User Guide of the TNSP

Wang Chao, Dong Shaojun

September 23, 2017

Contents

1 Installment **5**

2 Creation of a Tensor **7**
2.1 Declaration of a Tensor . . . 7
2.2 Set the Datatype and the Shape of a Tensor . . . 7
2.3 Set Value to a Tensor . . . 9
2.4 Set Names of Tensor Indices . . . 10
2.5 Useful Functions . . . 11

3 Get Information from a Tensor **13**
3.1 Output Information of a Tensor on Screen . . . 13
3.2 Get Datatype and Structure Information . . . 14
3.3 Get Tensor Elements . . . 16
3.4 Function on a Tensor . . . 17
3.5 Dot Product of two tensors . . . 18

4 Operation on Tensors **19**
4.1 Operation on elements . . . 19
4.2 Fusion and split of tensor indices . . . 20
4.3 Permutation of tensor indices . . . 20
4.4 Composition . . . 22
4.5 Decomposition . . . 24

5 Input/Output of a Tensor **27**

6 Parallel Programming **29**
6.1 The MPI mode . . . 29
6.2 Inter-core Transportation of Tensors . . . 29
6.3 Some MPI functions on tensors . . . 30

7 Others **33**
7.1 Options . . . 33
7.2 Random Number Generator . . . 33
7.3 Writemess . . . 34
7.4 Pointer . . . 34

Chapter 1

Installment

TNSP can be installed with make. Configuration have to be set in the make.inc file. A make.inc for a Linux machine(ubuntu 16.04) running GNU compilers is given in the main directory. Then the user can do "make" in the current directory to build the standard library "libTensor-x"(x for version), and do "make test" to run the test PEPS program.

Before using DSJ's Tensor Package, the following module should be used

```
use Tensor_type
use Tools
```

To use the TNSP in your code, you should use the following option when compiling your code

```
-I$(TNSP)/libTensor-x
```

and the following option to link your code with the TNSP

```
-L$(TNSP)/libTensor-x -llibTensor-x
```

, where $(TNSP) is the TNSP directory and x is the version. And make sure you have linked the TNSP with the blas and lapack.

The package is also required to be compiled by OpenMPI using the compiler mpif90. The TNSP uses many features of the Fortran 2003 standard, and can be successfully compiled by gfortran compiler later than 4.8.4 version.

gfortran compiler can be downloaded at https://gcc.gnu.org/fortran/. In Unbuntu, it can be installed by a quick command

```
sudo apt-get install gfortran
```

Openmpi can be downloaded at https://www.open-mpi.org/. In Unbuntu, it can be installed by a quick command

```
sudo apt-get install libopenmpi-dev
```

Lapack and Blas can be downloaded at http://www.netlib.org/lapack/. In Unbuntu, it can be installed by a quick command

```
sudo apt-get install liblapack-dev
```

A copy of the latest version of tensor package can be requested from sj.dong@outlook.com. The bugs of the package may also be reported to the same email.

Chapter 2

Creation of a Tensor

2.1 Declaration of a Tensor

The most basic data type in the Tensor package is `type(Tensor)`. A variable with such type is declared as

```
1 type(Tensor)::my_tensor
```

After declaration of a tensor, its status is empty. That means the tensor has no shape or datatype(type of data). Before we use the tensor, we need to determine its datatype and shape, which will be introduced in the next section.

2.2 Set the Datatype and the Shape of a Tensor

The datatype of a tensor is the datatype of its elements. The package currently doesn't not support elements of a tensor with different datatypes, so the datatype of all elements of a tensor is always the same.

An tensor can have three status:

1. An *empty* tensor has no shape or datatype.
2. A *static* tensor has a shape and a fixed datatype.
3. A *dynamic* tensor has a shape and a variable datatype.

A *static* tensor has fixed datatype. If we make an operation that may change its datatype, a proceeding type transformation will always be performed automatically to keep the datatype of the tensor unchanged. A *dynamic* tensor have no fixed datatype. Its datatype can always be changed by operations or assignments.

A tensor after declaration is *empty*. We can set the shape and the datatype of an *empty* tensor, and make it a *static* one, by the type-bound procedure:

```
1     allocate(int dims(:), int/chars datatype)
```

where the *dims* argument is an 1-D integer array that specifies the shape of the tensor (which includes the rank of the tensor as `size(dims)` and dimensions of indices of the tensor as `dims(1)`, `dims(2)` ...). The

Datatype		Data type of Tensor elements	Abbreviation
Integer	String		
1	'integer'	`integer`	i
2	'real' 'real*4' 'real(kind=4)'	`real(kind=4)`	s
3	'double' 'real*8' 'real(kind=8)'	`real(kind=8)`	d
4	'complex' 'complex*8' 'complex(kind=4)'	`complex(kind=4)`	c
5	'complex*16' 'complex(kind=8)'	`complex(kind=8)`	z
6	'logical'	`logical`	l
7	'character'	`character(len=*)`	a

Table 2.1: Pre-set datatypes and corresponding data types of tensor elements.

datatype argument is an integer or a string that specifies the datatype of tensor elements, which must be chosen within several pre-set values, as listed in Tab. 2.1.

