-algebra operations supported by TCAPI is provided in Appendix C 2 e.

Contraction. Tensor contraction is a fundamental primitive in TN applications. Let tensors A and B share a set of bonds S with matching bond dimensions. Denoting by I the ordered set of bonds A not in S, and by J the ordered set of bonds of B not in S, contraction over the shared bonds produces a tensor C with bonds $I \cup J$, whose elements are given by

$$C_{IJ} = \sum_S A_{IS} B_{SJ}. \tag{3}$$

This expression reduces to ordinary matrix multiplication when the numbers of nonshared and shared bonds—that is, the cardinalities of the corresponding bond sets—satisfy $|I| = |J| = |S| = 1$. As a concrete example, consider tensors A_{ijk} and B_{jklm} sharing bonds j and k. Their contraction yields

$$C_{ilm} = \sum_{j=0}^{d_j - 1} \sum_{k=0}^{d_k - 1} A_{ijk} B_{jklm}, \tag{4}$$

where d_j and d_k denote the corresponding bond dimensions. The graphical representation of this contraction is shown in Fig. 4(a).

Decomposition. Tensor decomposition is another fundamental primitive in TN applications. It extends matrix factorizations to higher-order tensors by expressing a tensor—exactly, or approximately when truncation is applied—as a contracted network of lower-order factors. As a representative example, we consider the tensor SVD. Let $A \in \mathbb{K}^{d_i \times d_j \times d_k}$ and matricize it with respect

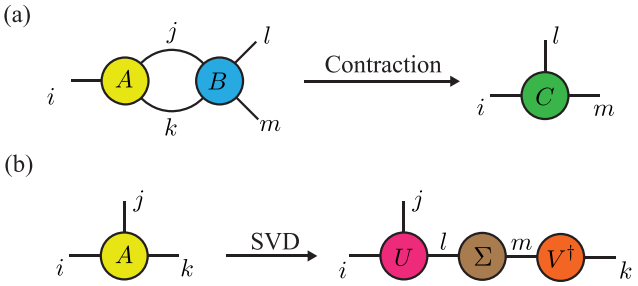


FIG. 4. Representative tensor linear-algebra operations. (a) Tensor contraction over shared bonds. (b) Tensor decomposition, illustrated by the SVD as a representative example.

to the bipartition $\{i, j\}|\{k\}$, yielding

$$A' \in \mathbb{K}^{(d_i \times d_j) \times d_k}, \quad A'_{pq} \equiv A_{ijk}, \tag{5}$$

where the composite index p corresponds to (i, j) and q corresponds to k. Performing an SVD gives $A' = U'\Sigma V^\dagger$, i.e.,

$$A'_{pq} = \sum_{l,m} U'_{pl} \Sigma_{lm} (V^\dagger)_{mq}. \tag{6}$$

Reshaping U'_{pl} to U_{ijl} and identifying $V^\dagger_{mq} = V^\dagger_{mk}$, we obtain the tensor decomposition $A = U * \Sigma * V^\dagger$, where $*$ denotes tensor contraction over the internal bonds l and m. The corresponding graphical representation is shown in Fig. 4(b).

III. BASIC USAGE OF THE TENSOR COMPUTING APPLICATION PROGRAMMING INTERFACE

In this section, we illustrate the basic usage of the `C++` TCAPI through a minimal working example, highlighting its concise and flexible syntax in a step-by-step manner. The `Python` interface follows the same operation semantics but uses run-time tensor-kind descriptors and backend-native tensor values; the full `C++` and `Python` specification is provided in Appendix C. The example considers generating a random many-body wavefunction, measuring the bipartite entanglement entropy (BEE) between the left and right halves of the system, and constructing a low-entanglement (low-rank) approximation of the state. Throughout the example, we assume that the `C++` TCAPI has been implemented on top of a TCF that provides a concrete `C++` tensor type `Ten` with element type `double`.

To use the `C++` TCAPI, we first include its header file `tcapi/tcapi.h`, along with the required `C++` standard library headers used in this example:

```
#include "tcapi/tcapi.h"
#include <random>
```

To make the code portable across different TCFs, we define a set of type aliases that abstract away TCF-dependent types used throughout the example:

```
using ten = Ten;
using elem = tcapi::elem_t<ten>;
```

TCAPI provides a collection of alias templates—such as `tcapi::elem_t` used here—that extract concrete types associated with a given tensor type (see Appendix C 1 for details). Note that the type `Ten` is directly accessible because the header `tcapi/tcapi.h` is supplied by the TCF exposing

`Ten`. To migrate this code to a different TCF, one simply replaces `Ten` with the corresponding tensor type provided by the new framework, without modifying the remaining code.

Before invoking any TCAPI tensor operations, a context must be created to manage resources associated with the TCF, such as GPU devices or thread pools. This step is typically performed at the beginning of the `main` function. The concrete context-handle type is obtained using the alias template `tcapi::context_handle_t`, and a context instance is initialized by calling `tcapi::create_context`:

```
tcapi::context_handle_t<ten> ctx;
tcapi::create_context(ctx);
```

We next generate a random wavefunction for a qubit chain of length six using the function `tcapi::random`:

```
std::mt19937 engine;
std::uniform_real_distribution<double> dis(
    -1.0, 1.0);
auto gen = [&dis, &engine]() {
    return dis(engine);
};
auto psi = tcapi::random<ten>(
    ctx, {2, 2, 2, 2, 2, 2}, gen);
```

The wavefunction `psi` is a sixth-order tensor. To satisfy the normalization condition, we renormalize `psi` using `tcapi::normalize`:

```
tcapi::normalize(ctx, psi);
```

To measure the BEE between the left and right halves of the chain, we perform a Schmidt decomposition of `psi` across the center of the system using the SVD function `tcapi::svd`:

```
ten u, sigma, vt;
tcapi::svd(ctx, psi, 3, u, sigma, vt);
```

Here, the third argument, `3`, is the TCAPI parameter `num_of_bds_as_row`; it specifies that the SVD is performed across the bipartition after the first three bonds of `psi`, corresponding to the split between the left and right halves of the six-qubit chain. The output `sigma` is a second-order diagonal tensor whose diagonal entries are the singular values; implementations may store this tensor compactly when supported by the underlying TCF. To compute the BEE, we extract the diagonal entries of `sigma` as a first-order tensor and traverse these values using `tcapi::for_each`:

```
ten sigma_vals;
tcapi::diag(ctx, sigma, sigma_vals);

elem ee = 0.0;
auto compute_ee = [&ee](const elem sigma_val){
    ee -= (sigma_val * sigma_val)
          * log(sigma_val * sigma_val);
};
tcapi::for_each(ctx, sigma_vals, compute_ee);
```

The resulting BEE is stored in `ee`.

We next investigate a low-entanglement approximation of `psi` and examine the effect of truncation on the wavefunction. Such an approximation is obtained by performing a truncated SVD using

`tcapi::trunc_svd`, retaining only the two largest singular values:

```
elem trunc_err = 0.0;
tcapi::trunc_svd(
    ctx, psi, 3,
    u, sigma, vt,
    trunc_err, 2, 0.0);
```

The approximate wavefunction `psi1` is reconstructed by contracting `u`, the diagonal tensor `sigma`, and `vt`. The contractions are carried out using `tcapi::contract`:

```
ten psi1;
tcapi::contract(
    ctx,
    u, "ijkl", sigma, "lm", psi1, "ijkm");
tcapi::contract(
    ctx,
    psi1, "ijkl", vt, "lmno", psi1, "ijkmno");
tcapi::normalize(ctx, psi1);
```

To quantify the truncation effect, we compute the fidelity between `psi` and the normalized `psi1` by evaluating their overlap via tensor contraction:

```
ten ovlp;
tcapi::contract(
    ctx,
    psi, "ijklmn", psi1, "ijklmn",
    ovlp, "");
auto ovlp_v = tcapi::get_elem(ctx, ovlp, {});
auto fide = std::norm(ovlp_v);
```

Since this contraction eliminates all bonds, `ovlp` is a zeroth-order tensor with shape `{}` and size 1, whose sole element is read using `tcapi::get_elem(ctx, ovlp, {})`. The fidelity `fide` should equal $1.0 - $ `trunc_err`. Finally, before exiting `main`, we release the resources managed by the context using `tcapi::destroy_context`:

```
tcapi::destroy_context(ctx);
```

The context handle `ctx` must not be passed to any TCAPI function after `tcapi::destroy_context` has been called.

The code presented in this section can be seamlessly ported to other `C++` TCFs by modifying only the line

```
using ten = Ten;
```

demonstrating the flexibility of TCAPI. For more realistic applications, readers are referred to the example code presented in Appendix A, which implements a TN algorithm for the imaginary-time evolution of a quantum spin chain. For guidance on efficient use of TCAPI in large-scale production environments, including supercomputers, we refer readers to the open-source library [76] used for Application B in the following section.

IV. NUMERICAL DEMONSTRATIONS

By design, TN applications developed with TCAPI are highly portable, while the interface introduces only a minimal abstraction overhead in the `C++` case, consistent with the zero-overhead principle of modern `C++`. As a result, their performance is expected to be comparable to that achieved using the native APIs of the underlying TCFs. We substantiate these claims—namely, high portability and low abstraction overhead for the `C++` interface—by benchmarking two representative TN applications arising in quantum many-body computations. The first targets ground-state calculations, while the second focuses on quantum dynamics. Both applications are implemented using the `C++` TCAPI as well as the native APIs of one of the adopted TCFs, enabling a direct and fair performance comparison.

A. Two representative TN applications

Our first TN application (Application A) computes the ground state of the one-dimensional (1D) transverse-field Ising model (TFIM) using imaginary-time evolution with the infinite time-evolving block decimation (iTEBD) algorithm [3]. The Hamiltonian is given by

$$H = -J \sum_i Z_i Z_{i+1} + g \sum_i X_i, \tag{7}$$

where Z_i (X_i) denotes the Pauli-Z (Pauli-X) operator acting on site i. Here, J is the nearest-neighbor Ising coupling, which we set to unity hereafter, and g is the transverse-field strength along the x direction. For $g = 1$, the value used in the benchmarks below, the system undergoes a quantum phase transition [77], accompanied by a closing of the excitation gap. In contrast to gapped one-dimensional systems, whose ground states obey an entanglement area law [78] and can therefore be efficiently approximated by finite-bond-dimension MPSs, critical one-dimensional systems exhibit logarithmic growth of bipartite entanglement with subsystem size, as described by conformal field theory [79]. A finite MPS bond dimension then imposes an effective entanglement and correlation-length cutoff, leading to finite-entanglement scaling [80,81]. This critical setting therefore provides a controlled way to increase the bond dimension and collect performance data for increasingly demanding tensor-network workloads. Implementation details of the `C++` TCAPI-based version of Application A are provided in Appendix A.

Our second TN application (Application B) simulates the dynamics of the kicked Ising model (KIM) on a heavy-hex lattice [82,83] using a two-dimensional tensor-network state (2dTNS) combined with belief propagation (BP). An overview of the 2dTNS-BP algorithm is given in Appendix B. One Floquet cycle is implemented by the quantum circuit

$$U = \prod_{\langle i,j \rangle} R_{ij}^{ZZ}(\theta_{ZZ}) \prod_i R_i^X(\theta_X), \tag{8}$$

where $R_{ij}^{ZZ}(\theta_{ZZ}) = \exp(-i\theta_{ZZ} Z_i Z_j/2)$ denotes a two-qubit ZZ rotation acting on sites i and j, and $R_i^X(\theta_X)$ is a single-qubit X rotation on site i. Here, $\langle i, j \rangle$ runs over the edges of the lattice connectivity graph. The lattice geometry used for benchmarking and a schematic of the corresponding circuit are shown in Figs. 5(a) and 5(b), respectively. After M Floquet cycles, the state evolves to

$$|\psi\rangle = U^M |00\cdots0\rangle, \tag{9}$$

starting from the all-zero product state. In the benchmarks presented below, we use $\theta_X = 0.7\pi$, $\theta_{ZZ} = 0.25\pi$, and $M = 10$.

B. Demonstration of high portability across heterogeneous platforms

The numerical demonstrations use `C++` TCAPI implementations on top of three TCFs. TCF I employs `gqten::tensor` from the *GraceQ/tensor* library [84] for CPU execution. TCF II targets

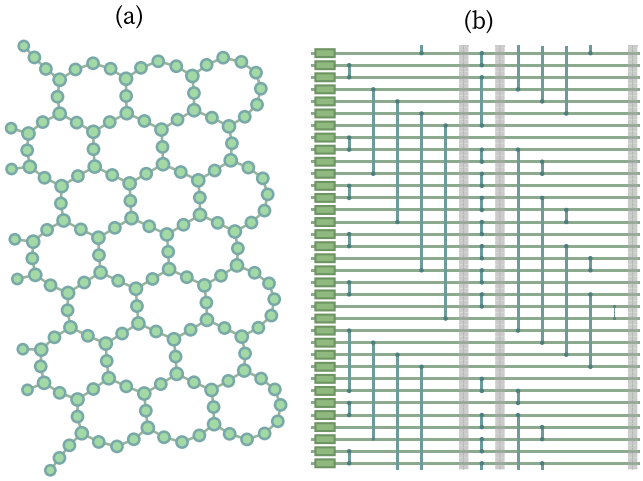


FIG. 5. Configuration used in Application B. (a) Heavy-hex lattice geometry corresponding to the 156-qubit superconducting chips, e.g., `ibm_marrakesh` and `ibm_kobe`, in IBM Quantum System Two. This connectivity is mapped directly onto the 2dTNS used in the simulation. (b) Representative portion of the quantum circuit implementing a single Floquet cycle of the KIM. Rectangles denote single-qubit R^X gates, while wires connecting pairs of qubits represent two-qubit R^{ZZ} gates. Gray vertical bands indicate groups of R^{ZZ} gates executed concurrently by the parallel algorithm (see Appendix B).

NVIDIA GPUs using `cuda_tensor`, a thin wrapper over *cuTENSOR* [85] and *cuTensorNet* [86]. TCF III is based on *Cytnx*.

We first assess portability between CPU and GPU platforms by compiling the `C++` TCAPI-based implementation of Application A against two backends: TCF I (CPU) and TCF II (GPU). The resulting binaries are executed on an Intel Xeon 6240R CPU and an NVIDIA H100 GPU, respectively. Figure 6 reports the wall-clock time as a function of the bond dimension χ. For $\chi > 1000$, TCF II achieves speedups exceeding one order of magnitude relative to TCF I. This result demonstrates that TCAPI-based TN applications can be retargeted to accelerator-based hardware with minimal effort, while delivering substantial performance gains.

We next evaluate portability between a local development environment and a massively parallel production environment by compiling the `C++` TCAPI-based implementation of Application B against TCF I and TCF III. The TCF I binary is executed on the supercomputer Fugaku using 4 and 156 Fujitsu A64FX nodes (production environment), whereas the TCF III binary is executed on an Apple M2 Ultra CPU (development environment). Figure 7 shows the wall-clock time of the BP procedure applied on the tensors in the 2dTNS. On Fugaku, we observe favorable strong scaling: increasing the node count from 4 to 156 reduces the run-time by approximately a factor of 15. Although the four-node configuration is slightly slower than the Apple M2 Ultra, the 156-node configuration achieves substantially improved performance. These results indicate that TN applications developed using TCAPI can be migrated seamlessly from a local shared-memory environment to a massively parallel supercomputing architecture and can effectively exploit its resources without any modifications to the application source code.

C. Demonstration of low abstraction overhead

We next demonstrate that TN applications developed using the `C++` TCAPI achieve performance comparable to implementations written directly against the native APIs of the underlying TCFs. This confirms that TCAPI introduces only a negligible abstraction overhead.

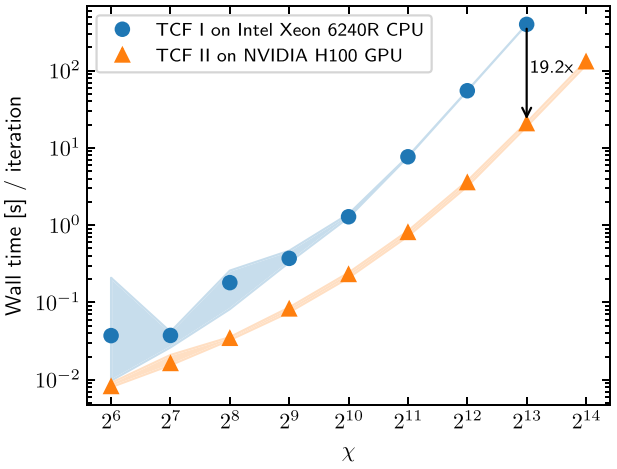


FIG. 6. Wall-clock time per iTEBD iteration as a function of the bond dimension χ for the ground-state calculation of 1D TFIM at $g = 1$ (Application A). Blue dots correspond to TCF I executed on an Intel Xeon 6240R CPU, while orange triangles correspond to TCF II executed on an NVIDIA H100 GPU. Marks indicate mean values computed over the final ten iterations, and shaded bands represent the corresponding minimum-maximum ranges.

To quantify the overhead, we reimplemented Applications A and B using the native API of TCF I (`gqten::tensor`) and repeated the same numerical benchmarks on the corresponding hardware platforms. For Application A executed on an Intel Xeon 6240R CPU, the TCAPI-based implementation exhibits performance close to that of the native API implementation across all tested bond dimensions, as shown in Fig. 8(a). The inset shows that the relative timing difference, $(t_{\mathrm{TCAPI}} - t_{\mathrm{native}})/t_{\mathrm{native}}$, remains small and does not grow systematically with the bond dimension, indicating that the abstraction overhead introduced by TCAPI is negligible compared with the

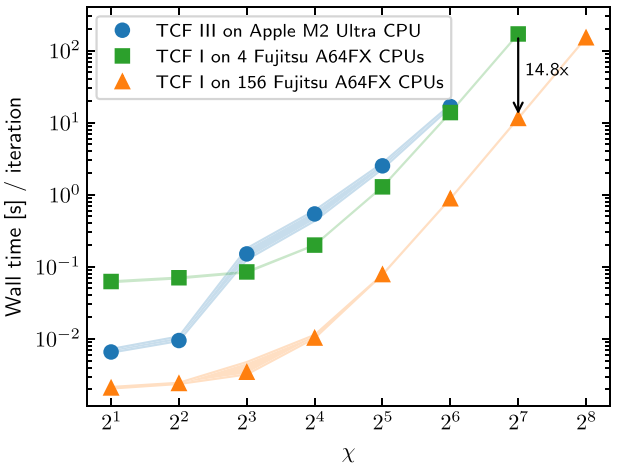


FIG. 7. Wall-clock time per BP iteration as a function of the bond dimension χ for the KIM simulation on a heavy-hex lattice (Application B). Blue dots correspond to TCF III executed on an Apple M2 Ultra CPU, while green squares and orange triangles correspond to TCF I executed on 4 and 156 nodes of supercomputer Fugaku, respectively. Marks indicate mean values over ten BP iterations, and shaded regions represent the corresponding minimum-maximum ranges.

