mpnum Documentation

Release git

Daniel Suess and Milan Holzäpfel

Jan 19, 2018

Documentation

CHAPTER 1

A matrix product representation library for Python

mpnum is a flexible, user-friendly, and expandable toolbox for the matrix product state/tensor train tensor format.

1.1 Introduction

mpnum is a flexible, user-friendly, and expandable toolbox for the matrix product state/tensor train tensor format. It is available under the BSD license at mpnum on Github. mpnum provides:

- support for well-known matrix product representations, such as:
  - matrix product states (*MPS*), also known as tensor trains (TT)
  - matrix product operators (*MPO*)
  - local purification matrix product states (*PMPS*)
  - arbitrary matrix product arrays (*MPA*)
- arithmetic operations: addition, multiplication, contraction etc.
- compression, canonical forms, etc. (see `compress()`, `canonicalize()`)
- finding extremal eigenvalues and eigenvectors of MPOs (see `eig()`)

In this introduction, we discuss mpnum's basic data structure, the `MPArray` (MPA). If you are familiar with matrix product states and want to see mpnum in action, you can skip to the IPython notebook `mpnum_intro.ipynb` (view mpnum_intro.ipynb on Github).

Contents

1.1.1 Matrix product arrays

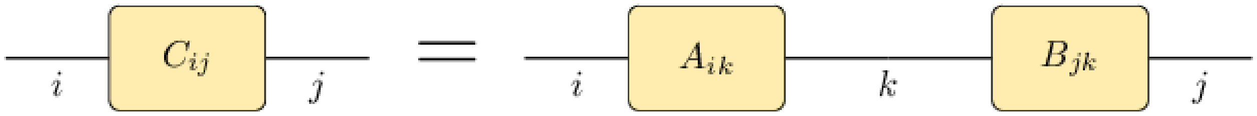
The basic data structure of mpnum is the class *`mpnum.mparray.MPArray`*. It represents tensors in matrix-product form in an opaque manner while providing the user with a high-level interface similar to numpy's `ndarray`. Special cases of MPAs include matrix-product states (MPS) and operators (MPOs) used in quantum physics.

Graphical notation

Operations on tensors such as contractions are much easier to write down using graphical notation [*Sch11*, Figure 38]. A simple case of of a tensor contraction is the product of two matrices:

$$C = AB^T$$

We represent this tensor contraction with the following figure:



Each of the tensors A, B and C is represented by one box. All the tensors have two indices (as they are matrices), therefore there are two lines emerging from each box, called *legs*. Connected legs indicate a contraction. The relation between legs on the left and right hand sides of the equality sign is given by their position. In this figure, we specify the relation between the indices in a formula like B_{kl} and the individual lines in the figure by giving specifying the name of each index on each line.

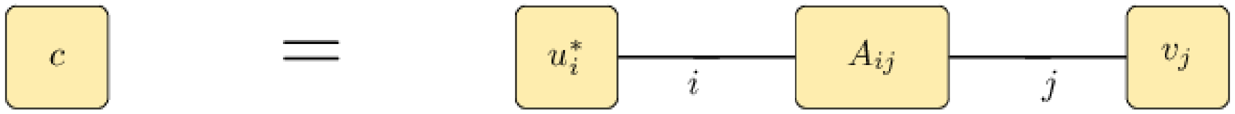
In this simple case, the figure looks more complicated than the formula, but it contains complete information on how all indices of all tensors are connected. To be fair, we should mention the indices in the formula as well:

$$C_{ij} = \sum_k A_{ik} B_{jk}$$

Another simple example is the following product of two vectors and a matrix:

$$c = u^\dagger A v = \sum_{ij} u_i^* A_{ij} v_j$$

This formula is represented by the following figure:

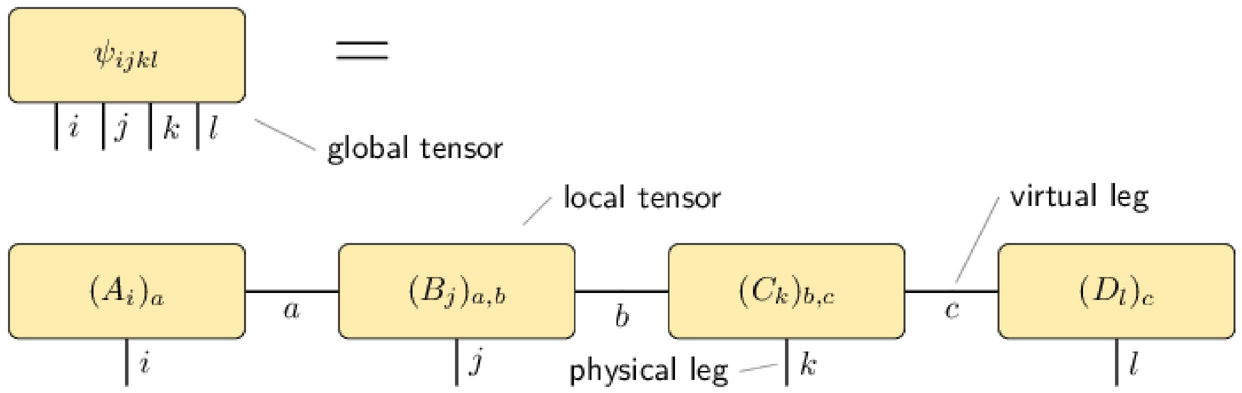


Matrix product states (MPS)