For example, the following statement generates a $3 \times 4 \times 2$ tensor with double-precision real elements.

```
use Tensor_type
type(Tensor)::my_tensor
complex*16::data2(3,2,2)

    data2=dcmplx(1.2,2)
    call my_tensor%allocalte([3,4,2],'real*8') ! my_tensor is now a 3×4×2 'static' tensor
                                               ! with double-precision real elements
                                               ! It's elements are 0 by default.
    my_tensor=data2  ! my_tensor is now a 3×2×2 'static' tensor with
                     ! double-precision real elements.
                     ! It's elements are all 1.2, since a type transformation
                     ! from complex*16 to real*8 has been implicitly conducted
```

There's no procedure to transform an *empty* tensor into a *dynamic* one. This is automatically performed when we assign an array or another tensor to an *empty* tensor. For example:

```
use Tensor_type
type(Tensor)::my_tensor,another_tensor

    !
    ! Here we may assign some values to another_tensor
    !
```

```
7
8     my_tensor=another_tensor ! my_tensor is now a dynamic tensor with the same
9                              ! shape and data as another_tensor
```

or

```
1 use Tensor_type
2 type(Tensor)::my_tensor
3 integer::data1(3,4)
4 real*8::data2(3,2,2)
5
6     data1=1
7     my_tensor=data1          ! my_tensor is now a 3 × 4 dynamic integer tensor with
8                              ! the same data as data1
9     data2=0.2d0
10    my_tensor=data2          ! my_tensor is now a 3 × 2 × 2 dynamic double precision real
11                             ! tensor with the same data as data2
```

As shown from the examples above, unlike *static* tensors, a *dynamic* tensor doesn't have fixed datatype. So it's more flexible but easier to cause precision problems. In practical use, *dynamic* tensors are usually used as immediate variables, and *static* tensors are usually used to save crucial data.

For a *dynamic* tensor, we can fix its datatype and render it *static* by type-bound procedure:

```
1     Static()
```

And for a *static* tensor, we can release the control of its datatype and render it *dynamic* by type-bound procedure:

```
1     Dynamic()
```

2.3 Set Value to a Tensor

As already shown in previous examples, we can set value to a tensor by assignment statements

```
1     my_tensor=tensor2
```

or

```
1     my_tensor=array
```

As already illustrated, if `my_tensor` is *empty* or *dynamic*, it will become exactly `tensor2` or `array`. If `my_tensor` is *static*, a type transformation may be automatically performed.

We can also change the value of a single tensor element by the type-bound procedure

```
    set_value(int pos(:,:,...), value)
```

where `pos` is a multi-dimensional array specifies the position of the element. For example

```
use Tensor_type
type(Tensor)::my_tensor
integer::data(3,4)

    data=1
    my_tensor=data                              ! my_tensor(1,2)=1
    call my_tensor%setValue([1,2],2.1d0)        ! my_tensor(1,2)=2
```

Note that a type transformation may be automatically performed even if we set value to a tensor element of a *dynamic* tensor, in order to satisfy that all elements of a tensor have the same datatype.

2.4 Set Names of Tensor Indices

A tensor may have several indices, which are conventionally labeled by integers. After a sequence of complex operations including contraction, SVD, QR decomposition . . . Its very hard to remember which integer means which index. To solve such problem, we may assign a name to each index of a tensor, which free us from keeping track of the indices during tensor operations. A standard name of an index has the syntex $A.b$, where the part before the dot is interpreted as the name of the tensor and the part after the dot is the actual name of the index. We can set the name of an index by the type-bound procedure

```
    setName(int n, chars name)
```

where `n` identifies the index we want to set the name and should be less than or equal to the rank of the tensor, and `name` is just the index name. For example:

```
use Tensor_type
type(Tensor)::my_tensor
integer::data(3,4)

    data=1
    my_tensor=data
    call my_tensor%setName(1,'tn.left')       ! we set the index name of the 1st index of
                                              ! my_tensor as tn.left
    call my_tensor%setName(2,'tn.right')      ! we set the index name of the 2nd index of
                                              ! my_tensor as tn.right
    call my_tensor%setName(1,'tn.up')         ! we replace the index name of the 1st index
                                              ! of my_tensor as tn.up
```

Have set the index name of some index of a tensor, we can replace it with some other index name by a variation of `setName`

```
    setName(chars old_name, chars new_name)
```

For example

```
use Tensor_type
type(Tensor)::my_tensor
integer::data(3,4)

    data=1
    my_tensor=data
    call my_tensor%setName(1,'tn.left')       ! we set the index name of the 1st index of
                                              ! my_tensor as tn.left
    call my_tensor%setName('tn.left','tn.up') ! we replace the index name of the 1st index
                                              ! of my_tensor by tn.up
```

Note that we should always make sure that there's no two indices of a tensor with the same index name.

2.5 Useful Functions

A quick way to generate ramdom matrix is through the type-bound procedure

```
    random()
```

For example

```
use Tensor_type
type(Tensor)::my_tensor

    call my_tensor%allocalte([3,4,2],'real*8')
    call my_tensor%random()                    ! now my_tensor is a random tensor
```

Note that `my_tensor` should be non-empty (thus has a shape) before we randomize it.