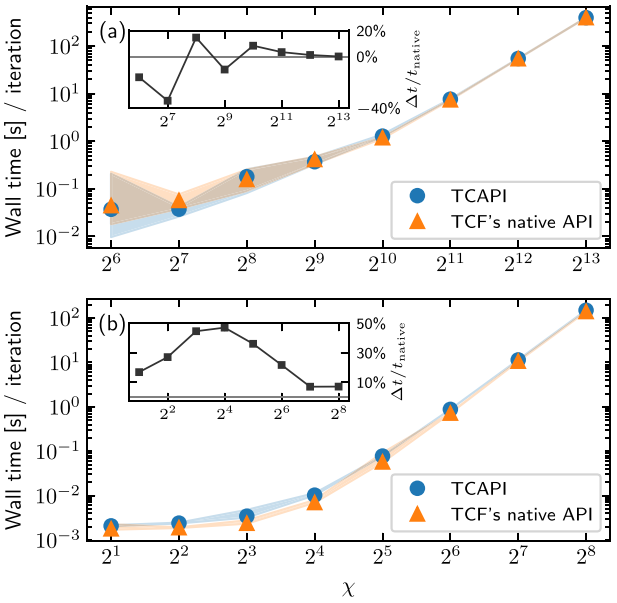


FIG. 8. Comparison between TCAPI-based implementations and native implementations using TCF I, analogous to Figs. 6 and 7. Blue dots correspond to TN applications developed using TCAPI, while orange triangles correspond to implementations written directly against the native API of TCF I. Insets show the relative timing difference, $(t_{\text{TCAPI}} - t_{\text{native}})/t_{\text{native}}$, quantifying the abstraction overhead introduced by TCAPI. (a) Application A. (b) Application B.

overall computational cost. For Application B executed on Fugaku using 156 nodes, the native implementation is marginally faster at small bond dimensions ($\chi = 4$–32), where fixed overheads are more visible because the absolute wall time is short. However, the two implementations exhibit nearly indistinguishable performance for $\chi > 64$, as shown in Fig. 8(b), and the inset confirms that the relative difference decreases as the computational workload increases. Overall, these results demonstrate that TCAPI does not introduce any significant performance degradation, even in large-scale supercomputing environments.

V. SUMMARY AND OUTLOOK

In summary, we have introduced TCAPI—an application-oriented, lightweight API that enables portable, high-performance TN applications across diverse TCFs—and specified it for both `C++` and `Python` with shared tensor semantics. TCAPI comprises (1) coherent type-system conventions that model tensor objects and their associated metadata, (2) a comprehensive set of primitives for essential tensor computations, and (3) run-time configuration environment variables that govern execution behavior. We implemented the `C++` specification across TCFs spanning shared-memory CPU systems, GPU-accelerated platforms, and massively parallel supercomputing architectures, and developed representative TN applications using TCAPI. Our benchmark results demonstrate that applications can be migrated across TCFs with minimal code modifications while achieving performance comparable to implementations written directly against native framework APIs. Taken together, these results position TCAPI as a viable pathway toward standardizing an application-level interface for generic, high-performance tensor computation on current and emerging HPC architectures.

Looking ahead, several developments are required to further mature TCAPI into a fully standardized interface for generic tensor computations. As modern computing platforms increasingly

rely on CPU–accelerator architectures—most notably GPUs—exploiting concurrency has become a first-class performance objective for TN workloads. Accordingly, future revisions of TCAPI will introduce asynchronous APIs designed to support highly concurrent execution. In addition, automatic differentiation has become an essential tool for developing and optimizing TN algorithms, and we plan to incorporate a standardized automatic-differentiation API in a subsequent version of TCAPI. While the current specification focuses primarily on dense tensors, symmetry support is an important future direction, especially for quantum many-body applications. A natural TCAPI extension would expose symmetry information through API-level symmetry metadata, such as quantum-number sectors, sector dimensions, leg directions, parity or grading, and context-level fusion or sign conventions, while leaving storage layouts and backend-specific implementations to each TCF. This would allow contractions, permutations, and decompositions to retain the same high-level calling structure for dense and symmetry-structured tensors. Designing such a portable specification requires careful treatment of the differing conventions used by existing libraries and is left to future work.

The `C++` specification targets the `C++17` standard to maximize portability across leadership-class HPC systems. Nonetheless, the absence of modern language features such as concepts and constraints limits the expressiveness and robustness of TCAPI-based applications. We therefore plan to migrate the `C++` specification to `C++20` or later standards in future releases. The `Python` specification introduced here provides a basis for integration with `Python`-based TCFs developed for machine learning, such as *JAX* and *PyTorch*. Given the widespread use of `Julia` in the tensor-computing community, a companion Julia specification remains an important future direction, particularly for quantum-physics TCFs such as *ITensors.jl* [87] and *TensorKit.jl* [88].

ACKNOWLEDGMENTS

We acknowledge helpful discussions with Q. Zhao, C. Li, H. Xu, and S. Niu. We thank K. F. Hui for developing the TCAPI implementations based on *ITensor* and *NumPy*. This work was primarily supported by the U.S. Department of Energy, Office of Basic Energy Sciences, under Grant No. DE-FG02-06ER46305 (R.-Y.S. and D.N.S.). This work was also supported in part by project JPNP20017, commissioned by the New Energy and Industrial Technology Development Organization (NEDO), and by JSPS KAKENHI Grant No. JP21H04446. Additional support was provided by JST COI-NEXT (Grant No. JPMJPF2221) and by the Program for Promoting Research of the Supercomputer Fugaku (Grant No. MXP1020230411) from MEXT, Japan. We further acknowledge support from the UTokyo Quantum Initiative and the RIKEN TRIP initiative (RIKEN Quantum). Numerical benchmarks were performed on the HOKUSAI supercomputer at RIKEN and the Fugaku supercomputer at the RIKEN Center for Computational Science (Projects No. ra000011 and No. ra010014), as well as through the HPCI System Research Project (Project No. hp250069).

DATA AVAILABILITY

The data that support the findings of this article are openly available [89].

APPENDIX A: IMPLEMENTATION OF APPLICATION A USING `C++` TCAPI

In this Appendix, we provide the `C++` TCAPI-based implementation of Application A, which performs the ground-state calculation of the 1D TFIM using iTEBD with imaginary-time evolution.

```cpp
#include "tcapi/tcapi.h"
using Ten = /* The concrete tensor type, for example gqten::tensor<gqten::Real_8> */;

#include <array>
#include <vector>
#include <random>

using ContextHandle = tcapi::context_handle_t<Ten>;
using Elem = tcapi::elem_t<Ten>;
using Real = tcapi::real_t<Ten>;

int main (int argc, char *argv[]) {
  // Create context
  ContextHandle context;
  tcapi::create_context(context);

  // Define the parameters of the model and the simulation
  Real J = 1.0, g = 1.0;
  Real dt = 0.005;
  int d = 2, chi = 5;
  int N = 3000;

  // Prepare Hamiltonian and imaginary-time evolution operator
  auto g2 = - 0.5 * g;
  auto h = tcapi::zeros<Ten>(context, {4, 4});
  tcapi::set_elem(context, h, {0, 0}, J); tcapi::set_elem(context, h, {3, 3}, J);
  tcapi::set_elem(context, h, {1, 1}, -J); tcapi::set_elem(context, h, {2, 2}, -J);
  std::vector<tcapi::elem_coors_t<Ten>> coors_set {
    {0, 1}, {0, 2}, {1, 0}, {1, 3},
    {2, 0}, {2, 3}, {3, 1}, {3, 2}
  };
  for (auto &coors : coors_set) { tcapi::set_elem(context, h, coors, g2); }
  Ten u;
  tcapi::scale(context, h, -dt, u);
  tcapi::exp(context, u, 1);
  tcapi::reshape(context, u, {d, d, d, d});

  // Initialize infinite MPS
  std::mt19937 engine;
  std::uniform_real_distribution<Real> dis(-1.0, 1.0);
  auto gen = [&dis, &engine]() { return dis(engine); };
  std::array<Ten, 2> Gamma {
      tcapi::random<Ten>(context, {chi, d, chi}, gen),
      tcapi::random<Ten>(context, {chi, d, chi}, gen)
  };
  std::array<Ten, 2> Lambda {
      tcapi::random<Ten>(context, {chi}, gen),
      tcapi::random<Ten>(context, {chi}, gen)
  };
  for (auto &lambda : Lambda) { tcapi::diag(context, lambda); }

  // iTEBD iterations
  for (int i = 0; i < N; i++) {
    auto A = i % 2;
    auto B = (i+1) % 2;
    Ten Theta;
    tcapi::contract(context, Lambda[B],"ij", Gamma[A],"jkl", Theta,"ikl");
    tcapi::contract(context, Theta, "ijk", Lambda[A],"kl", Theta,"ijl");
    tcapi::contract(context, Theta,"ijk", Gamma[B],"klm", Theta,"ijlm");
    tcapi::contract(context, Theta,"ijkl", Lambda[B],"lm", Theta,"ijkm");

```

```
    tcapi::contract(context, Theta,"ijkl", u,"jkmn", Theta,"imnl");
    Real trunc_err = 0.0;
    tcapi::trunc_svd(context,
      Theta, 2,
      Gamma[A], Lambda[A], Gamma[B],
      trunc_err,
      chi,
      1.0e-10
    );
    tcapi::normalize(context, Lambda[A]);

    auto LambdaB_inv = tcapi::copy(context, Lambda[B]);
    tcapi::diag(context, LambdaB_inv);
    tcapi::for_each(context, LambdaB_inv, [](Elem &elem) { elem = 1.0 / elem; });
    tcapi::diag(context, LambdaB_inv);
    tcapi::contract(context, LambdaB_inv,"ij", Gamma[A],"jkl", Gamma[A],"ikl");
    tcapi::contract(context, Gamma[B],"ijk", LambdaB_inv,"kl", Gamma[B],"ijl");
    if (i >= 1) {
      Elem E_iTEBD = 0.0;
      tcapi::for_each(context,
          Theta,
          [&E_iTEBD](const Elem &elem) { E_iTEBD += (elem*elem); });
      E_iTEBD = - log(E_iTEBD) / dt / 2.0;
      std::printf("Iter %5i E_iTEBD = %.15f\n", i, E_iTEBD);
    }
  }

  // Destroy context
  tcapi::destroy_context(context);
  return 0;
}
```

APPENDIX B: 2dTNS-BP ALGORITHM FOR QUANTUM CIRCUIT SIMULATION

This Appendix summarizes the algorithm used to simulate quantum circuits in Application B, as introduced in Sec. IV. The overall workflow is as follows. A quantum circuit is first partitioned into smaller segments, each of which—while still representing a valid subcircuit—is transformed into a tensor-network operator (TNO). These TNOs are then applied sequentially to a TNS. During this process, the bond dimensions are regulated through low-rank approximations, enabling efficient state updates while preventing exponential growth in computational cost.

The subsections below describe (1) the construction of TNOs from quantum circuits and (2) the BP-based approximation used to contract TNOs with TNSs, which together enables scalable and efficient circuit simulation.

1. Construction of TNOs from quantum circuits

In this implementation, quantum circuits specified in `OpenQASM` format [90] are converted into TNO representations suitable for TN processing, as outlined in Fig. 9. Within the standard `OpenQASM` specification, one-, two-, and three-qubit gates constitute the basic instruction set. For clarity, we restrict the present discussion to circuits composed of single- and two-qubit gates [Fig. 9(a)].

The conversion procedure starts from a bond-dimension-one TNO corresponding to the identity operator [Fig. 9(b)]. Gate tensors are then incorporated sequentially. Single-qubit gates are applied directly to the corresponding local operator tensors, whereas each two-qubit unitary gate—represented as a fourth-order tensor—is decomposed, typically using SVD, into two local tensors connected by an auxiliary virtual bond. Each resulting tensor acts on one qubit and the shared virtual bond [Figs. 9(b) and 9(c)], ensuring compatibility with the target TN topology. For two-dimensional geometries, these identity operators are placed along the shortest path connecting the target qubits on the hardware coupling graph.

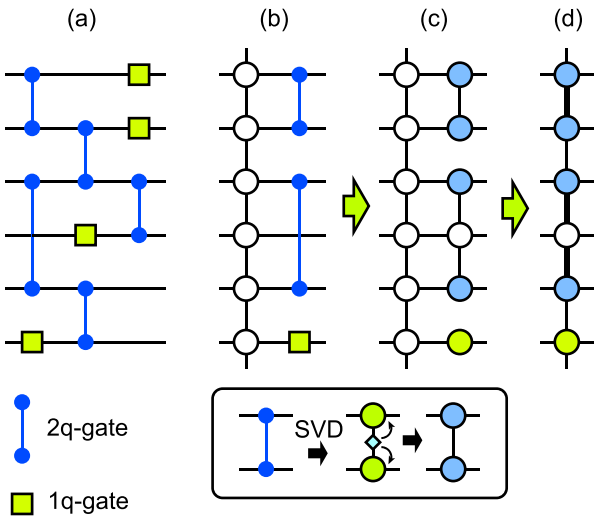


FIG. 9. (a) Schematic of a quantum circuit composed of single-qubit (1q) and two-qubit (2q) gates. (b) Construction of the base TNO representing the identity operator together with the first gate layer. White circles denote tensors of bond dimension one on the virtual bonds, corresponding to the operator $1 \otimes 1 \otimes \cdots \otimes 1$. (c) Decomposition of a two-qubit unitary gate into two site tensors—each acting on one qubit and a shared virtual bond—obtained via SVD. Non-nearest-neighbor gates are handled by inserting identity operators along the intermediate path connecting the two target qubits. (d) Assembly of the full TNO by attaching the decomposed site tensors to the base network. Tensors updated through gate contractions are highlighted in color, and thicker lines indicate enlarged bond dimensions.

To avoid the computational explosion that would arise from attaching all gates simultaneously, barriers are inserted between layers of parallelizable gates, and separate TNOs are constructed for each layer. All `OpenQASM` circuits used in this work were generated using *Qiskit* [91].

2. BP-based approximate contraction scheme

We employ a BP-based regauging procedure to efficiently contract a TNO with a TNS, while applying low-rank truncations to the updated tensors. The approach follows Ref. [92] and is summarized schematically in Fig. 10. The procedure proceeds as follows. First, the TNO site tensors O_i are contracted with the corresponding TNS site tensors V_i [Fig. 10(a)]. BP is then iterated over all bonds until the message tensors $M_{i\to j}$ converge [Fig. 10(b)]. Upon convergence, square-root and inverse square-root tensors, $M_{i\to j}^{1/2}$ and $M_{i\to j}^{-1/2}$, are constructed so as to satisfy the consistency relations illustrated in Fig. 10(c). These tensors are inserted on each bond [Fig. 10(d)], after which SVD-based low-rank truncation is applied to obtain the updated site tensors V_i. Although BP involves nearest-neighbor communication, all contractions are strictly local, allowing the algorithm to be parallelized naturally with only local data exchanges.

Relative to TEBD, the present approach offers greater flexibility. TEBD requires a fixed ordering in which adjacent pairs of sites are updated sequentially, whereas operating directly on the TNO enables more general update schedules. For example, gates acting on nonadjacent sites can be incorporated without explicit reordering. The main trade-off is that the computational cost increases with the bond dimension of the TNO; consequently, very large TNOs reduce overall efficiency. An additional distinction concerns the regauging strategy. In TEBD with the simple-update scheme, regauging is performed after truncation, whereas in the present method it is carried out during TNO attachment. In principle, this ordering can provide more context-aware local environments for approximation, although our numerical tests did not reveal a significant practical advantage.

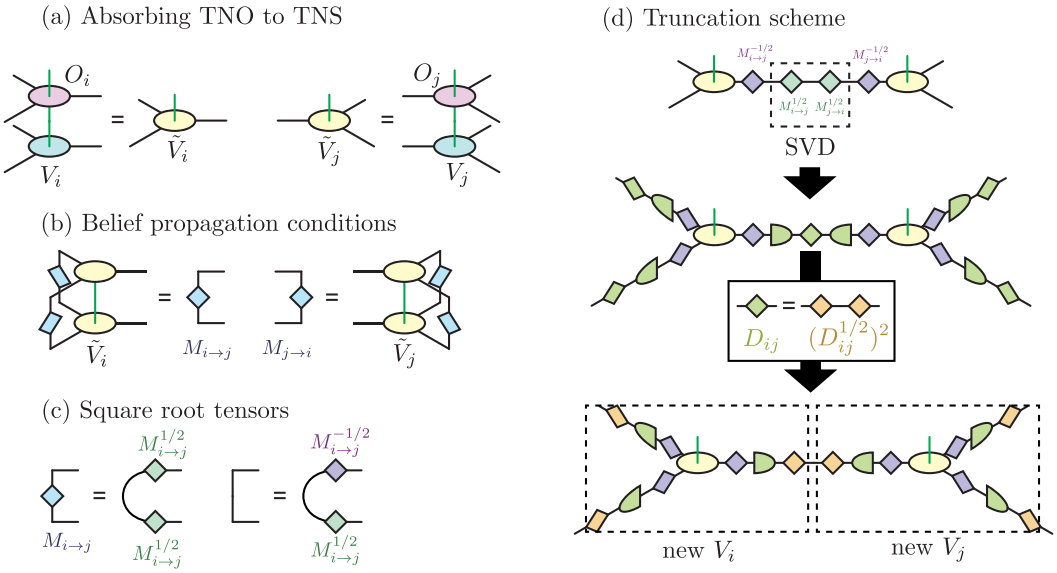


FIG. 10. Workflow for updating a TNS by applying a TNO using BP, followed by low-rank approximation of the resulting tensors. (a) Local contraction of TNO site tensors O_i with TNS site tensors V_i, producing intermediate tensors $\tilde{V}_i$. (b) Fixed-point relations for the BP message tensors. (c) Consistency constraints for the square-root and inverse square-root message tensors, $M_{i\to j}^{1/2}$ and $M_{i\to j}^{-1/2}$. (d) SVD-based truncation of the message tensors $M_{i\to j}$. With D_{ij} denoting the diagonal matrix of singular values, the largest χ singular values are retained to reconstruct the updated site tensors V_i within the prescribed maximum bond dimension. Dashed boxes highlight the subnetwork in which the updated tensors V_i are assembled.

