

# Short introduction to openNAMPS – an open, lightweight non-Abelian MPS software package

Miklós Antal Werner

*Department of Theoretical Physics, Institute of Physics,  
Budapest University of Technology and Economics, Műegyetem rkp. 3., H-1111 Budapest, Hungary  
MTA-BME Quantum Dynamics and Correlations Research Group,  
Budapest University of Technology and Economics,  
Műegyetem rkp. 3., H-1111 Budapest, Hungary and  
Strongly Correlated Systems 'Lendület' Research Group,  
Wigner Research Centre for Physics, P.O. Box 49, 1525 Budapest, Hungary  
(Dated: August 30, 2023)*

The openNAMPS software package is introduced and the main functionalities are described. Matlab R2022b or newer is required to run openNAMPS.

## I. ABOUT OPENAMPS

In this document we introduce the openNAMPS software, that is free software under the GNU General Public License (GPLv3 [1]). You are free to use and improve the software, but we kindly ask you to cite Ref. 2, where the theoretical foundations of openNAMPS are described in details.

### The current version of openNAMPS contains

- a general tensor class "NAtensor" that can be used both for Abelian and non-Abelian symmetries
- MPS structures for both Abelian and non-Abelian symmetries
- Time Evolving Block Decimation (TEBD) algorithm for both Abelian and non-Abelian symmetries
- Structures that can store symmetries, sites for specific models.
- Commonly used models like Heisenberg spin chains and the Hubbard model is also implemented.
- Example run scripts for both Abelian and non-Abelian models

**Important note:** Before running the scripts it is necessary to execute the `pathimport.m` script from the root folder of openNAMPS.

## II. OPENAMPS BASICS

Below we describe the basic constituents of the code. See also Ref. 2 and 3 for a more detailed introduction.

### A. Quantum numbers and irrep indices

States of a quantum system with  $N_{\text{sym}}$  symmetries are usually characterized by  $N_{\text{sym}}$  different quantum numbers. For example, a states of a system with an  $SU(2)$

spin and  $U(1)$  charge symmetry are characterized by numbers  $\{S, Q\}$ , where  $S$  is the total spin and  $Q$  is the total charge of the state  $|\{S, Q\}; \psi\rangle$ . From a group theoretical perspective,  $S$  marks an irreducible representation of the  $SU(2)$  group while  $Q$  marks an irrep of the  $U(1)$  group.

In the openNAMPS code, instead of using the physical quantum numbers like  $S$  and  $Q$ , representations of simple groups are always indexed by positive integer numbers  $\gamma \in \{1, 2, 3, \dots\}$ . For more then one symmetry, the collection  $\Gamma = [\gamma_1, \gamma_2, \dots, \gamma_{N_{\text{sym}}}]$  of these indices stands for the total irrep index that characterizes the state. In the following, we simply refer to  $\Gamma$  as the irrep index of the state, which index contains  $N_{\text{sym}}$  positive integers. Trivial representation is always  $\Gamma = [1, 1, \dots, 1]$ .

### Indexing conventions $SU(2)$ , $U(1)$ , and $\mathbb{Z}_2$ groups

While users are free to define their symmetries using any convention (except that the trivial representation must be  $\gamma = 1$ ), below we show the convention used in the predefined symmetry classes.

- For the  $SU(2)$  we simply use  $\gamma_{SU(2)} = 2S + 1$ .
- In case of  $U(1)$  the representation index corresponding to the (integer, but possibly negative) charge  $Q$  is the position index of  $Q$  in the following series:  $\{0, -1, +1, -2, +2, -3, +3, \dots\}$ , i.e.

$$\gamma_{U(1)} = 2|Q| + \mathbf{1}_{Q \geq 0}(Q), \quad (1)$$

where the indicator function  $\mathbf{1}_{Q \geq 0}(Q)$  is 1 if  $Q \geq 0$  and zero otherwise.