We can generate a diagonal matrix (as a rank-2 tensor) by the type-bound procedure

```
    eye(array,int D1, int D2)
    eye(array)
    eye(int D1, int D2)
```

The procedure has three varieties. If the input parameters include an array together with integers D1 and D2, the returned value is a $D1 \times D2$ matrix (as a dynamic tensor) with datatype and diagonal elements same as the input array. If the If dimension of the input array is larger that D1 or D2 then there's a cut-off. the input parameter is only an array, the returned value is a $D \times D$ matrix with datatype and diagonal elements same as the input array, where D is the dimension of the input array. If the input parameters are integers D1 and D2, the returned value is a $D1 \times D2$ integer matrix, with all diagonal elements 1.

For example

```
use Tensor_type
type(Tensor)::my_tensor

    my_tensor=eye([0.5,0.4],3,3)        ! now my_tensor is a 3 × 3 matrix: diag(0.5, 0.4, 0)
    my_tensor=eye(2,2)                  ! now my_tensor is a 2 × 2 identy matrix
```

Chapter 3

Get Information from a Tensor

We have learnt how to build a tensor. In this section we introduce some procedures to get information from an already-built tensor, which includes datatype, rank, dimension of each index, name of each index and tensor elements.

3.1 Output Information of a Tensor on Screen

During coding, we need to check if the tensor is what we want from time to time. This task is accomplished by procedures to output information of a tensor on screen.

Firstly we can output the basic information of a tensor on screen by the type-bound procedure

```
1    dimInfo()
```

Secondly we can output the elements of a tensor on screen by the type-bound procedure

```
1    print()
```

For example

```
1  use Tensor_type
2  type(Tensor)::my_tensor
3  integer::data(3,4)
4
5      data=reshape([1,2,3,4,5,6,7,8,9,0,1,2],[3,4])
6      my_tensor=data
7      call my_tensor%setName(1,'tn.left')      ! we set the index name of the 1st index of
8                                               ! my_tensor as tn.left
9      call my_tensor%setName(2,'tn.right')     ! we set the index name of the 2nd index of
10                                              ! my_tensor as tn.right
11     call my_tensor%dimInfo()                 ! output structure information of my_tensor
12                                              ! on screen
13     call my_tensor%print()                   ! output tensor elements on screen
```

The output on screen would be

```
==================
------------------

*** START ***
Dynamic class Tensor,data type is integer
The rank of the Tensor is
2
The number of  data of the Tensor is
12
***   Dimension Data    ***
3 , 4
***   Dimension END   ***
index Name are
tn.left , tn.right

***end***

==================
------------------
Dynamic,integer
*** START ***
1           4           7           0

2           5           8           1

3           6           9           2

*** END ***
```

3.2 Get Datatype and Structure Information

The datatype of a tensor can be accessed by the type-bound procedure

```
    int getType()
```

The returned value is an integer representing the datatype of the tensor, as explained in Tab. 2.1. Or by the type-bound prodcedure

```
    chars(20) getclassType()
```

The returned value is an string of 20 characters representing the datatype of the tensor, as explained in Tab. 2.1.

The rank of a tensor can be accessed by the type-bound procedure

```
int getRank()
```

The returned value is the rank of the tensor.

The dimension of each index of a tensor can be accessed by the type-bound procedure

```
int(:) dim()
int dim(int n)
int dim(chars indexName)
```

This procedure has three varieties. If there is no input parameter, the returned value is an array of dimensions of all indices. If the input parameter is an integer n, the returned value is the dimension of the nth index. If the input parameter is a string of characters, the returned value is the dimension of the index specified by the input string as the index name.

The name of an index of a tensor can be accessed by the type-bound procedure

```
chars(len=length of index name)(:) outName()
chars(len=length of index name) outName(int n)
```

This procedure has two varieties. If there is no input parameter, the returned value is an array of string of charactoers containing names of all indices. If the input parameter is an integer n, the returned value is the name of the nth index.

An example of the four procedures above is

```
use Tensor_type
type(Tensor)::my_tensor
integer::data(3,4),my_type,my_rank,dim1,dim2,dims(2)
character(len=20)::char_type,index_nm1,index_nms(2)

    data=reshape([1,2,3,4,5,6,7,8,9,0,1,2],[3,4])
    my_tensor=data
    call my_tensor%setName(1,'tn.left')      ! we set the index name of the 1st index of
                                             ! my_tensor as tn.left
    call my_tensor%setName(2,'tn.right')     ! we set the index name of the 2nd index of
                                             ! my_tensor as tn.right
    my_type=my_tensor%getType()              ! get the datatype of my_tensor which is 1
                                             ! for integer
    char_type=my_tensor%getclassType()       ! get the datatype of my_tensor which is
                                             ! 'integer'
    my_rank=my_tensor%getRank()              ! get the rank of my_tensor which is 2
    dims=my_tensor%dim()                     ! get the dimensions of all indices of
                                             ! my_tensor
    dim1=my_tensor%dim(1)                    ! get the dimension of 1st index of my_tensor

```

```
    dim2=my_tensor%dim('tn.right')          ! get the dimension of the index of my_tensor
                                            ! with index name 'tn.right'
    index_nm1=my_tensor%outName(1)          ! get the name of 1st index of my_tensor
                                            ! which is 'tn.left'
    index_nms=my_tensor%outName()           ! get the names of indices of my_tensor
                                            ! which is ['tn.left','tn.right']
```

3.3 Get Tensor Elements

The tensor functions introduced above return the basic information of a tensor, thus their returning values have definite types.