Conversely, unlike TEBD—which can omit certain regauging passes and truncate directly—the present TNO-TNS scheme requires regauging at every update step, since truncation is applied to the BP message tensors. As a result, the overall update loop is modestly more computationally demanding.

We provide an open-source implementation of the 2dTNS-BP algorithm, *TNBP* [76]. Beyond the benchmarks reported in this work, *TNBP* is ready for simulating generic large-scale quantum circuits on massively parallel computing systems.

APPENDIX C: TENSOR COMPUTING APPLICATION PROGRAMMING INTERFACE

In this Appendix, we present the TCAPI specification for the `C++` and `Python` language interfaces. The tensor terminology and operation semantics introduced above are shared by all TCAPI language interfaces. Each language exposes these concepts through type-system conventions appropriate to that language: the `C++` interface uses compile-time tensor types and traits, while the `Python` interface uses run-time tensor-kind descriptors and raw backend tensor values.

1. Type system

a. `C++` *type system*

To expose a uniform interface across heterogeneous TCFs, the `C++` interface relies heavily on generic programming. TCAPI therefore defines a self-contained type system for abstract tensor objects and their associated traits. Building on this foundation, TCAPI provides core tensor operations and run-time configuration variables for diagnostics and performance reporting. Unless

stated otherwise, C++ classes and functions reside in the tcapi namespace and are expected to be accessible through the single header tcapi/tcapi.h.

The C++ TCAPI type system consists of three components: an abstract tensor type, denoted by TenT; a set of trait types associated with TenT, exposed through tcapi::tensor_traits; and a collection of auxiliary types.

TenT. TenT is the core abstraction in the C++ TCAPI interface. It denotes a generic, compile-time-resolved tensor type whose concrete implementation and associated types, such as element and shape types, are accessed through tcapi::tensor_traits. An object of type TenT represents a tensor through its logical tensor elements and metadata, and may manage backend resources associated with that tensor, including data storage and representation-specific metadata. These logical tensor elements need not all be stored explicitly: Different objects of the same TenT may use dense storage or implementation-defined compact storage, such as diagonal storage. In the C++ TCAPI interface, TenT is not required to be copy constructible or copy assignable. TCAPI code that is intended to work across conforming backends must therefore not duplicate tensors by ordinary C++ copying; explicit standalone deep duplication is provided through tcapi::copy(ctx, orig), which receives the backend context required to perform the copy. Ownership transfer is provided through tcapi::move(ctx, from). If a backend provides ordinary C++ copy construction or copy assignment for TenT, those operations are implementation extensions and are not part of the portable TCAPI contract.

Each C++ TCAPI implementation exposes its concrete tensor types through the namespace-level alias tcapi::supported_ten_types_t, accessible through tcapi/tcapi.h:

```
namespace tcapi {
using supported_ten_types_t =
  std::tuple<
    /* implementation-provided tensor types */
  >;
}
```

This is a compile-time type catalog. Its ordering is implementation defined. The catalog lists the types provided by the implementation, while their compatibility with a particular execution context remains backend dependent. The number of listed types, an individual tensor type, and its element type can be obtained using standard tuple traits and tcapi::elem_t:

```
using Types = tcapi::supported_ten_types_t;
constexpr auto count = std::tuple_size_v<Types>;
using FirstTen = std::tuple_element_t<0, Types>;
using FirstElem = tcapi::elem_t<FirstTen>;
```

tcapi::tensor_traits. Properties intrinsically associated with TenT are exposed as member types of the tcapi::tensor_traits class template:

```
template <typename TenT>
struct tensor_traits;
```

Table I lists the required member types and their roles. To ensure a uniform API across heterogeneous backends, some associations are currently constrained; for example, tcapi::tensor_traits<TenT>::shape_t must be List<tcapi::tensor_traits<TenT>::bond_dim_t>. These constraints may be relaxed in future TCAPI revisions.

TABLE I. Member types defined in `tcapi::tensor_traits<TenT>` and their intended roles.

Member type	Description
`ten_t`	Concrete tensor type corresponding to `TenT`.
`order_t`	Integral type for the tensor order.
`shape_t`	Shape type in bond-index order; `List<bond_dim_t>`.
`bond_dim_t`	Integral type for a bond dimension.
`bond_idx_t`	Integral type for a bond index (zero based).
`bond_label_t`	Integral type for a user-defined bond label.
`ten_size_t`	Integral type for the number of tensor elements, i.e., the product of the dimensions in the tensor shape.
`elem_t`	Element (value) type of `ten_t`; a real or complex floating-point type supporting arithmetic (+, -, *, /), comparisons, and common math functions via argument-dependent lookup.
`elem_coor_t`	Integral type for a single coordinate along a bond (zero based).
`elem_coors_t`	Coordinate tuple type for an element; `List<elem_coor_t>` in bond-index order.
`real_t`	Real scalar type; equal to `elem_t` if `elem_t` is real, otherwise the real component type of `elem_t`.
`real_ten_t`	Tensor type identical to `ten_t` but with `elem_t` replaced by `real_t`.
`cplx_t`	Complex scalar type; equal to `elem_t` if `elem_t` is complex, otherwise the complex counterpart of `elem_t`.
`cplx_ten_t`	Tensor type identical to `ten_t` but with `elem_t` replaced by `cplx_t`.
`context_handle_t`	Handle to the backend context maintained by the underlying TCF that provides `ten_t`.

To streamline generic programming workflows, TCAPI provides alias templates that forward the associated types listed in Table I from `tcapi::tensor_traits`:

```
template <typename TenT>
using target_type =
typename tensor_traits<TenT>::target_type;
```

Here, `target_type` represents any required associated types (e.g., `order_t`, `shape_t`, or `elem_t`); these aliases eliminate repetitive boilerplate of the form `typename tensor_traits<TenT>::...`, so a function template that processes a single element can be written as

```
template <typename TenT>
void process_element(
    const tcapi::elem_t<TenT> el
);
```

Auxiliary types. In addition to the associated types described above, TCAPI defines a set of auxiliary types. These types are used both to construct the member types listed in Table I and as parameter or argument types for the core functions introduced in Appendix C 2. All auxiliary types are summarized in Table II.

For notational convenience in Appendix C 2, we introduce two implementation-level aliases: `bond_idx_pairs_t<TenT>`, defined as an alias for `List<Pair<tcapi::bond_idx_t<TenT>, tcapi::bond_idx_t<TenT>>>`, and `bond_idx_elem_coor_pair_map<TenT>`, defined as an alias for `Map<tcapi::bond_idx_t<TenT>, Pair<tcapi::elem_coor_t<TenT>, tcapi::elem_coor_t<TenT>>>`. These aliases are intended solely for internal specification

TABLE II. Auxiliary types defined in TCAPI and their intended roles.

Auxiliary type	Description
`List<T>`	Sequence container of values of type `T`; alias of `std::vector<T>`.
`CRef<T>`	Nonowning const reference to an object of type `T`; alias of `std::reference_wrapper<const T>`.
`Pair<T, U>`	Pair of values of types `T` and `U`; alias of `std::pair<T,U>`.
`Map<T, U>`	Associative container mapping `T` → `U`; alias of `std::unordered_map<T,U>`.

purposes. They are not part of the formal TCAPI specification and are therefore not exported in the `tcapi` namespace.

b. `Python` *type system*

`Python` TCAPI exposes tensor types through a run-time descriptor called `TenKind`. A `TenKind` is the `Python` analog of the `C++` tensor type `TenT`: It identifies a family of backend-native tensor values and carries the type information needed by TCAPI functions. A `TenKind` is not itself a tensor object, and `Python` TCAPI does not wrap returned tensors in TCAPI-defined tensor classes. Instead, `Python` TCAPI functions return backend-native tensors. A `TenKind` identifies the backend tensor family and element type, not a particular storage representation. Dense and diagonal tensors of the same backend family and dtype therefore have the same `TenKind`. If the underlying TCF does not support compact diagonal storage, TCAPI routines that produce diagonal tensors return dense second-order backend tensors with the diagonal entries determined by the operation and all off-diagonal entries set to zero.

The required metadata fields associated with a `TenKind` mirror the `C++` traits in `tcapi::tensor_traits<TenT>`: and are summarized in Table III.

Concrete `TenKind` objects are provided and documented by each TCAPI implementation. `Python` TCAPI provides the following ten-kind introspection facilities:

```
tcapi.ten_kind_ten_type(TenKind)
tcapi.ten_kind_dtype(TenKind)
tcapi.ten_kind_real_dtype(TenKind)
tcapi.ten_kind_cplx_dtype(TenKind)

tcapi.real_ten_kind(TenKind)
tcapi.cplx_ten_kind(TenKind)

tcapi.ten_kind(ctx, tensor)

tcapi.supported_ten_kinds
```

TABLE III. Metadata associated with a `Python` `TenKind` and their intended roles.

Metadata	Description
`ten_type`	Raw backend tensor class returned by TCAPI functions.
`dtype`	Backend-native dtype of elements; `Python` analog of `tcapi::elem_t<TenT>`.
`real_dtype`	Backend-native real dtype; `Python` analog of `tcapi::real_t<TenT>`.
`cplx_dtype`	Backend-native complex dtype; `Python` analog of `tcapi::cplx_t<TenT>`.
`real_ten_kind`	Tensor kind corresponding to `tcapi::real_ten_t<TenT>`.
`cplx_ten_kind`	Tensor kind corresponding to `tcapi::cplx_ten_t<TenT>`.

Here, TenKind denotes any Python tensor-kind descriptor. The function tcapi.ten_kind(ctx, tensor) returns the TenKind associated with a raw backend tensor in the context ctx. The module-level attribute tcapi.supported_ten_kinds is an implementation-wide tuple containing the TenKind descriptors provided by the TCAPI implementation, in implementation-defined order. Compatibility with a particular execution context remains backend dependent. The element dtype associated with each descriptor can be obtained using tcapi.ten_kind_dtype.

A Python TCAPI context is an implementation-defined run-time handle passed to context-dependent TCAPI operations, usually as the first argument. Its creation and destruction are specified by tcapi.create_context and tcapi.destroy_context.

Python TCAPI does not prescribe dedicated container classes for shapes, coordinates, bond indices, or bond labels. Instead, it uses ordinary Python sequences, such as tuples, lists, or implementation-defined sequencelike objects accepted by the backend.

Tensor shapes and coordinate tuples are Python sequences of non-negative integers in bond-index order. Bond indices are Python integers, while bond-index lists and bond-label lists are Python sequences of integers. String labels are accepted by operations that define a string-label API.

Element values and scalar return values are represented by ordinary Python scalar values or by backend-native scalar values consistent with the relevant TenKind.

2. Core functions

Building on the type system described above, TCAPI exposes a suite of core functions that cover the most common tensor operations. Operation semantics are shared across language interfaces, while each language exposes them through its own signatures and calling conventions. These functions are organized into six categories: (1) read-only queries, which provide access to tensor metadata and single-element values (e.g., order, shape, and element lookup); (2) construction and destruction routines, which allocate, initialize, and release tensor storage; (3) input/output (I/O) facilities for loading tensors from and saving tensors to the file system; (4) tensor-manipulation operations, such as reshaping, transposition, and concatenation; (5) tensor linear-algebra operations, including contractions and decompositions; and (6) miscellaneous routines, which provide auxiliary functionality that is not performance critical, such as pretty printing for debugging.

TCAPI represents scalar-valued tensor results as backend tensor objects rather than standalone scalar values. A zeroth-order tensor has empty shape, size 1, and a single empty coordinate sequence. Its sole element is read with get_elem and updated with set_elem using that empty coordinate sequence. In the examples below, this sequence is written as {} in C++ and () in Python. Operations whose definitions require selecting, reordering, inserting, removing, slicing, or partitioning existing bonds are not applicable to zeroth-order tensors unless explicitly specified by the operation.

In the C++ interface, many TCAPI routines are available in both in-place and out-of-place variants. In-place variants mutate the supplied tensor operand(s), whereas out-of-place variants leave all inputs unmodified and return the results either as a function return value for single-tensor outputs or via designated output parameters for multitensor outputs or when return-value usage would lead to name collisions.

In the Python interface, context-dependent tensor operations generally use the procedural form tcapi.function(ctx,...). Routines that do not require an existing context, such as tensor-kind introspection and context creation or query routines, use their individually specified signatures. Tensor-producing operations return raw backend tensors, multioutput operations return tuples, and constructor or loader functions that require a target tensor kind take a TenKind argument. Python TCAPI does not use C++-style output arguments. Tensor operations leave their input tensors unmodified. Unless a routine explicitly guarantees a deep copy, storage sharing between inputs and outputs is implementation defined. In the Python signature blocks below, calls are shown without

assignment; the accompanying text specifies the return value of each call, such as a tensor, a tuple, a scalar, or no value, and the examples show any required assignment or rebinding.

TCAPI treats a diagonal tensor as an ordinary second-order tensor with shape $\{N, N\}$ whose off-diagonal logical tensor elements are zero. Diagonal tensors have the same public tensor type or tensor kind as other tensors with the same backend family and element type; implementations may store them compactly or densely. Unless stated otherwise, TCAPI operations are defined by logical tensor elements, not by storage representation.

For each function, we provide a minimal usage example following the function signature and description. Throughout the examples, `Ten` denotes either a `C++` tensor type or a `Python` `TenKind` with 32-bit real floating-point elements, and `CplxTen` denotes its complex counterpart. In `C++`, this means a concrete tensor type with element type `float`; in `Python`, this means an implementation-provided `TenKind` with a backend-native 32-bit real floating-point `dtype`.

a. Read-only queries

This subsection specifies read-only query functions provided by TCAPI, which retrieve tensor metadata and individual elements without modifying the tensor.

order

C++:

```cpp
template <typename TenT>
tcapi::order_t<TenT> order(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a
);
```

Python:

```python
tcapi.order(ctx, a)
```

Return the order of `a`, i.e., the number of bonds (dimensions). For a zeroth-order tensor, this function returns 0. In the `Python` interface, the returned value is an integerlike value.

Example

C++:

```cpp
// a is a 3rd-order tensor
auto ord = tcapi::order(ctx, a);
// ord == 3
```

Python:

```python
# a is a 3rd-order tensor
ord = tcapi.order(ctx, a)
# ord == 3
```

shape

C++:

```cpp
template <typename TenT>
tcapi::shape_t<TenT> shape(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a
);
```

Python:

```python
tcapi.shape(ctx, a)
```

Return the shape (the bond dimensions) of `a`, listed in bond-index order. For a zeroth-order tensor, this function returns the empty shape, i.e., the empty sequence of bond dimensions. In the `Python` interface, the returned value is a sequence of integerlike values.

Example

C++:

```cpp
// a is a 3rd-order tensor
// with the shape {3, 4, 2}
auto s = tcapi::shape(ctx, a);
// s[0] == 3; s[1] == 4; s[2] == 2
```

Python:

```python
# a is a 3rd-order tensor
# with the shape (3, 4, 2)
s = tcapi.shape(ctx, a)
# s[0] == 3; s[1] == 4; s[2] == 2
```

size

C++:

```cpp
template <typename TenT>
tcapi::ten_size_t<TenT> size(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a
);
```

Python:

```python
tcapi.size(ctx, a)
```

Return the total number of tensor elements in `a`. For a zeroth-order tensor, this function returns 1. For a diagonal tensor with shape $\{N, N\}$, this value is N^2, independent of whether the tensor is stored compactly. In the `C++` interface, this is consistent with the semantics of `std::size`. In the `Python` interface, the returned value is an integerlike value.

Example

C++:

```cpp
// a is a 3rd-order tensor
// with the shape {3, 4, 2}
auto n = tcapi::size(ctx, a);
// n == 24
```

Python:

```python
# a is a 3rd-order tensor
# with the shape (3, 4, 2)
n = tcapi.size(ctx, a)
# n == 24
```

size_bytes

C++:

```cpp
template <typename TenT>
std::size_t size_bytes(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a
);
```

Python:

```python
tcapi.size_bytes(ctx, a)
```

Return the physical memory footprint of `a` in bytes. A compactly stored diagonal tensor may use fewer bytes than a dense tensor with the same shape. In the `Python` interface, the returned value is an integerlike value.

Example

C++:

```cpp
// a is a 3rd-order float32 tensor
// with the shape {3, 4, 2}
auto bytes = tcapi::size_bytes(ctx, a);
// bytes == 96
```

Python:

```python
# a is a 3rd-order float32 tensor
# with the shape (3, 4, 2)
num_bytes = tcapi.size_bytes(ctx, a)
# num_bytes == 96
```

get_elem

C++:

```
template <typename TenT>
tcapi::elem_t<TenT> get_elem(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::elem_coors_t<TenT> &coors
);
```

Python:

```
tcapi.get_elem(ctx, a, coors)
```

Return the element value of `a` at the zero-based coordinate tuple `coors`, specified in bond-index order. For a zeroth-order tensor, `coors` is the empty coordinate sequence, and the function returns the sole element. For a diagonal tensor, off-diagonal elements return zero. In the `Python` interface, `coors` is a `Python` sequence, and the returned value is a scalar value consistent with the relevant `TenKind`.

