DMRjulia (v0.8.7): Tensor recipes for entanglement renormalization computations

Thomas E. Baker†

Department of Physics, University of York, Heslington, York YO10 5DD, United Kingdom
and
Institut quantique & Département de physique, Université de Sherbrooke, Sherbrooke, Québec J1K 2R1 Canada

Martin P. Thompson

Institut quantique & Département de physique, Université de Sherbrooke, Sherbrooke, Québec J1K 2R1 Canada

(Dated: November 30, 2021)

Detailed notes on the functions included in the DMRjulia library are included here. This discussion of how to program functions for a tensor network library are intended to be a supplement to the other documentation dedicated to explaining the high level concepts. The chosen language used here is the high-level julia language that is intended to provide an introduction to provide a concise introduction and show transparently some best practices for the functions. This document is best used as a supplement to both the internal code notes and introductions to the subject to both inform the user about other functions available and also to clarify some design choices and future directions.

This document presently covers the implementation of the functions in the tensor network library for dense tensors. The algorithms implemented here is the density matrix renormalization group. The document will be updated periodically with new features to include the latest developments.

CONTENTS

I. Introduction 3
A. Starting to use the library 3
B. What paper should be cited? 3
C. Debugging tips 4
1. Debugging tips 4
D. Comment on time to run the code 4
E. Some common implementation features in julia 4
1. Use of where statements 5
2. Imaginary numbers 5
3. Other functionality 5
4. Uniformity of input numbers and types 5
F. Structure of the library 5
G. Some comments on the julia programming language 5

II. Acknowledgements 6

References 6

I Dense tensors 7

III. Main file: DMRjulia.jl & Overview 7

IV. Module: tensor.jl 8
A. Types: denstens and general tensor types 8
1. Type: denstens 8
2. Type: qarray 8
3. Type: TensType & denstensType 9
4. Type: intType 9
5. Type: intvecType 9
6. Type: genColType 9

B. Struct: tens 10

C. Constructors 10
1. Function: tens 10
2. Function: rand 11
3. Function: zero 11
4. Function: makeId 11
5. Function: convertTensor 12
6. Function: madeens 12
7. Function: makeArray 13

D. Helper functions 13
1. Function: convIn 13
2. Function: findnotcons 13
3. Function: checkType 14
4. Function: ind2pos! 14
5. Function: pos2ind! 15
6. Function: get ranges 15
7. Function: makepos & positionincrementer! 16

E. Elementary operations 16
1. Function: copy 16
2. Function: length 17
3. Function: size 17
4. Function: sum 17
5. Function: norm 17
6. Function: conj & conj! 18
7. Function: ndims (tensor rank) 18
8. Function: lastindex 18
9. Function: eltype 18
10. Function: elnumtype 18
11. Function: getindex & getindex! 19
12. Function: searchindex 19
13. Function: setindex & setindex! 20

F. Linear Algebra 21
1. Function: loadM! 21
2. Function: tensorcombination! 21

† Please send correspondence to: thomasedward.baker@york.ac.uk
3. Function: mult! 22
4. Function: add! 22
5. Function: sub! 23
6. Function: div! 23
7. Functions: +, -, *, / 23
8. Functions: sqrt! & sqrt 24
9. Functions: inverse_element & invmat 24
10. Function: exp 25

G. Tensor operations 25
1. Function: reshape! 25
2. Function: reshape 26
3. Function: unreshape & unreshape! 26
4. Function: permutedims! 27
5. Function: permutedims 27
6. Function: joinindex & joinindex! 27

H. Display 29
1. Function: showTens 29
2. Function: print 29
3. Function: println 29

V. Module: Opt.jl 30
1. Function: spinOps.jl 30
2. Function: fermionOps.jl 30
3. Function: tJOps.jl 31

VI. Module: contractions.jl 31
A. Conversion of tensors to and evaluation of matrix multiplication 32
1. Function: libmult! 32
2. Function: matrixequivalent & matrixequivalent! 32
3. Function: permutedims_2matrix! 33
4. Function: prepareT 33
5. Function: corecontractor 34
6. Function: maincontractor 34
B. Contraction functions 35
1. Functions: contract, ccontract, contractc, and ccontractc (scalar output) 35
2. Functions: contract, ccontract, contractc, and ccontractc (tensor output) 36
3. Functions: contract, ccontract, contractc, and ccontractc (re-ordered tensor output) 37
4. Function: trace 38
5. Function: checkcontract 39

VII. Module: decompositions.jl 39
1. Function: libsvd 39
2. Function: recursive_SVD 40
3. Function: findnewm 40
4. Function: safesvd 41
5. Function: svd 41
6. Function: libeigen 43
7. Function: eigen 43
8. Function: libqr & liblq 44
9. Function: qr & lq 45
10. Function: polar 46

VIII. Module: MPtask.jl 47
1. Types: MPS, MPO, and Env 47
2. Struct: matrixproductstate 48
3. Struct: matrixproductoperator 48
4. Struct: environment & vecenvironment 48
5. Constructor: environment 48
6. Constructor: makeoc 49
7. Constructor: MPS 49
A. Function: applyOps & applyOps! 50
1. Constructor: MPO 51
B. Function: randMPS 52
1. Elementary functions: elnumtype, eltype, size, length, lastindex, copy, conj, conj! and setindex! 52
2. Types: largeMPS, largeMPO, and largeEnv 55
3. Structs: largematrixproductstate, largematrixproductoperator, and largeenvironment 56
4. Structs: tensor2disc & tensorfromdisc 56
C. Function: largeMPS, largeMPO, largeLenv, largeRenv and largeEnv 56
1. Elementary functions: getindex, setindex!, lastindex, length, and eltype 58
D. Function: Load types loadMPS, loadMPO, loadLenv, and loadRenv 60
E. Function: copy (large types) 61
F. MPS gauges: movement of the orthogonality center 62
1. Function: moveR & moveR! 62
2. Function: moveL & moveL! 63
3. Function: movecenter! 63
4. Function: move! & move 64
G. Function: leftnormalize, leftnormalize!, rightnormalize, & rightnormalize! 64
1. Function: boundarymove! & boundarymove 65
H. Updating and making environment tensors 66
1. Function: Lupdate & Lupdate! 66
2. Function: Rupdate & Rupdate! 67
3. Function: makeBoundary 68
4. Function: makeEnds 68
5. Function: makeEnv 68
I. Measurements and related functions 69
1. Function: applyMPO 69
2. Function: expect 70
3. Function: correlationmatrix 70
4. Function: operator_in_order! 72
5. Function: permutations 72
6. Function: correlation 73
J. Constructing MPOs and MPSs 75
I. INTRODUCTION

This documentation of code is meant to be a "Numerical Recipes" style introduction to tensor network algorithms [1]. This introduction will cover the aspects of a tensor network library from the most basic operations to the implementation of algorithms.

This code is an implementation of the concepts introduced in Refs. 2–4. This introduction will focus on the details of the numerical implementation, so the interested reader is encouraged to start with the references given here if the larger physics concepts are not known.

If the user is unfamiliar with tensor networks in general, we recommend beginning with Ref. 2 which explains generally the ideas behind and uses of tensor networks. A thorough introduction to the density matrix renormalization group (DMRG), one of the most powerful original tensor network algorithms, is given in Ref. 3. Implementation of this code can be found at Ref. 5. Additional documentation and introductory materials will be made available in the near future. Tensors with quantum number symmetries are currently available in the code, and this document will be updated with a thorough account of the functions in a subsequent update.

If the user is attempting to get some practice manipulating the basic operations in the code, it is recommended to understand the functions reshape (Sec. IV G 1), permutedims (Sec. IV G 5), contract (Sec. VI), and svd (Sec. VII 5) which constitute the four basic operations necessary to understand tensor network methods as outlined in Ref. 2 and relied on heavily in Ref. 3. Where possible, the built-in julia functions are overloaded with functions acting on any new types that are introduced in DMRjulia, hence the code will look as close to traditional julia code as possible.

More advanced algorithms can always be broken down into these operations (with some important exceptions). So, in order to understand algorithms, the basic operations should be learned. The basic input structures from julia can be input into these functions, although some custom tensor types are implemented to reduce some extra compilation time in the code.

A. Starting to use the library

The DMRjulia library can be downloaded directly from the online repository [5]. To install the package, enter the following into the terminal to install it:

```
julia> pkg> add DMRJtensor
```

To use the package, the following command will load all functions

```
julia> using DMRJtensor
```

The best option for those wishing to add different algorithms or to explore different features of a tensor network code. To make all functions in the library available in the julia terminal or as the first line in a code using the functions here,

```
julia> path = "<path to library>"
julia> include(path*"/DMRjulia.jl")
```

Below is an indepth discussion of the functions used in the library. For a brief description, all of the functions are implemented in DMRjulia and can be accessed in the terminal by typing (for the function contract),

```
julia> ? help? DMRjulia #or other function
```

Some functions are not exported but are defined in the library. This is mainly to prevent those functions from being overwritten with new variable names in the main program and because they serve little purpose other than to aid other functions. To access these private objects after performing the above, type DMRjulia.get ranges to access that function in that case. Functions that are overloaded onto a function internal to julia are immediately publicly available without any other operation as public variables.

B. What paper should be cited?

If you would like to see the original reference papers based on the code that you have written, enter the following line at the end of your code or type the following into the command line after running your code.

Depending on what type of system you are solving with DMRG, for example, it may change which papers you should cite since some papers influence how the algorithm should be used in certain situations. For example, citations for quantum chemistry and two-dimensional systems would be different.

Some general references are listed here.

- Two-site version of DMRG
  
  \[
  \text{dmrg(...,method="twosite")} \quad [6, 7]
  \]

- Strictly single-site (3S) DMRG
  
  \[
  \text{dmrg(...,method="3S")} \quad \text{default}] \quad [8]
  \]

- Quantum chemistry Hamiltonians: \[9–11\]

- identification of the wavefunction ansatz with a matrix product state (MPS) for DMRG \[12, 13\]
In general, it is good to familiarize oneself with the literature so that the foundational literature is always cited in addition to the specific algorithms used above [2, 3, 14–19].

This list is meant to contain the original paper where an algorithm is given. If any paper is missing here, please feel free to message the lead author with a message suggesting another paper to add to the reference list here.

The DMRjulia literature is contained in Refs. 2–4. We do not anticipate that this document will be frequently cited, but if necessary, then Ref. 20 contains the citation style. More authors are expected to be added onto this document as the library progresses.

C. Debugging tips

1. Debugging tips

There is a lot that can go wrong with any code, and there are a few practical tips that can help to fix most issues.

If you run into something that is not contained in these documents, can not be diagnosed, and it is not too many years since this document was published, then try contacting the author.

1. Focus on the top error message output by the program and work down the error messages displayed

Sometimes, although definitely not always, the error at the top of the list will be the most relevant to solving the problem. Typically the first relevant error is contained in a file contained by the DMRjulia code itself. Identifying which error is occurring can help to find where in the program an error is.

2. Disable `@inbounds`, `@simd`, `Threads.@threads`

This can make error messages more readable and expose where segmentation faults might be.

3. Ensure that the MPS and MPO are defined with the right types.

If the computation requires complex numbers, it is best to define them at the start.

4. Design algorithms with `Array`, convert to `denstens`, and then start with `qarray`

For ease of debugging, julia’s internal array struct has many error messages for bad use. This can be very useful in order to make the algorithm work in the first place. The `denstens` is next and has less error messages. The bulk of the errors will occur when using the quantum number system, but there will be special checks that can be added to a code to ensure its proper function.

5. Comment parts of the code until the error goes away

It typically helps to draw diagrams while the code at each step.

6. When consulting an expert in the code.

Please include the full error message and, as accurately as possible, what you were trying to do. Other information like what machine is being used is very useful. Even better yet, solve a problem and tell us about it to be a contributor!

At this stage of development, the input tensors are generally the root of issues in the code. So, it is worthwhile to check twice that the inputs are programmed as needed.

It is also highly recommended to use an integrated development environment. At the time of this document, the popular software for this is Virtual Studio Code, although Sublime, Atom, Xcode, and others are also used.

D. Comment on time to run the code

The core philosophy of the DMRjulia library is to reduce the amount of time spent implementing algorithms and learning the basics of tensor network codes. Because the implementation here is in a high-level language, there will be some noticeable speed decreases from other specific, low-level implementations of tensor networks.

In most applications, the DMRjulia library performs as fast as other lower-level codes. However, in other very basic examples, upwards of a 10% slowdown has been noticed. This is mainly for simple implementations of quantum numbers (dense tensors appear to function as fast as other implementations). Asking the user to wait a small percentage to obtain results is preferable, in our opinion, to requiring the user to spend more time coding an algorithm. We will note that several highly intensive models have been run with the code (in excess of $m = 10,000$ for DMRG) and results were obtained reasonably with other codes.

Explicit time comparisons will not be made here, as we think that they are not useful in the long run. The main ingredients that will help run tensor network algorithms fast is a firm knowledge of how to use those algorithms and a flexible code that can respond to what a user needs to input. The implementation here aims to help the user maintain focus on theoretical concepts instead of spending an inordinate amount of time tending to code efficiency or implementing algorithms.

E. Some common implementation features in julia

There are several implementation features in julia that deserve some discussion here to both identify why they are being used in the code and how to interpret them if implementing a tensor network library in a lower level language.
1. Use of where statements

Note that there is a difference between defining the type of a variable in a function as

```julia
function test(vec::Array{Number,2})
end
```

and

```julia
function test(vec::Array{X,2})
    where X <: Number
end
```

The second implementation will assume all entries of the input vector `vec` are the same. Meanwhile, the first implementation requires a vector input that has the element of the vector as `Number` (meaning that a real number, a complex number, and an integer can be elements). This can be harder to define in the code and since there are many places where the input vector has a homogeneous type, the second implementation is used frequently.

2. Imaginary numbers

Imaginary numbers and complex numbers are implemented in julia. For example, \(3 + 4i\) would appear as `3+4im`. The library is fully compatible with imaginary numbers inside of matrices, but it is recommended to define all initial types with `ComplexF64` (or similar) before starting a problem (see Sec. VIII 7).

3. Other functionality

All other functionality not covered in this document is internal to julia. In all cases, the library tries to look as much like julia as possible.

4. Uniformity of input numbers and types

Where possible, the default numbers for a function are cast as the type of input tensor to avoid any extra allocations. However, input numbers from the user are not automatically converted. It is highly recommended to ensure that all tensor types and numbers input into functions are made to be the same type. For example, if using complex numbers, creating the MPS, MPO, and all input values as complex will greatly reduce the amount of time needed to solve the problem. If this is not done, then the program may not run.

F. Structure of the library

An overview of how each file is dependent on the others is given in Fig. 1. Each file contains code that is related to each other. In other implementations of the library (for example, if this code were to be implemented in a lower level language), these files would be modules containing operations for a specific purpose. In the julia language, it is convenient to simply use the `include` function to load all files into one module, so the extra effort of making each file a module is skipped here.

G. Some comments on the julia programming language

Some disadvantages that will be encountered with this language in general is the tendency for julia to return lazy types on application of an operation (for example, the transpose will return an object that has instructions to compute the transpose when the program believes it is necessary to). These types can often decrease performance of the code here. There are some other pitfalls that are remedied with the full library, but seeing how the basic `Array` type works on this code will inform what is going on in the full library. Note that all functions in the library are similar to the functions as they would be applied to regular `Array` types, and the functions defined here will admit those `Array` types if those are preferred over the library’s defined types. For example, the apostrophe acting on a matrix gives the transpose, but the full transpose operation will not be executed immediately.

When writing code, it is recommended to write the native julia `Array` type and then upgrade to the dense tensor for efficiency and backwards compatibility.

At any point in the julia terminal, once the DMRjulia is loaded, one can type “?” and enter the function name to obtain inline comments. The discussion here is to motivate why some functions are programmed the way that they are programmed and why other choices were made.

Whenever a function is in conflict with the pre-existing definitions in julia, it will be imported using the `import` feature.

Regular programming in julia is done without declarations of what types are input into functions. This is broken here by explicitly defining the types that are put into the julia programming language explicitly. Because the normal `Array` type has been eschewed here, the compiler sometimes has trouble determining what output types when using custom types such as the dense tensor and quantum number tensor without these declarations. Also, including them can help with debugging.

If there is any question as to why a function was programmed in such a way, it is contained here. The notes in the program are there for usage instructions.

There are four modifiers in lines of code that need to be discussed here. The first is the `@inline` function which tells the compiler to make the function inline in the code. This will increase the size of the executable because the program will be longer, but the time decreases. The compiler might normally do this operation for a code, but
FIG. 1. Dependencies of the DMRjulia library.

the functions where it is applied here are at the most
time-sensitive elements, so it is forced here to ensure this
happens. The functions are also not very large, so will
not add much to the execution time.

The second macro to keep track of is the @inbounds
flag. This means that there should be no bounds check-
ing on the retrieval and placement of elements from an
array. This can cause an overhead of 10-20% in some
cases. The macro is only applied to the core functions.

For the most optimal performance, the global macro
--check-bounds=no which can be applied at the start
of julia, will eliminate all bounds checking.

When running a simple for loop, an optimization flag
@simd can be used to generate a small speed increase.

There is also the option to parallelize some loops
with the Threads.@threads macro. There is a separate
@distributed macro which has a slightly different im-
plementation. Nested parallelization is not supported by
Threads.@threads as of the current version of julia.

At some points, the quantum number version is alluded
to. All of these functions will be written for the quantum
number conserving tensor. This will be added to the
documentation in due time.

There are also optional arguments that can be pro-
vided to functions, and there will be an entire se-
cret menu of inputs that can be applied to a particu-
lar function. Whenever an input has an equals sign,
the argument may be left off the function call. In
this case, the default is used. For example, if calling
dmrg, the special option of the method can be specified,
dmrg(psi,mpo,sweeps=20,method="twosite") where
sweeps and method are the special options here.

There is a convention in julia in-place operations to
append an exclamation point (!) onto the end of a func-
tion’s name. For example, reshape will make a copy of
the input and reshape! will change the data in-place.
Sometimes this is useful sometimes it is not. Tradition-
ally, the in-place function does not receive a return value.
However, it is sometimes useful for consistency to return
a value anyway since returning the same value has no
cost (it is returned as a pointer, not a copy).

II. ACKNOWLEDGEMENTS

T.E.B. is grateful to the US-UK Fulbright Commission
for financial support under the Fulbright U.S. Scholarship
programme as hosted by the University of York. This
research was undertaken in part thanks to funding from
the Bureau of Education and Cultural Affairs from the
United States Department of State.

This library would not exist without the financial sup-
port provided to T.E.B. provided by the postdoctoral
fellowship from Institut quantique and Institut Transdis-
ципinaire d’Information Quantique (INTRIQ). This re-
search was undertaken thanks in part to funding from
the Canada First Research Excellence Fund (CFREF).