There are also tensor functions that return the calculation result fo a tensor. The type of their returning values may depend on the datatype of themselves. Such kind of tensor functions are usually named as

```
    ?function(...)
```

, where '?' is absent or a character in i, s, d, c, z, l, a. When '?' is absent, the returning value is a tensor. When '?' is in i-a, the returning value has the type abbreviated by '?' as listed in Tab. 2.1.

The type-bound procedure

```
    ?i(int pos(:))
    ?i()
```

returns the elements of a tensor.

When there is no input, the output is an 1d array (if '?' in i-a) or a tensor (if '?' is absent) of all tensor elements. We can also use an integer array `pos` to specify the position of the output element. For example

```
use Tensor_type
type(Tensor)::my_tensor
real*8::data(3,4),data2(3,4),data3
integer::data4(3,4)

    data=reshape([1.1,2.1,3.1,4.1,5.1,6.1,7.1,8.1,9.1,0.1,1.1,2.1],[3,4])
    my_tensor=data
    data2=reshape(my_tensor%di(),[3,4])    ! get the elements of my_tensor
    data3=my_tensor%di([2,2])              ! get the element of my_tensor at [2,2]
    data4=reshape(my_tensor%ii(),[3,4])    ! get the elements of my_tensor as integers
```

The user-defined unitary operator

```
    .?i.
```

with syntax `T.?i.pos` serves the same job as `T%?i(pos)`.

3.4 Function on a Tensor

The largest element of a tensor can be derived by the type-bound procedure

```
    ?maxmin(chars ctr)
```

If input is absent, the output is the element with largest real part. If the input is 'maxr', the return value is the real part of the element with the maximal real part. If the input is 'maxi', the return value is the imaginary part of the element with the maximal image part. If the input is 'maxa', the return value is the absolute value of the element with the maximal absolute value. If the input is 'minr', 'mini', 'mina', minimal value is returned similarly.

For example

```
use Tensor_type
type(Tensor)::my_tensor
complex*16::data(2,2)
real*8::maxre,maxim,maxabs,minre,minim,minabs

    data=reshape([dcmplx(1.1,1.2),dcmplx(1.3,1.4),dcmplx(1.5,1.6),dcmplx(1.7,1.8)],[2,2])
    my_tensor=data

    maxre=my_tensor%dmaxmin('maxr')    ! = 1.70
    maxim=my_tensor%dmaxmin('maxi')    ! = 1.80
    maxabs=my_tensor%dmaxmin('maxa')   ! = 2.48
    minre=my_tensor%dmaxmin('minr')    ! = 1.10
    minim=my_tensor%dmaxmin('mini')    ! = 1.20
    minabs=my_tensor%dmaxmin('mina')   ! = 1.63
```

The sum of all elements of a tensor can be derived by the type-bound procedure

```
    ?sum()
```

The trace of a matrix (a rank-2 tensor) can be derived by the type-bound procedure

```
    ?trace()
```

The norm of a tensor defined by the sum of the squared absolute values of all elements can be derived by the type-bound procedure

```
    ?norm2()
```

whose squared root is returned by the type-bound procedure

```
    ?norm()
```

An example of the four functions above is

```
use Tensor_type
type(Tensor)::my_tensor
complex*16::my_sum,my_trace
real*8::my_norm,my_norm2

    call my_tensor%allocate([5,5],'complex*16')
    call my_tensor%random()

    my_sum=my_tensor%zsum()          ! my_sum is now the sum of all elements of my_tensor
    my_trace=my_tensor%ztrace()      ! my_trace is now the trace of my_tensor
    my_norm=my_tensor%dnorm()        ! my_norm is now the 2-norm of my_tensor
    my_norm2=my_tensor%dnorm2()      ! my_norm2 is now the squared 2-norm of my_tensor
```

3.5 Dot Product of two tensors

The dot product can be conducted conveniently using user defined operator

```
    .?x.
    .?dot.
```

When taking dot product, two tensors are treated as 1-D arrays. The `.?x.` operator will conjugate the first tensor while the `.?dot.` operator won't. For example

```
use Tensor_type
type(Tensor)::A,B
real*8::dot1,dot2

    call A%allocate([5],'complex*16')
    call A%random()
    call B%allocate([5],'complex*16')
    call B%random()

    dot1=A.dx.B      ! now dot1= \sum_i A_i^* B_i
    dot2=A.ddot.B    ! now dot2= \sum_i A_i B_i
```

Caution: Note that the priority of user-defined operators are always lower than in-built operators. So we may put the statement in brackets properly.

Chapter 4

Operation on Tensors

In this chapter we introduce some operations on tensors. The operations are classified into five categories according to their behaviors.

4.1 Operation on elements

In this section we introduce some operations that acts on each element independently. The first operation is to take conjugation on each element, performed by the over-loaded function

```
    conjg(tensor)
```

For example

```
use Tensor_type
type(Tensor)::my_tensor
complex*16::data(2,2)

    data=reshape([dcmplx(1.1,1.2),dcmplx(1.3,1.4),dcmplx(1.5,1.6),dcmplx(1.7,1.8)],[2,2])
    my_tensor=data
    my_tensor=conjg(my_tensor)  ! now my_tensor is conjugated
```

The user-defined unitary operator

```
    .con.
```

serves the same job.

Then there are four arithmetic operators $+$, $-$, $*$ and / overloaded to act on tensors.