The matrix product state representation of a state $|\psi\rangle$ on four subsystems is given by

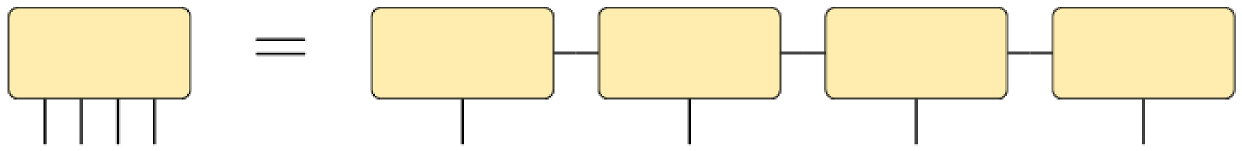
$$\langle ijkl|\psi\rangle = \psi_{ijkl} = A_i B_j C_k D_l$$

where each $A_i \in \mathbb{C}^{1\times D}$; $B_j, C_k \in \mathbb{C}^{D\times D}$ and $D_l \in \mathbb{C}^{D\times 1}$ (reference: e.g. *[Sch11]*; *exact definition*). This construction is also known as *tensor train* and it is given by the following simple figure:

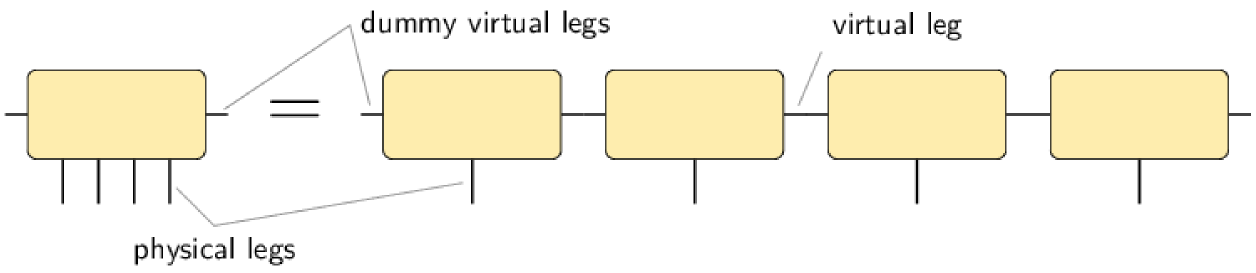


We call ψ a *global tensor* and we call the MPS matrices A_i, B_j etc. which are associated to a certain subsystem *local tensors*. The legs/indices $i, j, \dots$ of the original tensor $|\psi\rangle$ are called *physical legs*. The additional legs in the matrix product representation are called *virtual legs*. The dimension (size) of the virtual legs are called the *representation ranks* or *compression ranks*. In the physics literature, the virtual legs are often called *bonds* and the representation ranks are called *bond dimensions*.

Very often, we can omit the labels of all the legs. The figure then becomes very simple:



As explained in the next paragraph on MPOs, we usually add *dummy virtual legs* of size 1 to our tensors:

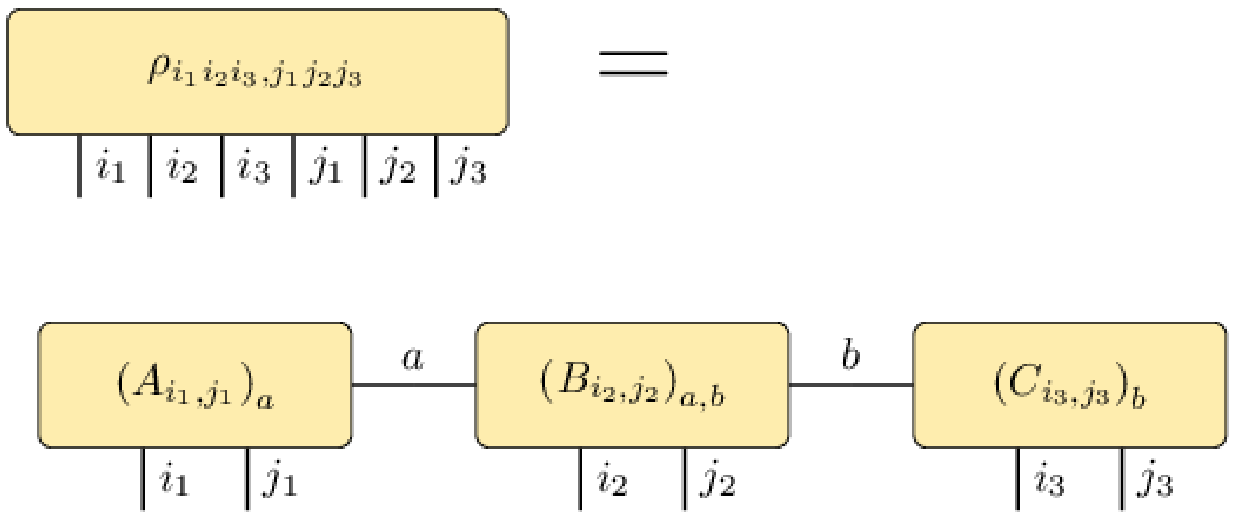


Matrix product operators (MPO)

The matrix product operator representation of an operator ρ on three subsystems is given by

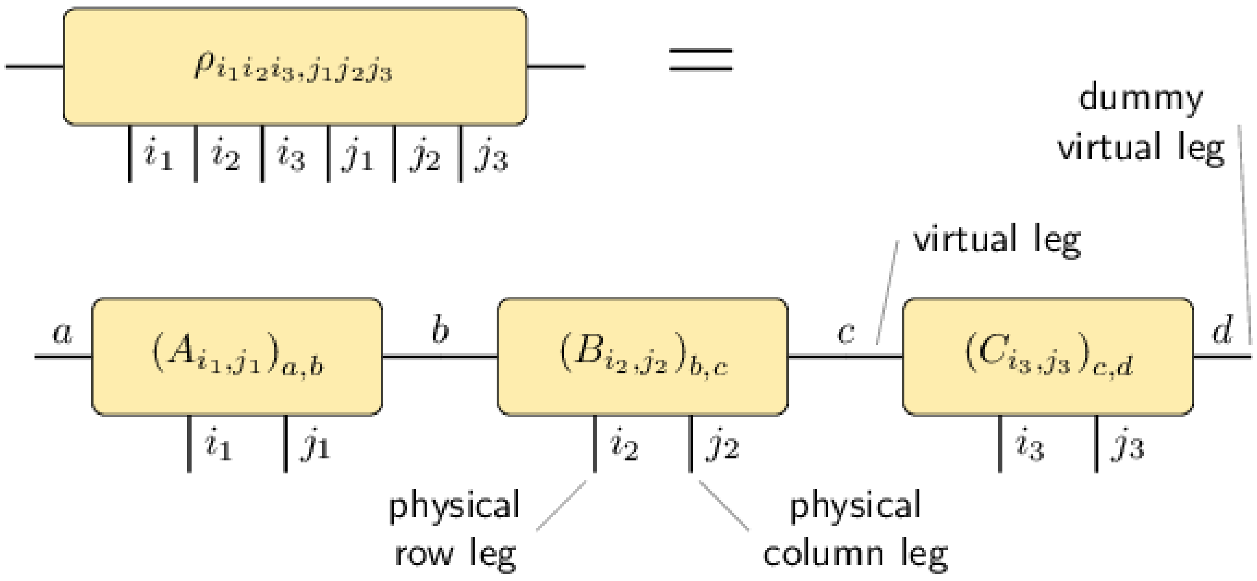
$$\langle i_1 i_2 i_3|\rho|j_1 j_2 j_3\rangle = \rho_{i_1 i_2 i_3, j_1 j_2 j_3} = A_{i_1 j_1} B_{i_2 j_2} C_{i_3 j_3}$$

where the $A_{i_1 j_1}$ are row vectors, the $B_{i_2 j_2}$ are matrices and the $C_{i_3 j_3}$ are column vectors (reference: e.g. *[Sch11]*; *exact definition*). This is represented by the following figure:


$\rho_{i_1 i_2 i_3, j_1 j_2 j_3}$
i_1 i_2 i_3 j_1 j_2 j_3
=
$(A_{i_1,j_1})_a$
a
$(B_{i_2,j_2})_{a,b}$
b
$(C_{i_3,j_3})_b$
i_1 j_1 i_2 j_2 i_3 j_3