- In the case of  $\mathbb{Z}_2$  parity symmetry  $\gamma = 1$  stands for the even (+), and  $\gamma = 2$  for the odd (-) states.

### B. NA-tensors

NA-tensors are the basic building blocks in the openNAMPS code. They are block-sparse tensors with a particular block structure that follows from the symmetry related quantum numbers which belong to the indices (or legs) of the NA-tensor. The data structure of NA-tensors is the following:

- NAtensors have  $N_{\text{leg}}$  legs. We refer to the legs by their names (strings) that are stored in the `NAtensor.leg_names` cell array.
- Legs have type (`NAtensor.leg_types`), that can be incoming ('i') and outgoing ('o'). Contracion rules are simple: incoming legs can be contracted only with outgoing ones.
- Blocks of the tensor are indexed by a list of irrep indices,  $[\Gamma_1, \Gamma_2, \dots, \Gamma_{N_{\text{rep}}}]$ , that we may call as the "key" of the block and which key contains a total of  $N_{\text{sym}} \cdot N_{\text{rep}}$  positive integer numbers.
- The data blocks ( $N_{\text{leg}}$ -dimensional arrays) are stored in `NAtensor.data` variable, that is a `containers.Map()` associative container (data is stored in `key`  $\rightarrow$  `block` pairs). The key of a block is just  $[\Gamma_1, \Gamma_2, \dots, \Gamma_{N_{\text{rep}}}]$  casted to `char` data type.
- Legs of the NAtensor depend on one or more irrep indices from the "key". The `NAtensor.dependencies` cell array contains the dependencies for each leg. For example, if dependencies of some leg are  $[d_1, d_2, \dots, d_m]$  that means that the leg depends on  $[\Gamma_{d_1}, \Gamma_{d_2}, \dots, \Gamma_{d_m}]$  irrep indices from the key. The order of dependencies is important. The different  $[\Gamma_{d_1}, \Gamma_{d_2}, \dots, \Gamma_{d_m}]$  lists characterize *sectors* of the leg.
- If we contract two tensors, the dependencies of the contracted legs are matched, i.e. only those blocks of the two tensors are contraced, where the  $[\Gamma_{d_1}, \Gamma_{d_2}, \dots, \Gamma_{d_m}]$  indices are the same. See also Refs. 2 and 3 for more details.

### Examples of NA-tensors

A simple example is the usual MPS tensor  $A_{m\sigma}^{m'}$  that rotates from the product basis  $|m\rangle \otimes |\sigma\rangle$  to the Schmidt-basis  $|m'\rangle$ ,

$$|m'\rangle = \sum_{m,\sigma} A_{m\sigma}^{m'} |m\rangle \otimes |\sigma\rangle \quad (2)$$

If we have symmetries, and the irrep indices of states  $m$ ,  $\sigma$ , and  $m'$  are  $\Gamma$ ,  $\Gamma^{\text{loc}}$ ,  $\Gamma'$ , respectively, then the MPS tensor can be written as an NA-tensor,  $A(\{\Gamma, \Gamma^{\text{loc}}, \Gamma'\})_{m\sigma}^{m'}$ , where the dependencies of the legs are  $\text{dep}(m) = [1]$ ,  $\text{dep}(\sigma) = [2]$ ,  $\text{dep}(m') = [3]$ , i.e leg  $m$  depends on the first, leg  $\sigma$  on the second, and leg  $m'$  on the third irrep index.