Example

C++:

```
// a is a 3-by-3 identity matrix
auto v = tcapi::get_elem(ctx, a, {1, 1});
// v == 1.0
v = tcapi::get_elem(ctx, a, {0, 1});
// v == 0.0
```

Python:

```
# a is a 3-by-3 identity matrix
v = tcapi.get_elem(ctx, a, (1, 1))
# v == 1.0
v = tcapi.get_elem(ctx, a, (0, 1))
# v == 0.0
```

b. Construction and destruction

This subsection specifies core routines for allocating, initializing, copying, moving, and releasing tensor objects. Tensor element type and precision are selected through `TenT` in the `C++` interface and `TenKind` in the `Python` interface. In the `Python` interface, constructors that create a tensor of a specified kind take a `TenKind` argument and return raw backend tensors. `C++`-specific ownership routines that do not have a `Python` analog are identified explicitly.

allocate

C++:

```
template <typename TenT>
TenT allocate(
    tcapi::context_handle_t<TenT> &ctx,
    const tcapi::shape_t<TenT> &shape
);
```

Python:

```
tcapi.allocate(ctx, Ten, shape)
```

Allocate and return a tensor with the specified shape `shape`. The empty shape is valid; in that case, the function returns a zeroth-order tensor whose sole element is uninitialized. Storage is allocated, but tensor elements are intentionally left *uninitialized* to avoid initialization overhead. Consequently, element values are indeterminate until they are explicitly assigned. In the `Python` interface, `Ten` is a `TenKind`, `shape` is a `Python` sequence, and the returned value is a raw backend tensor.

Example

C++:

```
auto a = tcapi::allocate<Ten>(ctx, {3, 4, 2});
auto v = tcapi::get_elem(ctx, a, {0, 2, 1});
// the value of v is indeterminate
```

Python:

```
a = tcapi.allocate(ctx, Ten, (3, 4, 2))
v = tcapi.get_elem(ctx, a, (0, 2, 1))
# the value of v is indeterminate
```

zeros

C++:

```
template <typename TenT>
TenT zeros(
    tcapi::context_handle_t<TenT> &ctx,
    const tcapi::shape_t<TenT> &shape
);
```

Python:

```
tcapi.zeros(ctx, Ten, shape)
```

Allocate and return a tensor of shape `shape` with all elements initialized to zero. The empty shape is valid; in that case, the function returns a zeroth-order tensor whose sole element is initialized to zero. In the `Python` interface, `Ten` is a `TenKind`, `shape` is a `Python` sequence, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::zeros<Ten>(ctx, {3, 4, 2});
auto v = tcapi::get_elem(ctx, a, {0, 2, 1});
// v == 0.0
```

Python:

```python
a = tcapi.zeros(ctx, Ten, (3, 4, 2))
v = tcapi.get_elem(ctx, a, (0, 2, 1))
# v == 0.0
```

assign_from_range

C++:

```cpp
template <typename TenT,
          typename RandomIt,
          typename Func>
TenT assign_from_range(
    tcapi::context_handle_t<TenT> &ctx,
    const tcapi::shape_t<TenT> &shape,
    RandomIt first,
    Func &&coors2idx
);
```

Python:

```python
tcapi.assign_from_range(
    ctx, Ten, shape, values, coors2idx
)
```

Construct and return a tensor of shape `shape` by assigning its elements from an input range. In `C++`, the range is a random-access range starting at `first`; in `Python`, it is the sequence `values`. The element at coordinate tuple `coors` is initialized as `*(first + coors2idx(coors))` in `C++` or `values[coors2idx(coors)]` in `Python`. The empty shape is valid; in that case, the function returns a zeroth-order tensor and invokes the coordinate-to-index map once with the empty coordinate sequence.

In the `Python` interface, `Ten` is a `TenKind`, `shape` is a `Python` sequence, `values` is a `Python` sequence, `coors2idx` is a callable, and the returned value is a raw backend tensor. In `C++`, the mapping function `coors2idx` must have the signature

```cpp
std::iterator_traits<
        RandomIt
>::difference_type coors2idx(
    const tcapi::elem_coors_t<TenT> &coors
);
```

and must map valid coordinate tuples to zero-based indices in the range.

Example

C++:

```cpp
using Elem = tcapi::elem_t<Ten>;
using ElemVec = std::vector<Elem>;
std::iterator_traits<
    typename ElemVec::iterator
>::difference_type coors2idx(
    const tcapi::elem_coors_t<Ten> &coors
) {
  return 3 * coors[0] + coors[1];
}
ElemVec els {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
auto a = tcapi::assign_from_range<Ten>(
    ctx, {2, 3}, els.begin(), coors2idx
);
auto el = tcapi::get_elem(ctx, a, {1, 1});
// el == 5.0
```

Python:

```python
def coors2idx(coors):
    return 3 * coors[0] + coors[1]

values = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
a = tcapi.assign_from_range(
    ctx, Ten, (2, 3), values, coors2idx
)
el = tcapi.get_elem(ctx, a, (1, 1))
# el == 5.0
```

random

C++:

```cpp
template <typename TenT, typename RandNumGen>
TenT random(
    tcapi::context_handle_t<TenT> &ctx,
    const tcapi::shape_t<TenT> &shape,
    RandNumGen &gen
);
```

Python:

```python
tcapi.random(ctx, Ten, shape, gen)
```

Construct and return a tensor of shape `shape` whose elements are generated by repeatedly invoking the random number generator `gen`. The empty shape is valid; in that case, the function returns a zeroth-order tensor and invokes the random-number generator once. Each call to `gen()` must return a value convertible to `tcapi::elem_t<TenT>` in `C++` or consistent with `tcapi.ten_kind_dtype(Ten)` in the `Python` interface. In the `Python` interface, `Ten` is a `TenKind`, `shape` is a `Python` sequence, `gen` is a callable, and the returned value is a raw backend tensor.

Example

C++:

```cpp
std::mt19937 engine;
std::uniform_real_distribution<float> dis(
  0.0, 1.0
);
auto gen = [&dis, &engine]() {
  return dis(engine);
};
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
auto el = tcapi::get_elem(ctx, a, {1, 2, 0});
// el is generated by gen and lies in
// [0.0, 1.0)
```

Python:

```python
import random

def gen():
    return random.random()

a = tcapi.random(ctx, Ten, (3, 4, 2), gen)
el = tcapi.get_elem(ctx, a, (1, 2, 0))
# el is generated by gen and lies in [0.0, 1.0)
```

eye

C++:

```cpp
template <typename TenT>
TenT eye(
    tcapi::context_handle_t<TenT> &ctx,
    const tcapi::bond_dim_t<TenT> N
);
```

Python:

```python
tcapi.eye(ctx, Ten, N)
```

Construct and return the $N \times N$ identity tensor as a diagonal tensor, with ones on the main diagonal and zeros elsewhere. Implementations may store the result compactly when supported by the underlying TCF. In the `Python` interface, `Ten` is a `TenKind`, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::eye<Ten>(ctx, 3);
auto el = tcapi::get_elem(ctx, a, {1, 1});
// el == 1.0
el = tcapi::get_elem(ctx, a, {1, 2});
// el == 0.0
```

Python:

```python
a = tcapi.eye(ctx, Ten, 3)
el = tcapi.get_elem(ctx, a, (1, 1))
# el == 1.0
el = tcapi.get_elem(ctx, a, (1, 2))
# el == 0.0
```

fill

C++:

```cpp
template <typename TenT>
TenT fill(
    tcapi::context_handle_t<TenT> &ctx,
    const tcapi::shape_t<TenT> &shape,
    const tcapi::elem_t<TenT> v
);
```

Python:

```python
tcapi.fill(ctx, Ten, shape, v)
```

Construct and return a tensor of shape `shape` with all elements initialized to `v`. The empty shape is valid; in that case, the function returns a zeroth-order tensor whose sole element is initialized to `v`. In the `Python` interface, `Ten` is a `TenKind`, `shape` is a `Python` sequence, `v` is a scalar value consistent with `tcapi.ten_kind_dtype(Ten)`, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::fill<Ten>(ctx, {3, 2, 4}, 2.0);
auto el = tcapi::get_elem(ctx, a, {0, 1, 3});
// el == 2.0
```

Python:

```python
a = tcapi.fill(ctx, Ten, (3, 2, 4), 2.0)
el = tcapi.get_elem(ctx, a, (0, 1, 3))
# el == 2.0
```

copy

C++:

```
template <typename TenT>
TenT copy(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &orig
);
```

Python:

```
tcapi.copy(ctx, orig)
```

Construct and return a *deep* copy of `orig`. The returned tensor has the same shape and element values as `orig`, and the two tensors do not share memory. In the `Python` interface, the returned value is a raw backend tensor.

Example

C++:

```
Ten a;
// perform operations on a
auto b = tcapi::copy(ctx, a);
// b == a
```

Python:

```
a = tcapi.zeros(ctx, Ten, (3, 4, 2))
# perform operations on a
b = tcapi.copy(ctx, a)
# b has the same shape and element values as a
```

move

C++:

```
template <typename TenT>
TenT move(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &from
);
```

Python: No `Python` interface is specified for `move`.

Transfer the data storage and metadata owned by `from` to a newly returned tensor. No deep copy is performed. Upon return, `from` is placed in a valid, default-constructed state (`TenT{}`). This is a `C++` ownership utility; `Python` tensor lifetime and ownership follow normal `Python` and backend-library semantics.

Example

C++:

```
Ten a;
// perform operations on a
auto a_cpy = tcapi::copy(ctx, a);
auto b = tcapi::move(ctx, a);
// b == a_cpy
// a == Ten{}
```

clear

C++:

```
template <typename TenT>
void clear(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &a
);
```

Python: No `Python` interface is specified for `clear`.

Reset `a` to an empty, default-constructed state, releasing its data storage and associated metadata. Upon return, `a == TenT{}`. This is a `C++` ownership utility; `Python` tensor lifetime and ownership follow normal `Python` and backend-library semantics.

Example

C++:

```
Ten a;
// perform operations on a
tcapi::clear(ctx, a);
// a == Ten{}
```

c. Input and output (I/O)

This subsection specifies routines for serializing tensors to persistent storage and deserializing them into memory.

load

C++:

```
template <typename TenT, typename Storage>
TenT load(
    tcapi::context_handle_t<TenT> &ctx,
    Storage &&strg
);
```

Python:

```
tcapi.load(ctx, Ten, strg)
```

Deserialize a tensor from the storage location identified by `strg` and return it. In `C++`, the type of `strg` is represented by the template parameter `Storage`; in `Python`, `strg` denotes a pathlike or handlelike object that identifies a file on the host file system. `C++` implementations are required to accept stringlike inputs, such as `std::string`, `std::string_view`, or `const char*`, and may additionally support `std::filesystem::path` or other storage abstractions. `Python` implementations are required to accept stringlike inputs, such as `str`, and may additionally support `pathlib.Path` or other storage abstractions. In the `Python` interface, `Ten` is a `TenKind`, and the returned value is a raw backend tensor. The on-disk format is backend-defined and must be compatible with the requested tensor kind and the execution context `ctx`.

Example

C++:

```cpp
// ./a.ten is a file on disk
// containing the serialized representation
// of a Ten tensor
auto a = tcapi::load<Ten>(ctx, "./a.ten");
```

Python:

```python
# ./a.ten is a file on disk
# containing the serialized representation
# of a tensor compatible with Ten
a = tcapi.load(ctx, Ten, "./a.ten")
```

save

C++:

```cpp
template <typename TenT, typename Storage>
void save(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    Storage &&strg
);
```

Python:

```python
tcapi.save(ctx, a, strg)
```

Serialize the tensor `a` and write it to the storage location identified by `strg`. In `C++`, the type of `strg` is represented by the template parameter `Storage`; in `Python`, `strg` denotes a pathlike or handlelike object that identifies a file on the host file system. `C++` implementations are required to accept stringlike inputs, such as `std::string`, `std::string_view`, or `const char*`, and may additionally support `std::filesystem::path` or other storage abstractions. `Python` implementations are required to accept stringlike inputs, such as `str`, and may additionally support `pathlib.Path` or other storage abstractions. This function is the counterpart to `load`.

Example

C++:

```
Ten a;
// do some operations on a
tcapi::save(ctx, a, "./a.ten");
// the file "./a.ten" is created on disk
```

Python:

```
a = tcapi.zeros(ctx, Ten, (3, 4, 2))
# do some operations on a
tcapi.save(ctx, a, "./a.ten")
# the file "./a.ten" is created on disk
```

d. Tensor-manipulation operations

This subsection specifies tensor-manipulation routines that operate on tensor structure or elementwise values without invoking tensor linear algebra. In the `Python` interface, tensor-manipulation routines return their results as raw backend tensors. Unless otherwise stated, TCAPI tensor-manipulation routines operate on tensor elements rather than stored elements. Implementations may preserve diagonal storage when the result remains diagonal and may densify when the result cannot be represented as a diagonal tensor.

set_elem

C++:

```
template <typename TenT>
void set_elem(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &a,
    const tcapi::elem_coors_t<TenT> &coors,
    const tcapi::elem_t<TenT> el
);
```

Python:

```
tcapi.set_elem(ctx, a, coors, el)
```

Assign the value `el` to the element at coordinates `coors`. For a zeroth-order tensor, `coors` is the empty coordinate sequence, and the function updates the sole element. In the `C++` interface, this operation updates `a` in place. In the `Python` interface, `coors` is a `Python` sequence, `el` is a scalar value consistent with the relevant `TenKind`, and the returned value is a raw backend tensor with the specified element updated.

Example

C++:

```cpp
auto a = tcapi::zeros<Ten>(ctx, {3, 4, 2});
tcapi::set_elem(ctx, a, {2, 1, 0}, 1.0);
auto v = tcapi::get_elem(ctx, a, {2, 1, 0});
// v == 1.0
```

Python:

```python
a = tcapi.zeros(ctx, Ten, (3, 4, 2))
a = tcapi.set_elem(ctx, a, (2, 1, 0), 1.0)
v = tcapi.get_elem(ctx, a, (2, 1, 0))
# v == 1.0
```

reshape

C++:

```cpp
template <typename TenT>
void reshape(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const tcapi::shape_t<TenT> &new_shape
);                                          // (1)

template <typename TenT>
void reshape(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const tcapi::shape_t<TenT> &new_shape,
    TenT &out
);                                          // (2)
```

Python:

```python
tcapi.reshape(ctx, a, new_shape)
```

Reshape a tensor to `new_shape`, with the total number of logical tensor elements unchanged. This operation also converts between the empty shape and any shape with total size 1, such as {1} or {1, 1}, because the number of logical elements is preserved. No element reordering, transposition, or value modification occurs; the linear order of logical tensor elements is preserved. For densely stored tensors, this operation may be implemented without copying the element data. For diagonal tensors or other compact storage formats, the same logical-element semantics apply; implementations may preserve compact storage when possible or densify the tensor when the reshaped result is no longer representable compactly. The operation also covers combining consecutive bonds into a composite bond or splitting a bond into several bonds, with the relevant dimensions multiplying to the same value; nonconsecutive bonds can first be made consecutive by `transpose` before reshaping. In the `C++` interface, (1) reshapes `inout` in place and (2) reads from `in` and writes the reshaped tensor into `out`. In the `Python` interface, `new_shape` is a `Python` sequence, and the returned value is a raw backend tensor with the requested shape.

Example

C++:

```cpp
auto a = tcapi::zeros<Ten>(ctx, {3, 4, 2});
tcapi::reshape(ctx, a, {4, 2, 3});
auto s = tcapi::shape(ctx, a);
// s == {4, 2, 3}

Ten b;
tcapi::reshape(ctx, a, {2, 3, 4}, b);
auto sb = tcapi::shape(ctx, b);
// sb == {2, 3, 4}
```

Python:

```python
a = tcapi.zeros(ctx, Ten, (3, 4, 2))
a = tcapi.reshape(ctx, a, (4, 2, 3))
s = tcapi.shape(ctx, a)
# s == (4, 2, 3)

b = tcapi.reshape(ctx, a, (2, 3, 4))
sb = tcapi.shape(ctx, b)
# sb == (2, 3, 4)
```

transpose

C++:

```cpp
template <typename TenT>
void transpose(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const List<
        tcapi::bond_idx_t<TenT>
    > &new_order
);                                          // (1)

template <typename TenT>
void transpose(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const List<
        tcapi::bond_idx_t<TenT>
    > &new_order,
    TenT &out
);                                          // (2)
```

Python:

```python
tcapi.transpose(ctx, a, new_order)
```

Permute tensor bonds according to the permutation `new_order`. Element values are preserved; only their coordinates are permuted (the memory layout may also change). In the `C++` interface, (1) applies the permutation to `inout` in place and (2) reads from `in` and writes the permuted tensor into `out`. In the `Python` interface, `new_order` is a `Python` sequence of bond indices, and the returned value is a raw backend tensor with the permuted bonds.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 2, 4}, gen
);
auto el1 = tcapi::get_elem(ctx, a, {1, 0, 0});
tcapi::transpose(ctx, a, {1, 0, 2});
auto el2 = tcapi::get_elem(ctx, a, {0, 1, 0});
// el1 == el2

Ten b;
tcapi::transpose(ctx, a, {2, 1, 0}, b);
auto sb = tcapi::shape(ctx, b);
// sb == {4, 3, 2}
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 2, 4), gen)
el1 = tcapi.get_elem(ctx, a, (1, 0, 0))
a = tcapi.transpose(ctx, a, (1, 0, 2))
el2 = tcapi.get_elem(ctx, a, (0, 1, 0))
# el1 == el2
b = tcapi.transpose(ctx, a, (2, 1, 0))
sb = tcapi.shape(ctx, b)
# sb == (4, 3, 2)
```

cplx_conj

C++:

```cpp
template <typename TenT>
void cplx_conj(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout
);                                          // (1)

template <typename TenT>
void cplx_conj(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.cplx_conj(ctx, a)
```

Apply elementwise complex conjugation. In the C++ interface, (1) conjugates `inout` in place and (2) conjugates `in` and writes the result into `out`. If `tcapi::elem_t<TenT>` is real, (1) performs no change and (2) produces a deep copy. In the `Python` interface, the returned value is a raw backend tensor; for a real-valued input, the result has the same shape and element values as `a`.

Example

C++:

```
auto a = tcapi::random<CplxTen>(
    ctx, {3, 2, 4}, gen
);
CplxTen a_conj;
tcapi::cplx_conj(ctx, a, a_conj);
auto el1 = tcapi::get_elem(ctx, a, {0, 0, 0});
auto el2 = tcapi::get_elem(
  ctx, a_conj, {0, 0, 0}
);
// std::conj(el1) == el2
```

Python:

```
a = tcapi.random(ctx, CplxTen, (3, 2, 4), gen)
a_conj = tcapi.cplx_conj(ctx, a)
el1 = tcapi.get_elem(ctx, a, (0, 0, 0))
el2 = tcapi.get_elem(ctx, a_conj, (0, 0, 0))
# conjugate(el1) == el2
```

to_cplx

C++:

```
template <typename TenT>
tcapi::cplx_ten_t<TenT> to_cplx(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in
);
```

Python:

```
tcapi.to_cplx(ctx, a)
```

Return a complex-valued tensor with the same shape as `in`. Real-valued elements are embedded as complex numbers with zero imaginary parts. If `tcapi::elem_t<TenT>` is already complex, the function returns a deep copy of `in`. In the `Python` interface, the returned value is a raw backend tensor whose `TenKind` is the complex counterpart of `tcapi.ten_kind(ctx, a)`.