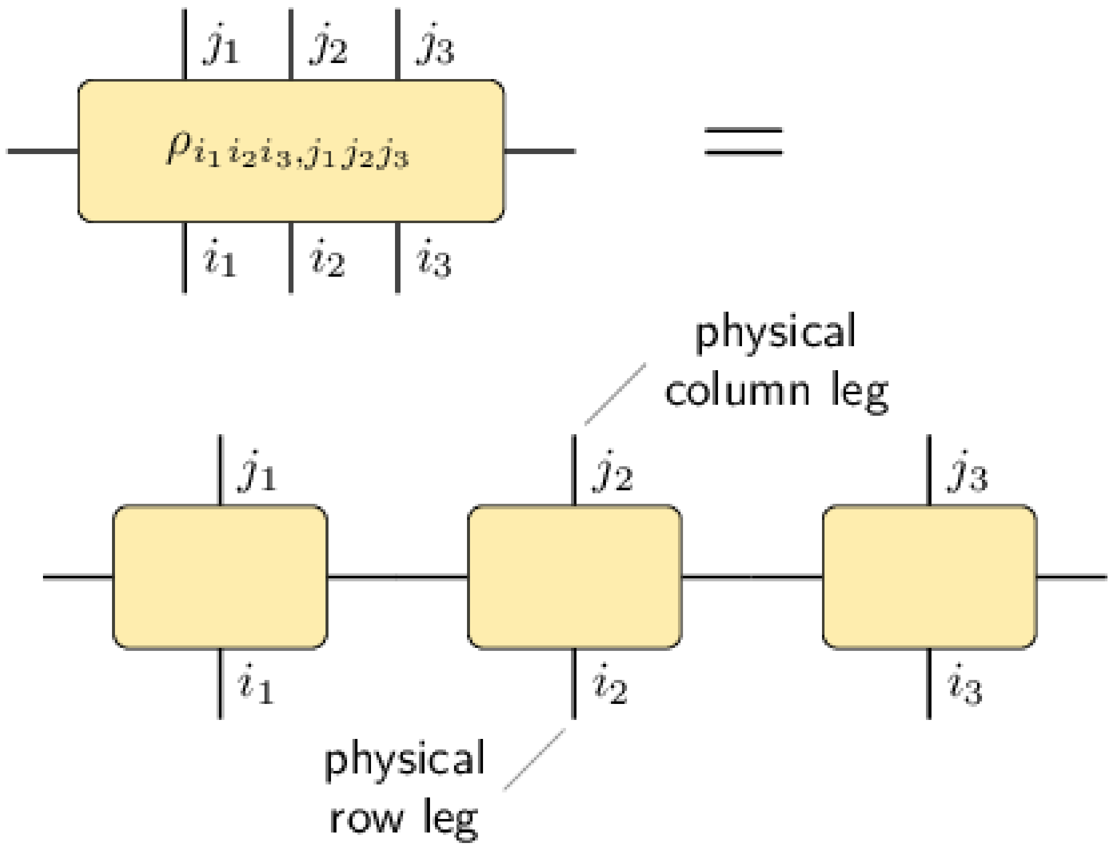
Be aware that the legs of ρ are not in the order $i_1 i_2 i_3 j_1 j_2 j_3$ (called *global order*) which is expected from the expression $\langle i_1 i_2 i_3 | \rho | j_1 j_2 j_3 \rangle$ and which is obtained by a simple reshape of the matrix ρ into a tensor. Instead, the order of the legs of ρ must match the order in the MPO construction, which is $i_1 j_1 i_2 j_2 i_3 j_3$. We call this latter order *local order*. The functions `global_to_local` and `local_to_global` can convert tensors between the two orders.

In order to simplify the implementation, it is useful to introduce *dummy virtual legs* with index size 1 on the left and the right of the MPS or MPO chain:

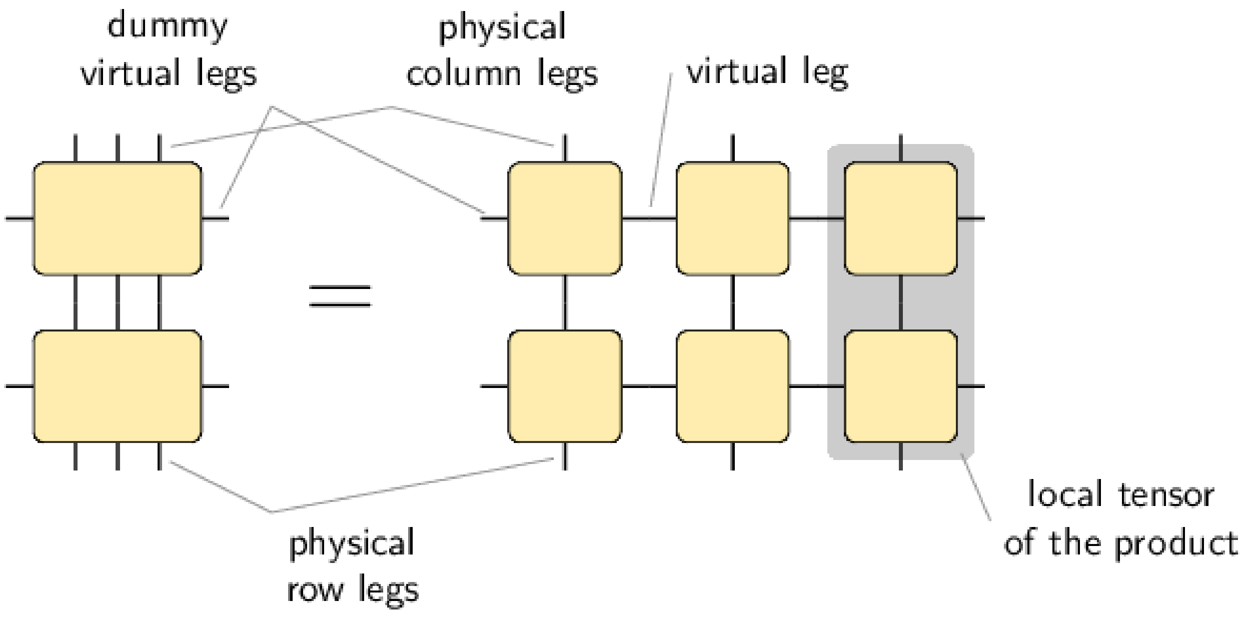


With these dummy virtual legs, all the tensors in the representation have exactly two virtual legs.

It is useful to draw the physical column indices upward from the global and local tensors while leaving the physical row indices downward:



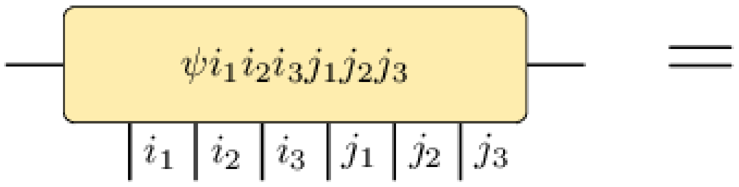
With this arrangement, we can nicely express a product of two MPOs:

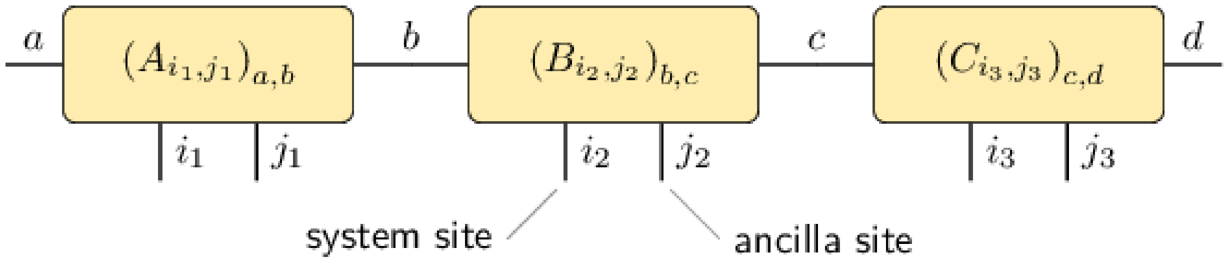


This figure tells us how to obtain the local tensors which represent the product: We have to compute new tensors as indicated by the shaded area. The figure also tells us that the representation rank of the result is the product of the representation rank of the two individual MPO representations.

Local purification form MPS (PMPS)

The local purification form matrix product state representation (PMPS or LPMPS) is defined as follows:

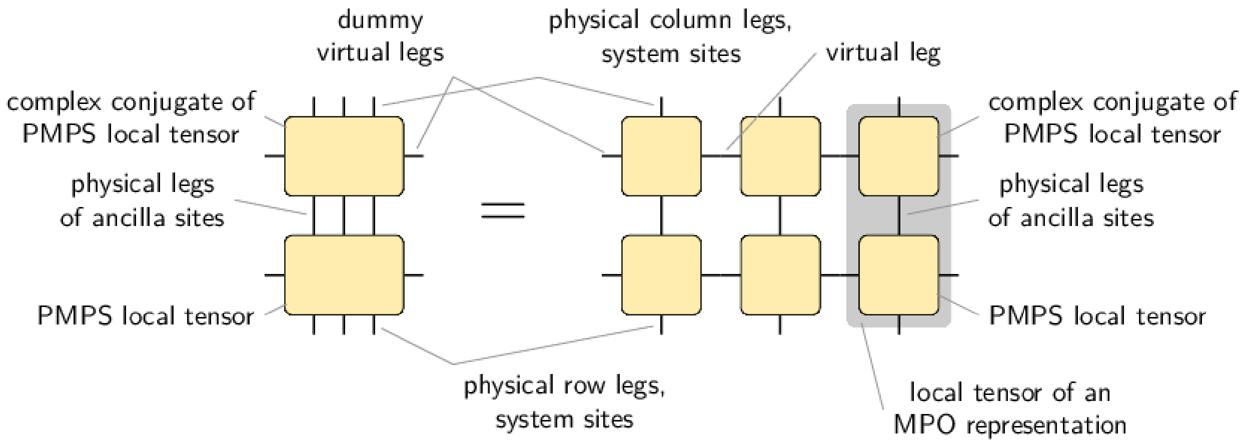




Here, all the i indices are actual sites and all the j indices are ancilla sites used for the purification (reference: e.g. [Cue13]; *exact definition*). The non-negative operator described by this representation is given by