An other, more interesting example is the Clebsch-Gordan tensor, that is built from the generalazied Clebsch-Gordan coefficients,

$$(\Gamma_1, m_1; \Gamma_2, m_2 | \Gamma, M)_{\alpha} . \quad (3)$$

These amplitudes are used, when we search for irreducible representations in the product of two irreps  $\Gamma_1$  and  $\Gamma_2$ . Here  $\Gamma$  is the outgoing irrep index, and  $\alpha$  is the so called outer multiplicity that enumerates the multiplets

in the  $\Gamma_1 \times \Gamma_2$  product having the same  $\Gamma$  irrep index. These Clebsch-Gordan coefficients can be stored in the NA-tensor  $C(\{\Gamma_1, \Gamma_2, \Gamma\})_{m_1 m_2}^{M\alpha}$  in which  $\text{dep}(m_1) = [1]$ ,  $\text{dep}(m_2) = [2]$ ,  $\text{dep}(M) = [3]$ , while the outer multiplicity index has a triple dependency  $\text{dep}(\alpha) = [1, 2, 3]$ .

The main purpose of this document is the description of the code, the construction of NA-tensors and their algebraic properties are described in more details in Refs. 2 and 3.

### Handling NA-tensors in opeNAMPS

In this short paragraph basic code examples are shown in which we define an NA-tensor, we set some blocks of it, we perform simple algebraic transformations, and contractions. The full script is in the file `run_scripts/NAtensor_demo.m`, in the example the number of symmetries is `NO_OF_SYMS = 2;`.

Our tensor will have a dependency structure similar to the Clebsch-Gordan tensor:

$$\begin{aligned} T(\{\Gamma_1, \Gamma_2, \Gamma_3\})_a^c \eta_b \\ \text{dep}(a) = [1], \quad \text{dep}(b) = [2], \\ \text{dep}(c) = [3], \quad \text{dep}(\eta) = [1, 2, 3] \end{aligned} \quad (4)$$

The tensor has two incoming legs ('a' and 'b'), and two outgoing legs ('c' and 'eta'), its possible definition in the code reads as

```
T = NAtensor({'a','b','c','eta'}, ...
    {'i','i','o','o'}, ...
    {[1],[2],[3],[1,2,3]}, ...
    NO_OF_SYMS);
```

Here the first argument is the cell array of the leg names, the second cell array contains the leg types, the third one contains the dependencies, and the fourth is just the number of symmetries. It is important, that in the dependencies  $\Gamma$  labels are enumerated starting from 1 to  $m$ , with no missing numbers. ( $m$  is 3 in our case): it is mandatory to index irrep labels this way.

T is now an empty NAtensor, with no active (nonzero) blocks. We can set blocks with the `NTset_block` function:

```
T = NTset_block(T, ...
    {'a',1}, {'b',1}, {'c',1}}, ...
    {[2,3], [3,4], [4,5]}, ...
    {'a','b','c','eta'}, ...
    ones(4,6,8,2));
```

Here, the first argument is the NAtensor itself<sup>1</sup>, the second argument specifies the irrep positions (irrep is specified by a [legname - which dependency of the leg] pairs), in the third argument we give the values of irrep indices ( $\Gamma$  quantum numbers), in the fourth argument we specify the order of the legs in which the tensorblock array

<sup>1</sup> In the opeNAMPS we avoid to use classes because class function calls have a rather large overhead in MatLab.

(fifth argument, in the example a 4-order tensor filled with ones) is passed.

In the second argument it is obligatory to mark all the irrep labels once. However, the order, and the way how we refer to the irrep labels is irrelevant. The following code is equivalent with the one above,

```
T = NTset_block(T, ...
    {{'b',1},{'eta',3}, {'eta',1}}, ...
    {[3,4], [4,5], [2,3]}, ...
    {'eta','b','a','c'}, ...
    ones(2,6,4,8));
```

We note that the user has a large freedom in the order in which the irreps are specified (2nd and 3rd argument), and also in the order of legs (4th and 5th argument). It is important to note one issue about the dimensions of legs within blocks: the dimensions of legs must depend only on the  $\Gamma$  values of its dependency irrep labels. E.g. in our example the leg 'a' is 4 dimensional, while its dependency ( $\Gamma_1$ ) contains the quantum numbers [2,3]. We may set other blocks in which  $\Gamma_1$  is still [2,3], while  $\Gamma_2$  and  $\Gamma_3$  may differ from the values in the example above: the 'a' leg must be 4-dimensional also in these blocks. The code in its present version does not check these dimension consistencies by default.