Example

C++:

```cpp
auto a = tcapi::eye<Ten>(ctx, 3);
auto cplx_a = tcapi::to_cplx(ctx, a);
auto el = tcapi::get_elem(ctx, cplx_a, {1, 1});
// el == 1.0 + 0.0j
```

Python:

```python
a = tcapi.eye(ctx, Ten, 3)
cplx_a = tcapi.to_cplx(ctx, a)
el = tcapi.get_elem(ctx, cplx_a, (1, 1))
# el == 1.0 + 0.0j
```

real

C++:

```cpp
template <typename TenT>
tcapi::real_ten_t<TenT> real(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in
);
```

Python:

```python
tcapi.real(ctx, a)
```

Extract the real part of `in` elementwise and return a tensor with a real-valued element type. If `tcapi::elem_t<TenT>` is already real, this function returns a deep copy of `in`. In the `Python` interface, the returned value is a raw backend tensor whose `TenKind` is the real counterpart of `tcapi.ten_kind(ctx, a)`.

Example

C++:

```cpp
auto a = tcapi::random<CplxTen>(
    ctx, {3, 2, 4}, gen
);
auto a_real = tcapi::real(ctx, a);
auto el1 = tcapi::get_elem(ctx, a, {0, 0, 0});
auto el2 = tcapi::get_elem(
  ctx, a_real, {0, 0, 0}
);
// std::real(el1) == el2
```

Python:

```
a = tcapi.random(ctx, CplxTen, (3, 2, 4), gen)
a_real = tcapi.real(ctx, a)
el1 = tcapi.get_elem(ctx, a, (0, 0, 0))
el2 = tcapi.get_elem(ctx, a_real, (0, 0, 0))
# real(el1) == el2
```

imag

C++:

```
template <typename TenT>
tcapi::real_ten_t<TenT> imag(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in
);
```

Python:

```
tcapi.imag(ctx, a)
```

Extract the imaginary part of `in` elementwise and return a tensor with a real-valued element type. If `tcapi::elem_t<TenT>` is already real, the function returns a real-valued tensor of the same shape whose elements are all zero. In the `Python` interface, the returned value is a raw backend tensor whose `TenKind` is the real counterpart of `tcapi.ten_kind(ctx, a)`.

Example

C++:

```
auto a = tcapi::random<CplxTen>(
    ctx, {3, 2, 4}, gen
);
auto a_imag = tcapi::imag(ctx, a);
auto el1 = tcapi::get_elem(ctx, a, {0, 0, 0});
auto el2 = tcapi::get_elem(
  ctx, a_imag, {0, 0, 0}
);
// std::imag(el1) == el2
```

Python:

```
a = tcapi.random(ctx, CplxTen, (3, 2, 4), gen)
a_imag = tcapi.imag(ctx, a)
el1 = tcapi.get_elem(ctx, a, (0, 0, 0))
el2 = tcapi.get_elem(ctx, a_imag, (0, 0, 0))
# imag(el1) == el2
```

expand

C++:

```cpp
template <typename TenT>
void expand(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const Map<
        tcapi::bond_idx_t<TenT>,
        tcapi::bond_dim_t<TenT>
        > &bond_idx_increment_map
);                                          // (1)

template <typename TenT>
void expand(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const Map<
        tcapi::bond_idx_t<TenT>,
        tcapi::bond_dim_t<TenT>
        > &bond_idx_increment_map,
    TenT &out
);                                          // (2)
```

Python:

```python
tcapi.expand(ctx, a, bond_idx_increment_map)
```

Expand selected bonds by increasing their sizes according to `bond_idx_increment_map`. The keys specify the bonds to expand, and the corresponding values specify the additional sizes to append along those bonds. Elements whose coordinates lie within the original bounds are preserved. Any element with at least one coordinate in an appended range is set to zero. In the `C++` interface, (1) applies the expansion to `inout` in place and (2) reads from `in` and writes the expanded tensor into `out`. In the `Python` interface, `bond_idx_increment_map` is a mapping from bond indices to bond-dimension increments, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::zeros<Ten>(ctx, {2, 2, 2});
tcapi::expand(ctx, a, {{1, 2}, {0, 1}});
auto s = tcapi::shape(ctx, a);
// s == {3, 4, 2}
auto el = tcapi::get_elem(ctx, a, {2, 3, 0});
// el == 0.0
```

Python:

```python
a = tcapi.zeros(ctx, Ten, (2, 2, 2))
a = tcapi.expand(ctx, a, {1: 2, 0: 1})
s = tcapi.shape(ctx, a)
# s == (3, 4, 2)
el = tcapi.get_elem(ctx, a, (2, 3, 0))
# el == 0.0
```

shrink

C++:

```cpp
template <typename TenT>
void shrink(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const bond_idx_elem_coor_pair_map<TenT> &
          bd_idx_el_coor_pair_map
);                                          // (1)

template <typename TenT>
void shrink(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const bond_idx_elem_coor_pair_map<TenT> &
          bd_idx_el_coor_pair_map,
    TenT &out
);                                          // (2)
```

Python:

```python
tcapi.shrink(ctx, a, bd_idx_el_coor_pair_map)
```

Shrink selected bonds by slicing them to half-open coordinate ranges [*coor_first*, *coor_second*) specified by `bd_idx_el_coor_pair_map`. Bonds not listed in `bd_idx_el_coor_pair_map` retain their original ranges (and hence their original dimensions). In the `C++` interface, (1) shrinks `inout` in place and (2) reads from `in` and writes the sliced tensor into `out`. In the `Python` interface, `bd_idx_el_coor_pair_map` is a mapping from bond indices to coordinate-pair sequences, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
auto el1 = tcapi::get_elem(ctx, a, {0, 1, 1});
tcapi::shrink(ctx,
  a,
```

```
  {{1, {1, 3}}, {0, {0, 2}}}
);
auto s = tcapi::shape(ctx, a);
// s == {2, 2, 2}
auto el2 = tcapi::get_elem(ctx, a, {0, 0, 1});
// el1 == el2
```

Python:

```
a = tcapi.random(ctx, Ten, (3, 4, 2), gen)
el1 = tcapi.get_elem(ctx, a, (0, 1, 1))
a = tcapi.shrink(
    ctx, a, {1: (1, 3), 0: (0, 2)}
)
s = tcapi.shape(ctx, a)
# s == (2, 2, 2)
el2 = tcapi.get_elem(ctx, a, (0, 0, 1))
# el1 == el2
```

extract_sub

C++:

```
template <typename TenT>
void extract_sub(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const List<
        Pair<
            tcapi::elem_coor_t<TenT>,
            tcapi::elem_coor_t<TenT>
        >> & coor_pairs
);                                          // (1)

template <typename TenT>
void extract_sub(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const List<
        Pair<
            tcapi::elem_coor_t<TenT>,
            tcapi::elem_coor_t<TenT>
        >> & coor_pairs,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.extract_sub(ctx, a, coor_pairs)
```

Extract a subtensor specified by per-bond half-open coordinate ranges [*coor_first*, *coor_second*), provided in `coor_pairs` in bond-index order. In the `C++` interface,

(1) overwrites `inout` with the extracted subtensor and (2) reads from `in` and writes the extracted subtensor into `out`. In the `Python` interface, `coor_pairs` is a `Python` sequence of coordinate-pair sequences, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
Ten sub;
tcapi::extract_sub(ctx,
  a,
  {{1, 3}, {0, 2}, {0, 2}},
  sub
);
auto el1 = tcapi::get_elem(ctx, a, {1, 0, 0});
auto el2 = tcapi::get_elem(
  ctx, sub, {0, 0, 0}
);
// el1 == el2
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 2), gen)
sub = tcapi.extract_sub(
    ctx, a, ((1, 3), (0, 2), (0, 2))
)
el1 = tcapi.get_elem(ctx, a, (1, 0, 0))
el2 = tcapi.get_elem(ctx, sub, (0, 0, 0))
# el1 == el2
```

replace_sub

C++:

```cpp
template <typename TenT>
void replace_sub(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const TenT &sub,
    const tcapi::elem_coors_t<TenT> &begin_pt
);                                          // (1)

template <typename TenT>
void replace_sub(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const TenT &sub,
    const tcapi::elem_coors_t<TenT> &begin_pt,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.replace_sub(ctx, a, sub, begin_pt)
```

Replace a subregion of a tensor with `sub`, placing the origin (all-zero coordinate) of `sub` at `begin_pt` in the target tensor. In the `C++` interface, (1) modifies `inout` in place and (2) reads from `in` and writes the updated tensor into `out`. In the `Python` interface, `begin_pt` is a `Python` sequence, and the returned value is a raw backend tensor with the specified subregion replaced.

Example

C++:

```
auto a = tcapi::zeros<Ten>(ctx, {3, 4, 2});
Ten sub = tcapi::random<Ten>(
    ctx, {2, 2, 2}, gen
);
tcapi::replace_sub(ctx,
  a,
  sub,
  {1, 2, 0}
);
auto el1 = tcapi::get_elem(ctx, a, {1, 2, 0});
auto el2 = tcapi::get_elem(
  ctx, sub, {0, 0, 0}
);
// el1 == el2
```

Python:

```
a = tcapi.zeros(ctx, Ten, (3, 4, 2))
sub = tcapi.random(ctx, Ten, (2, 2, 2), gen)
a = tcapi.replace_sub(ctx, a, sub, (1, 2, 0))
el1 = tcapi.get_elem(ctx, a, (1, 2, 0))
el2 = tcapi.get_elem(ctx, sub, (0, 0, 0))
# el1 == el2
```

concatenate

C++:

```
template <typename TenT>
TenT concatenate(
    tcapi::context_handle_t<TenT> &ctx,
    const List<CRef<TenT>> &ins,
    const tcapi::bond_idx_t<TenT> concat_bdidx
);
```

Python:

```
tcapi.concatenate(ctx, ins, concat_bdidx)
```

Concatenate the tensors in `ins` along bond `concat_bdidx` and return the result. In the `C++` interface, `ins` is a list of nonowning const references to existing tensors; the referenced tensor

objects are not copied, consumed, or modified and must remain valid for the duration of the call. All input tensors must have the same order and identical bond dimensions on every bond except `concat_bdidx`. The output dimension along `concat_bdidx` equals the sum of the corresponding input dimensions. In the `Python` interface, `ins` is a `Python` sequence of raw backend tensors, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {2, 3, 4}, gen
);
auto b = tcapi::random<Ten>(
  ctx, {2, 3, 4}, gen
);
auto c = tcapi::random<Ten>(
  ctx, {2, 3, 4}, gen
);
auto d = tcapi::concatenate<Ten>(
    ctx,
    {std::cref(a), std::cref(b), std::cref(c)},
    1
);
auto s = tcapi::shape(ctx, d);
// s == {2, 9, 4}
auto el1 = tcapi::get_elem(ctx, b, {0, 0, 0});
auto el2 = tcapi::get_elem(ctx, d, {0, 3, 0});
// el1 == el2
```

Python:

```python
a = tcapi.random(ctx, Ten, (2, 3, 4), gen)
b = tcapi.random(ctx, Ten, (2, 3, 4), gen)
c = tcapi.random(ctx, Ten, (2, 3, 4), gen)
d = tcapi.concatenate(ctx, (a, b, c), 1)
s = tcapi.shape(ctx, d)
# s == (2, 9, 4)
el1 = tcapi.get_elem(ctx, b, (0, 0, 0))
el2 = tcapi.get_elem(ctx, d, (0, 3, 0))
# el1 == el2
```

stack

C++:

```cpp
template <typename TenT>
TenT stack(
    tcapi::context_handle_t<TenT> &ctx,
    const List<CRef<TenT>> &ins,
    const tcapi::bond_idx_t<TenT> stack_bdidx
);
```

Python:

```
tcapi.stack(ctx, ins, stack_bdidx)
```

Stack the tensors in `ins`, which must all have identical shape, along a new bond inserted at index `stack_bdidx`. In the `C++` interface, `ins` is a list of nonowning const references to existing tensors; the referenced tensor objects are not copied, consumed, or modified and must remain valid for the duration of the call. If the inputs are rth-order tensors, the result is an $(r+1)$th-order tensor. The dimension of the newly inserted bond equals `ins.size()` in `C++` or `len(ins)` in `Python`. In the `Python` interface, `ins` is a `Python` sequence of raw backend tensors, and the returned value is a raw backend tensor.

Example

C++:

```
auto a = tcapi::random<Ten>(
  ctx, {2, 3, 4}, gen
);
auto b = tcapi::random<Ten>(
  ctx, {2, 3, 4}, gen
);
auto c = tcapi::random<Ten>(
  ctx, {2, 3, 4}, gen
);
auto d = tcapi::stack(
  ctx,
  {std::cref(a), std::cref(b), std::cref(c)},
  1
);
auto s = tcapi::shape(ctx, d);
// s == {2, 3, 3, 4}
auto el1 = tcapi::get_elem(ctx, b, {0, 2, 3});
auto el2 = tcapi::get_elem(
  ctx, d, {0, 1, 2, 3}
);
// el1 == el2
```

Python:

```
a = tcapi.random(ctx, Ten, (2, 3, 4), gen)
b = tcapi.random(ctx, Ten, (2, 3, 4), gen)
c = tcapi.random(ctx, Ten, (2, 3, 4), gen)
d = tcapi.stack(ctx, (a, b, c), 1)
s = tcapi.shape(ctx, d)
# s == (2, 3, 3, 4)
el1 = tcapi.get_elem(ctx, b, (0, 2, 3))
el2 = tcapi.get_elem(ctx, d, (0, 1, 2, 3))
# el1 == el2
```

for_each

C++:

```cpp
template <typename TenT, typename Func>
void for_each(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    Func &&f
);                                          // (1)

template <typename TenT, typename Func>
void for_each(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    Func &&f
);                                          // (2)
```

Python:

```python
tcapi.for_each(ctx, a, f)
```

Visit every element exactly once and apply `f`. The traversal order is implementation defined. In the `C++` interface, `Func` must be invocable with the corresponding signature. (1) In-place (mutating) traversal: `f` is invoked as `f(elem)`, where `elem` has type `tcapi::elem_t<TenT>&`. (2) Read-only *const* traversal: `f` is invoked as `f(elem)`, where `elem` has type `const tcapi::elem_t<TenT>` (or an equivalent const-qualified form, such as `const tcapi::elem_t<TenT>&`). In the `Python` interface, `f` is invoked as `f(el)`, where `el` is a scalar value consistent with the relevant `TenKind`. If `f(el)` returns `None`, the current element is copied unchanged into the result. Otherwise, the returned value is used as the replacement element value. `tcapi.for_each` returns a raw backend tensor with the resulting element values. The input tensor `a` is not modified.

Example

C++:

```cpp
auto a = tcapi::eye<Ten>(ctx, 3);
float tot = 0.0;
auto sum = [&tot](const float el) {
  tot += el;
};
const Ten &ca = a;
tcapi::for_each(ctx, ca, sum);  // call (2)
// tot == 3.0
auto plus_one = [](float &el) {
 el += 1.0;
};
tcapi::for_each(ctx, a, plus_one);  // call (1)
auto el = tcapi::get_elem(ctx, a, {1, 1});
// el == 2.0
```

Python:

```python
a = tcapi.eye(ctx, Ten, 3)
tot = {"value": 0.0}

def sum_elem(el):
    tot["value"] += el
    return None

tcapi.for_each(ctx, a, sum_elem)
# tot["value"] == 3.0

def plus_one(el):
    return el + 1.0

a = tcapi.for_each(ctx, a, plus_one)
el = tcapi.get_elem(ctx, a, (1, 1))
# el == 2.0
```

for_each_with_coors

C++:

```cpp
template <typename TenT, typename Func>
void for_each_with_coors(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    Func &&f
);                                          // (1)

template <typename TenT, typename Func>
void for_each_with_coors(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    Func &&f
);                                          // (2)
```

Python:

```python
tcapi.for_each_with_coors(ctx, a, f)
```

This function is similar to `for_each`, but invokes `f` with both the element and its coordinates. For a zeroth-order tensor, the callable is invoked once with the empty coordinate sequence. The traversal order is implementation defined. In the `C++` interface, the callable `f` is invoked as `f(elem, coors)`, where `coors` has type `const tcapi::elem_coors_t<TenT>&`. In the `Python` interface, `f` is invoked as `f(el, coors)`, where `el` is a scalar value consistent with the relevant `TenKind`, and `coors` is a `Python` coordinate sequence in bond-index order. If `f(el, coors)` returns `None`, the current element is copied unchanged into the result. Otherwise, the returned value is used as the replacement element value. `tcapi.for_each_with_coors` returns a raw backend tensor with the resulting element values. The input tensor `a` is not modified.