[1] William H Press, Saul A Teukolsky, William T Vetter-
ling, and Brian P Flannery, “Numerical recipes in c++,”
The art of scientific computing (1992).
[2] Thomas E Baker, Samuel Desrosiers, Maxime Trem-
Dense tensors

III. MAIN FILE: DMRJULIA.JL & OVERVIEW

The order in which the files are loaded are done in julia according to the following code.

```julia
module DMRjulia
import LinearAlgebra # julia’s interface to the BLAS linear algebra library
import Printf # julia’s printing interface for outputs to the terminal
import Serialization # julia’s efficient read/write of types to disk
import Distributed # julia’s internal utility for parallelization

filepath = "lib/" # folder where the following files are located

include(filepath*"tensor.jl") # basic tensors
include(filepath*"Opt.jl") # operators for some models
```

Part I

Dense tensors

[5] 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).
[6] Steven R White, “Density-matrix algorithms for quantum renormalization groups,” Physical Review B 91, 155115 (2015).
[7] Garnet Kin-Lic Chan and Martin Head-Gordon, “Highly correlated calculations with a polynomial cost algorithm: A study of the density matrix renormalization group,” The Journal of chemical physics 110, 4127–4130 (1999).
[8] Roberto Olivares-Amaya, Weifeng Hu, Naoki Nakatani, Sandeep Sharma, Jun Yang, and Garnet Kin-Lic Chan, “The ab-initio density matrix renormalization group in practice,” The Journal of chemical physics 142, 034102 (2015).
[9] Stellan Östlund and Stefan Rommer, “Thermodynamic limit of density matrix renormalization,” Physical review letters 75, 3537 (1995).
[10] Garnet Kin-Lic Chan and Martin Head-Gordon, “Highly correlated calculations with a polynomial cost algorithm: A study of the density matrix renormalization group,” The Journal of chemical physics 110, 4127–4130 (1999).
[11] Roberto Olivares-Amaya, Weifeng Hu, Naoki Nakatani, Sandeep Sharma, Jun Yang, and Garnet Kin-Lic Chan, “The ab-initio density matrix renormalization group in practice,” The Journal of chemical physics 142, 034102 (2015).
[12] Stellan Östlund and Stefan Rommer, “Thermodynamic limit of density matrix renormalization,” Physical review letters 75, 3537 (1995).
[13] Stefan Rommer and Stellan Östlund, “Class of ansatz wave functions for one-dimensional spin systems and their relation to the density matrix renormalization group,” Physical review b 55, 2164 (1997).
[14] Ulrich Schollwöck, “The density-matrix renormalization group,” Reviews of modern physics 77, 259 (2005).
[15] Ulrich Schollwöck, “The density-matrix renormalization group in the age of matrix product states,” Annals of physics 326, 96–192 (2011).
[16] S White, “How it all began: A personal account,” LECTURE NOTES IN PHYSICS-NEW YORK THEN BERLIN- , vii-xvi (1999).
[17] Karen A Hallberg, “New trends in density matrix renormalization,” Advances in Physics 55, 477–526 (2006).
[18] Román Orús, “A practical introduction to tensor networks: Matrix product states and projected entangled pair states,” Annals of Physics 349, 117–158 (2014).
[19] Jacob C Bridgeman and Christopher T Chubb, “Hand-waving and interpretive dance: an introductory course on tensor networks,” Journal of Physics A: Mathematical and Theoretical 50, 223001 (2017).
[20] Thomas E. Baker et al., “DMRjulia0: Tensor recipes for entanglement renormalization computations,” arXiv preprint arXiv:XXXX (2021).
[21] John S Townsend, A modern approach to quantum mechanics (University Science Books, 2000).
[22] John Hubbard, “Electron correlations in narrow energy bands,” Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences 276, 238–257 (1963).
[23] David Sénéchal, “An introduction to quantum cluster methods,” arXiv preprint arXiv:0806.2690 (2008).
[24] Jozef Spalek, “tj model then and now: a personal perspective from the pioneering times,” arXiv preprint arXiv:0706.4236 (2007).
[25] EM Stoudenmire and Steven R White, “Real-space parallel density matrix renormalization group,” Physical review B 87, 155137 (2013).
Each module will be discussed in depth in the following, but the first bit of code here allows for the discussion of the code strategy and to given an overview. Each set of functions that follow some theme will be labeled as a module in julia but not defined as such in the library for ease of use. Note that by only defining the modules in julia as regular files with functions in them, all the import statements can be included here in one step, hence motivating why the formal module was not defined for each file in the following.

IV. MODULE: TENSOR.JL

The tensor module contains the most basic operations relating to the definition of a tensor in the code. It is the first module defined, so it is dependent on nothing, although some functions from the `LinearAlgebra` package from julia are defined. The linear algebra functions are imported, making it so that the functions from this module must also be imported to avoid a conflict. In all cases, the functions defined will be matched closely with the definitions in julia and this will make the code easier to use.

A. Types: denstens and general tensor types

Some basic global types must be defined before moving onto the basic tensor type. These types act like generic names for a wide variety of types that are defined in the following. The export command makes the variables, types, and functions public once the library is loaded. All other variables, types, and functions are private to the library.

1. Type: denstens

The first type is a generic place-holder for the core tensor type. To define the core tensor type, a `DataType` (julia’s generic type for a `Float64`, `ComplexF64`, etc.) corresponding to the type of number stored in the tensor. This format allows for easier code and is named the denstens.

```julia
abstract type denstens end
export denstens
```

On its own, there is not much to do with this, but it will be used to define several types in the defined struct.

2. Type: qarray

This type of tensor is very different from the dense tensor (denstens). The quantum number conserving array will take advantage of quantum number symmetries to exploit a block symmetry in the tensor networks. For more information, see [4].

```julia
abstract type qarray end
export qarray
```
3. **Type: TensType & denstensType**

To define a type that encompasses all of the tensor types possible in the library, the `TensType` is defined. This can be used as a placeholder when defining a function for more than one input type. This mainly makes coding functions easier as will be see, when they can accept more than one type, Julia’s principle advantage that is used here.

```plaintext
const TensType = Union{qarray, denstens, AbstractArray}
export TensType

case denstensType = Union{AbstractArray, denstens}
export denstensType

case tenstype = TensType
export tenstype
```

The `denstensType` is useful when considering the non-quantum number types.

4. **Type: intType**

This type is the base integer type used in the library.

```plaintext
const intType = typeof(1)
export intType
```

There was no detectable difference with using another type of integer here; however, it may be useful for future users to have a common definition of the tensors here. One could reasonably replace it with `Int64` everywhere if desired. If for some reason the code is run on 32-bit architecture, then this variable can be changed to `Int32` for compatibility and potential speed increases.

5. **Type: intvecType**

Particularly for inputs for contraction, there are four possible input types that will be allowed as a generic column. One can enter a single integer, an array of integers (i.e., `[1, 2, 3]`), a matrix of integers with the row or column size equal to 1 (i.e. `[1; 2; 3]`), or as a `Tuple`.

```plaintext
const intvecType = Union{P, Array{P, 1}, Array{P, 2}, Tuple} where P <: Integer
export intvecType
```

The `Tuples` are the best as they natively carry the length of the tuple, allowing the compiler to more easily optimize the resulting code. The `Tuple` can store any type of input and can even mix inputs (for example, both integers and floats) while the traditional vector stores just one type. A `Tuple`’s contents can not be changed generically once it is made. If, for example, a `Tuple` was made with a vector as one of the elements, the entries of the vector can be changed, but the size of the vector can not. This makes tuples somewhat difficult to deal with when programming, but they tend to be more efficient. Whenever the vector does not need to change and it matches the syntax in Julia’s main functions, Tuples will be used, although strict attention will not be paid to it so that this code can act as a template for other languages where only vectors are used.

6. **Type: genColType**

For operations like getting an index, there are four types of inputs that should be accommodated to match Julia’s inputs for the natively defined functions.

```plaintext
const genColType = Union{UnitRange, Integer, Array{P, 1}, Colon, StepRange, Tuple} where P <: Integer
export genColType
```

There are four basic types of inputs, a Colon (·), a unit range (i.e., `1:10`), an integer, and an array of integers signifying the values to be kept on a particular index.
B. Struct: tens

The tens type is the core tensor type in the library. It is a sub-type of the denstens and carries the information of the tensor, no matter its rank.

```plaintext
mutable struct tens{W <: Number} <: denstens
    size::NTuple{N,intType} where N
    T::Array{W,1}
end
export tens
```

The fields of this tensor are callable (if the tens is A) as A.size and A.T for each field respectively. The size field stores a tuple representing the current size of the tensor and the T field stores the elements of a the tensor as a vector. The number of elements in the T field must always match the product of the elements in the size field.

Note that there is one major difference from the Array type in julia. The Array is defined with a DataType and a rank, but since the rank is a fluid concept, it is not necessary to carry that information around in the tensor network computation. In fact, because the rank changes so frequently, it is better to leave it out. This will make some inefficiency with the compiler in julia, but it turns out not to be too much. So, this tensor is defined somewhat for documentation purposes but also because it avoids the need to consider the rank in the type of a tensor.

C. Constructors

1. Function: tens

There are eight ways in which were convenient to initialize a tens. The native constructor is to define write something like tens(W,A) where W is the DataType used and A is the input vector for the T field. The compiler automatically determines the rest for the input function.

The first natively defined constructor is made inputting the DataType and returning a null vector. The tensor can be initialized as tens(), tens(Float64), tens(type=Float64) which comprise the first three functions below.

Another option is to take a tensor defined natively in julia and convert that type into the denstens type. The recipe is to simply take the size from the size function and initialize the size field. The tensor field can be set by reshaping the Array into a vector. The type of the resulting tensor can also be forced by adding it before the declaration of the Array.

```plaintext
function tens(;type::DataType=Float64)
    return tens{type}((,),type[])
end

function tens(type::DataType)
    return tens{type}((,),type[])
end

function tens{T}() where T <: DataType  
    return tens(type=T)
end
```

When a regular julia tensor should be converted to the tens type, the following functions will construct the tensor.

```plaintext
function tens(G::DataType,P::Array{W,N}) where W <: Number where N
    sizeP = size(P)
    vecP = reshape(P,prod(sizeP))
    if G != W
        rP = convert(Array{G,1},vecP)
    else
        rP = vecP
    end
    return tens{G}(sizeP,rP)
end
```
The conversion of the regular tensor type to a denstens costs only a moderate amount of allocations and should not be an issue in algorithm construction. The same is true when converting back as will be seen. If using another programming language, then converting to this type can be neglected or taken from the native type in that language.

2. Function: rand

It is sometimes useful to generate a random tensor. This version simply uses the julia version and converts to the dense tensor type.

```julia
import Base.rand
function rand(rho::tens{W}) where W <: Number
    return tens{W}(rand(W, size(rho)))
end
```

3. Function: zero

Similar to the julia function zero, a tensor with all zero entries can be created from a pre-existing dense tensor.

```julia
import Base.zero
function zero(M::tens{W}) where W <: Number
    return tens{W}(zeros(W,M.size))
end
```

4. Function: makeId

This function generates an identity tensor. This is most commonly used when contracting two indices on a tensor (i.e., tracing out an index).

```julia
function makeId(W::DataType,ldim::Integer; addone::Bool=false, addRightDim::Bool=false)
oneval = W(1)
if addone
    if addRightDim
        newsize = (ldim,ldim,1)
    else
        newsize = (1,ldim,ldim,1)
    end
else
    newsize = (ldim,ldim,1)
end
end
```
newsize = (ldim, ldim)
end
Id = zeros(W, ldim^2)
@simd for i = 1:ldim
   @inbounds Id[i + ldim*(i-1)] = oneval
end
return tens{W}(newsize, Id)
end

function makeId(A::tens{W}, iA ::R) where R <: intvecType where W <: Number
   Id = makeId(W, size(A, iA[1][1]), addone=true, addRightDim=true)
   for g = 2: length(iA)
      addId = makeId(W, size(A, iA[g][1]), addone=true, addRightDim=false)
      Id = contract(Id, ndims(Id), addId, 1)
   end
   newsize = ntuple(a-> size(Id, a), ndims(Id)-1)
   return reshape!(Id, newsize...)
end

function makeId(A::Array{W, N}, iA ::R) where N where {R <: intvecType, W <: Number}
   densA = tens(A)
   Id = makeId(densA, iA)
   return makedens(Id)
end
export makeId

5. Function: convertTensor

This function converts the element type of the dense tensor into another type (from the type of M to a type G).

function convertTens(W::DataType, M::denstens)
   return tens{W}(M.size, convert(Array{W,1}, M.T))
end
export convertTens

This is slightly contrasted with julia’s conversion function (convert), but it is easier to define and use. It is also a rarely used function, so it is recommended that if the data type must be changed, that this function should be read about and searched for. Often, it is much better to define all types at the beginning of a computation and continue from there instead of converting between tensor types.

6. Function: makedens

The function makedens converts a denstens to the denstens format (trivial operation).

function makedens(M::denstens)
   return M
end

function makedens(M::AbstractArray)
   return tens(M)
end
export makedens

While function does nothing for now, it is used later for quantum number tensors and having the unified interface can be useful in several situations.
7. Function: makeArray

The function *makedens* converts a *denstens* to the julia *Array* format.

```plaintext
function makeArray(M::denstens)
    return reshape(M.T, size(M))
end

function makeArray(M::AbstractArray)
    return M
end
export makeArray
```

D. Helper functions

1. Function: convIn

Converts any input of type *intvecType* to a Tuple. This is most useful for contractions, but it is used in many places in the library, so it is defined in *tensor*.

```plaintext
function convIn(iA::Union{Array{P,1},Array{P,2}}) where P <: Integer
    return ntuple(i->iA[i], length(iA))
end

function convIn(iA::Integer)
    return (iA,)
end

function convIn(iA::NTuple{N,intType}) where N
    return iA
end
export convIn
```

The tuples here carry the information about how many elements there are in the code. The conversion of the array types to the tuple will force the code to determine how long the tuple will be. So, the use of only the tuples (and integers) as input to this function will ultimately be slightly faster, but the overall performance increase with this change is very minor.

2. Function: findnotcons

Generates the complement set of an input. This is used for contractions, but it has uses in other functions that mandate its definition be here. For example, the complement of (1,2,6) for a total rank of 10 would be (3,4,5,7,8,9,10).

```plaintext
function findnotcons(nA::Integer,iA::NTuple{N,intType}) where N
    notcon = ()
    for i = 1:nA
        k = 0
        notmatchinginds = true
        while k < length(iA) & k notmatchinginds
            k += 1
            notmatchinginds = iA[k] != i
        end
        if notmatchinginds
            notcon = (notcon...,i)
        end
    end
```
3. Function: checkType

If one of the two inputs is not matched (i.e., an Array and a denstens), then the Array is converted to a denstens.

```julia
function checkType(A::R,B::S) where {R <: AbstractArray, S <: denstens}
    return tens(A),B
end

function checkType(A::S,B::R) where {R <: AbstractArray, S <: denstens}
    return A,tens(B)
end

function checkType(A::AbstractArray,B::AbstractArray)
    return A,B
end

function checkType(A::qarray,B::qarray)
    return A,B
end

function checkType(A::denstens,B::denstens)
    return A,B
end
export checkType
```

Note that the last function call is between two tensor types which share a type. The cost to evaluating this function is effectively removed if two identical tensor types are input into the function, and this will be the function used by the julia language when two identical tensor types are present.

4. Function: ind2pos!

This function converts a set of positions in indices to a position in a tensor. The essential operations that accomplish this are to first envision the tensor as a matrix of size of the first dimension of the first index by the rest of the indices reshaped together. Then, the same idea to find the row and column of an element in a matrix may be used to find the positions in the tensor.

First, the index is converted to the 0-indexed version (julia uses 1-indexed numbers, so the value is subtracted by 1). Then, the value is divided by the dimension of the first index and rounded down, \( \lfloor x/S_j \rfloor \) for index \( x \) and size of an index \( j \), \( S_j \). The column (first index’s) position is now known. To find the row value, the modulus of the index is taken and a one is added since the value is 1-indexed here, \( x \mod S_j \).

```julia
@inline function ind2pos!(currpos::Array{Array{X,1},1},k::X,x::Array{X,1},
    index::X,S::Array{X,1}) where X <: Integer
    @inbounds currpos[k][1] = x[index]-1
    @simd for j = 1:size(S,1)-1
        @inbounds currpos[k][j+1] = fld(currpos[k][j],S[j])
        @inbounds currpos[k][j] = (currpos[k][j]) % S[j] + 1
    end
    @inbounds currpos[k][size(S,1)] = currpos[k][size(S,1)] % S[size(S,1)] + 1
    nothing
end
```

The function is often used to identify index values for quantum number tensors.
5. **Function: pos2ind!**

It is frequently the case that an array containing the position in a tensor will need to be converted to a number and vice versa. This version of the function, `pos2ind!`, performs this operation in place with pre-existing vectors.

The first implementation of this function returns a single scalar value.

```julia
@inline function pos2ind(currpos::NTuple{N,P},S::NTuple{N,P}) where {N, P <: Integer}
    x = 0
    @simd for i = length(S):-1:2
        @inbounds x += currpos[i]-1
        @inbounds x *= S[i-1]
    end
    @inbounds x += currpos[1]
    return x
end
export pos2ind

@inline function pos2ind!(x::Array{X,1},j::Integer,
currpos::Union{Array{X,1},NTuple{N,intType}}
,S::NTuple{N,intType}) where {N, X <: Integer}
    @inbounds x[j] = currpos[end]
    @simd for i = length(S)-1:-1:1
        @inbounds x[j] -= 1
        @inbounds x[j] *= S[i]
        @inbounds x[j] += currpos[i]
    end
    nothing
end
export pos2ind!
```

The second function admits a predefined vector to eliminate any extra allocations from the first implementation. It can be noted that the definition of a single scalar value is very cheap in terms of both computational time and garbage collection. However, care is taken to allow for pre-defined vectors to be input into functions to avoid defining vectors inside of functions that may be called many times.

Use of the `@inbounds` and `@simd` flags is to reduce runtime but often has only a modest effect.

The function is most often useful for searching for an element of a tensor in the `denstens` format. By converting the position to an index, it can make it easier to search for an element in a tensor of the `denstens` format.

6. **Function: get_ranges**

This function converts any of the four input types for `genColType` to a vector with the relevant values to keep on a given index. This is then used in `setindex!` principally.

```julia
function get_ranges(C::Tuple,a::genColType...) ap = Array{genColType,1}(undef,length(a))
    for y = 1:length(ap)
        if typeof(a[y]) <: Colon
            ap[y] = 1:C[y]
        elseif typeof(a[y]) <: Integer
            ap[y] = a[y]:a[y]
        elseif typeof(a[y]) <: AbstractArray
            ap[y] = 1:length(a[y])
        else
            ap[y] = a[y]
        end
    end
    return ap
end
```
The generation of the ranges is somewhat more costly in terms of allocations for a temporary object, but the overall cost for this operation is very low inside of a given algorithm.

7. Function: makepos & position_incrementer!

When writing loops in general, there are two basic strategies if an array of numbers need to be iterated over. One option is to use the modulus operation to convert an index to a position (see the conversions between integers and positions in the pos2ind and ind2pos functions). The second option is to explicitly increment a position vector, starting at zero, and continue until the maximum length is reached. The functions makepos and position_incrementer! generate the initial position and then increment it. The second function is mainly used inside of a function.

```julia
function makepos(ninds::P; zero::P=0) where P <: Integer
    pos = Array{P,1}(undef,ninds)
    pos[1] = zero
    one = zero + 1
    for g = 2:ninds
        pos[g] = one
    end
    return pos
end
export makepos

function position_incrementer!(pos :: Array{G,1}, sizes :: Union{Array{G,1}, Tuple}) where G <: intType
    w = 1
    @inbounds pos[w] += 1
    @inbounds while w < length(sizes) && pos[w] > sizes[w]
        @inbounds pos[w] = 1
        w += 1
    @inbounds pos[w] += 1
    end
    nothing
end
export position_incrementer!
```

The use of the makepos function deserves some discussion. One can choose to initialize the vector as a 1-indexed vector or a 0-indexed vector by choosing either zero=0 or zero=-1, respectively.

E. Elementary operations

These functions will closely resemble julia’s interface, but will be necessary to define in other implementations.

1. Function: copy

The most robust copy function in julia is the deepcopy function, but using a deepcopy is inherently type unstable, meaning that the compiler can not always resolve the output type from the function. One can use this function whenever the need to copy an array is required, but defining a suitable copy function instead will make julia’s other function and keep the code type stable. The function deepcopy works immediately on the previously defined types.

```julia
import Base.copy
function copy(A::tens{W}) where {W <: Number}
    return tens{W}(A.size,copy(A.T))
end
```

The copy function must be careful to leave no possible way for changes in the original tensor to affect the new tensor. So, the main item that must be addressed here is the T field. For example, if the copy operation is not applied...
on this tensor, then conjugating the original elements will conjugate both tensors, even if only one of the tensors is to be conjugated! The reason that this happens because simple definitions of variables are not inherently copies in julia, instead they are passed by reference. So the new object only stores a pointer that references the original data. If the original data is changed, so will the new object. The tuple must remain unchanged, so it is not necessary to copy.

