SOFTWARE METAPAPER

GuiTeNet: A Graphical User Interface for Tensor Networks

Lisa Sahlmann[1] and Christian B. Mendl[2,3]

[1] Technische Universität Dresden, Faculty of Mathematics, Dresden, DE

[2] Technische Universität Dresden, Institute of Scientific Computing, Dresden, DE

[3] Technische Universität München, Department of Informatics and Institute for Advanced Study, Garching, DE

Corresponding author: Christian B. Mendl (christian.mendl@tum.de)

We introduce a graphical user interface for constructing arbitrary tensor networks and specifying common operations like contractions or splitting, denoted GuiTeNet. Tensors are represented as nodes with attached legs, corresponding to the ordered dimensions of the tensor. GuiTeNet visualizes the current network, and instantly generates Python/NumPy source code for the hitherto sequence of user actions. Support for additional programming languages is planned for the future. We discuss the elementary operations on tensor networks used by GuiTeNet, together with high-level optimization strategies. The software runs directly in web browsers and is available online at guitenet.org.

(1) Overview
### Introduction
Tensor networks have found a wide range of applications within mathematics [1, 2], physics and chemistry, in particular as matrix product states (MPS), projected entangled pair states (PEPS) or the multiscale entanglement renormalization ansatz (MERA) for strongly correlated quantum systems [3–5]. While tensor networks and associated operations are conveniently represented as graphical diagrams, a subsequent implementation of these operations is often tedious, especially if one has to keep track of arrangements of many indices. On the other hand, fundamental operations like finding the (quasi-)optimal contraction order and performing the partial contraction of a given tensor network are available as software packages [6–10] or via NumPy's `einsum` command [11, 12]. To bridge the gap between graphical representation and implementation, we introduce a graphical user interface (GUI) for constructing arbitrary tensor networks and specifying common operations on them, like contractions or splitting via QR- or SVD-decompositions. Our software framework then instantly generates source code for these operations; currently Python/NumPy is supported, with additional programming languages planned for the future. We use JavaScript and the D3.js library to make the GUI conveniently available via web browsers.

The GUI represents each tensor as a node with an arbitrary number of legs, corresponding to the number of dimensions (rank) of the tensor. The ordering of the dimensions is indicated by labels. **Figure 1** visualizes
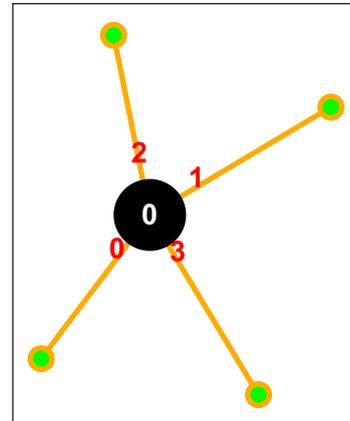
Figure 1: A single tensor with 4 legs (dimensions). The ordering of dimensions is indicated by the red labels.

a single tensor as it appears in the GUI. Note that this abstract representation leaves the actual dimensions open, i.e., it does not differentiate between, say, a $2 \times 3 \times 4$ tensor and a $8 \times 7 \times 6$ tensor, since both have rank 3.

The user interacts with the GUI mainly via drag-and-drop gestures, to add tensors to the network or attach legs to a tensor, and to specify operations like contractions and tensor splitting; see below for more details. The GuiTeNet framework visualizes the current tensor network, and simultaneously generates source code which implements the hitherto sequence of user actions. For example, the generated Python code for a contraction of three tensors followed by QR splitting reads:

```
import numpy as np

def f(T0, T1, T2):
    T3 = np.einsum(T0, (0, 1, 2), T1, (3, 2), T2, (0, 4, 5), (1, 3, 4, 5))
    T4 = np.transpose(T3, (3, 0, 2, 1))
    T5, T6 = np.linalg.qr(T4.reshape((np.prod(T4.shape[:2]),
                np.prod(T4.shape[2:]))), mode='reduced')
    T5 = T5.reshape(T4.shape[:2] + (T5.shape[1],))
    T6 = T6.reshape((T6.shape[0],) + T4.shape[2:])
    return(T5, T6)
```

Creating tensors

A new tensor is added to the network by a drag-and-drop gesture. The user drags a special "create tensor" symbol (blue circle in **Figure 2**) to the desired location. When "dropping" the symbol, a new tensor (black circle) appears there. Initially it has zero legs. The tensors are automatically labeled 0, 1, 2, … to provide a unique identifier. The "create tensor" symbol reappears at its default location after this operation, and can then be used to add another tensor to the network.