Example

C++:

```cpp
auto a = tcapi::eye<Ten>(ctx, 3);
auto plus_coor0 = [](
    float &el,
    const tcapi::elem_coors_t<Ten> &coors
) {
  el += coors[0];
};
tcapi::for_each_with_coors(
    ctx, a, plus_coor0
);  // call (1)
auto el = tcapi::get_elem(ctx, a, {1, 1});
// el == 2.0
```

Python:

```python
a = tcapi.eye(ctx, Ten, 3)

def plus_coor0(el, coors):
    return el + coors[0]

a = tcapi.for_each_with_coors(
    ctx, a, plus_coor0
)
el = tcapi.get_elem(ctx, a, (1, 1))
# el == 2.0
```

e. Tensor linear-algebra operations

This subsection specifies tensor linear-algebra operations provided by TCAPI. These routines operate on higher-order tensors by temporarily matricizing them according to a user-specified bond partition. They then apply standard linear-algebra kernels and fold the results back into tensor form. The functionality includes norms, normalization, scalar scaling, matrix functions, factorizations, and eigensolvers. Unless otherwise noted, dimensional compatibility conditions required by the underlying matrix operations must be satisfied. In the `Python` interface, tensor results are returned as raw backend tensors. Multioutput routines return tuples.

diag

C++:

```cpp
template <typename TenT>
void diag(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout
);                                          // (1)

template <typename TenT>
void diag(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    TenT &out
);                                          // (2)
```

Python:

```python
tcapi.diag(ctx, a)
```

If the input is first-order with shape $\{N\}$, the routine returns a second-order diagonal tensor with shape $\{N, N\}$, placing the input elements on the main diagonal and setting all off-diagonal elements to zero. Implementations may store the result compactly when supported by the underlying TCF. If the input is second-order with shape $\{M, N\}$, the routine extracts the main diagonal as a first-order tensor of length $\min(M, N)$. The input tensor must be either first- or second-order. In the `C++` interface, (1) modifies `inout` in place, and (2) reads from `in` and writes the result into `out`. In the `Python` interface, the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::eye<Ten>(ctx, 3);
auto a_cpy = tcapi::copy(ctx, a);
tcapi::diag(ctx, a);
auto r = tcapi::order(ctx, a);
// r == 1
tcapi::diag(ctx, a);
// a == a_cpy
```

Python:

```python
a = tcapi.eye(ctx, Ten, 3)
a_cpy = tcapi.copy(ctx, a)
a = tcapi.diag(ctx, a)
r = tcapi.order(ctx, a)
# r == 1
a = tcapi.diag(ctx, a)
# a == a_cpy
```

norm

C++:

```
template <typename TenT>
tcapi::real_t<TenT> norm(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a
);
```

Python:

```
tcapi.norm(ctx, a)
```

Return the Frobenius norm of `a`. In the `C++` interface, the returned value has type `tcapi::real_t<TenT>`. In the `Python` interface, the returned value is a scalar with the real dtype corresponding to `tcapi.ten_kind(ctx, a)`. For an rth-order tensor $A \in \mathbb{K}^{d_0 \times \cdots \times d_{r-1}}$, the Frobenius norm is

$$\|A\|_F = \sqrt{\sum_{i_0=0}^{d_0-1} \cdots \sum_{i_{r-1}=0}^{d_{r-1}-1} \left|A_{i_0 \cdots i_{r-1}}\right|^2}, \tag{C1}$$

where $|A_{i_0 \cdots i_{r-1}}|$ denotes the modulus when A is complex valued.

Example

C++:

```
auto a = tcapi::eye<Ten>(ctx, 3);
auto norm = tcapi::norm(ctx, a);
// norm == std::sqrt(3.0)
```

Python:

```
a = tcapi.eye(ctx, Ten, 3)
norm = tcapi.norm(ctx, a)
# norm == sqrt(3.0)
```

normalize

C++:

```
template <typename TenT>
tcapi::real_t<TenT> normalize(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout
);                                          // (1)

template <typename TenT>
tcapi::real_t<TenT> normalize(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.normalize(ctx, a)
```

Scale `a` to unit Frobenius norm. In the `C++` interface, (1) normalizes `inout` in place, and (2) reads from `in` and writes the normalized tensor into `out`; both overloads return the original norm as `tcapi::real_t<TenT>`. In the `Python` interface, the returned value is the tuple `(normalized_a, norm)`, where `normalized_a` is a raw backend tensor containing the normalized tensor and `norm` is the original Frobenius norm.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
tcapi::normalize(ctx, a);
auto norm = tcapi::norm(ctx, a);
// norm == 1.0

a = tcapi::random<Ten>(ctx, {3, 4, 2}, gen);
Ten b;
auto m = tcapi::normalize(ctx, a, b);
auto m2 = tcapi::norm(ctx, b);
// m2 == 1.0
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 2), gen)
a, orig_norm = tcapi.normalize(ctx, a)
norm = tcapi.norm(ctx, a)
# norm == 1.0
```

scale

C++:

```cpp
template <typename TenT>
void scale(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const tcapi::elem_t<TenT> s
);                                          // (1)

template <typename TenT>
void scale(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const tcapi::elem_t<TenT> s,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.scale(ctx, a, s)
```

Multiply a tensor by a scalar `s`. In the `C++` interface, `s` has type `tcapi::elem_t<TenT>`, (1) scales `inout` in place, and (2) reads from `in` and writes the scaled tensor into `out`. In the `Python` interface, `s` is a scalar value consistent with the dtype associated with `tcapi.ten_kind(ctx, a)`, and the returned value is a raw backend tensor.

Example

C++:

```
auto a = tcapi::eye<Ten>(ctx, 3);

tcapi::scale(ctx, a, 3.0);
auto el = tcapi::get_elem(ctx, a, {2, 2});
// el == 3.0

Ten c;
tcapi::scale(ctx, a, -2.0, c);
auto el2 = tcapi::get_elem(ctx, c, {2, 2});
// el2 == -6.0
```

Python:

```
a = tcapi.eye(ctx, Ten, 3)

a = tcapi.scale(ctx, a, 3.0)
el = tcapi.get_elem(ctx, a, (2, 2))
# el == 3.0
c = tcapi.scale(ctx, a, -2.0)
el2 = tcapi.get_elem(ctx, c, (2, 2))
# el2 == -6.0
```

trace

C++:

```
template <typename TenT>
void trace(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const bond_idx_pairs_t<TenT> &bdidx_pairs
);                                          // (1)

template <typename TenT>
void trace(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const bond_idx_pairs_t<TenT> &bdidx_pairs,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.trace(ctx, a, bdidx_pairs)
```

Perform a partial trace by summing over the bond pairs specified by `bdidx_pairs`. The type `bond_idx_pairs_t` is defined in Appendix C 1 a. In the `Python` interface, `bdidx_pairs` is a `Python` sequence of bond-index pairs.

Each paired bond must have the same dimension. If all bonds are paired, the result is a zeroth-order tensor. The relative order and dimensions of any remaining (untraced) bonds are preserved.

In the `C++` interface, (1) applies the trace to `inout` in place, and (2) reads from `in` and writes the result into `out`. In the `Python` interface, the returned value is a raw backend tensor.

Example

C++:

```
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2, 4, 2}, gen
);
tcapi::trace(ctx, a, {{1, 3}, {2, 4}});
auto s = tcapi::shape(ctx, a);
// s == {3}
```

Python:

```
a = tcapi.random(
    ctx, Ten, (3, 4, 2, 4, 2), gen
)
a = tcapi.trace(ctx, a, ((1, 3), (2, 4)))
s = tcapi.shape(ctx, a)
# s == (3,)
```

exp

C++:

```
template <typename TenT>
void exp(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const tcapi::order_t<TenT>
        num_of_bds_as_row
);                                          // (1)

template <typename TenT>
void exp(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.exp(ctx, a, num_of_bds_as_row)
```

Compute the matrix exponential by matricizing the tensor with the first `num_of_bds_as_row` bonds as row indices and the remaining bonds as column indices. The routine applies the matrix exponential to the resulting matrix and refolds the result back to the original tensor shape. The total row dimension must equal the total column dimension. In the `C++` interface, (1) modifies `inout` in place, and (2) reads from `in` and writes the result into `out`. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is a raw backend tensor.

Example

C++:

```
auto a = tcapi::eye<Ten>(ctx, 3);
tcapi::exp(ctx, a, 1);
auto el = tcapi::get_elem(ctx, a, {1, 1});
// el == 2.71828...
```

Python:

```
a = tcapi.eye(ctx, Ten, 3)
a = tcapi.exp(ctx, a, 1)
el = tcapi.get_elem(ctx, a, (1, 1))
# el == 2.71828...
```

inverse

C++:

```
template <typename TenT>
void inverse(
    tcapi::context_handle_t<TenT> &ctx,
    TenT &inout,
    const tcapi::order_t<TenT>
        num_of_bds_as_row
);                                          // (1)

template <typename TenT>
void inverse(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &in,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &out
);                                          // (2)
```

Python:

```
tcapi.inverse(ctx, a, num_of_bds_as_row)
```

Compute the matrix inverse by matricizing the tensor with the first `num_of_bds_as_row` bonds as row indices and the remaining bonds as column indices. The routine forms the corresponding

square matrix, inverts it, and refolds the result back to the original tensor shape. The total row dimension must equal the total column dimension, and the resulting matrix must be invertible. In the C++ interface, (1) modifies `inout` in place, and (2) reads from `in` and writes the result into `out`. In the Python interface, `num_of_bds_as_row` is an integerlike value, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::eye<Ten>(ctx, 3);
tcapi::inverse(ctx, a, 1);
auto el = tcapi::get_elem(ctx, a, {1, 1});
// el == 1.0
```

Python:

```python
a = tcapi.eye(ctx, Ten, 3)
a = tcapi.inverse(ctx, a, 1)
el = tcapi.get_elem(ctx, a, (1, 1))
# el == 1.0
```

contract

C++:

```cpp
template <typename TenT>
void contract(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const List<
        tcapi::bond_label_t<TenT>
        > &bd_labs_a,
    const TenT &b,
    const List<
        tcapi::bond_label_t<TenT>
        > &bd_labs_b,
    TenT &c,
    const List<
        tcapi::bond_label_t<TenT>
        > &bd_labs_c
);                                          // (1)

template <typename TenT>
void contract(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const std::string_view bd_labs_str_a,
    const TenT &b,
    const std::string_view bd_labs_str_b,
    TenT &c,
    const std::string_view bd_labs_str_c
);                                          // (2)
```

Python:

```
tcapi.contract(
    ctx, a, bd_labs_a, b, bd_labs_b,
    bd_labs_c
)
```

This routine performs a label-based Einstein contraction. Let the label lists for `a`, `b`, and the result be α, β, and γ, respectively. All labels that appear in both α and β but not in γ are contracted (summed over). The labels in γ define both the free bonds of the result and their ordering. Bond dimensions associated with identical labels must agree. In the `C++` interface, (1) provides labels as integerlike values of type `tcapi::bond_label_t<TenT>` and (2) provides labels as strings (e.g., `‘‘ijk’’`), where each character is interpreted as a label. In the `Python` interface, labels may be provided as sequences of integers or, for the string-label API, as strings; the returned value is a raw backend tensor. If γ is empty, the result is a zeroth-order tensor. This API guarantees correctness even when the `C++` output tensor `c` aliases `a`, `b`, or both. Repeating a label within a single operand (i.e., an implicit trace within `a` or `b`) is not supported in the current specification; use `trace` first if needed.

Example

C++:

```
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
auto b = tcapi::random<Ten>(
  ctx, {2, 4, 5}, gen
);
Ten c;
tcapi::contract(ctx,
  a, {1, -1, -2},
  b, {-2, -1, 0},
  c, {0, 1}
);
// which is equivalent to
tcapi::contract(ctx,
  a, "ijk", b, "kjl",
  c, "li"
);
auto s = tcapi::shape(ctx, c);
// s == {5, 3}
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 2), gen)
b = tcapi.random(ctx, Ten, (2, 4, 5), gen)
c = tcapi.contract(
    ctx,
    a, (1, -1, -2),
    b, (-2, -1, 0),
    (0, 1),
)
# which is equivalent to
c = tcapi.contract(
    ctx, a, "ijk", b, "kjl", "li"
)
s = tcapi.shape(ctx, c)
# s == (5, 3)
```

linear_combine

C++:

```cpp
template <typename TenT>
TenT linear_combine(
    tcapi::context_handle_t<TenT> &ctx,
    const List<CRef<TenT>> &ins
);                                          // (1)

template <typename TenT>
TenT linear_combine(
    tcapi::context_handle_t<TenT> &ctx,
    const List<CRef<TenT>> &ins,
    const List<tcapi::elem_t<TenT>> &coefs
);                                          // (2)
```

Python:

```python
tcapi.linear_combine(ctx, ins)
tcapi.linear_combine(ctx, ins, coefs)
```

Form and return the linear combination $\sum_{i=0}^{m-1} s_i A_i$, where $\{A_i\} = \texttt{ins}$ and $\{s_i\} = \texttt{coefs}$. In overload (1), all coefficients are implicitly set to $s_i = 1$. All input tensors A_i must have identical shapes. In the `C++` interface, `ins` is a list of nonowning const references to existing tensors; the referenced tensor objects are not copied, consumed, or modified and must remain valid for the duration of the call. In the `Python` interface, `ins` is a `Python` sequence of raw backend tensors, `coefs` is a `Python` sequence of scalar values consistent with the input tensor kind, and the returned value is a raw backend tensor.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
auto b = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
auto c = tcapi::random<Ten>(
  ctx, {3, 4, 2}, gen
);
auto d = tcapi::linear_combine(ctx,
  {std::cref(a), std::cref(b), std::cref(c)},
  {1.0, 2.0, 3.0}
);
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 2), gen)
b = tcapi.random(ctx, Ten, (3, 4, 2), gen)
c = tcapi.random(ctx, Ten, (3, 4, 2), gen)
d = tcapi.linear_combine(
    ctx,
    (a, b, c),
    (1.0, 2.0, 3.0),
)
```

svd

C++:

```cpp
template <typename TenT>
void svd(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &u,
    tcapi::real_ten_t<TenT> &sigma,
    TenT &v_dag
);
```

Python:

```python
tcapi.svd(ctx, a, num_of_bds_as_row)
```

Perform an SVD of an rth-order tensor `a` by first matricizing it into a matrix. Let `shape(a)` $= \{d_0, \ldots, d_{r-1}\}$ and $k =$ `num_of_bds_as_row` with $1 \leqslant k < r$. The tensor is matricized by grouping

the first k bonds into the row index and the remaining bonds into the column index:

$$A' \in \mathbb{K}^{I \times J}, \quad I = \prod_{b=0}^{k-1} d_b, \quad J = \prod_{b=k}^{r-1} d_b. \tag{C2}$$

An SVD

$$A' = U \Sigma V^{\dagger} \tag{C3}$$

is then computed, where $\Sigma = \mathrm{diag}(s_0, \ldots, s_{\kappa-1})$, with $s_0 \geqslant s_1 \geqslant \ldots \geqslant s_{\kappa-1} \geqslant 0$ and $\kappa = \min(I, J)$. Finally, the factors are folded back into tensors: `shape(u)` $= \{d_0, \ldots, d_{k-1}, \kappa\}$, `shape(sigma)` $= \{\kappa, \kappa\}$, and `shape(v_dag)` $= \{\kappa, d_k, \ldots, d_{r-1}\}$. `sigma` is a real second-order diagonal tensor corresponding to Σ; its diagonal elements are the singular values in nonincreasing order, and all off-diagonal elements are zero. The first k bonds of `u` match the first k bonds of `a`, and the remaining $r - k$ bonds of `v_dag` match the remaining bonds of `a`. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is the tuple `(u, sigma, v_dag)` of raw backend tensors.

Example

C++:

```
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 12}, gen
);
Ten u, sigma, v_t;
tcapi::svd(ctx, a, 2, u, sigma, v_t);
```

Python:

```
a = tcapi.random(ctx, Ten, (3, 4, 12), gen)
u, sigma, v_t = tcapi.svd(ctx, a, 2)
```

trunc_svd

C++:

```cpp
template <typename TenT>
void trunc_svd(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &u,
    tcapi::real_ten_t<TenT> &sigma,
    TenT &v_dag,
    tcapi::real_t<TenT> &trunc_err,
    const tcapi::bond_dim_t<TenT> chi_max,
    const tcapi::real_t<TenT> s_min
);                                          // (1)

template <typename TenT>
void trunc_svd(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &u,
    tcapi::real_ten_t<TenT> &sigma,
    TenT &v_dag,
    tcapi::real_t<TenT> &trunc_err,
    const tcapi::bond_dim_t<TenT> chi_max,
    const tcapi::real_t<TenT> s_min,
    const tcapi::bond_dim_t<TenT> chi_min,
    const tcapi::real_t<TenT> target_trunc_err
);                                          // (2)
```

Python:

```python
tcapi.trunc_svd(
    ctx, a, num_of_bds_as_row, chi_max, s_min
)
tcapi.trunc_svd(
    ctx, a, num_of_bds_as_row,
    chi_max, s_min, chi_min, target_trunc_err
)
```

This routine follows the same matricization and SVD convention as `svd`, but additionally truncates the singular values according to a prescribed strategy. For pretruncation singular values $s_0 \geqslant \cdots \geqslant s_{\kappa-1}$, define the (relative) truncation error for keeping χ values as

$$\epsilon = \frac{\sum_{i=\chi}^{\kappa-1} s_i^2}{\sum_{i=0}^{\kappa-1} s_i^2}, \tag{C4}$$

which is returned in `trunc_err`. (1) Fixed maximum bond dimension kept: keep at most `chi_max` singular values after discarding those below `s_min`. Equivalently, (2) with `chi_min` $= 1$ and `target_trunc_err` $= 0$. (2) General truncation strategy:

(1) Discard all $s_i <$ `s_min`.

(2) Among the singular values that survive the `s_min` cutoff, retain at least `chi_min` values when possible. If fewer than `chi_min` values survive the cutoff, retain all surviving values; values below `s_min` are not restored to satisfy `chi_min`.

(3) Increase χ (retaining additional values in descending order) until either $\epsilon \leqslant$ `target_trunc_err` or $\chi =$ `chi_max`.

The output `sigma` is a second-order diagonal tensor of retained singular values with shape $\{\chi, \chi\}$, where χ is the retained bond dimension. In the `Python` interface, `num_of_bds_as_row`, `chi_min`, and `chi_max` are integerlike values, `target_trunc_err` and `s_min` are real scalar values, and the returned value is the tuple `(u, sigma, v_dag, trunc_err)`.