When making a previously defined function, it is often the case that trial and error will determine which quantities need to be copied and which do not. For safety, one can simply copy every field, but there is a very small savings that can be found with careful copying.

2. **Function: length**

   This returns the number of elements in the tensor in julia, so it will perform the same operation here.

   ```julia
   import Base.length
   function length(M::denstens)
       return length(M.T)
   end
   ```

3. **Function: size**

   This returns the size of a given index on a tensor. If the full tensor is input, then the Tuple representing the size of the specified index is returned.

   ```julia
   import LinearAlgebra.size
   import Base.size
   function size(A::denstens)
       return A.size
   end
   
   function size(A::denstens,i::Integer)
       return A.size[i]
   end
   ```

4. **Function: sum**

   This returns the sum of the elements of a given index on a tensor. This can be accomplished by simply summing up the values in the T field (stored values) of the tensor.

   ```julia
   import Base.sum
   function sum(A::denstens)
       return sum(A.T)
   end
   ```

5. **Function: norm**

   This returns the $L^1$-norm (Froebenius norm) of the elements of a given index on a tensor.

   ```julia
   import LinearAlgebra.norm
   function norm(A::denstens)
       return norm(A.T)
   end
   export norm
   ```
6. **Function: conj & conj!**

To conjugate all elements of a tensor, the `conj` function can be used. This will return a copy of the original tensor. Alternatively, one can conjugate in-place with `conj!`. Both of these functions are overloaded from Julia’s internal implementations.

```julia
import LinearAlgebra
function conj!(currtens :: denstens)
    LinearAlgebra.conj!(currtens.T)
    return currtens
end

import LinearAlgebra
function conj(M :: tens{W}) where W <: Number
    newT = LinearAlgebra.conj(M.T)
    return tens{W}(M.size, newT)
end
```

7. **Function: ndims (tensor rank)**

This function returns the rank of the tensor, overloading Julia’s internal function.

```julia
import LinearAlgebra
function ndims(A :: denstens)
    return length(A.size)
end
```

8. **Function: lastindex**

This function is necessary to determine the last element of a vector. Applied to the `denstens` type, the return value is the last size. This is used in Julia when `end` is called on a particular tensor (i.e., `1:end`).

```julia
import Base
function lastindex(M :: denstens, i :: Integer)
    return M.size[i]
end
```

9. **Function: eltype**

This returns the element type in the dense tensor.

```julia
import Base
function eltype(A :: tens{W}) where W <: Number
    return W
end
```

10. **Function: elnumtype**

This returns the element type in the dense tensor. This is the same behavior as `eltype`, but this function will have a new meaning when dealing with MPSs and MPOs.
function elnumtype(A::tens{W}) where W <: Number
    return eltype(A)
end
export elnumtype

11. Function: getindex & getindex!

To obtain a single element of the tensor, or to extract a subset of the values in a tensor, the function getindex (and its in-place variation getindex!) can be called. The function getindex overloads julia’s internal function for the denstens type and the in-place version performs the same with a minor cost savings.

import Base.getindex
function getindex(M::denstens, a::genColType...) ...
    return getindex!(M,a...)
end

function getindex(C::tens{W}, a::G...) where W <: Number where G <: Array{Bool,1} ...
    M = makeArray(C)
    return tens{W}(M[a...])
end

function getindex!(C::AbstractArray, a::genColType...) ...
    return getindex(C, a...)
end

function getindex!(C::tens{W}, a::genColType...) where W <: Number ...
    i = 0
    allintegers = true
    while allintegers && i < length(a) ...
        i += 1
        allintegers = typeof(a[i]) <: Integer
        if allintegers ...
            val = pos2ind(a,C.size)
            return C.T[val]
        else ...
            dC = makeArray(C)[a...]
            return tens{W}(dC)
        end
    end
end
export getindex!

Much of the work of extract the elements relies on julia’s internal system which uses strided arrays. DMRjulia may implement a version of this in the future.

Note that there is a differentiation between inputting a value of 1 for a given entry of a and the entry 1:1 yield very different results. For the integer input of 1, a single value is returned and the index is truncated. If the range 1:1 is input, then the index is not truncated and in general the array properties of the tensor are kept. In other words, the range implies that a range should be output while the integer signifies that the index should be excluded from the final tensor.

12. Function: searchindex

The function getindex has many extra components in it that aren’t necessary if a single set of integers must be found in a tensor. The function searchindex is a faster method that is sometimes used in the library to search for a single element of a tensor. The basic strategy is to generate the index position (a single integer) of where the element should be in the dense tensor and then extract that element.
function searchindex(C::denstens,a::intType...)  
  if length(C.T) == 0
    outnum = 0  
  elseif length(C.size) == 0
    outnum = C.T[i]  
  else
    veca = ntuple(i->a[i],length(a))
    w = pos2ind(vec,a.size)
    outnum = C.T[w]
  end
  return outnum
end
export searchindex

Note that generating a single integer w inside of the for loop does not create much overhead for the julia language. So, the in-place version of pos2ind is not defined and used here. This function might not be useful in lower level implementations of the tensor networks, but it can be useful here to obtain a single element at low cost from a tensor, returned as a scalar, in a unified way between all tensor types.

13. Function: setindex & setindex!

In many cases, the elements of a tensor should be set. This is not strictly required for a generic tensor network computation, and is not a feature in some libraries, but in some cases it is found to be useful here. The function reduces to the julia function in line with the treatment for getindex.

import Base.setindex!
function setindex!(B::denstens,A::A,a::genColType...)  
  where G <: Union{denstens,AbstractArray}
  indexranges = get_ranges(B.size,a...)
  nterms = prod(newsize)

  Asize = size(A)
  Bsize = size(B)
  for z = 1:nterms
    ind2pos!(pos,z,Asize)
    @simd for i = 1:length(indexranges)
      @inbounds s = pos[i]
      @inbounds pos[i] = indexranges[i][s]
    end
    x = pos2ind(pos,Bsize)
    @inbounds B.T[x] = A.T[z]
  end
  nothing
end

function setindex!(B::tens{W},A::W,a::Integer...)  
  where W <: Number
    @inbounds index = a[end]-1
    @simd for q = length(a)-1:-1:1
      @inbounds index *= size(B,q)
      @inbounds index += a[q]-1
    end
    @inbounds B.T[index+1] = A
  nothing
end

Each of the values to replace is scanned over and copied from the input.
The function is parallelized over all the inputs that must be adjusted and the $z$ value that indexes the number of terms to be replaced in the tensor is computed using the relatively expensive modulus operation in `pos2ind!`. This is mostly offset by the parallelization for typical applications. Because in the worst case of a dense square matrix this would scale as $O(n^2)$, the choice was made to use parallelization instead of the incrementor position method from `position_incremener!`.

### F. Linear Algebra

Basic linear algebra operations on tensors are a fundamental function that must be defined for many algorithms. Many of these can be written into a single function `tensorcombination` that handles more simple algebraic operations.

1. **Function: loadM!**

   `loadM!` is a simple copy operation to the left input matrix from the right input matrix. Assumes same element type and useful for ensuring compiler efficiency.

   ```
   function loadM!(output::Array{W,N}, input::Array{W,N}) where {N, W <: Number}
       @simd for x = 1:length(output)
           @inbounds output[x] = input[x]
       end
       nothing
   end
   ```

   Otherwise, Julia may assume that a different type is being copied, thus forcing a conversion and an extra allocation to be thrown.

2. **Function: tensorcombination!**

   This function forms the basis of all algebraic manipulations of `denstens` going forward. This will create a new linear combination of all input tensors with coefficients supplied by `alpha`. For example, the output $K$ is a linear combination of input tensors

   $K = \sum_i \alpha_i M_i$

   where the set of $M_i$ tensors was input into the function with coefficients $\alpha_i$.

   ```
   function tensorcombination(M::denstens...; alpha::Tuple=ntuple(i->1, length(M)),
       fct::Function=*)
       alphamult = prod(alpha)
       Mmult = prod(i->M[i].T[1], 1:length(M))
       outType = typeof(alphamult * Mmult)
       newTensor = tens{outType}(M[1].size, zeros(outType, length(M[1])))
       for i = 1:length(M[1])
           @inbounds newTensor.T[i] = 0
           @simd for k = 1:length(M)
               @inbounds newTensor.T[i] += fct(M[k].T[i], alpha[k])
           end
       end
       return newTensor
   end
   ```

   ```
   function tensorcombination(M::AbstractArray...;
       alpha::Tuple=ntuple(i->1, length(M)), fct::Function=*)
       return sum(k->fct(M[k], alpha[k]), 1:length(M))
   end
   ```
The inputs to this function are allowed to best done with tuples, but this can be input with a vector as well. As per usual, the exclamation point (!) signifies that the leading tensor is modified while the rest are not. If the exclamation point is not included, then the first tensor is copied and then modified. The usefulness of this function is that it can reduce code for the other functions in this section.

Note that the combination function \( fct \) can be modified. This means that one could replace it with, for example, division (/) or any other operation defined for scalars and tensors or even to implement a square root function. The default is Eq. (1).

3. Function: mult!

The multiplication of a scalar with a tensor can be accomplished by simply calling the tensorcombination function with a single input.

4. Function: add!

The addition of two tensors \( A \) and \( B \) where \( B \) receives a scalar multiplicative factor is defined with the add! function.
5. **Function: sub!**

The subtraction of two tensors $A$ and $B$ where $B$ receives a scalar multiplicative factor is defined with the `sub!` function. The factor `mult` is simply negated to get the subtraction and then the `add!` function is called.

```plaintext
function sub!(A::P, B::P, mult::Number) where P <: TensType
    return add!(A, B, -mult)
end

function sub!(A::P, B::P) where P <: TensType
    return add!(A, B, -1)
end
export sub!
```

6. **Function: div!**

Division of a tensor by a scalar is identical to multiplication by the inverse of that scalar value. At least to the typically used double precision, this is accurate enough, so the `div!` function is defined as

```plaintext
function div!(M::P, num::Number) where P <: TensType
    return tensorcombination!(M, alpha=(num,), fct=/)
end
export div!
```

7. **Functions: +,-,*,/**

The basic functions of addition, subtraction, multiplication, and division can be defined to give a tensor that is copied from the original.

```plaintext
import LinearAlgebra.
function +(A::P, B::P) where P <: TensType
    return tensorcombination(A, B)
end

import LinearAlgebra.
function -(A::P, B::P) where P <: TensType
    return tensorcombination((1, -1), A, B)
end

import Base.
function *(num::Number, M::P) where P <: TensType
    return tensorcombination(M, alpha=(num,))
end
```
Each function calls the `tensorcombination` function and returns a completely new tensor, so no in-place functions are called here. To perform those operations, the `add!`, `sub!`, `mult!`, and `div!` functions can be used.

8. Functions: `sqrt!` & `sqrt`

The square root can be defined from `tensorcombination` just as before with the other functions. The output is either manipulated in-place on the first tensor (!) or not.

9. Functions: `inverse_element` & `invmat`

The inverse of an element can be taken in a tensor.
The same function is also defined for Julia’s base Array class. This is in case any algorithm must be run with the Array struct.

10. Functions: exp

Exponentiating a matrix is an internal function to Julia. The strategy is to take a matrix, $\hat{G}$, and perform an eigenvalue decomposition. The diagonal matrix has each diagonal element exponentiated. The unitary matrices remain unchanged as can be seen by an explicit Taylor series expansion

$$\exp(\hat{G}) = \hat{U} \exp(\hat{S}) \hat{U}^\dagger$$

where $\exp(\hat{S})$ means to take the elements of the diagonal $\hat{S}$ matrix (found by eigenvalue decomposition) and exponentiating each.

The operations covered here are for a single tensor. These are two of the four basic operations identified in Ref. 2.

G. Tensor operations

The first basic operation from Ref. 2 is the reshaping of a dense tensor. The action is very simple, the size field must be replaced by a new tuple. The number of elements does not change, so the elements of the new tuple must multiply to the same number as the original tuple.

There are several variations on the reshape function that are included here. One is to perform an in-place reshape, reshape!. This does a shallow copy on the elements of the tensor network (T field) and simply replaces the size. A version of this function is also defined for Julia’s array types.

function reshape!(M::tens{W}, S::NTuple{N,intType}; merge::Bool=false) where {N, W <: Number}
    M.size = S
    return M
end

function reshape!(M::tens{W}, S::intType...;merge::Bool=false) where W <: Number
    return reshape!(M,S)
end

function reshape!(M::tens{W}, S::Array{Array{P,1},1};merge::Bool=false)
    where {W <: Number, P <: Integer}
    newsize = ntuple(a->prod(b->size(M,b),S[a]),length(S))
    order = vcat(S...)
    pM = issorted(order) ? M : permutedims!(M,order)
At this time, julia does not have an established system for disabling errors in functions, and the option taken here is for fast code over error-proof code. If there was an option, then it would be added here, but instead we rely on using julia’s native tensor format to check if the reshapes are done correctly, then convert to the dense tensor type. Sometimes it is useful to input the reshape in terms of the bare indices on the tensor. This can be useful for bookkeeping and avoiding the need to type out lengthy expressions that compute the new size of a reshaped tensor. The example syntax is `reshape(A,[[1,2],[3]])` which reshapess the first two indices together of a rank-3 tensor. The equivalent in the other expression would look something like `reshape(A,size(A,1)*size(A,2),size(A,3))` generically, although other versions can be defined.

2. Function: reshape

Depending on the use of the `reshape` function, sometimes the function will need to be reshaped with a copy operation applied to the input tensor. This is accomplished with the following function.

```julia
import Base.reshape
function reshape(M::tens{W}, S::intType...;merge::Bool=false) where W <: Number
    newMsize = S
    newM = tens{W}(newMsize,M.T)
    return reshape!(newM,S)
end

function reshape(M::tens{W}, S::Array{Array{P,1},1};merge::Bool=false)
    where {W <: Number, P <: intType}
    return reshape!(copy(M),S)
end
```

3. Function: unreshape & unreshape!

This function was primarily introduced for documentation purposes. It is easier to read a reshape that restores the original shape if it goes by another name. Here, this merely acts as a `reshape` with a different name.

```julia
function unreshape!(M::AbstractArray, S::intType...;merge::Bool=false)
    where W <: Number
    return reshape(M,S...)
end

function unreshape!(M::tens{W}, S::intType...;merge::Bool=false)
    where W <: Number
    return reshape!(M,S)
end
export unreshape!

function unreshape(M::AbstractArray, S::intType...;merge::Bool=false)
    where W <: Number
    return reshape(M,S...)
end

function unreshape(M::tens{W}, S::intType...;merge::Bool=false)
    where W <: Number
    newM = tens{W}(M.size,M.T)
    return unreshape!(newM,S...)
end
```
There is a useful modification of this function for the quantum number case, and that was the second reason for defining this function.

4. Function: `permutedims!`

In addition to `permutedims`, there is also the possibility to perform this operations in place. To do this, the `permutedims!` operation from Julia is used, similarly to the discussion around `permutedims` (Sec. IV G 5).

```julia
import Base.permutedims!

function permutedims!(M::tens{W}, vec::Array{P,1})
    where {W <: Number, P <: intType}
    return permutedims!(M,(vec...),M.size...)
end

function permutedims!(M::tens{W}, vec::NTuple{N,intType}) where {N,W <: Number}
    return permutedims!(M,vec,M.size...)
end

function permutedims!(M::AbstractArray,
    vec::NTuple{N,intType}) where {N, W <: Number}
    return permutedims(M,vec)
end

function permutedims!(M::tens{W}, vec::NTuple{N,intType},
    x::intType...) where {N,W <: Number}
    rM = reshape(M.T,x)
    xM = permutedims(rM, vec)
    out = tens(W,xM)
    return out
end
```

There is only a small difference when using the two functions, but it can be convenient to use this function when creating new code instead of designating a new tensor each time.

5. Function: `permutedims`

Permuting the dimensions of a tensor is a basic operation that is probably defined in every language where tensor network methods will be used. It is a standard function in computer science, and while developing DMRjulia, it was questioned whether to recode this function from scratch. But the existing functions were so efficient and involved clever algorithms from computer science, that this function will be left to existing libraries, so that the focus here can be on the operations specifically necessary for tensor networks.

```julia
import Base.permutedims

function permutedims(M::tens{W}, vec::Array{P,1})
    where {W <: Number, P <: intType}
    return permutedims!(M,(vec...))
end
```

6. Function: `joinindex` & `joinindex!`

It is sometimes the case that two tensors must be joined along a given index. For example, if two rank-3 tensors are to be joined along the third index of each tensor (with all other tensors kept the same), then the function `joinindex` (or in-place `joinindex!`) can be called as `joinindex(A,B,[1,3])` to accomplish this. This is used mostly for the single site implementation of DMRG, although it can have other uses as well.
function joinindex!(bareinds::IntVecType,A::Array{R,N},
B::Array{S,N}::Array{typeof(R(1)*S(1)),N})
    where {R <: Number, S <: Number, N}
    inds = convIn(bareinds)
    nA = ndims(A)
    nB = nA
    finalsize = [size(A,i) for i = 1:nA]
    @simd for i = 1:length(inds)
        @inbounds a = inds[i]
        @inbounds finalsize[a] += size(B,a)
    end
    thisType = typeof(S(1)*R(1))
    if length(inds) > 1
        newTensor = zeros(thisType,finalsize...)
    else
        dim = length(finalsize)
        newTensor = Array{thisType,dim}(undef,finalsize...)
    end
    notinds = findnotcons(ndims(A),(inds...,))
    axesout = Array{UnitRange{IntType},1}(undef,nA)
    @simd for i = 1:length(notinds)
        @inbounds a = notinds[i]
        @inbounds axesout[a] = 1:finalsize[a]
    end
    @simd for i = 1:length(inds)
        @inbounds b = inds[i]
        @inbounds start = size(A,b)+1
        @inbounds stop = finalsize[b]
        @inbounds axesout[b] = start:stop
    end
    axesA = ntuple(a->1:size(A,a),nA)
    @inbounds newTensor[axesA...] = A
    @inbounds newTensor[axesout...] = B
    return newTensor
end

function joinindex!(inds::IntVecType,A::W,B::R...)
    where {W <: TensType, R <: TensType}
    if typeof(A) <: densTensType
        C = A
    else
        C = copy(A)
    end
    return joinindex!(inds,C,B...)
end

function joinindex(A::W,B::R,inds::IntVecType)
    where {W <: TensType, R <: TensType}
    mA,mB = checkType(A,B)
    return joinindex(inds,mA,mB)
end
export joinindex
function joinindex!(A::W,B::R,inds::intvecType)
    where {W <: TensType, R <: TensType}
    mA,mB = checkType(A,B)
    return joinindex!(inds,mA,mB)
end
export joinindex!

H. Display

1. Function: showTens

The function showTens outputs the most relevant information for an input denstens. The size is displayed, and the tensor is output to show digits (default 4).

function showTens(M::denstens; show::intType = 4)
    println("printing regular tensor of type: ", typeof(M))
    println("size = ", M.size)
    maxshow = min(show, size(M.T, 1))
    maxBool = show < size(M.T, 1)
    println("T = ", M.T[1:maxshow], maxBool ? "..." : "")
    nothing
end
export showTens

2. Function: print

This prints out the relevant information from showTens

import Base.print
function print(A::denstens...;show::intType = 4)
    showTens(A, show = show)
    nothing
end

3. Function: println

This is the print operation but an extra return is entered to start a new line. This is the most commonly used version in julia.

import Base.println
function println(A::denstens;show::intType = 4)
    showTens(A, show = show)
    println("\n")
    nothing
end

V. MODULE: OPT.JL

There are several operators that are frequently used for several lattice types. Normally, one would have to define some special properties with this, but here the julia arrays can be used, simply.