$+$ and $-$ take the sum or difference of two tensors of same rank and same dimension of each rank. For example

```
    T3=T1+T2
    T4=T1-T2
```

Note that the user should make sure that indices of the two tensors are of the same order. One can also add a number to or subtract a number from each element of a tensor by

```
1    T2=T1+num
2    T3=T1-num
```

Similarly one can multiply a number to each element of a tensor or divide each element of a tensor by a number by

```
1    T2=T1*num
2    T3=T1/num
```

4.2 Fusion and split of tensor indices

In this section we introduce two operations fuse and split indices.

To fuse some continuous indices into one index, we can use the type-bound procedure

```
1    fuse(int (i:j))
```

, which fuse ith to jth indices of the original tensor into one index.

The type-bound procedure

```
1    split()
```

splits all legs that has been fused.

4.3 Permutation of tensor indices

In this section we introduce some operations that permutes tensor indices, which may be useful in some cases.

The major way to permute the indices of a tensor is to use the type-bound procedure

```
1    permute(int/chars order())
```

The input value is an array of integers or an array of strings of characters. If we input an array of integer, it should be a permutation of $\{1 \ldots n\}$, where n is the rank of the tensor. Then we permute the indices of the tensor as defined by the input permutation. If we input an array of strings of characters, it should be a specific arrangement of the index names, and specifies the new order of the indices.

The same task can also be accomplished by the user-defined operator

```
1    .p.
```

Sometimes it's desired to permute an index of a tensor into the first or last order. This task is performed by the type-bound procedures

```
1    forward(int/chars index)
```

and

```
1    backward(int/chars index)
```

The `forward` procedure permutes the ith index or the index specifies by name into the first order. The `backward` procedure permutes the ith index or the index specifies by name into the last order.

The two procedures each have a variety

```
1    forward(int/chars index(:))
```

and

```
1    backward(int/chars index(:))
```

where multiple indices specified by the input are permuted to the first or last order.

The same task can also be accomplished by the user-defined operator

```
1    .pf.
```

and

```
1    .pb.
```

We give an example of the 3 operations talked above

```
1  use Tensor_type
2  type(Tensor)::my_tensor
3
4      call my_tensor%allocate([2,2,2,2],'real*8')
5      call my_tensor%setname(1,'tn.1st')
6      call my_tensor%setname(2,'tn.2nd')
7      call my_tensor%setname(3,'tn.3rd')
8      call my_tensor%setname(4,'tn.4th')        !order: 1st, 2nd, 3rd, 4th
9      call my_tensor%permute(['tn.4th','tn.3rd','tn.2nd','tn.1st'])
10                                               !order: 4th, 3rd, 2nd, 1st
11     call my_tensor%forward('tn.1st')          !order: 1st, 4th, 3rd, 2nd
12     call my_tensor%backward('tn.4th')         !order: 1st, 3rd, 2nd, 4th
```

Caution: Since fortran2003 only supports an array of strings of characters with the same length. If different index names have different length, we may need to add some blanks before or after the index name. The blanks are automatically trimmed inside the package and will not affect the result.

The quick way to permute a rank-2 tensor(or a matrix) is to use the over-loaded function

```
1    transpose(tensor)
```

The operator

```
1    .H.
```

performs the Hermitian transformation of a rank-2 tensor.

An example of the 2 operations above is

```
1 use Tensor_type
2 type(Tensor)::A,At,Ah
3 complex*16::data(2,2)
4
5    data=reshape([dcmplx(1.1,1.2),dcmplx(1.3,1.4),dcmplx(1.5,1.6),dcmplx(1.7,1.8)],[2,2])
6    A=data
7    At=transpose(A)      ! At = A^T
8    Ah=.H.A              ! Ah = A^†
```

4.4 Composition

In this section we introduce some operations that compose two tensors together.

Firstly, we can compose two tensors together by contracting some indices. This is performed by the function

```
1    contract(tensorA, chars indexA(:), tensorB, chars indexB(:))
2    contract(tensorA, tensorB)
```

The input values of the first variety contain two arrays of strings of characters: `indexA` is an array of index names of `tensorA`, and `indexB` is an array of index names of `tensorB`. We compose `tensorA` with `tensorB` by contracting `indexA(i)` index with `indexB(i)` index for all i. By the second variety, we compose `tensorA` and `tensorB` together by contracting indices with identical names.

For example

```
1 use Tensor_type
2 type(Tensor)::L,R,LR
3
4    call L%allocate([2,2,2,2],'real*8')
```

```
    call L%random()
    call L%setname(1,'L.up')
    call L%setname(2,'L.down')
    call L%setname(3,'L.right1')
    call L%setname(4,'L.right2')

    call R%allocate([2,2,2,2],'real*8')
    call R%random()
    call R%setname(1,'R.up')
    call R%setname(2,'R.down')
    call R%setname(3,'R.left1')
    call R%setname(4,'R.left2')

    LR=contract(L,['L.right1','L.right2'],R,['R.left1','R.left2'])
         ! L and R are composed into LR with L.right1 contracted with R.left1 and
         ! L.right2 contracted with R.left2
```

The whole process is illustrated in Fig. 4.1.