$$\rho = \mathrm{tr}_{j_1 j_2 j_3}(|\psi\rangle\langle\psi|)$$

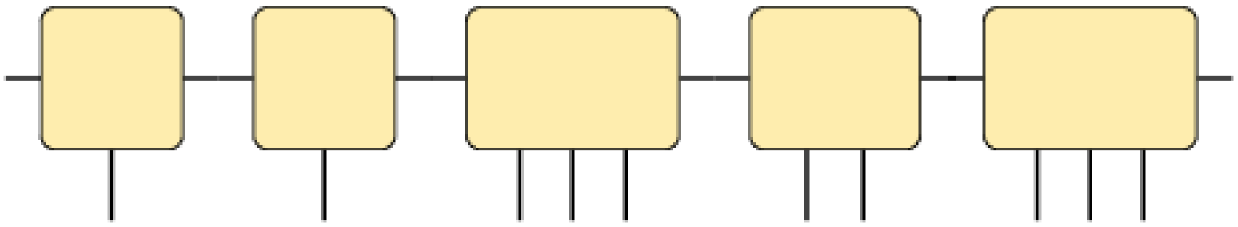
The following figure describes the relation:



It also tells us how to convert a PMPS representation into an MPO representation and how the representation rank changes: The MPO representation rank is the square of the PMPS representation rank.

General matrix product arrays

Up to now, all examples had the same number of legs on each site. However, the `MPArray` is not restricted to these cases, but can be used to express any local structure. An example of a inhomogenous tensor is shown in the following figure:



1.1.2 Next steps

The Jupyter notebook `mpnum_intro.ipynb` in the folder `Notebooks` provides an interactive introduction on how to use `mpnum` for basic MPS, MPO and MPA operations. Its rendered version can also be viewed in the *Introductory Notebook to mpnum*. If you open the notebook on your own computer, it allows you to run and modify all the commands interactively (more information is available in the section "Jupyter Notebook Quickstart" of the Jupyter documentation).

1.1.3 References

1.2 API reference

1.2.1 Module overview

- *`mpnum.mparray`*: Basic matrix product array (MPA) routines and compression
- `mppnum.mpstruct`: Underlying structure of MPAs to manage the local tensors
- *`mpnum.mpsmpo`*: Convert matrix product state (MPS), matrix product operator (MPO) and locally purifying MPS (PMPS) representations and compute local reduced states.
- *`mpnum.factory`*: Generate random, MPS, MPOs, MPDOs, MPAs, etc.
- *`mpnum.linalg`*: Compute the smallest eigenvalues & vectors of MPOs
- *`mpnum.special`*: Optimized versions of some routines for special cases
- *`mpnum.povm`*: Matrix product representation of Positive operator valued measures (POVM)
- *`mpnum.povm.localpovm`*: Pauli-like POVM on a single site
- *`mpnum.povm.mppovm`*: Matrix product POVM based on the Pauli-like POVM

1.2.2 `mparray`

Core MPArray data structure & general purpose functions

Todo: single site MPAs – what is left?

Todo: Local tensor ownership – see MPArray class comment

Todo: Possible optimization:

- replace integer-for loops with iterator (not obviously possible everwhere)
- replace internal structure as list of arrays with lazy generator of arrays (might not be possible, since we often iterate both ways!)
- more in place operations for addition, subtraction, multiplication

---

Todo: Replace all occurences of self._ltens with self[...] or similar & benchmark. This will allow easier transition to lazy evaluation of local tensors

---

class `mpnum.mparray.`**`MPArray`**(*ltens*)

Bases: `object`

Efficient representation of a general N-partite array A in matrix product form with open boundary conditions:

$$A_{i_1,\dots,i_N} = A^{[1]}_{i_1} \dots A^{[N]}_{i_N} \tag{1.1}$$

where the $A^{[k]}$ are local tensors (with N legs/dimensions). The matrix products in (1.1) are taken with respect to the left- and right-most legs (virtual indices) and the multi-index i_k corresponds to the true local legs. Open boundary conditions imply that $A^{[1]}$ is 1-by-something and $A^{[N]}$ is something-by-1.

For the details on the data model used for storing the local tensors see *`mpstruct.LocalTensors`*.