The three sets of operators will be defined for spins, Hubbard models (fermions), and $t-J$ models.
The spin operators that are generated are the standard operators for a given given spin system of dimension $s$. The standard relations are used to determine the matrix elements. The Pauli matrices are defined as [21]

$$
\hat{S}_a, b = (s + 1 - a) \delta_{a, b}, \quad \hat{S}_{m, m+1}^+ = \frac{\delta_{m, m+1}}{2} \sqrt{s(s+1) - m(m+1)}, \quad \hat{S}^- = (\hat{S}^+)\dagger
$$

where $a$ and $b$ index the rows and columns of the matrices.

The letter $O$ is reserved as a zero element. It is only full of zeros, simply. The matrix $\text{Id}$ is used as an identity operation. These are useful when constructing MPOs manually. The function was designed for integer and half-integer spin of any magnitude.

```plaintext
function spinOps(;s=0.5)
    states = convert(Int64,2*s+1) # number of quantum states
    O = zeros(Float64,states,states) # zero matrix
    Id = copy(O) + LinearAlgebra.I # identity matrix
    oz = copy(O) # z operator
    op = copy(O) # raising operator
    for (q,m) in enumerate(s:-1:-s) # counts from m to -m (all states)
        oz[q,q] = m
        if m+1 <= s
            op[q-1,q] = sqrt(s*(s+1) - m*(m +1))
        end
    end
    om = Array(op\') # lowering operator
    ox = (op+om)/2 # x matrix
    oy = (op-om)/(2*im) # y matrix
    return ox,oy,oz,op,om,Id
end

function spinOps(a::Float64)
    return spinOps(s=a)
end
export spinOps
```

Note the order output from the function and how it must appear when loading these operators into a program. If another operator is required, such as $\hat{S}_z \cdot \hat{S}_z$, then the syntax to generate this is simply $\hat{S}_z \cdot \hat{S}_z$.

2. Function: fermionOps.jl

The code to generate the fermion operators under this construction is [22, 23]

```plaintext
function fermionOps()
    states = 4 # fock space size
    O = zeros(Float64,states,states) # zero matrix
    Id = copy(O)+LinearAlgebra.I # identity
    Cup = copy(O) # annihilate (up)
    Cup[1,2] = 1.
    Cup[3,4] = 1.
    Cdn = copy(O) # annihilate (down)
    Cdn[1,3] = 1.
    Cdn[2,4] = -1.
    Nup = Cup\' * Cup # density (up)
end
```
Ndn = Cdn’ * Cdn # density (down)
Ndens = Nup + Ndn # density (up + down)

F = copy(Id) # Jordan-Wigner string operator
F[2,2] *= -1.
F[3,3] *= -1.

return Cup,Cdn,Nup,Ndn,Ndens,F,O,Id
end
export fermionOps

3. Function: tJOps.jl

The $t-J$ model is the same as the Hubbard model, but with double occupancy projected out. So, the operators from the Hubbard model are created and then only the first 3 states are kept.

```julia
function tJOps()
    # many of the Hubbard operators can be truncated
    Cup, Cdn, Nup, Ndn, Ndens, F, O, Id = fermionOps()
    states = 3 # fock space size
    s = states
    Cup = Cup[1:s,1:s]
    Cdn = Cdn[1:s,1:s]
    Nup = Nup[1:s,1:s]
    Ndn = Ndn[1:s,1:s]
    Ndens = Ndens[1:s,1:s]
    F = F[1:s,1:s]
    O = O[1:s,1:s]
    Id = Id[1:s,1:s]

    Sz = copy(O) # z-spin operator
    Sz[2,2] = 0.5
    Sz[3,3] = -0.5

    Sp = copy(O) # spin raising operator
    Sp[3,2] = 1.
    Sm = Array(Sp') # spin lowering operator

    return Cup,Cdn,Nup,Ndn,Ndens,F,Sz,Sp,Sm,O,Id
end
export tJOps
```

The spin operators are sometimes used in the context of $t-J$ models and are implemented here according to the relationship [24]

$$S = \langle \Psi | \hat{c}_{i,\sigma}^{\dagger} \hat{\sigma}_{\sigma',\sigma} \hat{c}_{i,\sigma'} | \Psi \rangle$$

(4)

To recover the matrix elements as was done for the fermionic operators to see how the matrix elements are given in the code.

VI. MODULE: CONTRACTIONS.JL

As the first example of one of the core operations in a tensor network, it is now shown how to code an efficient conversion algorithm to contract two tensors by using a matrix multiplication algorithm from a known library. The implementation may change for different languages that this is written in, but the basic concepts will remain the same.
A. Conversion of tensors to and evaluation of matrix multiplication

1. Function: `libmult!`

This is the core of the contraction algorithm. A function from the BLAS library is used to compute the relevant contraction. One feature that must be determined before the BLAS function is whether the types of all inputs is correct. That is done as a pre-processing step before activating the BLAS library.

```julia
function libmult(C::Array{T,2}, D::Array{X,2}, tempZ::Array{P,2};
    alpha::Number=1., beta::Number=1.)
    where {T <: Number, X <: Number, P <: Number}
    if length(tempZ) == 0
        if (eltype(C) == eltype(D)) && !(eltype(C) <: Integer)
            temp_alpha = typeof(alpha) <: eltype(C) ? alpha : convert(eltype(C), alpha)
            retMat = LinearAlgebra.BLAS.gemm('N','N', temp_alpha, C, D)
        else
            retMat = alpha * C * D
        end
    else
        if (eltype(C) == eltype(D) == eltype(Z[1])) && !(eltype(C) <: Integer)
            temp_alpha = convert(eltype(C), alpha)
            temp_beta = convert(eltype(C), beta)
            LinearAlgebra.BLAS.gemm!( 'N', 'N', temp_alpha, C, D, temp_beta, tempZ[1])
            retMat = tempZ[1]
        else
            retMat = alpha * C * D + beta * tempZ[1]
        end
    end
    return retMat
end
export libmult
```

The innermost matrix multiplication is often crucial to define. Even a small inefficiency can create a large time lag on the whole algorithm. At the present time, julia's BLAS library does not have all the features that could be implemented here. For example, changing the ‘N’ inputs (normal matrix) to ‘T’ for transpose cases almost not change in the computational speed even for very large matrices. If at a future time there are additional functionalities that are input into julia, then this is the function that must change to accommodate them.

A graphics processing unit (GPU) implementation of this function is still undergoing testing as of this version, but this is the place where such an implementation would go.

2. Function: `matrixequiv & matrixequiv!`

The function `matrixequiv` generates the matrix equivalent of a given input tensor. This can also happen in place. The main reason for defining this function is that it is slightly easier for the compiler to handle the input if the types are separated.

```julia
function matrixequiv(X::densTensType, Lsize::Integer, Rsize::Integer)
    return matrixequiv!(copy(X), Lsize, Rsize)
end

function matrixequiv!(X::AbstractArray, Lsize::Integer, Rsize::Integer)
    return reshape!(X, Lsize, Rsize)
end

function matrixequiv!(X::denstens, Lsize::Integer, Rsize::Integer)
    return reshape!(X.T, Lsize, Rsize)
end
```
The definition of these functions might seem to be an over-use of the function structure here, but Julia often optimizes code at the function barriers to ensure type stability. The functions are kept here largely in case a future compiler version is aided by these functions. The same is true for the next function.

3. Function: permutedims_2matrix!

The function \texttt{permutedims_2matrix!} generates the matrix equivalent of a given input tensor after first permuting some of the indices.

\begin{verbatim}
function permutedims_2matrix!(X::AbstractArray, vec::Tuple, Lsize::Integer, Rsize::Integer)
    xM = permutedims(X, vec)
    return reshape!(xM, Lsize, Rsize)
end

function permutedims_2matrix!(X::denstens, vec::Tuple, Lsize::Integer, Rsize::Integer)
    return permutedims_2matrix!(makeArray(X), vec, Lsize, Rsize)
end
\end{verbatim}

4. Function: prepareT

The function \texttt{prepareT} will make the matrix equivalent of the tensor based on the conjugation and contracted indices. The return values are then multiplied together and the result is then unreshaped when passed back to \texttt{maincontractor}.

\begin{verbatim}
function prepareT(A::densTensType, Lvec::Tuple, Rvec::Tuple, conjvar::Bool)
    vec = (Lvec..., Rvec...)
    Lsize = length(Lvec) > 0 ? prod(a->size(A,a), Lvec) : 1
    Rsize = length(Rvec) > 0 ? prod(a->size(A,a), Rvec) : 1
    issort = true
    a = 1
    while issort && a < length(Lvec)
        a += 1
        issort = Lvec[a-1] < Lvec[a]
    end
    if length(Rvec) > 0 && length(Lvec) > 0
        issort = issort && Lvec[end] < Rvec[1]
        a = 1
    end
    while issort && a < length(Rvec)
        a += 1
        issort = Rvec[a-1] < Rvec[a]
    end
    if issort
        if eltype(A) <: Complex && conjvar
            out = matrixequiv(A, Lsize, Rsize)
        else
            out = matrixequiv!(A, Lsize, Rsize)
        end
    else
        out = permutedims_2matrix!(A, vec, Lsize, Rsize)
    end
end
\end{verbatim}
This function takes care not compute a permutation or conjugation when it is not necessary (i.e., indices are already ordered or element type is real). It will only copy the input when it must. This is the function where it is convenient to conjugate the input tensor because the matrix equivalent can be regarded as a separate tensor entirely without changing the input tensor’s values.

5. Function: corecontractor

The tensors must now be converts to their matrix equivalent representation having defined a permutation and reshaping of the tensors. This function corecontractor will manipulate both tensors according to their contracted indices and their conjugation.

function corecontractor
  (conjA :: Bool, conjB :: Bool, A :: densTensType, iA :: intvecType,
   B :: densTensType, iB :: intvecType, Z :: densTensType...;
    alpha :: Number=1., beta :: Number=1.)

  notvecA = findnotcons(ndims(A), iA)
  C = prepareT(A, notvecA, iA, conjA)
  notvecB = findnotcons(ndims(B), iB)
  D = prepareT(B, iB, notvecB, conjB)

  if length(Z) > 0
    tempZ = reshape(Z[1], size(C, 1), size(D, 2))
    CD = libmult(C, D, tempZ... , alpha=alpha, beta=beta)
  else
    CD = libmult(C, D, alpha=alpha, beta=beta)
  end

  AAsizes = ()
  for w = 1:length(notvecA)
    AAsizes = (AAsizes... , size(A, notvecA[w]))
  end
  for w = 1:length(notvecB)
    AAsizes = (AAsizes... , size(B, notvecB[w]))
  end
  return CD, AAsizes
end

The function findnotcons was already defined in tensor.jl and returns the complement tuple to the input contracted indices. It represents the uncontracted indices.

6. Function: maincontractor

In the previously defined front-end contractors, each led to the maincontractor function. This will receive an argument indicating that it will be conjugated or not (conjA and conjB).

function maincontractor
  (conjA :: Bool, conjB :: Bool, A :: AbstractArray, iA :: intvecType,
   B :: AbstractArray, iB :: intvecType, Z :: AbstractArray...; alpha :: Number=1.,
   beta :: Number=1.)

  CD, AAsizes = corecontractor(conjA, conjB, A, iA, B, iB, Z... , alpha=alpha, beta=beta)
  return reshape(CD, AAsizes)
end
function maincontractor(conjA::Bool, conjB::Bool, A::tens{X}, iA::Tuple, B::tens{Y}, iB::Tuple, Z::tens{W}..., ; alpha::Number=1., beta::Number=1.) where {X <: Number, Y <: Number, W <: Number}

CD, AAsizes = corecontractor(conjA, conjB, A, iA, B, iB, Z..., alpha=alpha, beta=beta)
outType = typeof(X(1)*Y(1))
nelem = prod(size(CD))
return tens{outType}(AAsizes, reshape(CD, nelem))
end
export maincontractor

The reason for calling yet another function corecontractor is because beyond this point, the functions for the AbstractArray type and denstens converge.

At this point, having used so much of julia’s machinery to compute the contraction, it might be asked why bother defining the tensor with the denstens type. The answer is that neglecting the rank of the tensor allows for better compilation of the code by julia’s compiler, especially when a tensor is being reshaped frequently. This feature also makes it easier to handle at the higher levels. Since this function only requires a matrix input, then this makes a natural point where julia’s linear algebra operations become efficient to use. This also carries a message for lower level implementations. The conversion of the tensor to matrices for contraction (and also decomposition) means that programs to compute the matrix multiplication can be used as this point.

In terms of the DMRG algorithm, the contraction step is often practically the most expensive. The decomposition step will formally scale worse than the contraction step, but the need to compute several contractions (6 in a straightforward implementation of the two-site algorithm [3]) to form the reduced site representation of Schrödinger’s equation at each step. These tend to be where the code will spend most of its time, so it is important to ensure the contraction algorithm is efficient.

B. Contraction functions

The contract function is one of the four ore operations in a tensor network library as defined in Ref. 2. One overarching point about the implementation of the contract function here is that the code will not automatically catch errors. One strategy is to use these functions with julia’s Array type which will find errors in the input to some degree. If julia implements a good method for disabling error messages, then this will be considered in a future version.

1. Functions: contract, ccontract, contractc, and ccontractc (scalar output)

There are ways to ensure that the conjugation of a tensor is done efficiently if a tensor and one of the tensors needs to be conjugated. In order to account for the four types of contractions with and without conjugation between two tensors, the functions contract, ccontract, contractc, and ccontractc are defined to be dependent on a maincontractor function that will be defined above.

If the result of the contraction has no indices, it is useless to define the input indices. Inputting only the two tensors to be contracted will automatically assume that the tensors are correctly permuted and then output a scalar value.

function contract(A::TensType, B::TensType; alpha::Number=1.)
 mA, mB = checkType(A, B)
 vec_in = ntuple(i->i, ndims(mA))
 C = contract(mA, vec_in, mB, vec_in, alpha=alpha)
 return searchindex(C, 1, 1)
end

function ccontract(A::TensType, B::TensType; alpha::Number=1.)
 mA, mB = checkType(A, B)
 vec_in = ntuple(i->i, ndims(mA))
 C = ccontract(mA, vec_in, mB, vec_in, alpha=alpha)
 return searchindex(C, 1, 1)
end
function contractc(A::TensType, B::TensType; alpha::Number=1.)
    mA, mB = checkType(A, B)
    vec_in = ntuple (i->i, ndims(mA))
    C = contractc(mA, vec_in, mB, vec_in, alpha=alpha)
    return searchindex(C, 1, 1)
end

function ccontractc(A::TensType, B::TensType; alpha::Number=1.)
    mA, mB = checkType(A, B)
    vec_in = ntuple (i->i, ndims(mA))
    C = ccontractc(mA, vec_in, mB, vec_in, alpha=alpha)
    return searchindex(C, 1, 1)
end

One additional feature that is often useful for debugging (or to directly access the norm squared of a tensor) is to admit one single tensor as an argument. This essentially computes $A \cdot A$ (contraction of a tensor $A$ along all indices or conjugating one of the two terms with ccontract or contractc). This 'self-contraction' function is only useful for coding purposes in some cases, but the use of norm and then squaring the output is perfectly acceptable in this case and produces identical results.

function contract(A::TensType; alpha::Number=1.)
    return contract(A, A, alpha=alpha)
end

function ccontract(A::TensType; alpha::Number=1.)
    return ccontract(A, A, alpha=alpha)
end

function contractc(A::TensType; alpha::Number=1.)
    return contractc(A, A, alpha=alpha)
end

function ccontractc(A::TensType; alpha::Number=1.)
    return ccontractc(A, A, alpha=alpha)
end

2. Functions: contract, ccontract, contractc, and ccontractc (tensor output)

For each of the functions that result in a tensor (some indices remain), a prefactor can be applied as well as an additional tensor added onto the contraction. This is to mirror the axpy functions that are commonly in many libraries for matrix multiplication,

$$C = \alpha A \cdot B + \beta Z \quad (5)$$

but the multiplications here should be generalized to tensor contractions.

The inputs $iA$ and $iB$ can be any of the genvecType but will ultimately be converted to a tuple of integers. The reason for using a tuple is because it can be more efficient in the view of the compiler, although the difference here is negligible as was discussed for convIn.

function contract(A::TensType, iA::IntvecType, B::TensType, iB::IntvecType, Z::TensType...; alpha::Number=1., beta::Number=1.)
    mA, mB = checkType(A, B)
    return maincontractor(false, false, mA, convIn(iA), mB, convIn(iB), Z..., alpha=alpha, beta=beta)
end

function ccontract(A::TensType, iA::IntvecType, B::TensType, iB::IntvecType, Z::TensType...; alpha::Number=1., beta::Number=1.)
    mA, mB = checkType(A, B)
    return maincontractor(false, false, mA, convIn(iA), mB, convIn(iB), Z..., alpha=alpha, beta=beta)
end
Recall that the use of checkType is essentially zero if the same two types are input into the program. It is conceded that this implementation is especially easy in Julia where types can be defined generally. For lower level implementations, more function calls may be required. However, for learning purposes, this is sufficient.

Note the use of maincontractor as a unifying function here. The boolean inputs will determine where the tensors are conjugated (when constructing the matrix equivalent).

Note that the $Z$ tensor input does not need to be present even though it appears in the function call. This is the same for $alpha$ and $beta$. So, all of

contract(A,(1,2),B,(1,2))
contractc(A,(1,2),B,(1,2),alpha=3)
contract(A,[1,2],B,[1,2])
contractc(A,[1,2],B,[1,2],Z)
contractc(A,[1,2],B,[1,2],Z,alpha=3.,beta=4)

are valid and more combinations are possible. Note that if two indices are reshaped together on one tensor and that the other tensor has not, then the call

A = rand(20,30,40)
B = rand(20,30,40)
rB = reshape(B,[[1],[2,3]])
C = contract(A,(2,3),rB,(2,)) # = contract(A,(2,3),B,(2,3))

is valid.

3. Functions: contract, ccontract, contractc, and ccontractc (re-ordered tensor output)

One very frequently used feature of the contract function is to reorder the tensor once it has been contracted. This is accomplished by adding the reordering vector as the first argument, order.

function contract(order::IntVecType,A::TensType,iA::IntVecType,B::TensType,iB::IntVecType,Z::TensType...;alpha::Number=1.,beta::Number=1.)
    mA,mB = checkType(A,B)
    return maincontractor(true,false,mA,convIn(iA),mB,convIn(iB),Z...,
        alpha=alpha,beta=beta)
end

function ccontract(order::IntVecType,A::TensType,iA::IntVecType,B::TensType,iB::IntVecType,Z::TensType...;alpha::Number=1.,beta::Number=1.)
    mA,mB = checkType(A,B)
    return maincontractor(false,true,mA,convIn(iA),mB,convIn(iB),Z...,
        alpha=alpha,beta=beta)
end

export contract,ccontract,contractc,ccontractc
mA,mB = checkType(A,B)
newT = ccontract(mA,iA,mB,iB,Z...,alpha=alpha,beta=beta)
return permutedims!(newT,convIn(order))
end

function contractc(order::intvecType,A::TensType,iA::intvecType,B::TensType,
iB::intvecType,Z::TensType...;alpha::Number=1.,beta::Number=1.)
mA,mB = checkType(A,B)
newT = contractc(mA,iA,mB,iB,Z...,alpha=alpha,beta=beta)
return permutedims!(newT,convIn(order))
end

function ccontractc(order::intvecType,A::TensType,iA::intvecType,B::TensType,
iB::intvecType,Z::TensType...;alpha::Number=1.,beta::Number=1.)
mA,mB = checkType(A,B)
newT = ccontractc(mA,iA,mB,iB,Z...,alpha=alpha,beta=beta)
return permutedims!(newT,convIn(order))
end

An example of how to use this function is given here with the equivalent permute after the contraction is evaluated.

