julïacon

DMRGenie: Graphical interface for entanglement renormalization

Kiana Gallagher[1], Aaron Dayton[1], Drew Leske[2], Sarah Huber[2], and Thomas E. Baker[1, 3, 4]

[1]Department of Physics & Astronomy, University of Victoria, Victoria, British Columbia V8P 5C2, Canada
[2]Research Computing Services, University of Victoria, Victoria, British Columbia V8P 5C2, Canada
[3]Department of Chemistry, University of Victoria, Victoria, British Columbia V8P 5C2, Canada
[4]Centre for Advanced Materials and Related Technology, University of Victoria, Victoria, British Columbia V8P 5C2, Canada

ABSTRACT

We discuss implementation of a graphical user interface for tensor operations in scientific simulations related to entanglement renormalization. We call this software DMRGenie based on its heavy reliance on Genie which is a well-known software package for web applications. We also discuss the deployment of the software. Scientific simulations with tensor network-based framework of quantum algorithm methods are now possible through this tool without lengthy start-up or learning times.

Keywords

Julia, Genie, DMRjulia, TENPACK, Entanglement Renormalization, Tensor Networks, Graphical User Interface

1. Introduction

On one hand, for ideas to be implemented computationally, a need arises for easy-to-use high level languages. "On the other hand, scientific programming often requires computational resources at scale, driving optimization and performance requirements. To accomplish this second demand, a low-level language is required to control as much as possible in an implementation of an algorithm. This split between needs for different programming projects is known as the two-language problem.

The Julia programming language is a good if not great attempt to solve this two-language problem. By optimizing for loops, programs can be created in a rapid manner but still achieve performance on par with lower-level languages such as C++.

In any scientific endeavour, the need to use numerical techniques is highly necessary. And for new advances to be made, new methods must be able to be implemented into any given language to evolve beyond the state-of-the-art. For anyone developing new algorithms, the need for a programming framework that enables as many solutions of the two-language problem as possible is necessary, making a Julia a strong contender for algorithmic development.

Quantum information methods have become a tantalizing path to creating new algorithms and methods. These range from quantum algorithms implemented on quantum computers [11] (but since no
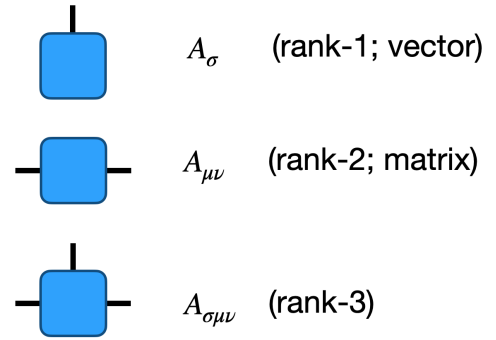

Fig. 1: Shown are diagrams for rank-1, 2, and 3 tensors. Each line signifies another index on the tensor.

perfect quantum computer exist, simulations with classical methods are still required) to entanglement renormalization techniques.[1] Methods of entanglement renormalization use quantities native to quantum information (*i.e.*, the density matrix or the entropy) in order to optimize a a quantum problem to find solutions to the eigenvalue problem (Schrödinger's equation) [14]. Entanglement renormalization techniques are formulated on a graph of tensors and have definite rules for tensor operations. Because tensor computations and operations can be cumbersome to represent mathematically in index notation, the field has adopted a diagram notation that allows for the improved communication of algorithms. Namely, the rank of the tensor corresponds to the number of lines on the diagram [12]. An example is shown for rank-1, 2, and 3 tensors in Fig. 1.

---

[1]This class of methods is more commonly referred to as tensor networks, simply, but with the rise of machine-learning and other methods particularly in quantum chemistry that are formulated on tensor networks, a distinguishing name is required. We choose to expand the usage of the term from Ref. [17] since all methods are fundamentally optimizing over quantities relevant for quantum information (*i.e.*, the entanglement) and renormalization was originally introduced to describe this class of methods [18, 9]. The methods required to implement an algorithm in this context is different from other contexts, notably machine learning.
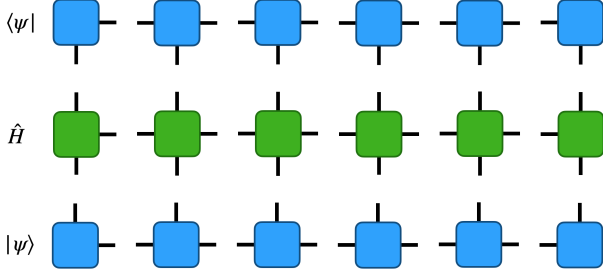
Fig. 2: The expectation value of an operator $\hat{H}$ is shown in tensor diagram notation for a 6-site system. When contracted, this would give $\langle \hat{H} \rangle$, or the energy E if $\hat{H}$ is the Hamiltonian. The wavefunction on the bottom row alone would be represented as in the text (Eq. 1), which is far more intricate.