Example

C++:

```cpp
Ten u, sigma, v_t;
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 12}, gen
);
float trunc_err = 0.0;
tcapi::trunc_svd(ctx,
  a, 2,
  u, sigma, v_t,
  trunc_err,
  6, 1e-12, 3, 1e-2
);
// trunc_err != 0.0
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 12), gen)
u, sigma, v_t, trunc_err = tcapi.trunc_svd(
    ctx,
    a, 2,
    6, 1e-12, 3, 1e-2,
)
# trunc_err != 0.0
```

qr

C++:

```cpp
template <typename TenT>
void qr(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &q,
    TenT &r
);
```

Python:

```python
tcapi.qr(ctx, a, num_of_bds_as_row)
```

Perform an economy (thin) QR decomposition of an rth-order tensor `a` using the same matricization convention as `svd`. Let `shape(a)` $= \{d_0, \ldots, d_{r-1}\}$ and $k =$ `num_of_bds_as_row` with $1 \leqslant k < r$. The tensor is first matricized by grouping the first k bonds into the row index and the remaining bonds into the column index:

$$A' \in \mathbb{K}^{I \times J}, \quad I = \prod_{b=0}^{k-1} d_b, \quad J = \prod_{b=k}^{r-1} d_b. \tag{C5}$$

A QR decomposition

$$A' = QR \tag{C6}$$

is then computed, where $Q \in \mathbb{K}^{I \times \rho}$ has orthonormal columns and $R \in \mathbb{K}^{\rho \times J}$ is upper triangular (upper trapezoidal if $I < J$), with $\rho = \min(I, J)$. Finally, the factors are folded back into tensors: `shape(q)` $= \{d_0, \ldots, d_{k-1}, \rho\}$ and `shape(r)` $= \{\rho, d_k, \ldots, d_{r-1}\}$. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is the tuple `(q, r)` of raw backend tensors.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 12}, gen
);
Ten q, r;
tcapi::qr(ctx, a, 2, q, r);
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 12), gen)
q, r = tcapi.qr(ctx, a, 2)
```

lq

C++:

```cpp
template <typename TenT>
void lq(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    TenT &l,
    TenT &q
);
```

Python:

```python
tcapi.lq(ctx, a, num_of_bds_as_row)
```

Perform an economy (thin) LQ decomposition of an rth-order tensor `a` using the same matricization convention as `svd`. Let `shape(a)` $= \{d_0, \cdots, d_{r-1}\}$ and $k =$ `num_of_bds_as_row` with $1 \leqslant k < r$. The tensor is first matricized by grouping the first k bonds into the row index and the remaining bonds into the column index:

$$A' \in \mathbb{K}^{I \times J}, \quad I = \prod_{b=0}^{k-1} d_b, \quad J = \prod_{b=k}^{r-1} d_b. \tag{C7}$$

An LQ decomposition

$$A' = LQ, \tag{C8}$$

is then computed, where $L \in \mathbb{K}^{I \times \rho}$ is lower triangular (lower trapezoidal if $I > J$) and $Q \in \mathbb{K}^{\rho \times J}$ has orthonormal rows, with $\rho = \min(I, J)$. Finally, the factors are folded back into tensors: `shape(l)` $= \{d_0, \cdots, d_{k-1}, \rho\}$ and `shape(q)` $= \{\rho, d_k, \cdots, d_{r-1}\}$. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is the tuple `(l, q)` of raw backend tensors.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 12}, gen
);
Ten l, q;
tcapi::lq(ctx, a, 2, l, q);
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 12), gen)
l, q = tcapi.lq(ctx, a, 2)
```

eigvals

C++:

```
template <typename TenT>
void eigvals(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    tcapi::cplx_ten_t<TenT> &w
);
```

Python:

```
tcapi.eigvals(ctx, a, num_of_bds_as_row)
```

Compute the eigenvalues of an rth-order tensor `a` by first reshaping it into a matrix. Let `shape(a)` $= \{d_0, \ldots, d_{r-1}\}$ and $k =$ `num_of_bds_as_row` with $1 \leqslant k < r$. The tensor is matricized by grouping the first k bonds into the row index and the remaining bonds into the column index:

$$A' \in \mathbb{K}^{I \times J}, \quad I = \prod_{b=0}^{k-1} d_b, \quad J = \prod_{b=k}^{r-1} d_b, \tag{C9}$$

where $I = J$ is required. The spectrum $\{\lambda_i\}_{i=0}^{I-1}$ of A' is then computed (e.g., via a dense eigensolver) and is returned as a first-order complex tensor `w`. Unlike `eig`, which returns the eigenvalue matrix as a second-order diagonal tensor for use in decompositions, this routine returns only the eigenvalue list as a first-order tensor. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is a raw backend tensor whose kind is the complex counterpart of the input tensor kind.

Example

C++:

```
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 12}, gen
);
CplxTen w;
tcapi::eigvals(ctx, a, 2, w);
```

Python:

```
a = tcapi.random(ctx, Ten, (3, 4, 12), gen)
w = tcapi.eigvals(ctx, a, 2)
```

eigvalsh

C++:

```cpp
template <typename TenT>
void eigvalsh(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    tcapi::real_ten_t<TenT> &w
);
```

Python:

```python
tcapi.eigvalsh(ctx, a, num_of_bds_as_row)
```

Hermitian/symmetric variant of `eigvals`. The input is interpreted as a real symmetric or complex Hermitian matrix after matricization, and the eigenvalues are returned as a real first-order tensor `w` in ascending order. Unlike `eigh`, which returns the eigenvalue matrix as a second-order diagonal tensor for use in decompositions, this routine returns only the eigenvalue list as a first-order tensor. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is a raw backend tensor whose kind is the real counterpart of the input tensor kind.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 3, 4}, gen
);
Ten a_t, w;
tcapi::transpose(ctx, a, {2, 3, 0, 1}, a_t);
auto a_sym = tcapi::linear_combine(
  ctx, {std::cref(a), std::cref(a_t)}
);
tcapi::eigvalsh(ctx, a_sym, 2, w);
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 3, 4), gen)
a_t = tcapi.transpose(ctx, a, (2, 3, 0, 1))
a_sym = tcapi.linear_combine(ctx, (a, a_t))
w = tcapi.eigvalsh(ctx, a_sym, 2)
```

eig

C++:

```
template <typename TenT>
void eig(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    tcapi::cplx_ten_t<TenT> &lambda_mat,
    tcapi::cplx_ten_t<TenT> &v
);
```

Python:

```
tcapi.eig(ctx, a, num_of_bds_as_row)
```

Perform a right eigendecomposition of an rth-order tensor `a` by first matricizing it into a matrix. Let `shape(a)` $= \{d_0, \ldots, d_{r-1}\}$ and $k =$ `num_of_bds_as_row` with $1 \leqslant k < r$. The tensor is matricized by grouping the first k bonds into the row index and the remaining bonds into the column index:

$$A' \in \mathbb{K}^{I \times J}, \quad I = \prod_{b=0}^{k-1} d_b, \quad J = \prod_{b=k}^{r-1} d_b, \tag{C10}$$

where $I = J$ is required. A right eigendecomposition,

$$A'V = V\Lambda, \tag{C11}$$

is then computed, where the diagonal elements of $\Lambda = \mathrm{diag}(\lambda_0, \ldots, \lambda_{I-1})$ are the eigenvalues and the columns of V are the corresponding right eigenvectors. The routine returns the eigenvalues as `lambda_mat`, a second-order diagonal tensor representing Λ with shape $\{I, I\}$, and returns the eigenvectors in `v`, obtained by folding V back to a tensor with `shape(v)` $= \{d_0, \ldots, d_{k-1}, I\}$, so that the last bond indexes eigenvectors and the first k bonds match those of `a`. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is the tuple `(lambda_mat, v)` of raw backend tensors whose kinds are the complex counterpart of the input tensor kind.

Example

C++:

```
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 12}, gen
);
CplxTen lambda_mat, v;
tcapi::eig(ctx, a, 2, lambda_mat, v);
auto s = tcapi::shape(ctx, v);
// s == {3, 4, 12}
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 12), gen)
lambda_mat, v = tcapi.eig(ctx, a, 2)
s = tcapi.shape(ctx, v)
# s == (3, 4, 12)
```

eigh

C++:

```cpp
template <typename TenT>
void eigh(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const tcapi::order_t<TenT>
        num_of_bds_as_row,
    tcapi::real_ten_t<TenT> &lambda_mat,
    TenT &v
);
```

Python:

```python
tcapi.eigh(ctx, a, num_of_bds_as_row)
```

Hermitian/symmetric variant of `eig`. The input is interpreted as a real symmetric or complex Hermitian matrix after matricization. The eigenvalues are returned as `lambda_mat`, a real second-order diagonal tensor whose diagonal elements are the eigenvalues in ascending order, and the corresponding eigenvectors are returned in `v`. In the `Python` interface, `num_of_bds_as_row` is an integerlike value, and the returned value is the tuple `(lambda_mat, v)` of raw backend tensors; `lambda_mat` has the tensor kind obtained by taking the real counterpart of the input tensor kind, while `v` has the same kind as the input tensor.

Example

C++:

```cpp
auto a = tcapi::random<Ten>(
  ctx, {3, 4, 3, 4}, gen
);
Ten a_t, lambda_mat, v;
tcapi::transpose(ctx, a, {2, 3, 0, 1}, a_t);
auto a_sym = tcapi::linear_combine(
  ctx, {std::cref(a), std::cref(a_t)}
);
tcapi::eigh(ctx, a_sym, 2, lambda_mat, v);
auto s = tcapi::shape(ctx, v);
// s == {3, 4, 12}
```

Python:

```python
a = tcapi.random(ctx, Ten, (3, 4, 3, 4), gen)
a_t = tcapi.transpose(ctx, a, (2, 3, 0, 1))
a_sym = tcapi.linear_combine(ctx, (a, a_t))
lambda_mat, v = tcapi.eigh(ctx, a_sym, 2)
s = tcapi.shape(ctx, v)
# s == (3, 4, 12)
```

f. Miscellaneous routines

This subsection specifies auxiliary routines that support context management, data movement, debugging, and interoperability. These routines are not performance critical.

create_context

C++:

```cpp
template <typename ContextHandleT,
          /* implementation-defined */ >
void create_context(
    ContextHandleT &ctx,
    /* implementation-defined */
);
```

Python:

```python
tcapi.create_context(Ten, **options)
```

Create and initialize a backend execution context. In the `C++` interface, the concrete type `ContextHandleT` selects the target TCF/backend. The handle `ctx` is default constructible and is initially not bound to a live execution context; on success, it becomes a valid context handle. In the `Python` interface, `Ten` is a `TenKind` that selects the implementation family for the context, and the returned value is an implementation-defined context handle. A context may support multiple `C++` tensor types `TenT` or `Python TenKind` objects provided by the same TCAPI implementation. Implementations may accept additional parameters, such as device IDs, streams/queues, allocators, or logging/debug flags.

Example

C++:

```cpp
// Pseudocode
// For a tensor on CPU with the type CPUTen
tcapi::context_handle_t<CPUTen> ctx_cpu;
// default threads, default allocator
tcapi::create_context(ctx_cpu);

// For a tensor on GPU with the type GPUTen
tcapi::context_handle_t<GPUTen> ctx_gpu;
int device_id = 0;
GpuStream stream = /* ... */;
// select device and stream/queue
tcapi::create_context(
  ctx_gpu, device_id, stream
);
```

Python:

```python
# Pseudocode
# For a tensor on CPU with the TenKind CPUTen
ctx_cpu = tcapi.create_context(CPUTen)

# For a tensor on GPU with the TenKind GPUTen
ctx_gpu = tcapi.create_context(
    GPUTen,
    device_id=0,
    stream=stream,
)
```

destroy_context

C++:

```cpp
template <typename ContextHandleT>
void destroy_context(ContextHandleT &ctx);
```

Python:

```python
tcapi.destroy_context(ctx)
```

Destroy the backend execution context managed by `ctx` and release any owned resources (e.g., devices, streams/queues, library handles, memory pools). After this call, `ctx` is invalidated and must not be passed to further TCAPI routines unless it is re-initialized via `create_context`.

Example

C++:

```cpp
tcapi::destroy_context(ctx);
```

Python:

```python
tcapi.destroy_context(ctx)
```

to_range

C++:

```cpp
template <typename TenT,
          typename RandomIt,
          typename Func>
void to_range(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    RandomIt first,
    Func &&coors2idx
);
```

Python:

```
tcapi.to_range(ctx, a, coors2idx)
```

Copy all elements of `a` into a one-dimensional container using a user-supplied index map. In the `C++` interface, elements are copied into a random-access output range beginning at `first`; for each coordinate tuple `coors`, the destination is written as `*(first + coors2idx(coors))`. The callable `coors2idx` must have the following signature:

```
std::iterator_traits<
        RandomIt
>::difference_type coors2idx(
    const tcapi::elem_coors_t<TenT> &coors
);
```

It must map every valid coordinate of `a` to a valid index in $[0, \texttt{tcapi::size(ctx,a)})$. In the `Python` interface, the returned value is a `Python list` with length `tcapi.size(ctx, a)`. The callable `coors2idx` is invoked as `coors2idx(coors)`, where `coors` is a `Python` coordinate sequence in bond-index order, and must return an integerlike index in $[0, \texttt{tcapi.size(ctx,a)})$. For a zeroth-order tensor, the output range or list has length 1; `coors2idx` is invoked once with the empty coordinate sequence and must return 0. Each list element is a scalar value consistent with the relevant `TenKind`. For a diagonal tensor, this routine writes all tensor elements, including off-diagonal zeros. To extract only diagonal elements, use `diag`.

Example

C++:

```
using Elem = typename tcapi::elem_t<Ten>;
using ElemVec = std::vector<Elem>;
std::iterator_traits<
    typename ElemVec::iterator
>::difference_type coors2idx(
    const tcapi::elem_coors_t<Ten> &coors
) {
  return 3 * coors[0] + coors[1];
}
ElemVec els {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
auto a = tcapi::assign_from_range<Ten>(
    ctx, {2, 3}, els.begin(), coors2idx
);
ElemVec els2(6);
tcapi::to_range(ctx,
  a,
  els2.begin(),
  coors2idx
);
// els2 == els
```

Python:

```python
def coors2idx(coors):
    return 3 * coors[0] + coors[1]

els = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
a = tcapi.assign_from_range(
    ctx, Ten, (2, 3), els, coors2idx
)
els2 = tcapi.to_range(ctx, a, coors2idx)
# els2 == els
```

show

C++:

```cpp
template <typename TenT>
void show(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a
);
```

Python:

```python
tcapi.show(ctx, a)
```

Print the contents of `a` to standard output in a human-readable form. The exact formatting (layout, precision, ordering, etc.) is implementation defined.

Example

C++:

```cpp
tcapi::show(ctx, a);
```

Python:

```python
tcapi.show(ctx, a)
```

close

C++:

```cpp
template <typename TenT>
bool close(
    tcapi::context_handle_t<TenT> &ctx,
    const TenT &a,
    const TenT &b,
    const tcapi::real_t<TenT> epsilon
);
```

Python:

```python
tcapi.close(ctx, a, b, epsilon)
```

This routine performs an elementwise absolute-tolerance comparison. The comparison succeeds iff `shape(a)` $=$ `shape(b)` $= \{d_0, \ldots, d_{r-1}\}$, and

$$\mathrm{Close}(A, B; \epsilon) \equiv \max_{0 \leqslant i_b < d_b} \left| A_{i_0 \cdots i_{r-1}} - B_{i_0 \cdots i_{r-1}} \right| \leqslant \epsilon, \tag{C12}$$

where $|\cdot|$ denotes absolute value, or modulus for complex values. The parameter `epsilon` must satisfy `epsilon` $\geqslant 0$. In the `C++` interface, the returned value is `true` when the comparison succeeds and `false` otherwise. In the `Python` interface, `epsilon` is a real scalar value and the returned value is a Boolean-like value.

Example

C++:

```cpp
auto A = tcapi::eye<Ten>(ctx, 3);
auto B = tcapi::copy(ctx, A);
tcapi::set_elem(ctx, B, {2, 0}, 1e-5);

bool ok_abs = tcapi::close(ctx, A, B, 1e-6);
// false
ok_abs = tcapi::close(ctx, A, B, 1e-4f);
// true
```

Python:

```python
A = tcapi.eye(ctx, Ten, 3)
B = tcapi.copy(ctx, A)
B = tcapi.set_elem(ctx, B, (2, 0), 1e-5)

ok_abs = tcapi.close(ctx, A, B, 1e-6)
# False
ok_abs = tcapi.close(ctx, A, B, 1e-4)
# True
```

convert

C++:

```cpp
template <typename Ten1T, typename Ten2T>
void convert(
    tcapi::context_handle_t<Ten1T> &ctx1,
    const Ten1T &t1,
    tcapi::context_handle_t<Ten2T> &ctx2,
    Ten2T &t2
);
```

Python:

```python
tcapi.convert(ctx1, t1, ctx2, Ten2)
```

Convert `t1` into a destination tensor, possibly changing the element type, the underlying TCF/backend, or both. This includes conversions between different floating-point precisions within the same TCF. Memory movement between `ctx1` (source TCF) and `ctx2` (destination TCF) is implementation defined. In the `C++` interface, the result is written into `t2`; on success, `t2` must be

fully constructed and ready for use. In the Python interface, Ten2 is the destination TenKind, and the returned value is a raw backend tensor associated with ctx2. If the source and destination have the same tensor type or tensor kind, this routine is equivalent to a deep copy.

In the precision-conversion examples below, Real64Ten denotes an implementation-provided C++ tensor type or Python TenKind with 64-bit real floating-point elements. Both Ten and Real64Ten are assumed to be provided by the same TCF and supported by ctx_cpu.

Example

C++:

```cpp
// CPU float -> CPU complex (promotion)
auto A = tcapi::random<Ten>(
  ctx_cpu, {4, 4}, gen
);
CplxTen Ac;
tcapi::convert(ctx_cpu, A, ctx_cpu, Ac);

// CPU float -> GPU float (device transfer)
GPUTen Ag;
tcapi::convert(ctx_cpu, A, ctx_gpu, Ag);

// CPU float32 -> CPU float64 (precision conversion)
Real64Ten A64;
tcapi::convert(ctx_cpu, A, ctx_cpu, A64);
```

Python:

```python
# CPU float -> CPU complex (promotion)
A = tcapi.random(ctx_cpu, Ten, (4, 4), gen)
Ac = tcapi.convert(
    ctx_cpu, A, ctx_cpu, CplxTen
)

# CPU float -> GPU float (device transfer)
Ag = tcapi.convert(ctx_cpu, A, ctx_gpu, GPUTen)

# CPU float32 -> CPU float64 (precision conversion)
A64 = tcapi.convert(ctx_cpu, A, ctx_cpu, Real64Ten)
```

version

C++:

```cpp
template <typename TenT>
std::string version();
```

Python:

```python
tcapi.version(Ten)
```

Return the TCAPI specification version implemented for the tensor type `TenT` in `C++` or for the `TenKind` `Ten` in `Python`. The version string has the form ‘‘`M.m`’’, where `M` is the major version and `m` is the minor version, both nonnegative integers.

Example

C++:

```cpp
auto version = tcapi::version<Ten>();
// may equal "1.0"
```

Python:

```python
version = tcapi.version(Ten)
# may equal "1.0"
```

3. Environment variables

The final component of TCAPI is a set of environment variables that specify optional run-time behavior of TCAPI implementations. These variables provide a lightweight mechanism to enable diagnostics and performance-related reporting without modifying application code.

`TCAPI_VERBOSE`: Control diagnostic output and optional performance reporting for TCAPI function calls. Similar in spirit to the verbose mode in Intel Math Kernel Library (MKL), a TCAPI implementation may emit implementation-defined messages whenever a TCAPI routine is invoked. Such messages may include the function name or signature, selected properties of the input tensors (e.g., order, shape, and element type), and, at higher verbosity levels, measured wall-clock execution time. The behavior of `TCAPI_VERBOSE` is defined as follows:

(1) `TCAPI_VERBOSE` = `0` (default): no diagnostic output is produced.

(2) `TCAPI_VERBOSE` = `1`: print a single-line message per TCAPI call containing the function name and salient input information; the exact format is implementation-defined.

(3) `TCAPI_VERBOSE` = `2`: in addition to the output at `TCAPI_VERBOSE` = `1`, include the measured execution time of each call.

---

[1] J. Ignacio Cirac, D. Pérez-García, N. Schuch, and F. Verstraete, Matrix product states and projected entangled pair states: Concepts, symmetries, theorems, Rev. Mod. Phys. **93**, 045003 (2021).

[2] S. R. White, Density matrix formulation for quantum renormalization groups, Phys. Rev. Lett. **69**, 2863 (1992).

[3] G. Vidal, Classical simulation of infinite-size quantum lattice systems in one spatial dimension, Phys. Rev. Lett. **98**, 070201 (2007).

[4] J. Haegeman, J. I. Cirac, T. J. Osborne, I. Pižorn, H. Verschelde, and F. Verstraete, Time-dependent variational principle for quantum lattices, Phys. Rev. Lett. **107**, 070601 (2011).

[5] J. Haegeman, C. Lubich, I. Oseledets, B. Vandereycken, and F. Verstraete, Unifying time evolution and optimization with matrix product states, Phys. Rev. B **94**, 165116 (2016).

[6] P. C. G. Vlaar and P. Corboz, Simulation of three-dimensional quantum systems with projected entangled-pair states, Phys. Rev. B **103**, 205137 (2021).

[7] M. Levin and C. P. Nave, Tensor renormalization group approach to two-dimensional classical lattice models, Phys. Rev. Lett. **99**, 120601 (2007).

[8] Z. Y. Xie, J. Chen, M. P. Qin, J. W. Zhu, L. P. Yang, and T. Xiang, Coarse-graining renormalization by higher-order singular value decomposition, Phys. Rev. B **86**, 045139 (2012).

[9] X. Chen, Z.-C. Gu, and X.-G. Wen, Classification of gapped symmetric phases in one-dimensional spin systems, Phys. Rev. B **83**, 035107 (2011).
[10] N. Schuch, D. Pérez-García, and I. Cirac, Classifying quantum phases using matrix product states and projected entangled pair states, Phys. Rev. B **84**, 165139 (2011).
[11] F. Pollmann, E. Berg, A. M. Turner, and M. Oshikawa, Symmetry protection of topological phases in one-dimensional quantum spin systems, Phys. Rev. B **85**, 075125 (2012).
[12] X. Chen, Z.-C. Gu, Z.-X. Liu, and X.-G. Wen, Symmetry protected topological orders and the group cohomology of their symmetry group, Phys. Rev. B **87**, 155114 (2013).
[13] O. Buerschaper, M. Aguado, and G. Vidal, Explicit tensor network representation for the ground states of string-net models, Phys. Rev. B **79**, 085119 (2009).
[14] J. I. Cirac, D. Poilblanc, N. Schuch, and F. Verstraete, Entanglement spectrum and boundary theories with projected entangled-pair states, Phys. Rev. B **83**, 245134 (2011).
[15] M. P. Zaletel and R. S. K. Mong, Exact matrix product states for quantum Hall wave functions, Phys. Rev. B **86**, 245305 (2012).
[16] M. P. Zaletel, R. S. K. Mong, and F. Pollmann, Topological characterization of fractional quantum Hall ground states from microscopic Hamiltonians, Phys. Rev. Lett. **110**, 236801 (2013).
[17] H. He, H. Moradi, and X.-G. Wen, Modular matrices as topological order parameter by a gauge-symmetry-preserved tensor renormalization approach, Phys. Rev. B **90**, 205114 (2014).
[18] I. L. Markov and Y. Shi, Simulating quantum computation by contracting tensor networks, SIAM J. Comput. **38**, 963 (2008).
[19] Y. Zhou, E. M. Stoudenmire, and X. Waintal, What limits the simulation of quantum computers? Phys. Rev. X **10**, 041038 (2020).
[20] F. Pan, K. Chen, and P. Zhang, Solving the sampling problem of the Sycamore quantum circuits, Phys. Rev. Lett. **129**, 090502 (2022).
[21] J. Tindall, M. Fishman, M. Stoudenmire, and D. Sels, Efficient tensor network simulation of IBM's kicked Ising experiment, PRX Quantum **5**, 010308 (2024).
[22] S. Anand, K. Temme, A. Kandala, and M. Zaletel, Classical benchmarking of zero noise extrapolation beyond the exactly-verifiable regime, arXiv:2306.17839.
[23] R.-Y. Sun, T. Shirakawa, and S. Yunoki, Improved real-space parallelizable matrix-product state compression and its application to unitary quantum dynamics simulation, Phys. Rev. B **110**, 085149 (2024).
[24] Y. Liu, Y. Chen, C. Guo, J. Song, X. Shi, L. Gan, Wenzhao Wu, Wei Wu, H. Fu, X. Liu, D. Chen, Z. Zhao, G. Yang, and J. Gao, Verifying quantum advantage experiments with multiple amplitude tensor network contraction, Phys. Rev. Lett. **132**, 030601 (2024).
[25] C. Oh, M. Liu, Y. Alexeev, B. Fefferman, and L. Jiang, Classical algorithm for simulating experimental Gaussian boson sampling, Nat. Phys. **20**, 1461 (2024).
[26] C. Schön, E. Solano, F. Verstraete, J. I. Cirac, and M. M. Wolf, Sequential generation of entangled multiqubit states, Phys. Rev. Lett. **95**, 110503 (2005).
[27] M. P. Zaletel and F. Pollmann, Isometric tensor network states in two dimensions, Phys. Rev. Lett. **124**, 037201 (2020).
[28] S.-J. Ran, Encoding of matrix product states into quantum circuits of one- and two-qubit gates, Phys. Rev. A **101**, 032310 (2020).
[29] R. Haghshenas, J. Gray, A. C. Potter, and G. K.-L. Chan, Variational power of quantum circuit tensor networks, Phys. Rev. X **12**, 011047 (2022).
[30] Z.-Y. Wei, D. Malz, and J. I. Cirac, Sequential generation of projected entangled-pair states, Phys. Rev. Lett. **128**, 010607 (2022).
[31] S. Anand, J. Hauschild, Y. Zhang, A. C. Potter, and M. P. Zaletel, Holographic quantum simulation of entanglement renormalization circuits, PRX Quantum **4**, 030334 (2023).
[32] D. Malz, G. Styliaris, Z.-Y. Wei, and J. I. Cirac, Preparation of matrix product states with log-depth quantum circuits, Phys. Rev. Lett. **132**, 040404 (2024).
[33] A. J. Ferris and D. Poulin, Tensor networks and quantum error correction, Phys. Rev. Lett. **113**, 030501 (2014).

[34] T. Farrelly, R. J. Harris, N. A. McMahon, and T. M. Stace, Tensor-network codes, Phys. Rev. Lett. **127**, 040507 (2021).
[35] C. Cao and B. Lackey, Quantum Lego: Building quantum error correction codes from tensor networks, PRX Quantum **3**, 020332 (2022).
[36] Y. Guo and S. Yang, Quantum error mitigation via matrix product operators, PRX Quantum **3**, 040313 (2022).
[37] S. Filippov, M. Leahy, M. A. C. Rossi, and G. García-Pérez, Scalable tensor-network error mitigation for near-term quantum computing, arXiv:2307.11740.
[38] A. Berezutskii *et al.*, Tensor networks for quantum computing, Nat. Rev. Phys. **7**, 581 (2025).
[39] M. Wang, Y. Pan, Z. Xu, G. Li, X. Yang, D. Mandic, and A. Cichocki, Tensor networks meet neural networks: A survey and future perspectives, arXiv:2302.09019.
[40] A. Zadeh, M. Chen, S. Poria, E. Cambria, and L.-P. Morency, Tensor fusion network for multimodal sentiment analysis, in *Conference on Empirical Methods in Natural Language Processing* (Association for Computational Linguistics, Stroudsburg, PA, 2017), pp. 1103–1114.
[41] H. Ben-Younes, R. Cadene, M. Cord, and N. Thome, Mutan: Multimodal Tucker fusion for visual question answering, in *Proceedings of the IEEE International Conference on Computer Vision* (IEEE, Piscataway, NJ, 2017), pp. 2631–2639.
[42] P. Zhang, Z. Su, L. Zhang, B. Wang, and D. Song, A quantum many-body wave function inspired language modeling approach, in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management* (Association for Computing Machinery, New York, NY, 2018), pp. 1303–1312.
[43] M. Hou, J. Tang, J. Zhang, W. Kong, and Q. Zhao, Deep multimodal multilinear fusion with high-order polynomial pooling, Adv. Neural Inf. Process. Syst. **32**, 12136 (2019).
[44] J. Kossaifi, A. Toisoul, A. Bulat, Y. Panagakis, T. M. Hospedales, and M. Pantic, Factorized higher-order CNNs with an application to spatio-temporal emotion estimation, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (IEEE, Piscataway, NJ, 2020), pp. 6060–6069.
[45] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, Tensorizing neural networks, Adv. Neural Inf. Process. Syst. **28**, 442 (2015).
[46] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov, Ultimate tensorization: Compressing convolutional and FC layers alike, arXiv:1611.03214.
[47] E. Stoudenmire and D. J. Schwab, Supervised learning with tensor networks, Adv. Neural Inf. Process. Syst. **29**, 4799 (2016).
[48] Y. Yang and T. M. Hospedales, Deep multi-task representation learning: A tensor factorisation approach, in *International Conference on Learning Representations* (ICLR, 2017).
[49] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, D. Song, and M. Zhou, A tensorized transformer for language modeling, Adv. Neural Inf. Process. Syst. **32**, 2232 (2019).
[50] K. Sozykin, A. Chertkov, R. Schutski, A.-H. Phan, A. S. Cichocki, and I. Oseledets, TTOpt: A maximum volume quantized tensor train-based optimization and its application to reinforcement learning, Adv. Neural Inf. Process. Syst. **35**, 26052 (2022).
[51] Y. Qiu, A. Wang, C. Li, H. Huang, G. Zhou, and Q. Zhao, Steps: Sequential probability tensor estimation for text-to-image hard prompt search, in *Proceedings of the Computer Vision and Pattern Recognition Conference* (IEEE, Piscataway, NJ, 2025), pp. 28640–28650.
[52] R. Han, R. Willett, and A. R. Zhang, An optimal statistical and computational framework for generalized tensor estimation, Ann. Stat. **50**, 1 (2022).
[53] A. Saiapin and K. Batselier, Tensor network based feature learning model, in *Proceedings of the 28th International Conference on Artificial Intelligence and Statistics* (PMLR, 2025), Vol. 258, pp. 3277–3285.
[54] V. Chekalina, G. Novikov, J. Gusak, A. Panchenko, and I. Oseledets, Efficient GPT model pre-training using tensor train matrix representation, in *Proceedings of the 37th Pacific Asia Conference on Language, Information and Computation* (Association for Computational Linguistics, 2023), pp. 600–608.
[55] Z. Chen, R. Dangovski, C. Loh, O. Dugan, D. Luo, and M. Soljacic, QuanTA: Efficient high-rank fine-tuning of LLMs with quantum-informed tensor adaptation, Adv. Neural Inf. Process. Syst. **37**, 92210 (2024).

[56] M. Xu, Y. L. Xu, and D. P. Mandic, TensorGPT: Efficient compression of large language models based on tensor-train decomposition, arXiv:2307.00526.
[57] A. Anjum, M. E. Eren, I. Boureima, B. Alexandrov, and M. Bhattarai, Tensor train low-rank approximation (TT-LoRA): Democratizing AI with accelerated LLMs, in *2024 International Conference on Machine Learning and Applications (ICMLA)* (IEEE, Piscataway, NJ, 2024), pp. 583–590.
[58] Z. Yang, Z. Liu, S. Choudhary, X. Xie, C. Gao, S. Kunzmann, and Z. Zhang, CoMERA: Computing-and memory-efficient training via rank-adaptive tensor optimization, Adv. Neural Inf. Process. Syst. **37**, 77200 (2024).
[59] Z. Tao, Y. Takida, N. Murata, Q. Zhao, and Y. Mitsufuji, Transformed low-rank adaptation via tensor decomposition and its applications to text-to-image models, in *Proceedings of the IEEE/CVF International Conference on Computer Vision* (IEEE, Piscataway, NJ, 2025), pp. 16333–16344.
[60] W. Huggins, P. Patil, B. Mitchell, K. B. Whaley, and E. M. Stoudenmire, Towards quantum machine learning with tensor networks, Quantum Sci. Technol. **4**, 024001 (2019).
[61] R. Dilip, Y.-J. Liu, A. Smith, and F. Pollmann, Data compression for quantum machine learning, Phys. Rev. Res. **4**, 043007 (2022).
[62] J. Dborin, F. Barratt, V. Wimalaweera, L. Wright, and A. G. Green, Matrix product state pre-training for quantum machine learning, Quantum Sci. Technol. **7**, 035014 (2022).
[63] H.-M. Rieser, F. Köster, and A. P. Raulf, Tensor networks for quantum machine learning, Proc. R. Soc. A **479**, 20230218 (2023).
[64] S. Shin, Y. S. Teo, and H. Jeong, Dequantizing quantum machine learning models using tensor networks, Phys. Rev. Res. **6**, 023218 (2024).
[65] M. Fishman, S. R. White, and E. M. Stoudenmire, The ITensor software library for tensor network calculations, SciPost Phys. Codebases 4 (2022).
[66] K.-H. Wu, C.-T. Lin, K. Hsu, H.-T. Hung, M. Schneider, C.-M. Chung, Y.-J. Kao, and P. Chen, The Cytnx library for tensor networks, SciPost Phys. Codebases 53 (2025).
[67] C. R. Harris *et al.*, Array programming with NumPy, Nature (London) **585**, 357 (2020).
[68] J. Bradbury *et al.*, JAX: Composable transformations of Python + NumPy programs, 2018, https://github.com/jax-ml/jax.
[69] A. Paszke *et al.*, PyTorch: An imperative style, high-performance deep learning library, in *Advances in Neural Information Processing Systems* (Curran Associates, Inc., Red Hook, NY, 2019), Vol. 32, pp. 8026–8037.
[70] TCAPI (Tensor computing application programming interface)—Cytnx backend, 2025, https://github.com/r-ccs-cms/tensor-computing-interface-backend-cytnx.
[71] TCAPI (Tensor computing application programming interface)—ITensor backend, 2026, https://github.com/AngelaHui01/tcapi_itensor.
[72] TCAPI (Tensor computing application programming interface)—NumPy backend, 2026, https://github.com/AngelaHui01/tcapi_numpy.
[73] TCAPI (Tensor computing application programming interface) specification, 2026, https://tensorcomputingapi.github.io.
[74] J. Brandejs, N. Hörnblad, E. F. Valeev, A. Heinecke, J. Hammond, D. Matthews, and P. Bientinesi, Tensor algebra processing primitives (TAPP): Towards a standard for tensor operations, arXiv:2601.07827.
[75] R. Penrose *et al.*, Applications of negative dimensional tensors, Comb. Math. Appl. **1**, 3 (1971).
[76] TNBP, 2026, https://github.com/r-ccs-cms/tnbp.
[77] P. Pfeuty, The one-dimensional Ising model with a transverse field, Ann. Phys. **57**, 79 (1970).
[78] M. B. Hastings, An area law for one-dimensional quantum systems, J. Stat. Mech. (2007) P08024.
[79] P. Calabrese and J. Cardy, Entanglement entropy and quantum field theory, J. Stat. Mech. (2004) P06002.
[80] L. Tagliacozzo, T. R. de Oliveira, S. Iblisdir, and J. I. Latorre, Scaling of entanglement support for matrix product states, Phys. Rev. B **78**, 024410 (2008).
[81] F. Pollmann, S. Mukerjee, A. M. Turner, and J. E. Moore, Theory of finite-entanglement scaling at one-dimensional quantum critical points, Phys. Rev. Lett. **102**, 255701 (2009).

[82] Y. Kim, A. Eddins, S. Anand, K. X. Wei, E. van den Berg, S. Rosenblatt, H. Nayfeh, Y. Wu, M. Zaletel, K. Temme, and A. Kandala, Evidence for the utility of quantum computing before fault tolerance, Nature (London) **618**, 500 (2023).
[83] K. Shinjo, K. Seki, T. Shirakawa, R.-Y. Sun, and S. Yunoki, Unveiling clean two-dimensional discrete time crystals on a digital quantum computer, npj Quantum Inf. **12**, 41 (2026).
[84] GraceQ/tensor, accessed 29 October 2026, https://github.com/gracequantum/tensor.
[85] NVIDIA Corporation, cuTENSOR: A high-performance CUDA library for tensor primitives, 2024, https://developer.nvidia.com/cutensor.
[86] NVIDIA Corporation, cuTensorNet: A CUDA tensor network library, 2024, https://developer.nvidia.com/cutensornet.
[87] Itensors.jl, 2020, https://github.com/ITensor/ITensors.jl.
[88] Tensorkit.jl, accessed 29 October 2026, https://github.com/Jutho/TensorKit.jl.
[89] R.-Y. Sun, T. Shirakawa, H. Kohshiro, D. N. Sheng, and S. Yunoki, Supporting data, 2026, https://github.com/sun-rongyang/data-arXiv.2512.23917.
[90] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, Open quantum assembly language, arXiv:1707.03429.
[91] Qiskit Community, Qiskit: An open-source framework for quantum computing, 2017, https://github.com/Qiskit/qiskit.
[92] J. Tindall and M. Fishman, Gauging tensor networks with belief propagation, SciPost Phys. **15**, 222 (2023).