A = rand(15,20,5)
B = rand(5,10,20)
C = contract(A,(3,2),B,(1,3))
D = permutedims!(C,[2,1])
#or
D = contract([2,1],A,(3,2),B,(1,3))

4. Function: trace

Where the contract function had a self-contract function that contracted over all indices in a tensor with a copy of that tensor, there is also the need to perform a partial trace over some of the indices on a tensor. This will be handled by the trace function. For example,

A = rand(10,20,40,30,10,5,20)
B = trace(A,[[1,5],[2,7]])

will trace over the indices first and fifth indices on A as well as the second and seventh indices. For a single set of indices, the command can reduce to

B = trace(A,[1,5])

This function makes use of the makeId function that was introduced earlier to accomplish the necessary contractions. Essentially, an identity matrix (or tensor) is contracted onto the indices in question.

function trace(A::TensType,iA::W) where W <: Union{intvecType,AbstractArray}
    if typeof(iA) <: intvecType
        Id = makeId(A,iA)
        conA = iA
    else
        Id = makeId(A,iA)
        conL = [iA[w][1] for w = 1:length(iA)]
        conR = [iA[w][2] for w = 1:length(iA)]
        conA = vcat(conL,conR)
    end
    return contract(A,conA,Id,[i for i = 1:ndims(Id)])
end
export trace
5. Function: checkcontract

It is sometimes the case that the contraction must be checked as a part of debugging a program. The checkcontract function does this. Remember that DMRjulia does not implement inherent error checks and instead provides functions such as checkcontract to accomplish this. The dense case simply tests the sizes of the indices that are being contracted over are the same. The quantum number case also implements a check for the fluxes of each index [4].

```julia
function checkcontract(A::TensType, iA::intvecType, B::TensType, iB::intvecType, Z::TensType..., alpha::Number=1., beta::Number=1.)
    mA, mB = checkType(A, B)
    if size(mA)[iA] == size(mB)[iB]
        println("contracting over indices with equal sizes")
    else
        error("some indices in A or B are not equal size; A->", size(mA)[iA],", B->",size(mB)[iB])
    end
    if typeof(mA) <: qarray
        println("checking flux:")
        checkflux(mA)
        checkflux(mB)
        for a = 1:length(iA)
            AQNs = recoverQNs(iA[a], A)
            BQNs = recoverQNs(iB[a], B)
            println("contracted index $a (A's index: ",iA[a],", B's index: ",iB[a],")")
            if length(AQNs) == length(BQNs)
                for w = 1:length(AQNs)
                    if AQNs[w] != inv(BQNs[w])
                        error("unmatching quantum numbers on indices on index ", iA[a]," of A and index ",iB[a]," of B: value ",w)
                    end
                end
            else
                error("unmatching quantum number vector lengths")
            end
            println("maching quantum numbers on both indices")
        end
    end
    nothing
end
export checkcontract
```

If the function finds not errors, then nothing will happen (except for a slowdown due to checking the relevant quantities). However, if there is an error, the program will be interrupted and a partially explanatory message will be displayed.

VII. MODULE: DECOMPOSITIONS.JL

The module containing the decomposition functions will contain all of SVD, eigen, and QR and LQ decompositions.

1. Function: libsvd

As of the present version, julia occaisionally uses a lazy permute type for transposes, permutes, and other operations. This means the operation is not performed when the function is called. A new data type is created with special instruction to perform the operation when the language determines that it must be done. It was noticed in an earlier
version that this action generated a tremendous slow down in the code because it did not handle parallelization well. To guard against this, the library’s SVD function explicitly converts the transpose of Ŷ to a generic Array.

```plaintext
function libsvd(X::Array{T,2}) where T <: Number
    F = LinearAlgebra.svd(X)
    return F.U,F.S,Array(F.Vt)
end
export libsvd
```

In the online version of this library, another safeguard was added to ensure that some undefined flags would not cancel a computation. These only appear in a few contrived circumstances and can mostly be resolved by implementing some low-level functions. However, the basic implementation here does not need them, and they can be guarded against by properly defining the core BLAS or other library’s function.

2. Function: recursive_SVD

This recursive version of the SVD will compute all singular values with greater precision. The native SVD implementation in many linear algebra solvers will limit precision based on the largest singular value.

```plaintext
function recursive_SVD(AA :: Array{W,2},
    tolerance :: Float64 = 1E-4) :: Tuple{Matrix{W},Vector{Float64},Matrix{W}}
where W <: Number
    U,D,V = safesvd(AA)
    counter = 2
    anchor = 1
    while counter <= size(D,1)
        if D[counter]/D[anchor] < tolerance
            Lpsi = U[:,counter:size(D,1)]
            Rpsi = V[counter:size(D,1),:]
            X = LinearAlgebra.dot(Lpsi,AA,Rpsi)
            Up,Dp,Vp = safesvd(X)
            for a = counter:size(D,1)
                U[:,a] = sum(p->U[:,p]*Up[p-counter+1,a-counter+1],counter:size(D,1))
                V[a,:] = sum(p->V[p,:]*Vp[p-counter+1,a-counter+1],counter:size(D,1))
            end
            D[counter:size(D,1)] = Dp
            anchor = counter
        end
        counter += 1
    end
    newD = Array(LinearAlgebra.Diagonal(D))
    return U,newD,V
end
```

This function is rarely useful for algorithms but may be useful for plotting singular values over a wide range of magnitudes. It can be called through `svd(...,recursive=true)`

3. Function: findnewm

This function obtains the number of singular values to keep on truncation. The input is the $D$ object from the SVD in a vector form (equivalent to a diagonal matrix), the maximal bond dimension $m$, the magnitude of the original input matrix $mag$, the cutoff parameter $cutoff$, the effective value of zero $effZero$ (set to machine precision for doubles of $10^{-16}$ in this library), $nozeros$ will remove all approximately zero singular values, $power$ is the exponent to which the singular values should be raised to, and the boolean $keepdeg$ will avoid truncating degenerate singular values if set to true.

```plaintext
function findnewm(D::Array{W,1},m::Integer,minm::Integer,mag::Float64,
    cutoff::Real,effZero::Real,nozeros::Bool,power::Number,
```

This function also applies to the truncation of eigenvalues, often where $\text{power} = 1$ in that case.

4. Function: safesvd

For all practical purposes, this function is equivalent to `LinearAlgebra.svd`. Some incompatibilities for highly degenerate input matrices were noticed, so backup implementations of the SVD from Ref. 3 were implemented to solve those issues. At the present time, Julia is not fully interfaced with the BLAS library’s implementation of the SVD in that some options are pre-selected. In principle, choosing these options fully could make the SVD work here, but for safety and backward compatibility, this function will remain here. There has been no testable consequence for this by us and the library’s implementation works on all tested cases.

See the current code version for the current implementation of this function [5].

5. Function: svd

The primary SVD function truncates the singular values according to the discussion in the text. The power that the singular values are taken to before truncation can be adjusted, but there is rarely a use to do this. Instead, the function is uniform between the SVD and the eigenvalue solver below, which justifies its purpose.
There are many aspects of this function that are discussed here. For the physics interpretation of the decomposition or some of the approximation parameters such as `cutoff`, the reader is directed to Ref. 3.

The options for this function can be useful in many situations. The parameter `power` simply gives the effective norm at which the singular values should be truncated (here, the values are squared since those are the values of the density matrix [3]). The parameter `cutoff` is the maximum sum of singular values to truncate that can override the `m` parameter (maximum bond dimension of the newly introduced index).

The `mag` is the Frobenius norm of the input tensor and is the value that the cutoff is scaled to. This often matters when truncating two tensors that are joined together. There is a miniscule cost savings with setting this parameter, but it can be done and avoid the computation (for example, the two-site DMRG algorithm is always norm 1 for all tensors, so this can be set to be unity always).

Mainly for testing purposes, the `minm` parameter can be set to guarantee a bond dimension greater than `minm`. This can be have a use in other contexts, but it is rare to need it. The `dmrg` function does set `minm=2` by default because it is assumed that all models will search for an entangled ground state which guarantees \( m \geq 2 \). This is often useful for stability reasons but again is not strictly necessary. There is no problem using this with a purely classical model with bond dimension of 1 everywhere.

There are also sometimes when only the non-zero values of the SVD should be kept and this can be toggled with `nozeros`. If the recursive SVD from Ref. 25 is demanded, then `recursive` can be toggled to true. The recursive SVD is almost never required for an algorithm, although it can be useful for displaying highly accurate singular values.

The `effZero` parameter specifies the effective zero value for the computation and `keepdeg` will not truncate degenerate singular values.

Note that the output arrays are ensured to be of the `Array` type. A large slowdown was noticed when using the special array types in julia. Conversion to the array types did not create a large slowdown or excessive allocations, so this was used instead of possibly causing a single-thread slow-down for some large computation later on.

The outputs from the `svd` function are the \( U, D, \) and \( V^\dagger \) tensors with some additional information that can be useful. The fourth output is the truncation error. The final value is the magnitude. The julia language allows for the specifying of only (for example) three of the outputs and will leave off those not specified. If one output is set, then a tuple is stored in that variable with all five outputs, so the previous trick of reducing the outputs will work for 2 or more outputs here.

There is a similar function defined for the `denstens` input.
One convenient format for the SVD is to group indices as was done for the reshape function. An input such as
\[ \text{svd}(A, [[1, 2], [3]]) \]
will group together indices 1 and 2, perform the SVD, and then unreshape those indices on the returning \( U \) tensor.

### Function: libeigen

The eigenvalue solver used in the library assumes a symmetric matrix input. This is both for efficiency in Julia and because it is the expected case. The function handles both orthogonal basis sets (regular eigenvalue decomposition) and problems requiring a generalized eigenvalue problem.

```julia
function libeigen(AA::Array{W,2}, B::Array{W,2}...) where W <: Number
    if eltype(AA) <: Complex
        M = LinearAlgebra.Hermitian((AA + AA') / 2)
    else
        M = LinearAlgebra.Symmetric((AA + AA') / 2)
    end
    return length(B) == 0 ? LinearAlgebra.eigen(M) : LinearAlgebra.eigen(M, B[1])
end
```

Note that this implementation will therefore assumes Hermitian input Hamiltonians.

### Function: eigen

The eigenvalue decomposition can be similarly defined to the SVD. The truncation on the \( \hat{D} \) matrix is done without squaring the value as was done for the SVD.

```julia
function eigen(AA::Array{W,2}, B::Array{W,2}...; cutoff::Float64 = 0.,
    m::Integer = 0, mag::Float64 = 0., minm::Integer = 1,
    nozeros::Bool = false, power::Number = 1,
    effZero::Float64 = 1E-16,
    keepdeg::Bool = false) where {W <: Number}
    Dsq, U = libeigen(AA, B...)
end
```
thism, sizeD, truncerr, sumD = findnewm(Dsq, m, minm, mag, cutoff, effZero, 
    nozeros, power, keepdeg)

if sizeD > thism
    @inbounds Utrunc = U[:, sizeD - thism + 1:sizeD]
    @inbounds Dtrunc = Dsq[sizeD - thism + 1:sizeD]
elseif sizeD < minm
    Utrunc = zeros(eltype(U), size(U, 1), minm)
    Dtrunc = zeros(eltype(Dsq), minm)
    @inbounds Utrunc[:, 1:thism] = U
    @inbounds Dtrunc[1:thism] = Dsq
else
    Utrunc = U
    Dtrunc = Dsq
end
finalD = Array(LinearAlgebra.Diagonal(Dtrunc))::Array{Float64,2}
return finalD, Utrunc, truncerr, sumD
end
export eigen

Note that the eigenvalue decomposition is taken to be symmetric in the center.
The inputs are nearly identical to the svd function. The exception is that the overlap matrix B can be provided in the even that a generalized eigenvalue problem is to be solved of the form

$$\mathcal{H}\psi = E \hat{B}\psi$$  \hspace{1cm} (6)

The function is also defined for a denstens input

function eigen(AA::tens{W}, B::tens{R}..., cutoff::Float64 = 0., 
m::Integer = 0, mag::Float64=0., minm::Integer=2, 
nozeros::Bool=false, recursive::Bool = false, 
keepdeg::Bool=false) where {W <: Number, R <: Number}
    X = reshape(AA.T, size(AA)...) 
    D, U, truncerr, sumD = eigen(X, B..., cutoff=cutoff, m=m, mag=mag, minm=minm, 
        nozeros=nozeros, keepdeg=keepdeg)
    tensD = tens(Float64, D)
    tensU = tens(T, U)
    return tensD, tensU, truncerr, mag
end

Similarly for the reshape and svd functions, the eigenvalue decomposition can conveniently group indices together and then unreshape them. An input such as eigen(A, [[1, 2], [3]]) will group together indices 1 and 2, perform the eigenvalue decomposition, and then unreshape those indices on the returning \( \hat{U} \) tensor.

function eigen(AA::TensType, vecA::Array{Array{W, 1}, 1}, 
    B::TensType..., cutoff::Float64 = 0., m::Integer = 0, 
    mag::Float64=0., minm::Integer=1, nozeros::Bool=false, 
    keepdeg::Bool=false) where W <: Integer
    AB, Lsizes, Rsizes = getorder(AA, vecA)
    D, U, truncerr, newmag = eigen(AB, B..., cutoff=cutoff, m=m, mag=mag, minm=minm, 
        nozeros=nozeros, keepdeg=keepdeg)
    outU = unreshape!(U, Lsizes..., size(D, 1))
    return D, outU, truncerr, newmag
end

8. Function: libqr & liblq

The QR decomposition and LQ decompositions are implemented here using the LinearAlgebra package.
function libqr(R::TensType; decomposer::Function=LinearAlgebra.qr)
    A = makeArray(R)
    P, Q = decomposer(A)
    if size(P,2) > size(Q,1)
        U = P[:,1:size(Q,1)]
        V = Array(Q)
    elseif size(P,2) < size(Q,1)
        U = Array(P)
        V = Q[1:size(P,2),:]
    else
        U = Array(P)
        V = Array(Q)
    end
    return U, V
end

function liblq(R::TensType; decomposer::Function=LinearAlgebra.lq)
    return libqr(R, decomposer = decomposer)
end

The LQ decomposition can be defined very nearly the same way but by changing out the function from julia’s LinearAlgebra package. It turns out that in this version of julia, using a function as an argument has little overhead.

9. Function: qr & lq

The LQ and QR decompositions produce a unitary tensor \( \hat{Q} \) and another weighted tensor \( \hat{L} \) or \( \hat{R} \). One realization of the LQ and QR decompositions is the SVD with the \( \tilde{D} \) contracted onto one of the \( \tilde{U} \) or \( \tilde{V}^\dagger \) tensors. However, the LQ and QR decomposition have the special property that \( \hat{R} \) (\( \hat{L} \)) is upper (lower) triangular.

It is in principle possible to truncate the decomposition, but this will not be implemented here. The return values for the truncation error and magnitude will be set to 0 and 1 respectively.

function qr(AA::TensType, vecA::Array{Array{W,1},1}) where W <: Integer
    AB, Lsizes, Rsizes = getorder(AA, vecA)
    Qmat, Rmat, truncerr, newmag = qr(AB)
    innerdim = size(Qmat,2)
    outU = unreshape!(Qmat, Lsizes..., innerdim)
    outV = unreshape!(Rmat, innerdim, Rsizes...)
    return outU, outV, truncerr, newmag
end

function qr(AA::AbstractArray; decomposer::Function=LinearAlgebra.qr)
    return libqr(AA, decomposer = decomposer)
end

function qr(AA::denstens; decomposer::Function=LinearAlgebra.qr)
    rAA = makeArray(AA)
    Q, R = libqr(rAA, decomposer = decomposer)
    return tens(Q), tens(R), 0., 1.
end

function lq(AA::TensType, vecA::Array{Array{W,1},1}) where W <: Integer
    AB, Lsizes, Rsizes = getorder(AA, vecA)
    Lmat, Qmat, truncerr, newmag = lq(AB)
    innerdim = size(Qmat,1)
outU = unreshape!(Lmat, Lsizes..., innerdim)
outV = unreshape!(Qmat, innerdim, Rsizes...)
return outU, outV, truncerr, newmag
end

function lq(AA::AbstractArray; decomposer::Function=LinearAlgebra.lq)
    return libqr(AA, decomposer=decomposer)
end

function lq(AA::denstens; decomposer::Function=LinearAlgebra.lq)
    rAA = makeArray(AA)
    L, Q = libqr(rAA, decomposer=decomposer)
    return tens(L), tens(Q), 0., 1.
end

Passing functions as arguments has very little overhead in julia, hence the definition of lq being based on qr. Just as with the SVD, a vector of indices to group together can be input to these functions.

10. Function: polar

The polar decomposition can be thought of as a post-processing step onto the SVD. The purpose is to maintain the outer basis states on the inner index generated by the SVD. There are two types of decompositions that can be obtained,

\[
\hat{U} \hat{D} \hat{U}^\dagger, \hat{U} \hat{V} \hat{V}^\dagger \quad \text{or} \quad \hat{U} \hat{V} \hat{U}^\dagger, \hat{V} \hat{D} \hat{V}^\dagger
\]

which can be selected by specifying right = false (left form) or right = true (right form).

The group of indices must be specified similar to the vector input formats for both the svd and eigen functions. The last new option is to specify the outermaxm which allows for a truncation on the outer index. This is rarely used since the tensor should be preserved but it can limit the inner bond dimension. All other options are the same as the SVD functions defined earlier.

function polar(AA::TensType, group::Array{Array{W,1},1};
    right::Bool=true, cutoff::Float64 = 0., m::Integer = 0,
    mag::Float64 = 0., minm::Integer = 1, nozeros::Bool=false,
    recursive::Bool=false, outermaxm::Integer = 0,
    keepdeg::Bool=false) where W <: Integer
    AB, Lsizes, Rsizes = getorder(AA, group)
    U, D, V, truncerr, newmag = svd(AB, cutoff=cutoff, m=m, mag=mag, minm=minm,
        nozeros=nozeros, keepdeg=keepdeg)

    U = unreshape!(U, Lsizes..., size(D,1))
    V = unreshape!(V, size(D,2), Rsizes...)

    if right
        if outermaxm > 0
            outermtrunc = m > 0 ? min(m, outermaxm, size(V,2)) : min(outermaxm, size(V,2))
            truncV = getindex(V, :, 1:outermtrunc)
            modV = V
        else
            truncV = V
            modV = V
        end
        DV = contract(D, 2, truncV, 1)
        rR = ccontract(modV, 1, DV, 1)
        rU = contract(U, 3, modV, 1)
        leftTensor, rightTensor = rU, rR
    else
        if outermaxm > 0
            outermtrunc = m > 0 ? min(m, outermaxm, size(U,1)) : min(outermaxm, size(U,1))
            truncU = getindex(U, :, 1:outermtrunc)
            modU = U
        else
            truncU = U
            modU = U
        end
        DU = contract(U, 1, truncU, 1)
        rR = ccontract(modU, 1, DU, 1)
        rU = contract(U, 3, modU, 1)
        rightTensor, leftTensor = rU, rR
    end
end
outmtrunc = m > 0 ? min(m, size(U, 1), outermaxm) : min(outermaxm, size(U, 1))
truncU = getindex(U, 1: outmtrunc, :)
modU = U
else
    truncU = U
    modU = U
end
UD = contract(truncU, 2, D, 1)
1R = contractc(U, 2, U, 2)
1U = contract(U, 2, V, 1)
leftTensor, rightTensor = 1R, 1U
end
return leftTensor, rightTensor, D, truncerr
end
export polar

VIII. MODULE: MPTASK.JL

This module will initialize the MPS and MPO for a given system. There is an interest in preserving the memory footprint of the system on a computer in terms of random access memory (RAM). To reduce the size of the tensors stored, the tensors of the MPS and MPO can be written to a disk and only called when necessary. Remarkably, this strategy works well even with an expected increase in garbage collection time (most likely due to a forced garbage collection step when the tensor is written or read). Further remarkably, the overhead costs very little time when used. Even further remarkably, no high level functions need to be rewritten for this feature to appear, so just as will be seen for quantum number tensors, this improvement adds almost no practical hurdle to implementation, but it will be required for some of the larger systems.