Realistically, only about a rank-14 tensor can be represented on a laptop machine. This is because each of the tensor indices has dimension of at least two, approaching the memory limitations of these devices. Therefore, the rank of the tensors represented here are small and often only a few indices are present.

When contracting two tensors together, one can connect two indices on the same tensor (*i.e.*, tracing over indices) or between tensors (contraction). When representing a network of tensors, the indices are typically not kept track of. Instead, indices that point towards each other or have matching labels are assumed to be contracted eventually. This rarely causes confusion when implementing algorithms.

The use of the diagram notation is useful for representing large numbers of tensors in a coherent way without representing cumbersome mathematics (see Fig. 2). For example, the wavefunction when decomposed through a series of reshapes and singular value decompositions (SVD) takes the form [4]

$$|\psi\rangle = \sum_{\substack{\sigma_1 \sigma_2 \sigma_3 \\ \sigma_4 \sigma_5 \sigma_6}} \sum_{\substack{a_1 a_2 a_3 \\ a_4 a_5}} A_{a_1}^{\sigma_1} A_{a_1 a_2}^{\sigma_2} A_{a_2 a_3}^{\sigma_3} A_{a_3 a_4}^{\sigma_4} A_{a_4 a_5}^{\sigma_5} A_{a_5}^{\sigma_6}$$
$$\times |\sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 \sigma_6\rangle \qquad (1)$$

which is far more intricate than the diagram notation but conveys no extra information. Any quantum operator (for example, the Hamiltonian) can be decomposed into a similar site-by-site form involving many tensors.

Building these algorithms from a purely diagrammatic level is a desirable possibility. This opens the possibility to using a graphical user interface to denote the various steps in the implementation of the entanglement renormalization algorithms

Julia's solution of the two-level problem allows for an easy implementation of algorithms and simultaneous fast execution of scientific code. We discuss in this article efforts to construct a fully working scientific software package for entanglement renormalization that is simultaneously fast and lightweight. This software package has then been extended to include a user interface that allows for the graphical construction of algorithms without any coding background, our principle goal here.

We will first discuss the background software (DMRjulia and TEN-PACK; available in Julia's package system as `DMRJtensor` and `TensorPACK`) originally developed in 2016 and first introduced to Github in 2020 and full debut on the Julia package environment in 2021. This means that the library has spanned many version of the Julia version starting originally in v0.5 and now continuing to v1.11 and beyond. We believe that some choices from our implementation may be interesting to the broader community, although many of the improvements that were introduced here later appeared in this development independently appeared in Julia from other groups, showing that common issues are being encountered and handled in similar ways.

DMRGenie is the natural extension of the previously developed software which we debut here. This builds on top of DMRjulia to include a graphical interface for easy development of new algorithms. This is made publicly available through a web interface and available for use with some limitations on the maximum size of the tensors and enforce reasonable computational requirements [8].

2. Entanglement renormalization with DMRjulia

DMRjulia is a software package in the Julia package environment that is 100% written in the Julia programming language. This makes the software fast and light-weight, ultimately taking only a few kilobytes to install, and optimized to the point of being as fast as lower-level implementations of the same algorithms.

The library implements several classes of tensor network anstaz to represent wavefunctions of quantum problems. This includes the matrix product state (MPS) [16, 5], projected entangled pair states (PEPS) [15], multiscale entanglement renormalization anstaz (MERA) [17], and classical tensor network algorithms such as the tensor renormalization group (TRG) [10]. Each of these implementations is done with care towards ease of use, culminating in the DMRGenie implementation here. All of these tensor networks are implemented in a fully flexible way.

The algorithms can be accessed and coded in a variety of styles from the basic manipulation of tensors with four basic operations (reshaping, permuting tensors, contraction, decomposition) [4, 3, 6]. Other operations such as joining tensor indices together (as a generalization of the direct sum operation on vectors or matrices) is also available.

We have also introduced a `nametens` system to attach strings to each tensor index and formulate rules such that the $*$ operator is overloaded to find all common names and contract them.[2] This provides another interface to develop algorithms that is useful in some cases.