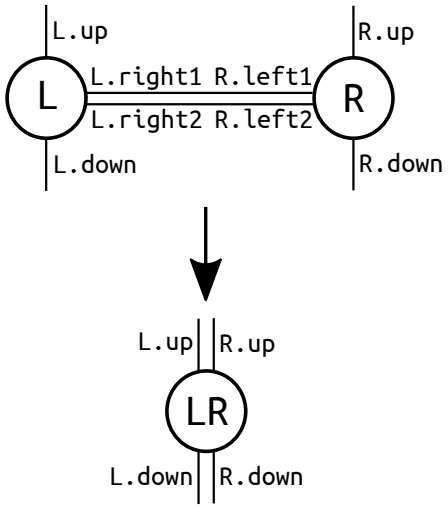

Figure 4.1: Contractiong of two tensors.

Secondly we can take the direct product of two tensors by the user-defined operator

```
    .kron.
```

For example

```
use Tensor_type
type(Tensor)::L,R,LR

    call L%allocate([2,2],'real*8')
```

```
    call L%random()
    call L%setname(1,'L.up')
    call L%setname(2,'L.down')

    call R%allocate([2,2],'real*8')
    call R%random()
    call R%setname(1,'R.up')
    call R%setname(2,'R.down')

    LR=L.kron.R    ! L and R are composed into LR by direct product
```

4.5 Decomposition

In this section we introduce some operations that decompose a tensor into multiple tensors.

Firstly we can subtract a smaller tensor from the original tensor by fixing an index. This is performed by the type-bound procedure

```
    subtensor(chars indexname,int value)
```

The subtensor is obtained by fixing the index specified by `indexname` at `value`.

Secondly we can perform a QR decomposition using the type-bound procedure

```
    QRTensor(tensor A, chars nameQ, chars nameR)
```

The input value `nameQ` and `nameR` tell us how to make the decomposition. The input tensor should satisfies that each index has the name `nameQ.???` or `nameR.???`. Then the indices started with `nameQ` is fused into one index, and the indices started with `nameR` is fused into another index. Then the tensor A becomes a rank-2 tensor. We can perform a QR decomposition, and decompose it into two tensors Q and R. Finally we split the indices that has been fused, and return the tensors as an array `[Q,R]`. Note that the last index of the tensor Q and the first index of the tensor R are newly generated and have no names.

For example

```
use Tensor_type
type(Tensor)::T,QR(2),Q,R

    call T%allocate([2,2,2,2],'real*8')
    call T%random()
    call T%setname(1,'Q.up')
    call T%setname(2,'Q.left')
    call T%setname(3,'R.down')
    call T%setname(4,'R.right')

    QR=T%QRTensor('Q','R')
    Q=QR(1)
```

```
13     call Q%setname(Q%getRank(),'Q.new')  ! Q now contains indices: Q.up, Q.left, Q.new
14     R=QR(2)
15     call R%setname(1,'R.new')            ! R now contains indices: R.down, R.right, R.new
```

The whole process is illustrated in Fig. 4.2

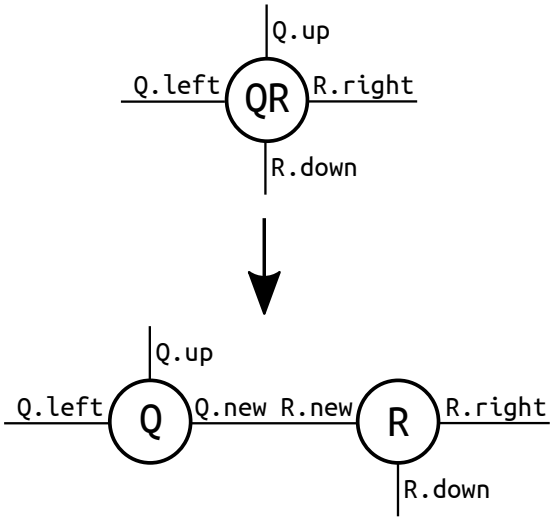

Figure 4.2: QR decomposition.

Similarly we can perform an LQ decomposition using the type-bound procedure

```
1     LQTensor(tensor, chars nameL, chars nameQ)
```

and the result is an array `[L,Q]`.

Finally we can perform a singular value decomposition(SVD) using the type-bound function

```
1     SVDTensor(tensor, chars nameL, chars nameR)
2     SVDTensor(tensor, chars nameL, chars nameR, int Dcut)
```

The result is an array `[L,S,R]`. S is the array of singular values. The last index of the tensor L and the first index of the tensor R are newly generated and has no name. The optional input `Dcut` specifies the truncation we made during the process of SVD. If there's no input `Dcut`, no truncation will be performed. If we input `Dcut`, S will be `Dcut` dimensional, and only the first `Dcut` largest singular values are retained.

For example

```
1 use Tensor_type
2 type(Tensor)::T,SVD(3),L,S,R
3
4     call T%allocate([2,2,2,2],'real*8')
5     call T%random()
```

```
6     call T%setname(1,'L.up')
7     call T%setname(2,'L.left')
8     call T%setname(3,'R.down')
9     call T%setname(4,'R.right')
10
11    SVD=T%SVDTensor('L','R',3)      ! Only first 3 largest singular values are retained
12    L=SVD(1)                        ! L is unitary (or orthogonal)
13    call L%setname(L%getRank(),'Q.new')
14    S=eye(SVD(2)%di())              ! S is diagonal
15    call S%setname(1,'S.left')
16    call S%setname(2,'S.right')
17    R=SVD(3))                       ! R is unitary (or orthogonal)
18    call R%setname(1,'R.new')
```