1. Types: MPS, MPO, and Env

There are several types that should be defined for defining the MPS, MPO, and the environment tensors. It is not strictly required to define the environment tensors as a new type. One could use the basic definition in julia, yet defining this type is much easier to think through in terms of the code and it makes the code compatible with the large types defined below.

For now, all tensors will be kept in the available memory (not the hard disk) and be regarded as “regular” (hence the prefix reg found in some places). The types to define are MPS, MPO, and Env (referring to the environment.

abstract type MPS end
export MPS

abstract type MPO end
export MPO

abstract type regMPS <: MPS end
export regMPS

abstract type regMPO <: MPO end
export regMPO

abstract type envType end
export envType

abstract type regEnv <: envType end
export regEnv

const Env = Union{AbstractArray, envType}
export Env

2. Struct: matrixproductstate

The MPS is contained in a series of tensors that have two fields. One is the tensor $A$ (named after the traditional pen and paper notation) and the orthogonality center $oc$.

```plaintext
mutable struct matrixproductstate{W} <: regMPS where W <: Array{TensType,1}
    A::W
    oc::intType
end
```

Whenever needed, this struct’s name is never written out explicitly. Instead, the generic type MPS is used.

3. Struct: matrixproductoperator

Similar to the matrixproductstate, the MPO is contained in the matrixproductoperator struct. There is no gauge condition on the MPO, so it is not a field that is stored in the struct.

```plaintext
mutable struct matrixproductoperator{W} <: regMPO where W <: Array{TensType,1}
    H::W
end
```

Whenever needed, this struct’s name is never written out explicitly. Instead, the generic type MPO is used.

4. Struct: environment & vecenvironment

The environment could have been defined synonymously with the matrixproductoperator, but it is useful to have a separate name for this particular field. It is therefore included here as the following. There is no established symbol for the environment tensor, so the $V$ is borrowed for the most pronounced letter of the word.

```plaintext
mutable struct environment{W} <: regEnv where W <: Array{TensType,1}
    V::Array[W,1]
end

mutable struct vecenvironment{W} <: regEnv where W <: Array{TensType,1}
    V::Array[Array[W,1]]
end
export vecenvironment
```

Whenever needed, this struct’s name is never written out explicitly. Instead, the generic type Env is used. The main reason for defining the vector of environments (vecenvironment) is to be able to use the global type Env for this quantity as well. There are a few situations where this is useful for general MPS optimization.

5. Constructor: environment

To make an environment from a list of tensors, the following function can be defined.

```plaintext
function environment(T::TensType...) return environment([T...]) end
export environment
```
6. Constructor: makeoc

The orthogonality center must be defined on the lattice. In one of the rare instances where a check is implemented, the makeoc function will check that the orthogonality center is properly defined and return 1 if nothing is defined.

```julia
function makeoc(Ns::Integer, oc::IntType, ...
    return length(oc) > 0 && 0 < oc[1] <= Ns ? oc[1] : 1
end
```

7. Constructor: MPS

This constructor will make the MPS with the orthogonality center optionally defined (default: site 1). The optional regtens will produce a julia defined Array for the output in case this is necessary.

```julia
function MPS(psi::Array{W,1}; regtens::Bool=false, oc::Integer=1,
    type::DataType=eltype(psi[1])) where W <: densTensType
    if eltype(psi[1]) != type && !regtens
        MPSvec = [convertTens(type, copy(B[i])) for i = 1:size(B,1)]
    else
        MPSvec = psi
    end
    return matrixproductstate(MPSvec,oc)
end

function MPS(psi::MPS; regtens::Bool=false, oc::Integer=psi.oc,
    type::DataType=eltype(psi))
    return MPS(psi.A,regtens=regtens,oc=oc,type=type)
end

function MPS(B::Array{W,1}; regtens::Bool=false, oc::Integer=1,
    type::DataType=eltype(B[1])) where W <: AbstractArray
    if !regtens
        MPSvec = [tens(convert(Array{type,ndims(B[i])), copy(B[i])))
            for i = 1:size(B,1)]
    else
        MPSvec = [convert(Array{type,ndims(B[i])),copy(B[i])) for i = 1:size(B,1)]
    end
    return matrixproductstate(typeof(MPSvec))(MPSvec,oc)
end

function MPS(type::DataType, B::Union{MPS, Array{W,1}}; regtens::Bool=false,
    oc::Integer=1) where W <: AbstractArray
    return MPS(B,regtens=regtens,oc=oc,type=type)
end

function MPS(physindvec::Array{W,1}; regtens::Bool=false, oc::Integer=1,
    type::DataType=Float64) where W <: Integer
    Ns = length(physindvec)
    if regtens
        vec = Array{Array{type,3},1}(undef,Ns)
        for w = 1:Ns
            vec[w] = zeros(type,1,physindvec[w],1)
            vec[w][1,1,1] = 1
        end
    else
        vec = Array{tens{type},1}(undef,Ns)
        for w = 1:Ns
```
temp = zeros(type,1,physindvec[w],1)
temp[1,1,1] = 1
vec[w] = tens(temp)
end
end
return MPS(vec,oc=oc)
end

function MPS(physindvec::Array{W,1},Ns::Integer;regtens::Bool=false,
oc::Integer=1,type::DataType=Float64) where W <: Integer
physindvecfull = physindvec[(w-1) % length(physindvec) + 1 for w = 1:Ns]
return MPS(physindvecfull,regtens=regtens,oc=oc,type=type)
end

function MPS(physindsize::Integer,Ns::Integer;regtens::Bool=false,
oc::Integer=1,type::DataType=Float64)
return MPS([physindsize for w = 1:Ns],oc=oc,type=type)
end

function MPS(type::DataType,physindvec::Array{W,1},Ns::Integer;
regtens::Bool=false,oc::Integer=1) where W <: Integer
return MPS(physindvec,Ns,regtens=regtens,oc=oc,type=type)
end

The MPS is often defined to be rank 3 even on the edge tensors. This form of the MPS constructor could allow for rank-2 on the edges (and then some modification to the eventual DMRG function), but the compiler in julia has an easier time if all the tensors are defined uniformly. This should be the case in several other languages as well, so it should be considered to keep the rank-3 on each tensor.

Note that a DataType can be provided in the first argument which will automatically cause the input arrays to be converted. This is useful when using complex numbers as the MPS initially defined here is useful for ensuring consistent operations throughout a computation.

Note that the initialization call of the form

physindsize = 2
Ns = 10
psi = MPS(physindsize,Ns)

generates a matrix product state with physical index size of 2 on each tensor with 10 sites. A vector can also be placed into the first argument with variable physical index sizes. The state can be initialized with applyOps.

A. Function: applyOps & applyOps!

The applyOps function provides a quick interface for applying single site operators to the MPS. This can be especially useful for creating starting wavefunctions from the MPS initialization of the starting MPS.

```julia
function applyOps!(psi::MPS,sites::Array{W,1},Op::TensType;
trail::TensType=ones(1,1)) where W <: Integer
for i = 1:length(sites)
    site = sites[i]
p = site
    psi[p] = contract([2,1,3],Op,2,psi[p],2)
    if trail != ones(1,1)
        for j = 1:p-1
            psi[j] = contract([2,1,3],trail,2,psi[j],2)
        end
    end
end
return psi
```
The MPO can sometimes be useful to define with only matrices (operators) as the constituent tensors. These will be rank-2 on each site and the constructor which reshapes those tensors formats them into the rank-4 tensor (with trivial horizontal indices of size 1) to make the indices match.
Just as with the MPS, the MPO can admit a leading `DataType` that converts another MPO or the tensors themselves to another format.

### B. Function: `randMPS`

The function `randMPS` creates a set of tensors with bond dimension 1 on the link indices and randomly chooses which element of the tensor to be assigned a value of 1 (the rest are zero).

```plaintext
function randMPS(T::DataType,physindsize::Integer,Ns::Integer;oc::Integer=1,
m::Integer=1)
    physindvec = [physindsize for i = 1:Ns]
    return randMPS(T,physindvec,oc=oc,m=m)
end

function randMPS(T::DataType,physindvec::Array{W,1};oc::Integer=1,
m::Integer=1) where W <: Integer
    Ns = length(physindvec)
    vec = Array{Array{T,3},1}(undef,Ns)
    if m == 1
        for w = 1:Ns
            vec[w] = zeros(1,physindvec[w],1)
            state = rand(1:physindvec[w],1)[1]
            vec[w][1,state,1] = 1
        end
        psi = MPS(vec,oc)
    else
        Lsize,Rsize = 1,prod(w->physindvec[w],2:length(physindvec))
        accumsize = physindvec[1]
        for w = 1:Ns
            physindsizew = physindvec[w]
            currRsize = Rsize < accumsize ? min(Rsize,m) : min(accumsize,m)
            vec[w] = rand(T,Lsize,physindsizew,currRsize)
            vec[w] /= norm(vec[w])
            Lsize = currRsize
            Rsize = cld(Rsize,physindsizew)
            accumsize *= physindsizew
        end
        psi = MPS(vec,oc)
        psi[oc] /= expect(psi)
    end
    return psi
end

function randMPS(physindsize::Integer,Ns::Integer;oc::Integer=1,m::Integer=1)
    return randMPS(Float64,physindsize,Ns,oc=oc,m=m)
end

function randMPS(physindvec::Array{W,1};oc::Integer=1,
m::Integer=1) where W <: Integer
    return randMPS(Float64,physindvec,oc=oc,m=m)
end
export randMPS
```

1. *Elementary functions: elnumtype, eltype, size, length, lastindex, copy, conj, conj! and setindex!*

The elementary functions that were defined for the tensor itself are now defined for the holder types here.
function elnumtype(op...)
    opnum = eltype(op[1])(1)
    for b = 2:length(op)
        opnum *= eltype(op[b])(1)
    end
    return typeof(opnum)
end

function size(H::MPO)
    return size(H.H)
end

function size(H::MPO,i::Integer)
    return size(H.H,i)
end

function size(psi::MPS)
    return size(psi.A)
end

function size(psi::MPS,i::Integer)
    return size(psi.A,i)
end

function size(G::regEnv)
    return size(G.V)
end

function size(G::regEnv,i::Integer)
    return size(G.V,i)
end

function length(H::MPO)
    return length(H.H)
end

function length(psi::MPS)
    return length(psi.A)
end

function length(G::regEnv)
    return length(G.V)
end

function eltype(Y::regMPS)
    return eltype(Y.A[1])
end

function eltype(H::regMPO)
    return eltype(H.H[1])
end

function eltype(G::regEnv)
    return eltype(G.V[1])
end

function getindex(A::regMPS,i::Integer)
return A.A[i]
end

function getindex(A::regMPS,r::UnitRange{W}) where W <: Integer
    if A.oc in r
        newoc = findfirst(w->w == A.oc,r)
    else
        newoc = 0
    end
    return MPS(A.A[r],oc=newoc)
end

function getindex(H::regMPO,i::Integer)
    return H.H[i]
end

function getindex(H::regMPO,r::UnitRange{W}) where W <: Integer
    return MPO(H.H[r])
end

function getindex(G::regEnv,i::Integer)
    return G.V[i]
end

function getindex(G::regEnv,r::UnitRange{W}) where W <: Integer
    return environment(G.V[r])
end

function lastindex(A::regMPS)
    return lastindex(A.A)
end

function lastindex(H::regMPO)
    return lastindex(H.H)
end

function lastindex(G::regEnv)
    return lastindex(G.V)
end

function setindex!(H::regMPO,A::TensType,i::intType)
    H.H[i] = A
    nothing
end

function setindex!(H::regMPS,A::TensType,i::intType)
    H.A[i] = A
    nothing
end

function setindex!(G::regEnv,A::TensType,i::intType)
    G.V[i] = A
    nothing
end

function copy(mps::matrixproductstate{W}) where W <: TensType
    return matrixproductstate{W}([copy(mps.A[i]) for i = 1:length(mps)],
                                  oc=copy(mps.oc))
function copy(mpo::MatrixProductOperator{W}) where W <: TensType
    return MatrixProductOperator{W}([copy(mpo.H[i]) for i = 1:length(mpo)])
end

function copy(mps::RegMPS)
    return MPS([copy(mps.A[i]) for i = 1:length(mps)],
                oc=copy(mps.oc))
end

function copy(mpo::RegMPO)
    return MPO([copy(mpo.H[i]) for i = 1:length(mpo)])
end

function copy(G::RegEnv)
    T = eltype(G[1])
    return envVec{T}([copy(G.V[i]) for i = 1:length(G)])
end

function conj!(A::RegMPS)
    conj!.(A.A)
    return A
end

function conj(A::RegMPS)
    B = copy(A)
    conj!.(B.A)
    return B
end

The `length` function returns the number of sites for a given type. Note also that `elnumtype` returns the type of number stored in a tensor of the MPS, MPO, or environment. This can be useful in some situations and extends the definition in `tensor.jl`.

2. Types: `largeMPS`, `largeMPO`, and `largeEnv`

Storing large matrices in memory, purely, will cause a difficulty in that only so much memory is available on a computer. For a truly large tensor, writing it to the disk is useful. Fortunately, Julia has an internal writing protocol to write tensors on the disk. The package, native to the basic functions in Julia, is called `Serialization` and makes this process easy.

The abstract types that must be defined are extensions of the previous `Reg` types defined above. Each of `largeMPS`, `largeMPO`, and `largeEnv`.

```julia
abstract type largeMPS <: MPS end
export largeMPS

abstract type largeMPO <: MPO end
export largeMPO

abstract type largeEnv <: envType end
export largeEnv
```

When writing the data to the disk, it is often useful to have an extension. In this case, “.dmrjulia” has been chosen.

```julia
const file_extension = " .dmrjulia"
```
3. **Structs: largematrixproductstate, largematrixproductoperator, and largeenvironment**

Each of `largematrixproductstate`, `largematrixproductoperator`, and `largeenvironment` will store a string denoting a file name and a data type that records what type of data is stored in the tensor. This second piece of information is only to avoid pulling the tensor from memory when the type is requested.

The large structs are made with the following constructors. Note the use of the `tensor2disc` to write the tensors. All three constructors follow the same basic pattern: 1) tensor is written to disk, 2) type of saved for later, 3) file names are saved. The environment constructor requires both the left and right environments to be made.

```julia
mutable struct largematrixproductstate <: largeMPS
    A::Array{String,1}
    oc::IntType
    type::DataType
end

mutable struct largematrixproductoperator <: largeMPO
    H::Array{String,1}
    type::DataType
end

mutable struct largeenvironment <: largeEnv
    V::Array{String,1}
    type::DataType
end
```

The MPS also stores the orthogonality center just as the regular version did.

4. **Structs: tensor2disc & tensorfromdisc**

Writing tensors to and from the disk is accomplished here using the `Serialization` package that is internal to Julia itself.

```julia
import Serialization

function tensor2disc(name::String,tensor::TensType; ext::String=file_extension)
    Serialization.serialize(name*ext,tensor)
    nothing
end

function tensorfromdisc(name::String; ext::String=file_extension)
    return Serialization.deserialize(name*ext)
end
```

When using this function, a set of files with extension “.dmrjulia” (default) will appear.

C. **Function: largeMPS, largeMPO, largeEnv, largeRenv and largeEnv**

Constructor for the large tensor types (i.e., stored on disk) containing MPS, MPO, and environment tensors.

```julia
function largeMPS(psi::MPS; label::String="mps_",
    names::Array{String,1}=[label*"$i" for i = 1:length(psi)],
    ext::String=file_extension)
    lastnum = 1
    for b = 1:length(psi)
        C = psi[b]
        tensor2disc(names[b],C,ext=ext)
        lastnum *= eltype(C)(1)
    end
```
return largematrixproductstate(names, psi.oc, typeof(lastnum))
end

function largeMPS(Ns::Integer; label::String = "mps_",
    names::Array{String,1}=[label*"$i" for i = 1:Ns],
    ext::String=file_extension, thisoc::Integer=1, type::DataType=Float64)
    return largematrixproductstate(names, thisoc, type)
end

function largeMPS(type::DataType, Ns::Integer; label::String = "mps_",
    names::Array{String,1}=[label*"$i" for i = 1:length(mpo)],
    ext::String=file_extension, thisoc::Integer=1)
    return largeMPS(Ns, label=label, names=names, ext=ext, thisoc=thisoc, type=type)
end

function largeMPO(mpo::P; label::String = "mpo_",
    names::Array{String,1}=[label*"$i" for i = 1:length(mpo)],
    ext::String=file_extension) where P <: Union{AbstractArray, MPO}
    lastnum = 1
    for b = 1:length(mpo)
        C = mpo[b]
        tensor2disc(names[b], C, ext=ext)
        lastnum *= eltype(C)(1)
    end
    return largematrixproductoperator(names, typeof(lastnum))
end

function largeMPO(Ns::Integer; label::String = "mpo_",
    names::Array{String,1}=[label*"$i" for i = 1:Ns],
    ext::String=file_extension)
    return largematrixproductoperator(names, type)
end

function largeMPO(type::DataType, Ns::Integer; label::String = "mpo_",
    names::Array{String,1}=[label*"$i" for i = 1:Ns],
    ext::String=file_extension)
    return largeMPO(Ns, label=label, names=names, ext=ext, type=type)
end

function largeLenv(lenv::P; label::String = "Lenv_",
    names::Array{String,1}=[label*"$i" for i = 1:length(lenv)],
    ext::String=file_extension) where P <: Union{AbstractArray, MPO}
    lastnum = 1
    for b = 1:length(lenv)
        C = env[b]
        tensor2disc(names[b], C, ext=ext)
        lastnum *= eltype(C)(1)
    end
    return largeenvironment(names, typeof(lastnum))
end

function largeLenv(Ns::Integer; label::String = "Lenv_",
    names::Array{String,1}=[label*"$i" for i = 1:Ns],
    ext::String=file_extension, type::DataType=Float64)
    return largeenvironment(names, type)
end

function largeLenv(type::DataType, Ns::Integer; label::String = "Lenv_",
    names::Array{String,1}=[label*"$i" for i = 1:Ns],
    ext::String=file_extension)
    return largeenvironment(names, type)
names::Array{String,1}=[label*$i$ for i = 1:Ns], ext::String=file_extension) return largeLenv(Ns,label=label,names=names,ext=ext,type=type) end export largeLenv

function largeRenv(renv::P,label::String="Renv_", names::Array{String,1}=[label*$i$ for i = 1:length(renv)], ext::String=file_extension) where P <: Union{AbstractArray,MPO} return largeLenv(renv,label=label,names=names,ext=ext) end export largeRenv

function largeRenv(Ns::Integer,label::String="Renv_", names::Array{String,1}=[label*$i$ for i = 1:Ns], ext::String=file_extension,type::DataType=Float64) return largeenvironment(names,type) end export largeRenv

function largeEnv(lenv::P,renv::P;Llabel::String="Lenv_", Rlabel::String="Renv_", Lnames::Array{String,1}=[Llabel*$i$ for i = 1:Ns], Rnames::Array{String,1}=[Rlabel*$i$ for i = 1:Ns], ext::String=file_extension, type::DataType=Float64) where P <: Union{AbstractArray,MPO} return largeLenv(lenv,names=Lnames,type=type), largeRenv(renv,names=Rnames,type=type) end export largeEnv

1. Elementary functions: getindex, setindex!, lastindex, length, and eltype

The standard elementary functions getindex, setindex!, lastindex, length, and eltype that were used for tensor.jl earlier are defined here. The type field is used to avoid reading the tensor from disk in eltype.
function getindex(A::largeMPS,i::Integer)
    return tensorfromdisc(A.A[i])
end

function getindex(A::largeMPO,i::Integer)
    return tensorfromdisc(A.H[i])
end

function getindex(A::largeEnv,i::Integer)
    return tensorfromdisc(A.V[i])
end

function setindex!(H::largeMPS,A::TensType,i::IntType;
                    ext :: String = file_extension)
    tensor2disc(H.A[i],A,ext=ext)
    nothing
end

function setindex!(H::largeMPO,A::TensType,i::IntType;
                   ext :: String = file_extension)
    tensor2disc(H.H[i],A,ext=ext)
    nothing
end

function setindex!(H::largeEnv,A::TensType,i::IntType;
                   ext :: String = file_extension)
    tensor2disc(H.V[i],A,ext=ext)
    nothing
end

function lastindex(A::largeMPS; ext :: String = file_extension)
    return tensorfromdisc(A.A[end],ext=ext)
end

function lastindex(H::largeMPO; ext :: String = file_extension)
    return tensorfromdisc(H.H[end],ext=ext)
end

function lastindex(H::largeEnv; ext :: String = file_extension)
    return tensorfromdisc(H.V[end],ext=ext)
end

function length(A::largeMPS)
    return length(A.A)
end

function length(H::largeMPO)
    return length(H.H)
end

function length(H::largeEnv)
    return length(H.V)
end

function eltype(op::largeType)
    return op.type
end
Note a critical feature of this construction. When using the `getindex` or `setindex` commands, the tensor is automatically read from the disk. This is a major time saver when writing algorithms. The same syntax (i.e., `psi[1]`) can be used, avoiding the need to recode each function. These functions can be used with any tensor type in the library.