NAtensor can be multiplied by scalars, can be added to each other, or can be complex conjugated

```
twoT = NTmult(T,2);    %2 * T
TplusT = NTadd(T,T);   % T + T
TminustwoT = NTsubtr(T,twoT); % T - 2 * T
minusT = NTneg(T);     % -T
conjT = NTconj(T);     % T*
```

We note that complex conjugation reverses the direction of the legs ('o'  $\leftrightarrow$  'i').

Tensors can also be contracted, i.e. incoming and outgoing legs can be connected to each other, if the sector structure (sector dimensions) of the legs are equivalent. For example the 'a' leg of T and conjT can be contracted. However, the result tensor would have two 'b', two 'c' and two 'eta' legs. To resolve this naming issue, we have to rename the legs of one of the tensors. The first solution uses NTprime\_all\_legs

```
%NTprime_all_legs: adds '~' to leg names
conjT_primed = NTprime_all_names(conjT);
TconjT = NTdot(T,conjT_primed, ...
    {'a'},{'a~'});
```

The result is an NAtensor with six legs: {'b','b~','c','c~','eta','eta~'}.

In the second solution we rename the legs by hand prior contraction. This solution is lengthier, but gives more freedom: we can choose arbitrary new names. In the following example we change the uncontracted leg names of conjT to {'d','e','xi'}

```
conjT_renamed = NTrename_legs(conjT, ...
    {'b','c','eta'},{'d','e','xi'});
```

```
TconjT = NTdot(T,conjT_renamed, ...
    {'a'},{'a'});
```

There is also a third solution: in NTdot we have an optional 5th argument called **renaming**. If we pass this, we can set the new names directly in NTdot().

### III. ABELIAN MPS AND TEBD

While the general non-Abelian routings can also be used for Abelian symmetries, there are also simpler Abelian routines prepared that can also be used. In these routines everything looks more or less like in a simple non-symmetric code, and only the simple fusion rules of the corresponding Abelian symmetry group are used.

#### A. The ABSITE structure

Necessary information about single Abelian sites are encoded in ABSITE structures. These structures contain basic information about the site, and contain also the local operators (in NAtensor structures) and their basic properties. For fermionic sites (modes), the ABSITE.fermions field is 1, and in that case the Jordan-Wigner phase operator ABSITE.operators('ph') must be defined. The NAtensor operators are collected in the ABSITE.operators() map container, and for fermionic sites the ABSITE.fermionicities() container sets the fermionic sign ( $\pm 1$ ) of the operator. The operators are represented by two-leg NAtensors  $O_{\bar{\tau}\tau}$ , where the 'tau' leg will be contracted with the 'tau' leg of the MPS wavefunction.

##### 1. Example: the $U(1)$ spinless fermion site

$U(1)$  symmetric fermion site can be defined by the ABSITE\_U1\_spinless\_fermion(U1sym,Q\_empty,Q\_occ) function. Here U1sym is the  $U(1)$  symmetry structure, generated by the generate\_U1\_symmetry([],[],[]) function, Q\_empty is the (integer) charge associated to the empty state, Q\_occ is the (integer) charge associated to the occupied state. In this case ABSITE.fermions == 1, i.e. we have fermions. The ABSITE.operators container has 6 operators defined whose names are {'id', 'f', 'fdag', 'fdag\_f', 'n', 'ph'}. Actually 'fdag\_f' and 'n' encode equal operators. The phase operator 'ph' is diagonal, its value is +1 for the empty, and -1 for the occupied state. The ABSITE.fermionicities are -1 for 'f' and 'fdag' operators, while +1 for the others.