Attaching tensor legs

Each leg represents one dimension of the tensor. The user creates a new leg by "pulling" it out of the tensor (i.e., drag-and-drop on the tensor), when simultaneously holding the Control key. Each tensor and its legs can still be freely moved around within the GUI window.
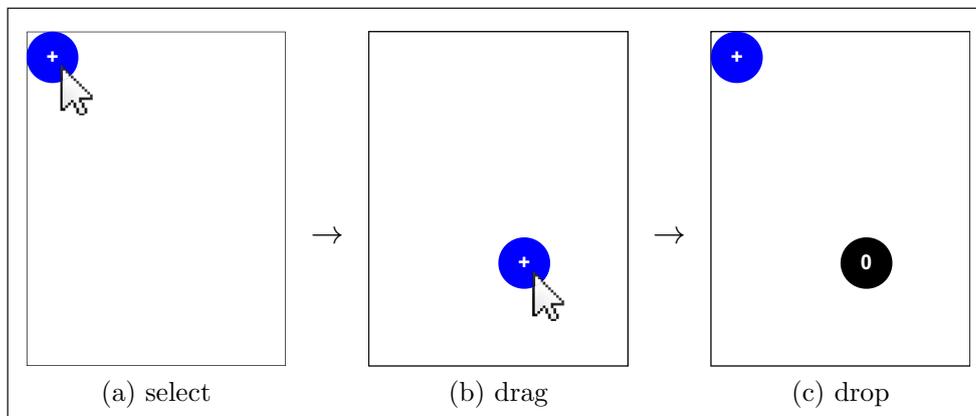
Tensor contractions

Tensor contractions are specified by connecting the tips of tensor legs. The tips snap to each other when brought into close contact. The actual contraction (possibly of several tensors) is executed when pressing the "Contract" button of the GUI, see **Figure 3** for an example.
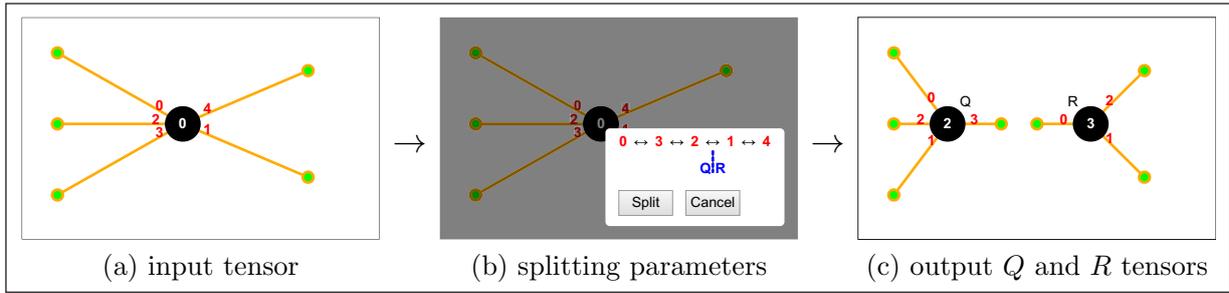
Splitting a tensor

The splitting of a tensor by QR or singular value decomposition (SVD) is a ubiquitous operation in tensor network algorithms, in particular for reducing "bond dimensions" by devising a singular value cut-off tolerance, and a prerequisite for working with left- and right-orthogonal tensors in the MPS framework [3]. The first step for decomposing a tensor A is its "matricization": a subset of legs is grouped together into one "fat" leg and the remaining (complementary) legs into a second "fat" leg. The two fat legs are interpreted as the rows and columns of a matrix, which is then decomposed. **Figure 4** illustrates this process (as it appears in the GUI) for the QR decomposition of a tensor with initially 5 legs. (An analogous SVD decomposition is currently still under development.) The user first right-clicks on a tensor to initiate the splitting operation. An overlay window then asks for the ordering and partitioning of

(a) select   (b) drag   (c) drop

Figure 2: Creating a new tensor by a drag-and-drop gesture. The mouse pointer is enlarged for visual clarity.