The implementation here for methods of entanglement renormalization is given as 100% of Julia code with no dependencies beyond the basic library.[3]

3. Tensor algebra with TENPACK

The issue of type stability has been of paramount concern to ensure that any code developed in the language avoids generating too much overhead because of Julia's multiple dispatch capabilities. Since the compiler in Julia must assume that any possible input can be entered into a function, the assumptions of how types are converted and exactly what type is created can generate excess allocations which ultimately slow a program. Eliminating these excess allocations is necessary to create efficient code that is competitive with lower level languages.

---

[2]Details on all functions in the library are available by typing ? followed by the name of the function in the Julia terminal.

[3]As of this article, the library is in an extended beta-testing phase. As we add more users, we are creating useful error checks for algorithms that can be created. The library will be moved to the alpha phase after this is complete, and we welcome feedback and development from anyone even if this phase has passed.

For example, when reshaping a tensor, the rank of the tensor changes. The implementation of the basic `Array` type in Julia contains both the element data type and also the rank of the tensor. Because the rank of the tensor is a fluid quantity in entanglement renormalization methods, this naturally creates issues with the base implementation of Julia's library and a desire to reduce allocations for greater efficiency with an alternative struct.

As a bridge between the operations on tensors in DMRjulia and Julia, as well as the lower level BLAS functions that Julia relies on, the Tensor (Linear Algebra) Package (TENPACK) was created. The package is implemented entirely in Julia with no dependencies beyond the basic Julia implementation.

The basic tensor type in TENPACK is the `denstens` abstract type which has the `tens` struct. This closely mirrors the definition of the `Array` type in base Julia, but it only records the element type of the tensor. There are two fields for this struct: `.size` (to store the current size of the tensor) and `.T` field (storing the elements of the tensor as a vector, no matter the current shape). The `.size` field contains a `Memory` type that was recently introduced in v1.11. This means practically that this implementation will allow for better determination of the type when using these custom structs and throw allocations in some non-standard places (*i.e.*, when the size field is requested as a tuple, the `Memory` in `.size` is converted to a tuple generating one allocation). We find overall that the changes we imposed make for a faster implementation especially on smaller systems, although it is difficult to compare fully because it is merely a trade-off and makes compilation into an executable easier, particularly with the recently developed trimming feature to make executables far smaller than previous implementations.

The design choices of TENPACK purposefully backtracks somewhat on the original promise of Julia in that most every function is fully typed for fixed inputs, so the code is not fully flexible and makes a stronger play for efficiency over flexibility. This can make the lowest levels of the library somewhat more difficult to maintain and manage with future feature improvements. However, we are able to run scalable simulations in this library that is created entirely in Julia.

We also found it necessary to re-code some of the basic Julia functions into a form that natively admits the `tens` type. Notably, the implementation of `permutedims`, `setindex`, and `getindex` were coded to avoid issues when updating versions that sometimes caused type instabilities, although care is taken to ensure that these functions (and others such as `svd`) are as efficient as the base Julia library). The goal when doing in this in the library is to always to match the performance and memory consumption of the basic Julia functions, but we want to ensure type-stability across versions. In some cases, a more rudimentary version of the function is used [13] or an alternative LAPACK function is called to ensure stability on the types of input tensors we provide.[4]

The original hope for Julia was to solve the two-language problem, and that is still largely solved in DMRjulia and TENPACK. We have merely made some compromises to ensure that there is efficient implementation because our use case is narrower than the most broad implementation that Julia strives to deploy.

---

[4] For example, the basic implementation of `svd` now uses `LAPACK.gesdd!` (divide and conquer method) but this can return an error in some cases that we have noticed for our inputs. The `svd` function in TENPACK natively calls `LAPACK.gesvd!`. For the faster divide and conquer method, we have made a new function `fsvd` (fast SVD, divide and conquer) and similar to still provide access to this and we do use this in some cases.

4. Graphical interface with DMRGenie

The DMRGenie graphical user interface (GUI) allows for tensor network algorithms to be run with minimal to no coding required, and using diagrams which are easier to read. DMRGenie consists of the Algorithm Runner and the Tensor Network Builder. The Algorithm Runner executes tensor network algorithms such as the density matrix renormalization group (DMRG) using known models or custom Hamiltonians [6]. The Tensor Network Builder allows the user to build their own tensor network and create their own tensor network algorithm using a drag-and-drop system. The frontend was built using HTML, JavaScript, React, and CSS for styling. The backend uses a Model View Controller (MVC) infrastructure. The packages used to run the tensor network algorithms are DMRjulia and TENPACK. So, while both DMRjulia and TENPACK are 100% from-scratch Julia implementations, DMRGenie is not, but this makes the code easier to use and work with and adapt to changing software standards in the future.