## B. The ABMPS structures

The `AB_MPS` structures are used to store Abelian matrix product states. This Abelian structure is able to store both left-canonical, right-canonical, and mixed-canonical matrix product states. The `ABMPS.cut_position` value stores the position of the cut. Matrices for `pos <= ABMPS.cut_position` are left canonical, while for `pos > ABMPS.cut_position`. The left canonical matrices are stored in the `ABMPS.left_matrices` array, while `ABMPS.right_matrices` contains the right canonical matrices. `ABMPS.schmidt` contains the Schmidt-tensor at the cut, while `ABMPS.schmidt_list` contains the Schmidt-tensors for every bond. The latter list is used by ABTEBD routines, that work for fully left canonical MPS.

The MPS matrices (both left and right canonical ones) have three legs `{'t_in','tau','t_out'}` and three quantum numbers (representation labels) associated with the three legs. In case of left canonical tensors the `'t_in'` name denotes the leg incoming from left, while the `'t_out'` legs goes out to the right direction. In case of right canonical tensors the incoming leg is in the right, and the outgoing goes to left. The `'tau'` leg is always incoming. The Schmidt-tensor has always two incoming legs, `'t_left'` and `'t_right'`.

## C. The simple ABTEBD

To start ABTEBD simulations an `ABTEBD_env` structure must be prepared. This 'ABTEBD environment' variable contains all the necessary information about our ABTEBD simulation: the `ABTEBD_env.StateMPS` wavefunction, the array of symmetry structures (`ABTEBD_env.Symmetries`), some basic information (chain length, number of symmetries, fermions (true or false)). The `ABTEBD_env.infinite` logical variable is true if the infinite chain TEBD algorithm is used[5], and false if we treat a finite chain. The nearest neighbor (2-site) Hamiltonians are collected in the `ABTEBD_env.TwoSiteHlist` array. These NAtensors have always four legs: `{'tau_1', 'tau_2', 'tau_1~', 'tau_2~'}`, where the first two legs are contracted with the `'tau'` legs of the MPS. The exponentialized two site evolvers are collected

in the `ABTEBD_env.Ulist` and `ABTEBD_env.Uhalflist`, the latter contains the evolvers for  $\Delta t/2$ , and is used by the 2nd order Trotter evolver. The time step  $\Delta t$  is a parameter of the function that creates the `ABTEBD_env` variable.

### 1. Example: the $U(1)$ spinless fermion ABTEBD

A good starting example for Abelian TEBD is again the  $U(1)$  spinless fermion model with nearest neighbor interactions. In the `run_scripts/` folder the `ABTEBD_U1_nearest_neighbor_spinless_fermion` standalone script simulates the real time dynamics of the model

$$\hat{H} = \sum_i \left( -c_i^\dagger c_{i+1} - c_{i+1}^\dagger c_i + V n_i n_{i+1} \right) \quad (5)$$

starting from a charge density wave state `|...010101...>`. In the script first we define the parameters of the model and the simulation, then the  $U(1)$  symmetry is defined. As we want to use the infinite lattice algorithm, it is essential to define the site charges in a way, that the total charge remains zero. Therefore `Q_EMPTY = -1` and `Q_OCCUPIED = +1` is used. After fixing these values, the initial MPS state is set. Equipped with all these variables we can create the `ABTEBD_env` variable using the `MOD_ABTEBD_U1_nearest***_init` function. Below this line the simulation is performed and data is written to the screen.

If one would like to define an other model, a model initializer function (like `MOD_ABTEBD_U1_nearest***_init`) must be prepared. Once the `ABTEBD_env.StateMPS` and the `ABTEBD_env.Ulist` is set, the TEBD simulation is ready to run.

## IV. NON-ABELIAN MPS AND TEBD

### A. Non-Abelian SITE structures

### B. Non-Abelian MPS

### C. Non-Abelian TEBD

---

[1] GPLv3

[2] NATEBD paper

[3]

[4] Vidal 2004

[5] Vidal 2007

[6] Schollwöck 2011