Todo: As it is now, e.g. `__imul__()` modifies items from `self._ltens`. This requires e.g. *`chain()`* to take copies of the local tensors. The data model seems to be that an *MPArray* instance owns its local tensors and everyone else, including each new *MPArray* instance, must take copies. Is this correct?

`__init__`(*ltens*)

Parameters **`ltens`** – local tensors as instance of *`mpstruct.LocalTensors`* or simply as a list of `numpy.ndarray` in the format described at *`mpstruct.LocalTensors`*

`__len__`()

Returns the number of sites

`T`

Transpose (=reverse order of) physical legs on each site. See also *`transpose()`* for more fine grained control.

`adj`()

Hermitian adjoint. Equivalent to self.T.conj().

`axis_iter`(*axes=0*)

Returns an iterator yielding Sub-MPArrays of `self` by iterating over the specified physical axes.

Example: If `self` represents a bipartite (i.e. length 2) array with 2 physical dimensions on each site `A[(k,l), (m,n)]`, `self.axis_iter(0)` is equivalent to:

```
(A[(k, :), (m, :)] for m in range(...) for k in range(...))
```

Parameters **`axes`** – Iterable or int specifiying the physical axes to iterate over (default 0 for each site)

Returns Iterator over *`MPArray`*

`canonical_form`

See *`mpstruct.LocalTensors.canonical_form`*

canonicalize(*left=None*, *right=None*)

Brings the MPA to canonical form in place [*Sch11*, Sec. 4.4]

Note that we do not support full left- or right-canonicalization. In general, the right- (left- resp.)most local tensor cannot be in a canonical form since at least one local tensor must be non-normalized.

The following values for *left* and *right* will be needed most frequently:

Left-/Right- canonicalize:	Do Nothing	To canonicalize maximally
left	`None`	`'afull'`, `len(self) - 1`
right	`None`	`'afull'`, `1`

`'afull'` is short for "almost full" (we do not support normalizing the outermost sites).

Arbitrary integer values of `left` and `right` have the following meaning:

- `self[:left]` will be left-normalized
- `self[right:]` will be right-normalized

In accordance with the last table, the special values `None` and `'afull'` will be replaced by the following integers:

	`None`	`'afull'`
left	`0`	`len(self) - 1`
right	`len(self)`	`1`

Exceptions raised:

- Integer argument too large or small: `IndexError`
- Matrix would be both left- and right-normalized: `ValueError`

compress(*method='svd'*, ***kwargs*)

Compress `self`, modifying it in-place.

Let $|u\rangle$ the original vector and let $|c\rangle$ the compressed vector. The compressions we return have the property (cf. [*Sch11*, Sec. 4.5.2])

$$\langle u|c\rangle = \langle c|c\rangle \in (0, \infty).$$

It is a useful property because it ensures

$$\min_{\phi \in \mathbb{R}} \|u - re^{i\phi}c\| = \|u - rc\|, \quad r > 0,$$
$$\min_{\mu \in \mathbb{C}} \|u - \mu c\| = \|u - c\|$$

for the vector 2-norm. Users of this function can compute norm differences between u and a normalized c via

$$\|u - rc\|^2 = \|u\|^2 + r(r-2)\langle u|c\rangle, \quad r \geq 0.$$

In the special case of $\|u\| = 1$ and $c_0 = c/\|c\|$ (pure quantum states as MPS), we obtain

$$\|u - c_0\|^2 = 2(1 - \sqrt{\langle u|c\rangle})$$

Returns Inner product $\langle u|c\rangle \in (0, \infty)$ of the original u and its compression c.

Parameters **method** – `'svd'` or `'var'`

Parameters for `'svd'`:

Parameters

- **`rank`** – Maximal rank of the result. (default: `None`)
- **`relerr`** – Maximal fraction of discarded singular values. Default `0`. If both rank and relerr are given, the smaller resulting rank is used.
- **`direction`** – `'right'` (sweep from left to right), `'left'` (inverse) or `None` (choose depending on canonicalization). (default: `None`)
- **`canonicalize`** – SVD compression works best when the MPA is brought into full left-/right-cannonical form first. This variable determines whether cannonical form is enforced before compression (default: `True`)
- **`svdfunc`** – Which SVD function to use during the compression. It should follow the conventios of *`truncated_svd()`*, which is also the default choice. In some circumstances, a partial SVD as provided by `scipy.sparse.linalg.svds()` or a randomized SVD such as *`randomized_svd()`* might speed up computations with no or little loss of accuracy.

Parameters for `'var'`:

Parameters

- **`rank`** – Maximal rank for the result. Either `startmpa` or `rank` is required.
- **`num_sweeps`** – Number of variational sweeps (required).
- **`startmpa`** – Start vector, also fixes the rank of the result. Default: Random, with same norm as self.
- **`randstate`** – `numpy.random.RandomState` instance used for random start vector. (default: `numpy.random`).
- **`var_sites`** – Number of connected sites to be varied simultaneously (default 1)

Increasing `var_sites` makes it less likely to get stuck in a local minimum but is generally slower.