4.1 DMRGenie Backend

The DMRGenie website was built using a Model View Controller (MVC) infrastructure, which was initialized using the Genie.jl Julia package. Genie was chosen because it allowed for full-stack development with Julia as the backend. This improved the development cycle as it made it simple to run tensor network functions with DMRjulia and TENPACK. The models and controls are written in Julia, while the views are written in Julia and HTML using a `.jl.html` file extension with supporting JavaScript files for functionality.

The model is a struct which is used to pass data between the frontend and backend of the application. When a page is initially loaded, a model is initialized with a default set of data. Once the user inputs their options and submits, that raw data is sent to the backend where tensor network computations can then be performed accordingly.

The view file contains the HTML for the application. This file is capable of containing Julia code due to its ".jl.html" file extension, allowing the controller to pass Julia variables to be output on the frontend.

The controller is responsible for starting up the HTML page with the relevant data when the website is initialized. When a tensor network is created or a tensor network function is run, the controller is responsible for creating the tensors and running the algorithm using DMRjulia and TENPACK. The results of any operation are stored in variables and returned to the view component.

Much of the data collection is handled in the routes. When a computation is submitted, the routes receives this request, parses through the data, then sends it in a usable format to the corresponding controller. Data is sent to the backend in a POST request and collected using the `postpayload` Genie function for the Algorithm Runner, while for the Tensor Network Builder it is sent as JSON then parsed using the JSON3 Julia package.

4.2 DMRGenie Frontend

The two main components of DMRGenie are the tensor network builder and the algorithm runner. The frontend of DMRGenie was built using HTML, CSS, and JavaScript along with ReactJS. The only ReactJS component of the app was the Tensor Network Builder. ReactJS was used along with ReactFlow which is a package meant for building node-based editors and interactive diagrams.

Fig. 3: The Heisenberg Model input for the Algorithm Runner written term-by-term as in Eq. (2).

4.3 Algorithm Runner

The Algorithm Runner is an interface that allows the user to build their own models or run known models to get the ground state energy of the system. The number of sites for a system and its Hamiltonian must be specified. The current supplied Hamiltonians are for the Hubbard model, the $t - J$ model, and the Heisenberg model [7]. Custom Hamiltonians can be built by the user by specifying the physical dimension of the system, the Hamiltonian constant, and the components of the Hamiltonian.

For example, if the user wanted to input a Heisenberg Hamiltonian of

$$\hat{H} = -J \sum_i \left(\sigma_i^x \sigma_{i+1}^x + \sigma_i^y \sigma_{i+1}^y + \sigma_i^z \sigma_{i+1}^z \right) \qquad (2)$$

where J is the Hamiltonian constant, then it would be possible to do so term by term. An example of the input is given in Fig. 3.

It is also possible to compute the correlations and impose symmetries on the system. The correlation matrix operators and correlation function operators must be specified to compute correlations. The matrix operators and function operators will then be evaluated at all positions on the lattice.

Details of the symmetries and how they need to be implemented are included in Appendix B.

4.4 Tensor Network Builder

The Tensor Network Builder is a drag and drop interface to build tensor networks. The rank of any tensor is specified before they
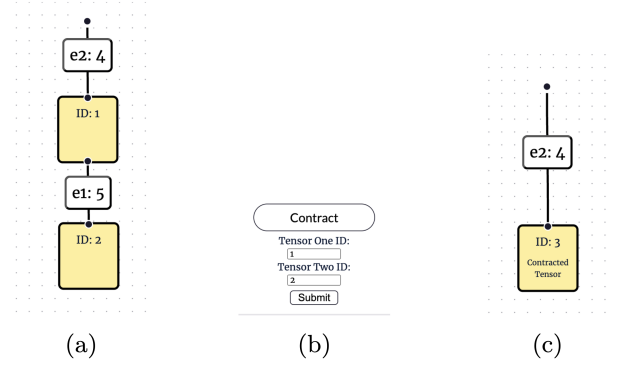

Fig. 4: Contraction two tensors with the Tensor Network Builder interface. a) The initial tensors. b) The input given to the contract menu operation. c) The output tensor.

are dropped onto the canvas. The number of connections points on a tensor is equal to the rank of the tensor. Indices can be created by dragging from a connection point on a tensor. It is possible to specify the name and dimension of all indices, and it is possible to rename an index if needed. Possible operations on tensors include contractions (see example of interface in Fig. 4), singular value decompositions, eigenvalue decompositions, QR decompositions, and LQ decompositions. When the user is done, the "submit" button can be clicked and the algorithm will be run on the backend. The output of the algorithm will be the tensors that resulted from the last operation. The output is given as a downloadable file.