The whole process is illustrated in Fig. 4.3

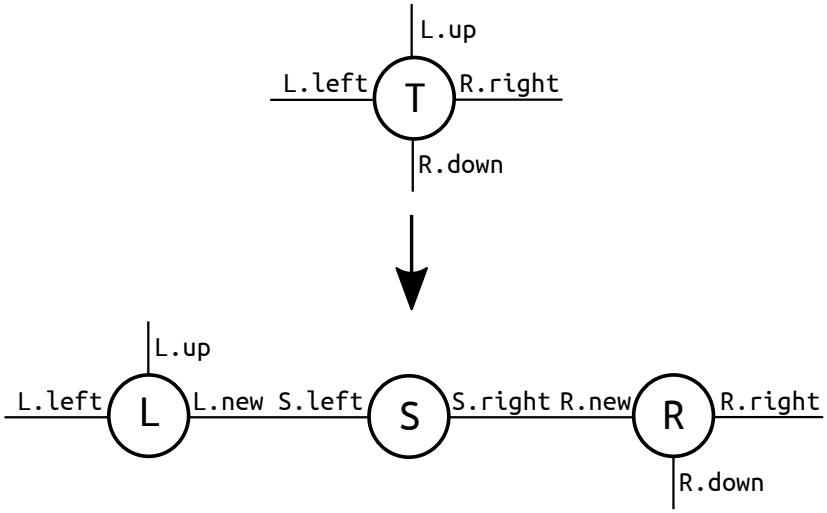

Figure 4.3: Singular value decomposition.

Chapter 5

Input/Output of a Tensor

In the design of a program, we sometimes want to save the tensor in a file, or to read the previously saved tensor from a file. These tasks are accomplished by the type-bound procedures

```
write(int file)
```

and

```
read(int file)
```

where `file` is the file unit.

For example

```
use Tensor_type
type(Tensor)::my_tensor, new_tensor
integer::data(3,4)

    data=reshape([1,2,3,4,5,6,7,8,9,0,1,2],[3,4])
    my_tensor=data

    call my_tensor%setName(1,'tn.left')      ! we set the index name of the 1st index of
                                             ! my_tensor as tn.left
    call my_tensor%setName(2,'tn.right')     ! we set the index name of the 2nd index of
                                             ! my_tensor as tn.right

    open(1,file='tensor.dat')
    call my_tensor%write(1)                  ! save my_tensor to tensor.dat
    close(1)

    open(2,file='tensor.dat')
    call new_tensor%read(2)                  ! read new_tensor from tensor.dat
                                             ! now new_tensor is identical to my_tensor
    close(2)
```

Chapter 6

Parallel Programming

The TNSP works with MPI to perform parallel programing. In this chapter, we introduce some MPI related functions.

6.1 The MPI mode

The MPI mode of the tensor package can be evoked by the procedure

```
1    set_output_cpu_info(int id,int nproc,int ierr)
```

The input `id` specifies the processor of the output procedure `writemess` which will be introduced in next chapter. The input `nproc` should be the number of processors running MPI, and the input `ierr` is the standard error code of the package. Note that this subroutine should be called after the initialization of MPI.

6.2 Inter-core Transportation of Tensors

We can send a tensor from a processor to another using the function

```
1    MPI_Send_Tensor(Tensor_from,Tensor_to,ID_from,ID_to,ierr,MPI_communicator)
```

We can broadcast a tensor to all processors in a communicator using the function

```
1    MPI_BCAST_Tensor(Tensor,ID,ierr,MPI_communicator)
```

For example

```
1 include 'mpi.h'
2 use Tensor_type
3 type(Tensor)::rand_tensor
4 integer::num_procs,id,ierr,i,seed
```

```
5
6     call MPI_Init(ierr)
7     call MPI_Comm_size(MPI_COMM_WORLD,num_procs,ierr)
8     call MPI_Comm_rank(MPI_COMM_WORLD,id,ierr)
9
10    set_output_cpu_info(0,num_procs,ierr)
11
12    call rand_tensor%allocate([3,3],'real*8')
13    call rand_tensor%random()           ! generate random tensors
14    if(num_procs>=3)then                ! send tensor at No.0 processor to No.2 processor
15        call MPI_Send_Tensor(rand_tensor,rand_tensor,0,2,ierr,MPI_COMM_WORLD)
16    end if
17    call MPI_BCAST_Tensor(rand_tensor,1,ierr,MPI_COMM_WORLD) ! send tensor at No.1 processor
18                                                             ! to all
19
20    call MPI_Finalize(ierr)
```

6.3 Some MPI functions on tensors

We can perform element-wise operation(taking sum, taking minimal or maximal value) on tensors at all processors in a communicator, and save the result in some tensor at all processors, using the function

```
1     MPI_Sum_Tensor(Tensor_in,Tensor_out,ierr,MPI_communicator)
2     MPI_Max_Tensor(Tensor_in,Tensor_out,ierr,MPI_communicator)
3     MPI_Min_Tensor(Tensor_in,Tensor_out,ierr,MPI_communicator)
```