(a) input network   (b) output tensor
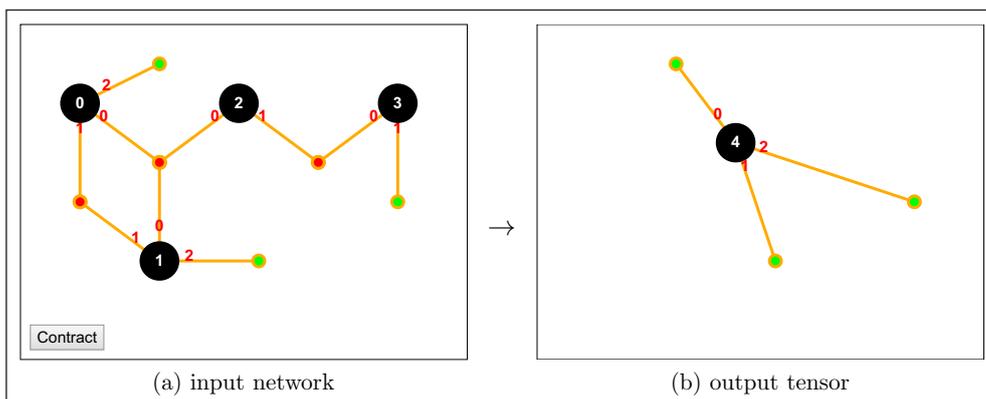
Figure 3: Illustration of an elementary contraction operation.

(a) input tensor   (b) splitting parameters   (c) output Q and R tensors

Figure 4: QR splitting of a tensor. In **(b)**, the user specifies the ordered dimensions attributed to the Q and R tensors, respectively.

dimensions attributed to the rows and columns in the matricization process. In the example, the "row" consists of dimensions 0, 3, 2 (in this order) and the "column" of dimensions 1, 4 (in this order). After the decomposition, the resulting Q and R matrices are finally reshaped to restore the original dimensions, with an additional dimension for the shared bond (last dimension of Q, first dimension of R). Thus the dimensions 0, 1, 2 of Q match the original dimensions 0, 3, 2, and dimensions 1, 2 of R the original dimensions 1, 4.

The initial reordering of dimensions becomes a separate "elementary transposition operation", as described below. The generated code uses a temporary tensor for this purpose. In **Figure 4**, this temporary tensor has index 1, and hence the Q and R tensors are consecutively labeled 2 and 3.

After this reordering, the partitioning is simply a reinterpretation of the data stored in the tensor, since the "row" group now consists of the first l leading dimensions, and the "column" group of the remaining $r - l$ trailing dimensions, where the rank r is the total number of dimensions.

Implementation and architecture

Somewhat analogous to an intermediate representation in source code compilation, we decompose the actions supported by the GUI into the following elementary operations on tensor networks:

(i) Elementary contraction of tensors

The GuiTeNet framework supports general contraction operations on a tensor network. An *elementary* contraction acts on a subset of tensors such that these tensors are joined (directly or indirectly) by shared legs, yielding a single tensor after the contraction. Note that "multi-bond" contractions, i.e., the simultaneous contraction of multiple legs as in **Figure 3**, is explicitly allowed. In principle, a contraction of several tensors could be decomposed into a sequence of pairwise contractions, e.g., computing the matrix product ABC by first "contracting" A with B to obtain $T = AB$ and then multiplying T with C. However, in general the optimal order of these pairwise contractions poses a delicate optimization problem [7] and is not straightforwardly applicable to multi-bond contractions. Hence we regard the contraction of (possibly more than two) tensors as elementary operation, and leave the optimized implementation to backend software packages.