ReactJS is used for the tensor network builder because of component reusability and the fast rendering. The ReactFlow package was designed to build node-based editors and interactive diagrams; however, graphs in computer science have similar structures to tensor networks. The nodes and edges in ReactFlow were generalized to tensors and indices.

5. Deployment

The application is packaged into a Docker container using GenieDeployDocker and Docker tools. The container image is then pushed to a local image registry and scanned for vulnerabilities before deployment to a Kubernetes cluster, where it is made available on the web. The registry, cluster and web access are components of a self-serve application deployment platform.

6. Conclusion

A graphical user interface for entanglement renormalization has been presented here in DMRGenie. This builds upon the previous DMRjulia and TENPACK packages to make coding novel quantum algorithms far easier and more visual.

7. Installation and usage

The basic libraries for DMRjulia and TENPACK can be obtained from online repositories [1, 2] or from the Julia package environment:

```
julia> ]
pkg> add DMRGenie DMRJtensor TensorPACK
```

The DMRGenie GUI can be used online [8].

8. Acknowledgement

9. References

[1] Thomas E. Baker. DMRjulia. `https://github.com/bakerte/DMRJtensor.jl`, 2020.

[2] Thomas E. Baker. TENPACK. `https://github.com/bakerte/TensorPACK.jl`, 2020.

[3] Thomas E Baker, Samuel Desrosiers, Maxime Tremblay, and Martin P Thompson. Basic tensor network computations in physics. *arXiv preprint arXiv:1911.11566, p. 20*, 2019.

[4] Thomas E Baker, Samuel Desrosiers, Maxime Tremblay, and Martin P Thompson. Méthodes de calcul avec réseaux de tenseurs en physique. *Canadian Journal of Physics*, 99:4, 2021. doi:10.1139/cjp-2019-0611.

[5] Thomas E Baker and Negar Seif. Bundled matrix product states represent low-energy excitations faithfully. *Journal of Physics A: Mathematical and Theoretical*, 57(39):395306, 2024. doi:10.1088/1751-8121/ad770f.

[6] Thomas E Baker and Martin P Thompson. DMRjulia I. Basic construction of a tensor network library for the density matrix renormalization group. *arXiv preprint arXiv:2109.03120*, 2021.

[7] Eduardo Fradkin. *Field theories of condensed matter physics*. Cambridge University Press, 2013.

[8] Kiana Gallagher, Aaron Dayton, and Thomas E. Baker. DMRGenie. `https://dmrgenie.rs.uvic.ca/`.

[9] HR Krishna-Murthy, JW Wilkins, and KG Wilson. Renormalization-group approach to the anderson model of dilute magnetic alloys. i. static properties for the symmetric case. *Physical Review B*, 21(3):1003, 1980. doi:10.1103/physrevb.21.1003.

[10] Michael Levin and Cody P Nave. Tensor renormalization group approach to two-dimensional classical lattice models. *Physical review letters*, 99(12):120601, 2007. doi:10.1103/physrevlett.99.120601.

[11] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[12] Roger Penrose. Angular momentum: an approach to combinatorial space-time. *Quantum theory and beyond*, 151, 1971.

[13] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. Numerical recipes in c++. *The art of scientific computing*, 1992. doi:10.1088/0143-0807/24/3/701.

[14] John S Townsend. *A modern approach to quantum mechanics*. University Science Books (Sausalito, California), 2000. doi:10.1119/1.17394.

[15] Frank Verstraete and J Ignacio Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv preprint cond-mat/0407066*, 2004.

[16] Frank Verstraete and J Ignacio Cirac. Matrix product states represent ground states faithfully. *Physical Review B—Condensed Matter and Materials Physics*, 73(9):094423, 2006. doi:10.1103/physrevb.73.094423.

[17] Guifre Vidal. Entanglement renormalization. *Physical review letters*, 99(22):220405, 2007. doi:10.1103/PhysRevLett.99.220405.