For example

```
1 use mpi
2 use Tensor_type
3 type(Tensor)::rand_tensor,tnsum,tnmax,tnmin
4 integer::num_procs,id,ierr,i,seed
5
6     call MPI_Init(ierr)
7     call MPI_Comm_size(MPI_COMM_WORLD,num_procs,ierr)
8     call MPI_Comm_rank(MPI_COMM_WORLD,id,ierr)
9
10    set_output_cpu_info(0,num_procs,ierr)
11
12    call rand_tensor%allocate([3,3],'real*8')
13    call rand_tensor%random()           ! generate random tensors
14
15    call MPI_sum_Tensor(rand_tensor,tnsum,ierr,MPI_COMM_WORLD) ! get sum
16    call MPI_max_Tensor(rand_tensor,tnmax,ierr,MPI_COMM_WORLD) ! get max
```

```
17     call MPI_min_Tensor(rand_tensor,tnmin,ierr,MPI_COMM_WORLD) ! get min
18
19     call MPI_Finalize(ierr)
```

Chapter 7

Others

7.1 Options

The package will automatically check if theres any repeated index name in a tensor. This feature costs some time on tensors with a large number of indices, and can be enabled and disabled using the functions

```
1    set_check_dimension()
2    unset_check_dimension()
```

One may set the maximal length of characters used in the package by

```
1    set_max_len_of_cha(int length)
```

This value is by default 5000, and is the upper bound of the length of index name and various other things.

7.2 Random Number Generator

The package provide a simple 16807 random number generator. The random seed is set automatically, and can be obtained by function

```
1    int out_randomseed()
```

Note that after evoking the MPI mode, the package automatically provides different random seeds on each processor. One may also set the seed manually by subroutine

```
1    set_seed(int my_seed)
```

The function to generate random number is

```
1    real*8 randomnumber()
```

and the returning value is a real*8 random number from 0 to 1.

7.3 Writemess

`Writemess` is a neatly designed subroutine to print characters on screen and/or to a log file. The syntax is very simple:

```
1    call writemess(chars message)
```

As have been introduced, in MPI mode, one can specify which processor to do the output. If the following subroutine is called

```
1    set_output_log_unit(int log_unit)
```

every message that `writemess` prints on screen will be printed to the log file of unit `log_unit`.

7.4 Pointer

There is a specially designed way to allow the users to DIY their own tensor functions. This is done through the type-bound procedure named

```
1    pointer(pointer data_pointer)
```

where `data_pointer` is a pointer that points to an 1D array of the same data type with the tensor. On return, it will points to the elements of the tensor that align in column-first manner (just like the physical realization of multi-dimensional arrays in fortran). Thus, a general way to DIY a tensor function/subroutine would be

```
1  subroutine my_sub(T)
2  use Tensor_type
3  implicit none
4
5      type(Tensor),intent(inout)::T
6      real(8),pointer::Tdata(:)    !supposing T is of double precision
7
8      call T%pointer(Tdata)
9
10     !!!!!!!!!!!!!!!!!!!!!!!!
11     ! DO SOMETHING ON TDATA !
12     !!!!!!!!!!!!!!!!!!!!!!!!
13
14 end subroutine
```

Note that the user can always perform the same task by directly visit the tensor element using type-bound procedures `?i` and `setValue`. However its usually fastest to use `pointer`. Furthermore, this is almost the only possible way to write some deep-level optimized code to get best performance.

Finally, we give a realistic example on how to use `pointer`. In the optimization of a value as a function of some variables in the form of tensor, a widely used method is call stochastic gradient descent(SGD) method. First we calculate an approximate value of the gradient of the optimization value with respect to tensor elements. That is, we get an approximate gradient tensor G. Next, we randomize the gradient tensor to get $R(G)$, and evolve the tensor variables in the opposite direction of $R(G)$ with a given step length. A common randomization function R has the form

$$R(G)_{i,j,k\ldots} = sign(G_{i,j,k\ldots}) * r_{i,j,k\ldots} \tag{7.1}$$

where $r_{i,j,k\ldots}$ is a random number in [-1,1] for each $i, j, k \ldots$.

The randomization process can be realized using `pointer` as

```
1  subroutine randomize(T)
2  use Tensor_type
3  implicit none
4
5      type(Tensor),intent(inout)::T
6      real(8),pointer::Tdata(:)    !supposing T is of double precision
7      integer::length,i
8      call T%pointer(Tdata)
9
10     length = T%gettotaldata()
11
12     do i=1,length
13         Tdata(i)=sign(randomnumber(),Tdata(i))
14     end do
15
16 end subroutine
```

Index

+, −, ∗, /, 19
.?dot., 18
.?i., 16
.?x., 18
.H., 22
.con., 19
.kron., 23
.p., 21
.pb., 21
.pf., 21
?i, 16
?maxmin, 17
?norm, 18
?norm2, 17
?sum, 17
?trace, 17

allocate, 7

backward, 21

conjg, 19
contract, 22

datatype, 8
dim, 15
dimInfo, 13
dynamic, 7

empty, 7
eye, 12

forward, 21
fuse, 20

getclassType, 14
getRank, 15
getType, 14

LQTensor, 25

MPI_BCAST_Tensor, 29
MPI_Max_Tensor, 30
MPI_Min_Tensor, 30
MPI_Send_Tensor, 29
MPI_Sum_Tensor, 30

outName, 15

permute, 20
pointer, 34
print, 13

QRTensor, 24

random, 11
randomnumber, 33
read, 27

set_output_cpu_info, 29
set_seed, 33
setName, 10
split, 20
static, 7
subtensor, 24
SVDTensor, 25

transpose, 22

write, 27
writemess, 34