On the other hand, a sequence of tensor network operations can be optimized by merging subsequent elementary contractions into a single elementary contraction. As simple (toy model) illustration why this might be useful, consider the contraction $C = AB$ (matrix-matrix multiplication) followed by the contraction $y = Cx$ (matrix-vector product). Merging these two contractions leads to $y = ABx$, for which a backend algorithm would naturally choose the order $y = A(Bx)$.

To uniquely specify a contraction operation, we follow NumPy's `einsum` command convention in the form $\text{einsum}(T_0, s_0, T_1, s_1, ..., s_{out})$. Here the T_i refer to tensors, and s_i are lists of integer labels for the corresponding dimensions, with multiply occurring labels to be summed over. The last argument s_{out} determines the ordering of dimensions in the output tensor after the contraction. For the example in **Figure 3** with 4 tensors,

$$s_0 = (0, 1, 2), \quad s_1 = (0, 1, 3), \quad s_2 = (0, 4), \quad s_3 = (4, 5) \quad \text{and} \quad s_{out} = (2, 3, 5).$$

Thus, the three dimensions of tensor T_0 are labeled 0, 1, 2, the three dimensions of tensor T_1 are labeled 0, 1, 3 etc. The dimensions labeled 0, 1 and 4 will be contracted since they appear multiple times, and the remaining dimensions are ordered as (2, 3, 5) in the output tensor. The generated Python source code follows exactly this scheme and reads explicitly

```
T4 = np.einsum(T0, (0, 1, 2), T1, (0, 1, 3),
               T2, (0, 4), T3, (4, 5), (2, 3, 5))
```

(ii) General transposition of a tensor

Formally, a tensor transposition is a permutation of dimensions, generalizing the usual transposition of matrices. For example, applying the permutation (1, 2, 0) to a $10 \times 11 \times 12$ tensor T yields a $11 \times 12 \times 10$ tensor, such that the (i, j, k)-th entry of T is the (j, k, i)-th entry of the transposed tensor. Since the tensor elements are typically stored as a contiguous array in memory, a transposition implies a reshuffling of the array elements. Thus, while a transposition does not involve arithmetic calculations besides computing memory addresses, its cost can still be significant, in particular due to the inherent "cache-unfriendliness".

Specifying a transposition only requires designating the permutation of dimensions. We follow the convention of NumPy's `transpose` function.

Regarding transpositions as separate elementary operations — instead of first step for splitting a tensor for

example — facilitates additional optimizations. A plausible scenario is integrating the transposition into a preceding contraction operation [13], generalizing $(AB^T)^T = BA^T$ for matrices A and B.

(iii) QR decomposition of a tensor

The elementary QR decomposition considered here does not involve any reordering of dimensions. Thus, it is uniquely specified by the number l of leading tensor dimensions to be interpreted as "row" dimension in the matricization process, and correspondingly the remaining dimensions as "column" dimension.

To illustrate, the generated Python code (up to renaming variables) for the elementary QR decomposition of a tensor T and $l = 3$ leading dimensions reads as follows:

```
Q, R = np.linalg.qr(T.reshape((np.prod(T.shape[:3]),
       np.prod(T.shape[3:]))), mode='reduced')
Q = Q.reshape(T.shape[:3] + (Q.shape[1],))
R = R.reshape((R.shape[0],) + T.shape[3:])
```

The `reshape` functions implement the matricization before and "de-matricization" after the actual QR decomposition, `T.shape` stores the tensor dimensions, `np.prod` computes the product of the leading and trailing dimensions, and `np.linalg.qr` implements the conventional QR decomposition of matrices.

(iv) Singular-value decomposition of a tensor

The (de-)matricization process for an elementary singular-value decomposition (SVD) of a tensor is analogous to the elementary QR decomposition. The output now consists of three tensors, corresponding to the U, S and V matrices of the matrix-SVD, with the diagonal S matrix storing the singular values. Additional parameters (compared to the QR decomposition) are a cut-off tolerance for the singular values, and optionally the maximally allowed number of singular values (the maximal "bond" dimension).