[18] Kenneth G Wilson. The renormalization group and critical phenomena. *Reviews of Modern Physics*, 55(3):583, 1983. doi:10.1103/RevModPhys.55.583.

APPENDIX

A. Running locally

DMRGenie can be run locally and we show how to do this here.

A.1 Setup

Navigate to the 'tensez' directory once the repository has been cloned and then into the 'DMRGenie' folder within it.
Install the necessary packages with instantiate.

```
julia> ]
pkg> instantiate
```

A.2 Running the App

The app must be activated by running the following from the 'DMRGenie' directory

```
julia> ]
pkg> activate .

julia> using Genie
julia> Genie.loadapp()
```

B. Quantum symmetries

TENPACK currently supports local $U(1)$ and $\mathbb{Z}_n$ symmetries. The basic idea is to establish the relevant quantum numbers on each physical index (drawn as the vertical index on the MPS). Specifying the quantum numbers on each of the physical indices is sufficient to determine the quantum numbers on all other indices, provided that

the total flux of the input Hamiltonian is a net zero (*i.e.*, symmetry-preserving Hamiltonian). Given this case, all tensors exhibit a block structure, where each block has a set of non-zero values and no non-zero values are available between quantum number sectors.

Given a symmetry preserving Hamiltonian, one can use rules similar to Kirchoff's loop rules (with total flux equal to zero) to determine the quantum numbers on each index. All of this is automatically handled in the quantum number tensor implementation in TENPACK for abelian quantum numbers.

For example, consider the Hamiltonian for two spin-half objects on two different sites

$$\mathbf{S}_1 \cdot \mathbf{S}_2 = \frac{1}{4} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 2 & 0 \\ 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3}$$

where $\mathbf{S} = \boldsymbol{\sigma}/2$ for the Pauli vector $\boldsymbol{\sigma}$ [14] and subscripts denote which site the spin vector is located. The matrix can be divided into three blocks, and in general, we can write a block structure of completely independent pieces as

$$\mathbf{S}_1 \cdot \mathbf{S}_2 = 1 \oplus \begin{pmatrix} -1 & 2 \\ 2 & -1 \end{pmatrix} \oplus 1 \tag{4}$$

where $\oplus$ is a direct sum. Each of these three independent pieces can be assigned a quantum number (+1, 0, -1; in order). This means that if we were to multiply this matrix with another matrix respecting the same symmetries, we could break the full matrix multiplication into parts. In this case, consider multiplying $\mathbf{S}_1 \cdot \mathbf{S}_2$ by itself and noting that one could perform three matrix multiplications, one on each quantum number sector, and obtain the same result.

This idea generalizes to all tensors that respect a quantum number symmetry. When reshaping an index, one can simply sum the abelian quantum numbers together to find the quantum number on the resulting grouped index. Eventually, one will need to contract a tensor. By identifying which indices are contracted and which are not, one can reshape any tensor into a matrix that must be divisible into quantum number blocks.

The implementation in TENPACK has been optimized so that it is fast. This methodology also avoids any issue with numerical round-off error accidentally changing the desired quantum number sector for a given problem. For example, not imposing quantum number symmetries for fermions can lead to a loss or gaining a fermion, which may resulting in finding an answer for a different symmetry sector than was originally intended. it is recommended to use quantum number symmetries whenever the problem displays quantum numbers.

C. Builder's guide

The frontend for the Tensor Network Builder was built using HTML, CSS, JavaScript, and React. It is recommended that the developer has experience in these languages. The majority of the interface is coded in React, and the styling is done in CSS. The React package used is **ReactFlow**. This specific package was chosen because it is used to create graphs commonly found in computer science, and the package was well documented. Computer science graphs and tensor networks have quite a bit of overlap in their structure. The nodes and edges can easily be generalized to tensors and indices. The website has two main components: the network canvas and the drag and drop menu.

Any functions called using ReactFlow are of the form **ReactFlow.FunctionName()**. This was due to an issue importing ReactFlow into the Genie environment. It should be noted that all the ReactFlow elements should be inside the <**ReactFlow.ReactFlowProvider**> element as this is the React-Flow environment.

C.1 Network Canvas

The network canvas is where the tensor network is displayed. The current canvas is provided by the ReactFlow package. The network canvas is a **div** element inside the <**ReactFlow.ReactFlowProvider**> element. The canvas is where the tensors are displayed, moved, and edited.