D. Function: Load types `loadMPS`, `loadMPO`, `loadLenv`, and `loadRenv`

Each of these functions loads a given MPS, MPO, or environment from the disk. All tensors must be saved on the hard disk in the correct filepath (determined by the `label` and `names` fields) before using these functions. All tensors are assumed to be of a uniform type and that the file extensions are the same between each object as a default. The only required input to these functions is the number of sites.

```plaintext
function loadMPS(Ns :: Integer; label :: String = "mps_", names :: Array{String,1} = [label * "$i" for i = 1:Ns], ext :: String = file_extension, oc :: Integer = 0)  
lastnum = 1  
storeoc = [1.]  
for i = 1:Ns  
   name = names[i]  
   A = tensorfromdisc(name, ext=ext)  
   lastnum *= eltype(A)(1)  
end  
lastnum = 1  
for i = 1:Ns  
   name = names[i]  
   A = tensorfromdisc(name, ext=ext)  
   lastnum *= eltype(A)(1)  
end  
return largematrixproductstate(names, thistype)  
end  
export loadMPS

function loadMPO(Ns :: Integer; label :: String = "mpo_", names :: Array{String,1} = [label * "$i" for i = 1:Ns], ext :: String = file_extension, oc :: Integer = 0)  
lastnum = 1  
for i = 1:Ns  
   name = names[i]  
   A = tensorfromdisc(name, ext=ext)  
   lastnum *= eltype(A)(1)  
end  
lastnum = 1  
for i = 1:Ns  
   name = names[i]  
   A = tensorfromdisc(name, ext=ext)  
   lastnum *= eltype(A)(1)  
end  
return largematrixproductoperator(names, thistype)  
end  
export loadMPO

function loadLenv(Ns :: Integer; label :: String = "Lenv_", names :: Array{String,1} = [label * "$i" for i = 1:Ns], ext :: String = file_extension, oc :: Integer = 0)  
lastnum = 1  
for i = 1:Ns  
   name = names[i]  
   A = tensorfromdisc(name, ext=ext)  
   lastnum *= eltype(A)(1)  
end  
lastnum = 1  
for i = 1:Ns  
   name = names[i]  
   A = tensorfromdisc(name, ext=ext)  
   lastnum *= eltype(A)(1)  
end  
return largematrixproductoperator(names, thistype)  
end  
export loadLenv
```
thistype = typeof(lastnum)
return largeenvironment(names, thistype)
end
export loadLenv

function loadRenv(Ns :: Integer; label :: String="Renv_",
names :: Array{String,1}=[label*"$i" for i = 1:Ns],
ext :: String=file_extension, oc :: Integer=0)
lastnum = 1
for i = 1:Ns
name = names[i]
A = tensorfromdisc(name, ext=ext)
lastnum *= eltype(A)(1)
end
thistype = typeof(lastnum)
return largeenvironment(names, thistype)
end
export loadRenv

function loadEnv(Ns :: Integer; Llabel :: String="Lenv_", Rlabel :: String="Renv_",
Lnames :: Array{String,1}=[Llabel*"$i" for i = 1:Ns],
Rnames :: Array{String,1}=[Rlabel*"$i" for i = 1:Ns],
ext :: String=file_extension, oc :: Integer=0)
lastnumL = 1
lastnumR = 1
for i = 1:Ns
name = Lnames[i]
A = tensorfromdisc(name, ext=ext)
lastnumL *= eltype(A)(1)
B = tensorfromdisc(name, ext=ext)
lastnumR *= eltype(B)(1)
end
thistypeL = typeof(lastnumL)
thistypeR = typeof(lastnumR)
return largeenvironment(Lnames, thistypeL), largeenvironment(Rnames, thistypeR)
end
export loadEnv

E. Function: copy (large types)

Copying objects for the large tensor types requires that the tensor be read from disk and then given a new filename
to place the new object. Hence, each of these files take a new file name as an argument.
This copy operation for large types is very different from the same operation for dense types. This means that an
algorithm that requires this operation will need to take this into account if designed for large types. However, most
of the algorithms work equally well with this interface.

function copy(names :: Array{String,1}, X :: largeMPS; ext :: String=file_extension,
copyext :: String=ext)
newObj = copy(X)
newObj.A = names
for i = 1:length(X)
Y = tensorfromdisc(names[i], ext=ext)
tensor2disc(X.A[i], Y, ext=copyext)
end
return newObj
end
F. MPS gauges: movement of the orthogonality center

As explained in Ref. 3, the MPS can be re-gauged so that the center of orthogonality is contained on another site. These functions will perform the necessary decompositions to change the gauge of the MPS.

1. Function: moveR & moveR!

This function will perform a decomposition on the rank-3 tensor of the MPS and move the orthogonality center one site to the right. Note that there are two applicable decompositions, qr and svd. The svd admits truncation so it is used in general. However, the movement of the orthogonality center often does not need to be truncated as is the case in the bulk of the MPS. So, when the size of the newly generated index is guaranteed not to be truncated, the qr decomposition is used. When using the svd, an extra contraction of D into the new tensor must be performed. Conversely, the qr decomposition requires no extra contraction.
There is also the possibility to update the tensor in-place on the MPS's elements with the `moveR!` function.

``` Julio
function moveR!(psi::MPS; cutoff::Float64=0., m::Integer=0, minm::Integer=0,
    condition::Bool=false, recursive::Bool=false, mag::Number=0.)
    il = psi.oc
    psi[il], psi[il+1], D, truncerr = moveR(psi[il], psi[il+1], cutoff=cutoff, m=m,
        minm=minm, condition=condition, recursive=recursive, mag=mag)
    psi.oc += 1
    return D, truncerr
end
export moveR!
```

2. Function: `moveL` & `moveL!`

The same rationale for the right move will equally apply to the left movement. However, instead of the qr decomposition for the non-truncating operation, the lq decomposition is used.

``` Julio
function moveL(Lpsi::TensType, Rpsi::TensType; cutoff::Float64=0., m::Integer=0,
    minm::Integer=0, condition::Bool=false, recursive::Bool=false, mag::Number=0.,
    fast::Bool=false)
    if (size(Rpsi,1) <= m) && !isapprox(cutoff,0.) && fast
        modU , Rtens = lq(Rpsi,[[1],[2,3]])
        UD = (condition ? getindex!(modU,1:size(Lpsi,3),:) : modU)
        D = UD
        truncerr = 0.
    else
        U,D, Rtens, truncerr, sumD = svd(Rpsi,[[1],[2,3]], cutoff=cutoff, m=m, minm=minm,
            recursive=recursive, mag=mag)
        modU = (condition ? getindex!(U,1:size(Lpsi,3),:) : U)
        UD = contract(modU,[2], D,[1])
    end
    Ltens = contract(Lpsi,3,UD,1)
    return Ltens, Rtens, D, truncerr
end
export moveL
```

There is also the possibility to update the tensor in-place on the MPS's elements with the `moveL!` function.

``` Julio
function moveL!(psi::MPS; cutoff::Float64=0., m::Integer=0, minm::Integer=0,
    condition::Bool=false, recursive::Bool=false, mag::Number=0.)
    iR = psi.oc
    psi[iR-1], psi[iR], D, truncerr = moveL(psi[iR-1], psi[iR], cutoff=cutoff, m=m,
        minm=minm, condition=condition, recursive=recursive, mag=mag)
    psi.oc -= 1
    return D, truncerr
end
export moveL!
```

3. Function: `movecenter!`

The combination of `moveR` and `moveL` can be combined into a single function `movecenter!` that when given a new orthogonality center will move the center until the new orthogonality center is obtained.
function movecenter!(psi::MPS, pos::Integer; cutoff::Float64=1E-14, m::Integer=0, minm::Integer=0, Lfct::Function=moveR, Rfct::Function=moveL)
    if m == 0
        m = maximum([maximum(size(psi[i])) for i = 1:size(psi,1)])
    end
    while psi.oc != pos
        if psi.oc < pos
            iL = psi.oc
            iR = psi.oc+1
            psi[iL], psi[iR], D, truncerr = Lfct(psi[iL], psi[iR], cutoff=cutoff, m=m, minm=minm, fast=true)
            psi.oc = iR
        else
            iL = psi.oc-1
            iR = psi.oc
            psi[iL], psi[iR], D, truncerr = Rfct(psi[iL], psi[iR], cutoff=cutoff, m=m, minm=minm, fast=true)
            psi.oc = iL
        end
    end
    nothing
end

This function is defined since the main interface function that the user sees can be named more simply. However, that function can be amended with another core function (like movecenter!) if another type of wavefunction ansatz is gauged and must be moved.

4. Function: move! & move

The move! function is the function that the user will use. The core operation of moving the orthogonality center can be switched out for another function if necessary. The tensors are updated in-place.

function move!(mps::MPS, pos::Integer; m::Integer=0, cutoff::Float64=1E-14, minm::Integer=0)
    movecenter!(mps, pos, cutoff=cutoff, m=m, minm=minm)
    nothing
end
export move!

function move(mps::MPS, pos::Integer; m::Integer=0, cutoff::Float64=1E-14, minm::Integer=0)
    newmps = copy(mps)
    movecenter!(newmps, pos, cutoff=cutoff, m=m, minm=minm)
    return newmps
end
export move

The move function simply copies the input MPS and returns a completely new MPS.

G. Function: leftnormalize, leftnormalize!, rightnormalize, & rightnormalize!

The leftnormalize and rightnormalize function generate the left- or right- normalization of the MPS [3]. That is, all the tensors are left as the \( \hat{U} \) or \( \hat{V}^\dagger \) tensors simply. The other elements returned are the \( \hat{D} \) matrix and the last \( \hat{V}^\dagger \) tensors for the case of the left-normalization. The right-normalization returns also the \( \hat{U} \) and \( \hat{D} \) tensors. The orthogonality center is set to zero after performing this operation.
```
function leftnormalize(psi::MPS)
    newpsi = move(psi, length(psi))
    U, D, V = svd(psi[end], [[1, 2], [3]])
    newpsi[end] = U
    newpsi.oc = 0
    return newpsi, D, V
end
export leftnormalize

function leftnormalize!(psi::MPS)
    move!(psi, length(psi))
    U, D, V = svd(psi[end], [[1, 2], [3]])
    psi[end] = U
    psi.oc = 0
    return D, V
end
export leftnormalize!

function rightnormalize(psi::MPS)
    newpsi = move(psi, 1)
    U, D, V = svd(psi[1], [[1], [2, 3]])
    newpsi[1] = V
    newpsi.oc = 0
    return U, D, newpsi
end
export rightnormalize

function rightnormalize!(psi::MPS)
    psi = move!(psi, 1)
    U, D, V = svd(psi[1], [[1], [2, 3]])
    psi[1] = V
    psi.oc = 0
    return U, D
end
export rightnormalize!
```

1. **Function: boundarymove! & boundarymove**

Sometimes it is useful to move the MPS's orthogonality center and then simultaneously update the environment tensors at the same time. This function `boundarymove!` will do so and manipulate tensors in place.

```
function boundaryMove!(psi::MPS, i::Integer, Lenv::Env,
                      Renv::Env, mpo::MPO...; mover::Function=move!)
    origoc = psi.oc
    if origoc < i
        mover(psi, i)
        for w = origoc:i-1
            Lupdate!(w, Lenv, psi, mpo...)
        end
    elseif origoc > i
        mover(psi, i)
        for w = origoc:-1:i+1
            Rupdate!(w, Renv, psi, mpo...)
        end
    end
end
nothing
```
function boundaryMove!(dualpsi::MPS, psi::MPS, i::Integer, Lenv::Env, Renv::Env, mpo::MPO...) ; mover::Function = move!)

origoc = psi.oc
if origoc < i
    mover(psi, i)
mover(dualpsi, i)
    for w = origoc:i-1
        Lenv[w+1] = Lupdate(Lenv[w], dualpsi[w], psi[w], [mpo[a][w] for a = 1:length(mpo)]...)
    end
elseif origoc > i
    mover(psi, i)
mover(dualpsi, i)
    for w = origoc:-1:i+1
        Renv[w-1] = Rupdate(Renv[w], dualpsi[w], psi[w], [mpo[a][w] for a = 1:length(mpo)]...)
    end
end
nothing
end

export boundaryMove

A second function boundaryMove will copy the input tensor and output a copy of the tensor, leaving the original unchanged.

function boundaryMove(dualpsi::MPS, psi::MPS, i::Integer, mpo::MPO, Lenv::Env, Renv::Env)
    newpsi = copy(psi)
    newdualpsi = copy(dualpsi)
    newLenv = copy(Lenv)
    newRenv = copy(Renv)
    boundaryMove!(newdualpsi, newpsi, i, mpo, newLenv, newRenv)
    return newdualpsi, newpsi, newLenv, newRenv
end

function boundaryMove(psi::MPS, i::Integer, mpo::MPO, Lenv::Env, Renv::Env)
    newpsi = copy(psi)
    newLenv = copy(Lenv)
    newRenv = copy(Renv)
    boundaryMove!(newpsi, newpsi, i, mpo, newLenv, newRenv)
    return newpsi, newLenv, newRenv
end

export boundaryMove

H. Updating and making environment tensors

DMRjulia uses a specific form for the environment tensors. The indices are defined for the left and right environments according to the diagram in Ref. 3. This will play a crucial role in several algorithms from here on out.

1. Function: Lupdate & Lupdate!

The function Lupdate will update the environment tensors in the environment according to the convention in the figure above.
The input for `Lupdate!` can be defined with an explicitly defined `dualpsi` (\(\langle \psi \rangle\), the dual of the wavefunction) or the wavefunction only (\(| \psi \rangle\)). If only the wavefunction is defined, then it is used as the dual wavefunction. Note that these functions admit any number of MPOs.

2. Function: `Rupdate & Rupdate!`

The function `Rupdate` will update the environment tensors in the environment according to the convention in the figure above.

The conventions are listed above and the functions perform similarly to the `Lupdate` functions.
3. **Function: makeBoundary**

This will make the leftmost or right most edge tensor in the network for the environment tensors.

```plaintext
function makeBoundary(dualpsi::MPS, psi::MPS, mpovec::MPO...;
  left::Bool=true, rightind::Integer=3)
  retType = elnumtype(dualpsi, psi, mpovec...)
  nrank = 2 + length(mpovec)
  boundary = ones(retType, ones(intType, nrank)...)
  if typeof(psi[1]) <: denstens
    return tens(boundary)
  else
    return boundary
  end
end
export makeBoundary
```

4. **Function: makeEnds**

This function generates the edge tensors in the tensor network for the matrix product state and however many matrix product operators are in use.

```plaintext
function makeEnds(dualpsi::MPS, psi::MPS, mpovec::MPO...;
  Lbound::TensType=typeof(psi[1])(), Rbound::TensType=typeof(psi[end])(())
  if abs(norm(Lbound)) <= 1E-13
    Lout = makeBoundary(dualpsi, psi, mpovec...)
  else
    Lout = copy(Lbound)
  end
  if abs(norm(Rbound)) <= 1E-13
    Rout = makeBoundary(dualpsi, psi, mpovec..., left=false)
  else
    Rout = copy(Rbound)
  end
  return Lout, Rout
end

function makeEnds(psi::MPS, mpovec::MPO...;
  Lbound::TensType=typeof(psi[1])(), Rbound::TensType=typeof(psi[1])())
  return makeEnds(psi, psi, mpovec..., Lbound=Lbound, Rbound=Rbound)
end
export makeEnds
```

5. **Function: makeEnv**

Using the `makeEnds` function to generate the edge tensors, this function will also perform all contractions between those tensors and the current orthogonality center of the input MPS. This will generate the environment tensors for the system.

```plaintext
function makeEnv(dualpsi::MPS, psi::MPS, mpo::MPO...;
  Lbound::TensType=typeof(psi[1])(), Rbound::TensType=typeof(psi[1])())
  Ns = length(psi)
  numtype = elnumtype(dualpsi, psi, mpo...)
  C = psi[1]
```
if typeof(psi) <= largeMPS || typeof(mpo) <= largeMPO
    Lenv, Renv = largeLRenv(numtype, Ns)
else
    Lenv = environment(psi[1], Ns)
    Renv = environment(psi[1], Ns)
end
Lenv[1], Renv[Ns] = makeEnds(dualpsi, psi, mpo..., Lbound=Lbound, Rbound=Rbound)

for i = Ns:-1:psi.oC+1
    Rupdate!(i, Renv, dualpsi, psi, mpo...)
end
for i = 1:psi.oC-1
    Lupdate!(i, Lenv, dualpsi, psi, mpo...)
end
return Lenv, Renv
end

function makeEnv(psi::MPS, mpo::MPO; Lbound::TensType=[0], Rbound::TensType=[0])
    return makeEnv(psi, psi, mpo, Lbound=Lbound, Rbound=Rbound)
end
export makeEnv

I. Measurements and related functions

Some of the most compact and yet most versatile functions will be defined here. The \texttt{expect} function will evaluate any correlation function provided the MPS and any number of MPOs. Similarly, the correlation function evaluates any \textit{N}-point function that is input. There are efficient algorithms for performing these operations and are described in the following.

1. Function: applyMPO

The application of the MPO to the MPS can be accomplished locally. The diagram in Ref. 3 conveys the algorithm used. The MPO tensor is contracted onto the MPS. Then, the resulting tensor is decomposed according to the SVD and the result is contracted onto the next tensor.

function applyMPO(psi::MPS, H::MPO; m::Integer=0, cutoff::Float64=0.)
    if m == 0
        m = maximum([size(psi[i], ndims(psi[i])) for i = 1:size(psi.A,1)])
    end
    thissize = size(psi,1)
    newpsi = [contract([1,3,4,2,5], psi[i], 2, H[i], 2) for i = 1:thissize]
    finalpsi = Array{typeof(psi[1]),1}(undef, thissize)
    finalpsi[thissize] = reshape!(newpsi[thissize], [[1], [2], [3], [4, 5]], merge=true)
    for i = thissize:-1:2
        currTens = finalpsi[i]
        newsize = size(currTens)
        temp = reshape!(currTens, [[1, 2], [3, 4]])
        U, D, V = svd(temp, m = m, cutoff = cutoff)
        finalpsi[i] = reshape!(V, size(D, 1), newsize[3], newsize[4], merge=true)
        tempT = contract(U, 2, D, 1)
One can use this to contract the MPO onto the MPS, making a resulting that that is $|\mathcal{H}\psi\rangle$. However, this is ultimately inefficient for making measurements. By using the locality of the MPS, the cost can be brought down considerably.