Strategies for optimization

Based on the elementary tensor network operations, several high-level optimization strategies are conceivable, solely based on the rank of each tensor instead of the actual dimensions. The implementation of the following ideas is left for future work.

- A natural representation for the sequence of user actions is a directed acyclic graph (DAG), storing an elementary operation or input tensor at each node. Such a representation clarifies dependencies, and allows to determine which operations can be executed in parallel.
- A more subtle optimization strategy tailored to tensor networks is the merging of subsequent contractions, i.e., if the tensor resulting from a contraction is immediately used in another contraction. A toy model example consists of merging $C = AB$ followed by $y = Cx$ into $y = ABx$, which can then be evaluated in the order $y = A(Bx)$. Note that the optimized computational cost for the merged contraction cannot be higher than performing the contractions sequentially (since the latter restricts the allowed contraction order), but actually determining the optimal order

for the merged contraction (by a backend software package) is in general more difficult [7].
- Another optimization strategy is avoiding explicit transpositions (i.e., permutations of tensor dimensions), and aiming for advantageous dimension ordering. As mentioned, the transposition of a tensor resulting from a contraction can be integrated into the contraction (see also [13]), generalizing $(AB^T)^T = BA^T$ for matrices. Transpositions may also be pushed through the computational graph; for example, instead of permuting the leading dimensions of the Q-tensor resulting from a QR decomposition, one could already permute these dimensions in the input tensor, or vice versa. Conversely, in case a transposition has to be performed, optimized software packages are available [9, 14].

Quality control

The software has been extensively tested by comparing the tensor contraction and splitting indices computed by the software with the expected output, and by running the generated Python code (tested with Python 2.7.15 and 3.6.5, NumPy 1.13). The software should run on any modern web browser with JavaScript enabled (tested with Firefox 78.0, Chrome 84.0, Microsoft Edge 44.19041.1, Apple Safari 13.1.2).

(2) Availability

Operating system
Any, runs directly in a web browser.

Programming language
HTML and JavaScript combined with the D3.js library (v5).

Dependencies
D3.js library (v5) (no installation necessary).
The generated Python code requires NumPy (version 1.6.0. or higher).

List of contributors
Lisa Sahlmann and Christian B. Mendl both designed and documented the software, and tested its functionality. Christian B. Mendl implemented the software and maintains the GitHub repository and associated web page.

Software location
Code repository GitHub
  Name: guitenet
  Persistent identifier: https://github.com/GuiTeNet/guitenet
  Licence: MIT License
  Date published: 01/08/2018

Language
English

(3) Reuse potential
The canonical use case of the GuiTeNet software is code generation for tensor network operations. In its present form, the software framework is well suited to handle a relatively small number of tensors, but manually

constructing a network with hundreds of tensors is cumbersome. Instead, generating code for subroutines or blocks inside loops is a plausible scenario for employing GuiTeNet in larger software projects. As specific example, rather than instantiating all tensors of a matrix product state, the GuiTeNet framework could be used to generate a local contraction operation required during a left-right sweep over the chain.

We also want to point out the pedagogical value which GuiTeNet might offer, including the seamless transition from vectors and matrices to general tensors.

Still, there are many desirable features left for future work, including code generation for other programming languages and software libraries, the implementation of high-level optimization strategies as described above, or a timeline of previous network states (e.g., before a contraction) with associated *Undo* functionality. Tensors with special properties (like orthogonal tensors resulting from a SVD or QR decomposition) should be marked, e.g., using a different symbol, and ideally such properties should be exploited in the generated code. Furthermore, one could take $U(1)$-symmetries into account by endowing the legs with additive quantum numbers (like particle or spin) and a directional arrow. Conceptually, the sum of quantum numbers flowing into a tensor must be equal to the sum of quantum numbers leaving the tensor, enforcing a block sparsity structure of the tensor. Another worthwhile goal is incorporating more exotic tensor network operations, like "loop skeletonization" [15]. Contributions of such or similar features to GuiTeNet are very welcome; in practice, these should be realized by opening a pull request at https://github.com/GuiTeNet/guitenet/pulls.