The <**ReactFlow.ReactFlow**> element is used to specify the functions called when specific events occur. These events include deleting tensors, deleting indices, moving tensors, and validating connections. If no function is passed through the element then the default functions are used which can be found on the React-Flow website documentation. The <**ReactFlow.Controls**> element is for the controls at the bottom left corner of the canvas. The <**ReactFlow.MiniMap**> element is for the mini version of the canvas at the bottom right. The <**ReactFlow.Background**> element is for the background of the canvas

C.2 Drag and Drop Menu

The drag and drop menu allows the user to create random tensors, upload tensors, and apply operations on created tensors. It is contained inside the <**ReactFlow.ReactFlowProvider**> element. The drag and drop menu has a tag of <**DNDMenu**>. It also contains all the elements for the tensor operations.

The variables that are used for all the tensor operations are passed through the drag and drop menu element. The drag and drop menu also keeps track of which operation is currently active and only allows one active operation at a time. This was a design choice, but it is not necessary.

C.3 Tensors

Nodes in ReactFlow are created by making an object in JavaScript that has specific properties and passing it to the **setNodes** function. This function will create a node on the canvas with the specified values. The object must contain certain properties such as **id**, **position**, and **type**. There are properties that are optional such as **style** and **data**. There is no strict limit to the properties, but any custom properties should be passed through the value linked to the data property.

Two custom nodes were created to represent tensor networks. The two types are **tensor** and **invisible**. The code for these tensors can be found in **TensorNode.js** and **InvisibleNode.js**. To make the simplest node, a empty div element can be returned, but we want to be able to connect tensors. The <**ReactFlow.Handle**> element is used to create a handle that allows edges to be connected to it. So a custom node is created by creating a div and specifying the handles and any other desired HTML or React elements .The tensor type is used to represent a tensor in a tensor network. The invisible type is required for free indices because the package required edges to have two connection points.

When creating a tensor type node, the **data** value is an object that must include certain properties such as the **rank** as an Int, **mutable** as a boolean, **label** as a string, and **color** in the form of an Int from 1-100. Furthermore, the style property is an object that should include a **backgroundColor** property in the form of a string; it is currently "hsl(" + tensorColor + ", 100%, 80%)". If the tensor is being created by uploading then the data property must also include

a **input** property as a string. The string is currently "", but this can be changed.

In the case of an invisible node, it is much simpler. The **data** value must contain a **mutable** property which should be a boolean value. The rank of the tensor is equal to the number of handles for the tensor, so a tensor of rank 3 can have only have 3 connection (a connection to any type of node is considered a connection). Additionally, the IDs must be unique, so a **nodeId** variable is incremented every time a tensor node is created. Similarly, if an invisible node is created then its ID is 'f${freeNodesId}' where **freeNodesId** is incremented every time an invisible node is created. If a node is deleted then all its edges will be deleted or split. If the tensor is connected to another tensor then that edge will be split otherwise the edge will be deleted. This is done by the **onNodesDelete** event.

C.4 Indices

Connecting two tensors is done by dragging from one handle of a tensor to a handle of another tensor. Free indices are created by dragging from a handle of a tensor onto the network canvas. Similar to nodes, edges are created by making an object in JavaScript with specific properties and passing it to the **setEdges** function. This function will create the edges on the canvas. The object must contain certain properties such as **id**, **sourceX**, **sourceY**, **targetX**, **targetY**, **sourcePosition**, **targetPosition** and **type**. There are properties that are optional such as **style** and **data**.

Similar to the nodes, custom edges can be made. There is only one type of custom edge required which is of type **index**. The code for the edge type can be found in **DimensionEdge.js**. For an index type edge, the data value must contain the property **dim** which is a positive Int, **chosenDim** which is a boolean, and **lineStyle** which is "Bezier" or "Free". The dimension of an index is immutable, so once the user choses the index then **chosenDim** becomes true. The **lineStyle** is "Free" if the edge is connected to an invisible type node otherwise it is "Bezier".

Since IDs must be unique, if a edge is created then it's ID is 'e${eId}' where **eId** is incremented every time an invisible edge is created. If an edge is deleted and it connected to an invisible node then the invisible node will be deleted as well. This is done by the **onEdgesDelete** function.