2. Function: `expect`

The `expect` function evaluates any correlation of the form $\langle \psi | H^p | \psi \rangle$ where $p \geq 0$ and is an integer, the number of MPOs input. Using a self-similar pattern that emerges in the tensors, the function can be programmed to be very short and concise. The particular pattern is displayed in Ref. 3.

Tracing over the tensors using this pattern results in the best scaling computation to produce the end result.

```plaintext
function expect(dualpsi::MPS, psi::MPS, H::MPO...;
    Lbound::TensType=typeof(psi[1])(),
    Rbound::TensType=Lbound, order::IntVecType=IntType[])
    Ns = size(psi, 1)
    nMPOs = size(H, 1)
    nLsize = nMPOs + 2
    Lenv, Renv = makeEnds(dualpsi, psi, H..., Lbound=Lbound, Rbound=Rbound)

    for i = 1:length(psi)
        Lenv = contractc(Lenv, 1, dualpsi[i], 1)
        for j = 1:nMPOs
            Lenv = contract(Lenv, [1, nLsize], H[j][i], [1, 3])
        end
        Lenv = contract(Lenv, [1, nLsize], psi[i], [1, 2])
    end

    if order == IntType[
        permvec = vcat([ndims(Renv)], [i for i = 2:ndims(Renv)-1], [1])
        modRenv = permutedims(Renv, permvec)
    else
        modRenv = Renv
    end

    return contract(Lenv, modRenv)
end

export expect
```

The additional function with only one input MPS is for ease of programming.

3. Function: `correlationmatrix`

For the special case of two-point correlation functions of the form, for example, $\langle \psi | \hat{c}_i^{\dagger} \hat{c}_j | \psi \rangle$, the locality of the orthogonality center can be used to make efficient operations to make the resulting quantities.
function correlationmatrix(dualpsi::MPS, psi::MPS, mCc::TensType, mCa::TensType, F::TensType..., silent::Bool = true)
    if typeof(mCc) <: Array && eltype(psi) <: denstens
        Cc = tens(mCc)
    else
        Cc = mCc
    end
    if typeof(mCa) <: Array && eltype(psi) <: denstens
        Ca = tens(mCa)
    else
        Ca = mCa
    end
    rho = Array{eltype(psi[1]),2}(undef, size(psi,1), size(psi,1))
    onsite = contract(Cc,2,Ca,1)
    if size(F,1) != 0
        FCC = contract(Cc,2,F[1],1)
    else
        FCC = Cc
    end
    diffTensors = !(psi == dualpsi)
    for i = 1:size(psi,1)
        move!(psi,i)
        if diffTensors
            move!(dualpsi,i)
        end
        TopTerm = contract([1,3,2], psi[i], [2], onsite, [1])
        rho[i,i] = contractc(TopTerm, dualpsi[i])
    end
    for i = 1:size(psi,1)-1
        move!(psi,i)
        if diffTensors
            move!(dualpsi,i)
        end
        TopTerm = contract(psi[i],[2],FCC,[1])
        Lenv = contractc(TopTerm,[1,3], dualpsi[i],[1,2])
        for j = i+1:size(psi,1)
            Renv = contract(psi[j],[2],Ca,[1])
            Renv = contractc(Renv,[2,3], dualpsi[j],[3,2])
            DMElement = contract(Lenv,Renv)
            if j < size(psi,1)
                if size(F,1) != 0
                    Lenv = contract(Lenv,1,psi[j],1)
                    Lenv = contract(Lenv,2,F[1],1)
                    Lenv = contractc(Lenv, [1,3], dualpsi[j],[1,2])
                else
                    Lenv = contract(Lenv,1,psi[j],1)
                    Lenv = contractc(Lenv, [1,2], dualpsi[j],[1,2])
                end
            end
            rho[i,j] = DMElement
            rho[j,i] = conj(DMElement)
        end
        if !silent
            println("Printing element: ",i," , ",j," ",DMElement)
        end
    end
end
return rho
function correlationmatrix(psi::MPS, Cc::TensType, Ca::TensType, F::TensType...; silent::Bool = true)
    return correlationmatrix(psi, psi, Cc, Ca, F...; silent=silent)
end
export correlationmatrix

A key point is to use the orthogonality of the MPS to recycle the left or right environment to keep the computational cost down.

4. Function: operator_in_order!

This function aids the computation of the full correlation function calculator. This function modifies a vector of any size with input sizes representing how many lattice points receive a particular operator. The return value is a vector incremented in the last position. This allows for the evaluation of all operators in order on the lattice. For example, a four-operator correlation function on the indices \( i, j, k, \) and \( \ell \) sites such that \( i \leq j \leq k \leq \ell \) and that the environments are recycled at each step.

function operator_in_order!(pos::Array{G,1}, sizes::IntVecType) where G <: Integer
    w = length(pos)
    pos[w] += 1
    while w > 1 && pos[w] > sizes[w]
        w -= 1
        @inbounds pos[w] += 1
        @simd for x = w:length(pos)-1
            @inbounds pos[x+1] = pos[x]
        end
    end
    nothing
end

5. Function: permutations

This is the standard heap algorithm for obtaining permutations of a vector. The original input in the main correlation function is \([i \text{ for } i = 1:nrank]\) for a rank \(nrank\) tensor [1].

function permutations(nelem::Integer)
    vec = [i for i = 1:nelem]
    numvecs = factorial(nelem)
    storevecs = Array{Array{intType,1},1}(undef, numvecs)
    saveind = zeros(intType, nelem)
    i = 0
    counter = 1
    storevecs[1] = copy(vec)
    while i < nelem
        if saveind[i+1] < i
            if i % 2 == 0
                a, b = 0, i
            else
                a, b = saveind[i+1], i
            end
            vec[a+1], vec[b+1] = vec[b+1], vec[a+1]
        end
        counter += 1
    end
end
storevecs[counter] = copy(vec)

    saveind[i+1] += 1
    i = 0
else
    saveind[i+1] = 0
    i += 1
end
end
return storevecs
end

6. Function: correlation

This function evaluates any correlation function with any number of operators. For example, \(\langle \hat{c}_i \hat{c}_j \hat{c}_k \hat{c}_\ell \rangle\). The operation is kept efficient by recycling the environment tensors for a given order of indices, for example \(i \leq j \leq k \leq \ell\). Then, the permutations function is used to change the order and the computation is repeated. There is no assumed symmetry in the resulting tensor output. For a symmetric evaluation of a two-operator expression, see correlationmatrix.

```plaintext
function correlation(dualpsi::MPS, psi::MPS, inputoperators::S...;
    sites::IntvecType=ntuple(i->1:length(psi),length(inputoperators)),
    trail::Tuple=()) where S <: Union{Array{TensType,1},TensType}

operators = Array{ Array{ typeof(psi[1]),1 }, 1}(undef,length(inputoperators))
lenghthops = Array{ IntType, 1}(undef,length(operators))
for k = 1:length(operators)
    if eltype(inputoperators[k]) <: TensType
        operators[k] = inputoperators[k]
        lengthops[k] = length(operators[k])
    else
        operators[k] = [inputoperators[k]]
        lengthops[k] = 1
    end
end

Ns = length(psi)
maxOplength = maximum(lengthops)
retType = typeof(eltype(dualpsi[1])(1) * eltype(psi[1])(1) *
    prod(w->prod(a->eltype(operators[w][a])(1),1:lengthops[w]),
    1:length(operators)))

base_sizes = [length(sites[i]) - (lengthops[i] - 1) for i = 1:length(sites)]
omega = Array{retType,length(operators)}(undef,base_sizes...)
perm = permutations(length(operators))

move!(psi,1)
move!(dualpsi,1)

Lenv,Renv = makeEnv(dualpsi,psi)
for b = 1:length(Renv)
    Renv[b] = permutedims(Renv[b],[2,1])
end
isId = [true for r = 1:length(operators)]```

```
if length(trail) > 0
    for r = 1:length(isId)
        index = 0
        while isId[r]
            index += 1
            isId[r] = searchindex(bigtrail,index,index) == 1
        end
    end
end
for i = 1:length(perm)
    order = perm[i]

    base_pos = ones(intType,length(operators))

    pos = [sites[1][1] for i = 1:length(operators)]
    prevpos = [sites[1][1] for i = 1:length(operators)]

    while sum(base_sizes - pos) >= 0
        startsite = 1
        while startsite < length(pos) && pos[startsite] == prevpos[startsite]
            startsite += 1
        end
        while startsite > 1 && pos[startsite-1] == prevpos[startsite]
            startsite -= 1
        end
        beginsite = prevpos[startsite]
        finalsite = pos[end]

        thisLenv = Lenv[beginsite]

        for w = beginsite:finalsite
            newpsi = psi[w]
            for g = 1:length(pos)
                opdist = w - pos[g]
                if 0 <= opdist < lengthops[g]
                    newpsi = contract([2,1,3],operators[order[g]][opdist + 1],2,newpsi,2)
                end
            end
            for r = 1:length(pos)
                if w < pos[r] & & !isId[r]
                    newpsi = contract([2,1,3],trail[r],2,newpsi,2)
                end
            end
            thisLenv = Lupdate(thisLenv,dualpsi[w],newpsi)
            if w < Ns
                Lenv[w+1] = thisLenv
            end
        end
    end
thisRenv = Renv[finalsite]
res = contract(thisLenv,thisRenv)
finalpos = pos[order]
\begin{verbatim}
@inbounds omega[finalpos...] = res

@simd for b = 1:length(pos)
    @inbounds prevpos[b] = pos[b]
end
operator_in_order!(base_pos,base_sizes)
@simd for b = 1:length(pos)
    @inbounds pos[b] = sites[b][base_pos[b]]
end
end
end
return omega
end

export correlation

J. Constructing MPOs and MPSs

Constructing the MPOs and MPSs can be tedious using basic operations. These functions make it easier to define
the MPO. The convention used for MPSs is given in Ref. 3 as is the convention for the MPOs.

In a subsequent work, the automatic generation the MPO will be used to make these quantities.

1. Function: makeMPO

This function converts the MPO form in a systematic way. The input MPO tensors can be entered as a function
or as a vector of rank-2 tensors. If rank-4 tensors are made, then they can be loaded into the MPO easily by
defining MPO(rank4s). See Ref. 3 for more details on writing MPO functions. There are two inputs that could be
accommodated for the makeMPO function. One is a simple matrix of numbers. Operators can be found from the
Opt.jl module.
Alternatively, the input matrix could be encoded in a function whose only argument is the site index.
In order to properly convert the rank-2 flattened MPO format, the physical index size must be known, hence the
physindsiz parameter which can be constant for each site (integer input) or vary with a vector of inputs. The
vector will repeat on, for example, an input of 3-site vector for a 10-site lattice will assume that a modular assignment
is taken on the lattice.

```plaintext
function makeMPO(H::Array{X,1}, physSize::Array{Y,1}, Ns::Integer;
    lower::Bool=true, regtens::Bool=false)
where {X <: AbstractArray, Y <: Integer}
retType = typeof(prod(a->eltype(H[a])(1),1:Ns))
finalMPO = Array{Array{retType,4},1}(undef,Ns)
for i = 1:Ns
    thisH = lower ? H[i] : transpose(H[i])
    states = physSize[(i-1)%size(physSize,1) + 1]
    a1size = div(size(thisH,1),states) #represented in LEFT link indices
    a2size = div(size(thisH,2),states) #represented in RIGHT link indices
    P = eltype(thisH)
    currsize = [a1size,states,states,a2size]
    G = Array(P,4)(undef,currsize...)
    for l = 1:a1size
        for j = 1:states
            for k = 1:states
                @simd for m = 1:a2size
                    @inbounds G[l,j,k,m] = thisH[j + (l-1)*states, k + (m-1)*states]
                end
            end
        end
    end
end
return G
end
```

```
finalMPO[i] = G

if lower
    finalMPO[1] = finalMPO[1][end:end,:, :, :]
    finalMPO[end] = finalMPO[end][:,:, :, 1:1]
else
    finalMPO[1] = finalMPO[1][1:1,:, :, :]
    finalMPO[end] = finalMPO[end][:,:, :, end:end]
end

return MPO(finalMPO, regtens=regtens)
end

function makeMPO(H::Array{X,2}, physSize::Array{Y,1}, Ns::Integer; lower::Bool=true, regtens::Bool=false)
    where {X <: Number, Y <: Integer}
    return makeMPO([H for i = 1:Ns], physSize, Ns)
end

function makeMPO(H::Array{X,2}, physSize::Y, Ns::Integer; lower::Bool=true, regtens::Bool=false) where {X <: Number, Y <: Integer}
    return makeMPO([H for i = 1:Ns], [physSize], Ns)
end

function makeMPO(H::Array{X,1}, physSize::Y, Ns::Integer; lower::Bool=true, regtens::Bool=false)
    where {X <: AbstractArray, Y <: Integer}
    return makeMPO(H, [physSize], Ns, lower=lower, regtens=regtens)
end

function makeMPO(H::Array{X,1}, physSize::Y;
    lower::Bool=true, regtens::Bool=false)
    where {X <: AbstractArray, Y <: Integer}
    return makeMPO(H, [physSize], length(H), lower=lower, regtens=regtens)
end

function makeMPO(H::Function, physSize::Array{X,1}, Ns::Integer;
    lower::Bool=true, regtens::Bool=false) where X <: Integer
    thisvec = [H(i) for i = 1:Ns]
    return makeMPO(thisvec, physSize, Ns, lower=lower, regtens=regtens)
end

function makeMPO(H::Function, physSize::Integer, Ns::Integer;
    lower::Bool=true, regtens::Bool=false)
    return makeMPO(H, [physSize], Ns, lower=lower, regtens=regtens)
end

export makeMPO
Given a vector that could result from an exact diagonalization computation, the `makeMPS` function converts to the MPS form. To find the MPS form, the size of the physical index must be input, `physInd`, in addition to the vector.

```plaintext
function makeMPS(vect :: Array{W,1}, physInd :: Array{intType,1};
    Ns :: intType = length(physInd), left2right :: Bool = true,
    oc :: intType = left2right ? Ns : 1,
    regtens :: Bool = false) where W <: Number

mps = Array{Array{W,3},1}(undef, Ns)
if left2right
    M = reshape(vect, physInd[1], div(length(vect), physInd[1]))
    Lindsize = 1 # current size of the left index
    for i = 1: Ns - 1
        U, DV = qr(M)
        temp = reshape(U, Lindsize, physInd[i], size(DV, 1))
        mps[i] = temp
        Lindsize = size(DV, 1)
        if i == Ns - 1
            temp = unreshape(DV, Lindsize, physInd[i + 1], 1)
            mps[Ns] = temp
        else
            Rsize = cld(size(M, 2), physInd[i + 1])  # integer division, round up
            M = unreshape(DV, size(DV, 1)*physInd[i + 1], Rsize)
        end
    end
    finalmps = MPS(mps, Ns, regtens = regtens)
else
    M = reshape(vect, div(length(vect), physInd[end]), physInd[end])
    Rindsize = 1 # current size of the right index
    for i = Ns : -1 : 2
        UD, V = lq(M)
        temp = reshape(V, size(UD, 2), physInd[i], Rindsize)
        mps[i] = temp
        Rindsize = size(UD, 2)
        if i == 2
            temp = unreshape(UD, 1, physInd[i - 1], Rindsize)
            mps[1] = temp
        else
            Rsize = cld(size(M, 1), physInd[i - 1])  # integer division, round up
            M = unreshape(UD, Rsize, size(UD, 2)*physInd[i - 1])
        end
    end
    finalmps = MPS(mps, 1, regtens = regtens)
end
move!(finalmps, oc)
return finalmps
end

function makeMPS(vect :: W, physInd :: Array{intType,1};
    Ns :: intType = length(physInd),
    left2right :: Bool = true,
    oc :: intType = left2right ? Ns : 1,
    regtens :: Bool = false) where W <: denstens

newvect = copy(vect.T)
return makeMPS(newvect, physInd; Ns = Ns, oc = oc, left2right = left2right,
    regtens = regtens)
end
```
Note that the qr (or lq) decomposition is used, expanding on the method in Ref. 2.

3. Function: penalty! & penalty

In order to find excitations on the MPS, there is a strategy to add a penalty term to the MPO based on the ground state of the form

$$\mathcal{H} = \mathcal{H}_0 + \lambda |\psi_0\rangle\langle \psi_0|$$

(8)

where $\psi_0$ is the ground state found from $\mathcal{H}_0$. Given the penalty $\lambda$, the new ”ground state” will be the first excitation. This method can suffer from poor convergence issues and vastly increasing bond dimension of the MPO. There are many other ways to compute the excitations that produce higher quality results and are more efficient.
4. Function: `transfermatrix`

One very powerful, but expensive and not commonly used, analysis tool in a tensor network is the determination of the correlation length from the transfer matrix. Ref. 3 contains more information on this technique. The function contracts the link indices between the input indices \( i \) and \( j \) to obtain the transfer matrix. The function can be reused by placing the output transfer matrix back into the function with the optional `transfermat` option to the function.

```sh
function transfermatrix(dualpsi :: MPS, psi :: MPS, i :: Integer, j :: Integer; startsite :: Integer = i, transfermat :: TensType = ccontract([1,3,2,4], dualpsi[startsite],2, psi[startsite],2))
    for k = startsite + 1: j
        transfermat = contractc(transfermat,3, dualpsi[k],1)
        transfermat = contract(transfermat,[3,4], psi[k], [1,2])
    end
    return transfermat
end

function transfermatrix(psi :: MPS, i :: Integer, j :: Integer; startsite :: Integer = i, transfermat :: TensType = ccontract([1,3,2,4], psi[startsite],2, psi[startsite],2))
    return transfermatrix(psi, psi, i, j, transfermat = transfermat)
end
```

K. Converting to and from exact diagonalization computations

1. Function: `fullpsi`

The function `fullpsi` contracts all link indices in the MPS and returns the full wavefunction suitable for exact diagonalization.

```sh
function fullpsi(psi :: MPS)
    Ns = length(psi)
    fullpsi = psi[1]
    for p = 2:Ns
        fullpsi = contract(fullpsi, ndims(fullpsi), psi[p], 1)
    end
    return reshape!(fullpsi, prod(size(fullpsi)))
end
export fullpsi
```

2. Function: `fullH`

The function `fullH` will contract the entire MPO and produce a matrix representing the Hamiltonian. The function simply contracts the link indices and reorders the remaining physical indices so that the matrix is properly represented.

```sh
function fullH(mpo :: MPO)
    Ns = length(mpo)
    fullH = mpo[1]
    for p = 2:Ns
        fullH = contract(fullH, ndims(fullH), mpo[p], 1)
    end
    dualinds = [i+1 for i = 2:2:Ns]
    ketinds = [i+1 for i = 1:2:Ns]
    finalinds = vcat([1], ketinds, dualinds, [ndims(fullH)])
end
```
pfullH = permutedims(fullH,finalinds)
size1 = size(pfullH,1)
size2 = prod(a->size(fullH,a),ketinds)
size3 = prod(a->size(fullH,a),dualinds)
size4 = size(pfullH,ndims(pfullH))

rpfullH = reshape(pfullH,size1,size2,size3,size4)
return rpfullH[size(rpfullH,1),:,:1]
end
export fullH

This method will only work for a few sites before running into memory problems on the classical computer. The exact number depends on the size of the physical index of each site. Using double precision allows for approximately 10 Lanczos coefficients to be determined accurately. Upgrading the computation to handle quadruple precision is possible but requires a (slower) eigenvalue solver capable of handling this.