Technical support is available via the "issues" page of the GitHub repository, or by contacting one of the authors by email.

An interesting open question is how GuiTeNet could inspire or profit from software and hardware architectures tailored to tensor operations, like contractions beyond conventional BLAS routines [13] or Google's Tensor Processing Units (TPUs) [16] employed in the TensorFlow machine learning framework.

During revision of this work we became aware of TensorTrace [17], a software package with a similar focus and approach as GuiTeNet.

Acknowledgements

Competing Interests
The authors have no competing interests to declare.

References
1. **Hackbusch, W** and **Kühn, S** 2009 A new scheme for the tensor representation. *J. Fourier Anal. Appl.*, 15: 706–722. DOI: https://doi.org/10.1007/s00041-009-9094-9

2. **Hackbusch, W** 2014 Numerical tensor calculus. *Acta Numerica*, 23: 651–742. DOI: https://doi.org/10.1017/S0962492914000087

3. **Schollwöck, U** 2011 The density-matrix renormalization group in the age of matrix product states. *Ann. Phys.*, 326: 96–192. DOI: https://doi.org/10.1016/j.aop.2010.09.012

4. **Verstraete, F, Murg, V** and **Cirac, J I** 2008 Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Adv. Phys.*, 57: 143–224. DOI: https://doi.org/10.1080/14789940801912366

5. **Vidal, G** 2008 Class of quantum many-body states that can be efficiently simulated. *Phys. Rev. Lett.*, 101: 110501. DOI: https://doi.org/10.1103/PhysRevLett.101.110501

6. **Pfeifer, R N C, Evenbly, G, Singh, S** and **Vidal, G** 2014 NCON: A tensor network contractor for MATLAB. *arXiv:1402.0939*.

7. **Pfeifer, R N C, Haegeman, J** and **Verstraete, F** 2014 Faster identification of optimal contraction sequences for tensor networks. *Phys. Rev. E.*, 90: 033315. DOI: https://doi.org/10.1103/PhysRevE.90.033315

8. **Evenbly, G** and **Pfeifer, R N C** 2014 Improving the efficiency of variational tensor network algorithms. *Phys. Rev. B.*, 89: 245118. DOI: https://doi.org/10.1103/PhysRevB.89.245118

9. **Solomonik, E, Matthews, D, Hammond, J R, Stanton, J F** and **Demmel, J** 2014 A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74: 3176–3190. DOI: https://doi.org/10.1016/j.jpdc.2014.06.002

10. **Springer, P** and **Bientinesi, P** 2016 Design of a high-performance GEMM-like tensortensor multiplication. *arXiv:1607.00145*.

11. **van der Walt, S, Colbert, S C** and **Varoquaux, G** 2011 The NumPy array: A structure for effcient numerical computation. *Comput. Sci. Eng.*, 13: 22–30. DOI: https://doi.org/10.1109/MCSE.2011.37

12. https://docs.scipy.org/doc/numpy/reference/generated/numpy.einsum.html.

13. **Matthews, D** 2018 High-performance tensor contraction without transposition. *SIAM J. Sci. Comput.*, 40: C1–C24. DOI: https://doi.org/10.1137/16M108968X

14. **Springer, P, Su, T** and **Bientinesi, P** 2017 HPTT: A high-performance tensor transposition C++ library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2017, pages 56–62. ACM. DOI: https://doi.org/10.1145/3091966.3091968

15. **Ying, L** 2017 Tensor network skeletonization. *Multiscale Model. Sim.*, 15: 1423–1447. DOI: https://doi.org/10.1137/16M1082676

16. **Jouppi, N P, Young, C, Patil, N,** et al. 2017 In-datacenter performance analysis of a Tensor Processing Unit. In *ISCA'17 Proceedings of the 44th Annual International Symposium on Computer Architecture*. DOI: https://doi.org/10.1145/3079856.3080246

17. **Evenbly, G** 2019 TensorTrace: an application to contract tensor networks. *arXiv:1911.02558*.