C.5 Contract

The contract operation accepts two different tensor IDs and contracts the tensors with these IDs. The contraction will cause the two tensors to be replaced with a new tensor that represents the contraction of the original tensors. The two tensors are contracted along indices that are named the same. Therefore an index that directly connects two tensors will be contracted. If tensor one and tensor two both have an index labeled the same thing and they are the same dimension then that index will also be contracted along. This is done by deleting the given two tensors and all indices that are being contracted along. Any indices that are not contracted will still be connected to the new resulting tensor.

An error will occur if the contraction is invalid or too many indices have the same name, or either tensor does not exist.

C.6 SVD Decomposition

The SVD composition accepts the ID of a single tensors and two groups of the tensor's indices. The decomposition will result in 3 tensors. The tensors will be two complex unitary matrices and a diagonal matrix. This is done by replacing the original tensor by these 3 tensors. The tensor that results from the first group of in-

dices will be connected to the tensors that the original edges were connected to (excluding the original tensor) and the diagonal matrix. The same applied to the tensor produced by the second group of indices. Th diagonal matrix is connected to the unitary matrices. Errors will occur if the matrix does not exist or not all the indices are grouped properly.

C.7 QR/LQ Decompositions

The QR/LQ decomposition accept the same input as the SVD decomposition. The resulting tensors will be the Q(L) and R(Q) matrices if the QR(LQ) decomposition is done. The tensor that results from the first group of indices will be connected to the tensors that the original edges were connected to (excluding the original tensor). The same applied to the tensor produced by the second group of indices.

An error will appear if the original tensor does not exist or the indices are not grouped properly.

C.8 Eigenvalue Decomposition

The eigenvalue decomposition accepts the ID of a single tensor are returns three square matrices. These matrices are the diagonal matrix of eigenvalues, the matrix of eigenvectors, and the matrix of inverse eigenvectors. This is done by replacing the original matrix with these square matrices and connecting them (in the order of eigenvectors-eigenvalues-inverse eigenvectors).

An error will occur if the selected tensor is not isolated or it is not a square matrix.

C.9 Split Edge

Splitting an edge does not change the tensors in the backend, but it makes the tensor network easier to understand visually. Splitting an edge takes the label of the edge as input. The tensors that the original edge is connected to will then be connected to an invisible tensor instead.

This is done by creating two new edges and two new invisible nodes. The new edges will have the same label as the original edge.

C.10 Connect Edge

Connecting an edge does not change the tensors in the backend, but it makes the tensor network easier to understand visually. Connecting edges accepts two edge labels that are connected to invisible nodes as input. It will then connect the tensor nodes directly. This is done by deleting the two original edges and replacing it with a single edge. The new edge will have the label of the first input edge. An error will occur if the dimension of the indices do not match or one of the indices does not exist.

C.11 Copy and Paste

Tensors are copied by selecting a group of isolated tensors and either clicking the copy button or using **command+c**. It does not matter if an edge or a invisible node is selected. An error will occur if the tensors are not isolated. First, the copy function will gather all the tensors that have been selected. It will then copy all the invisible nodes connected to the tensor nodes. The edges connected to all nodes are then copied. The copying step requires the IDs of all edges, and nodes to be updated as they must be unique. Furthermore, the target/targetHandle and source/sourceHandle must be updated for all the edges or they will connect to the original target and source.

Pasting what has been copied will add the saved nodes and edges to the network canvas. In the case that the original tensor(s) need to be duplicated multiple times then the pasting function will preform a copy on the current copied contents. This guarantees that it is possible to duplicate the same thing multiple times. Otherwise the IDs would not be unique and thus would cause bugs.

C.12 Undo and Redo

A history is saved for each operation. The status of each tensor and edge is saved as a history, so it possible to revert the status. If a new operation is done then the previous history will be rewritten.

C.13 Adding New Operations

Adding new operations requires the frontend to produce an accurate representation of the operation being done. It is easiest to use tensor IDs and edge labels as the input. Once those have been identified, it is important to determine which edges or nodes must be removed, updated, or added. It is possible to get the lists of nodes and edges using const **getNodes** and **getEdges**. It is then possible to set the nodes and edges using **setNodes** and **setEdges**. These functions can be called using **ReactFlow.useReactFlow()**.

C.14 Connecting to Backend

When an operation is done, it passes the name of the operation done and the JavaScript objects of the tensors and edges involved.

C.15 Chosen Conventions

—**invisible** nodes are **ALWAYS** the target node, never the source. Changing this will cause some errors in the code.

C.16 Additional Info

—Multiple elements can be selected by **holding command+clicking**
—React elements must start with a capital letter or errors may occur (this is a convention but it is easy to forget when learning)