References:

- `'svd'`: Singular value truncation, [*Sch11*, Sec. 4.5.1]
- `'var'`: Variational compression, [*Sch11*, Sec. 4.5.2]

`compression`(*method='svd'*, ***kwargs*)

Return a compression of `self`. Does not modify `self`.

Parameters: See *`compress()`*.

Returns `(compressed_mpa, overlap)` where `overlap` is the inner product returned by *`compress()`*.

`conj`()

Complex conjugate

`copy`()

Returns a deep copy of the MPA

`dtype`

Returns the dtype that should be returned by `to_array`

dump (*target*)

Serializes MPArray to `h5py.Group`. Recover using *`load()`*.

Parameters **`target`** – `h5py.Group` the instance should be saved to or path to h5 file (it's then serialized to /)

`classmethod from_array` (*array*, *ndims=None*, *has_virtual=False*)

Create MPA from array in local form.

See `mpnum.tools.global_to_local()` for global vs. local form.

Computes the (exact up to numerical accuracy) representation of *array* as MPA with open boundary conditions, i.e. rank 1 at the boundary. This is done by factoring off the left and the "physical" legs from the rest of the tensor via QR decomposition and working its way through the tensor from the left. This yields a left-canonical representation of *array*. [*Sch11*, Sec. 4.3.1]

The result is a chain of local tensors with `ndims` physical legs at each location and has `array.ndim // ndims` number of sites (assuming `ndims` has the same value for each site)

`has_virtual` allows to treat a part of the linear chain of an MPA as MPA as well. The rank on the left and right can be different from one and different from each other in that case. This is useful to apply SVD compression only to part of an MPA.

Parameters

- **`array`** (*`np.ndarray`*) – Dense array with global structure `array[(i0), ..., (iN)]`, i.e. the legs which are factorized into the same factor are already adjacent. (For me details see `tools.global_to_local()`)
- **`ndims`** – Number of physical legs per site (default array.ndim) or iterable over number of physical legs
- **`has_virtual`** (*`bool`*) – `True` if array already has indices for the left and right virtual legs

`classmethod from_array_global` (*array*, *ndims=None*, *has_virtual=False*)

Create MPA from array in global form.

See `mpnum.tools.global_to_local()` for global vs. local form.

Parameters and return value: See `from_array()`. `has_virtual=True` is not supported yet.

`classmethod from_kron` (*factors*)

Returns the (exact) representation of an n-fold Kronecker (tensor) product as MPA with ranks 1 and n sites.

Parameters **`factors`** – A list of arrays with arbitrary number of physical legs

Returns The kronecker product of the factors as MPA

`get` (*indices*, *astype=None*)

Returns the current MPA but with the first index at each sites evaluated at the corresponding value of `indices`

Parameters **`indices`** – Length `len(self)` sequence of index values for first physical leg at each site

Returns `type(self)` object

`group_sites` (*sites_per_group*)

Group several MPA sites into one site.

The resulting MPA has length `len(self) // sites_per_group` and `sites_per_group * self.ndims[i]` physical legs on site `i`. The physical legs on each sites are in local form.

Parameters **`sites_per_group`** (*`int`*) – Number of sites to be grouped into one

Returns An MPA with `sites_per_group` fewer sites and more ndims

leg2vleg(*pos*)
Performs the inverse operation to *vleg2leg()*.

Parameters **pos** – Number of the virtual to perform the transformation

Returns read-only MPA with transformed virtual

Todo: More appropriate naming for this functions?

---

classmethod load(*source*)
Deserializes MPArray from `h5py.Group`. Serialize using *dump()*.

Parameters **target** – `h5py.Group` containing serialized MPArray or path to a single h5 File containing serialized MPArray under /

lt

ndims
Tuple of number of legs per site

pad_ranks(*rank=None*, *force_rank=False*)
Increase rank by padding with zeros

This function is useful to prepare initial states for variational compression. E.g. for a five-qubit pure state with ranks *(2, 2, 4, 2)* it is desirable to increase the ranks to *(2, 4, 4, 2)* before using it as an initial state for variational compression.

Parameters

- **rank** (*int*) – Increase rank to this value, use `max(self.rank)` if `None` (default: `None`)
- **force_rank** – Use full rank even at the beginning and end of the MPS. See *full_rank()* for more details. (default: `False`)

Returns MPA representation of the same array with padded rank

ranks
Tuple of ranks

ravel()
Flatten the MPA to an MPS, shortcut for `self.reshape((-1,))`

reshape(*newshapes*)
Reshape physical legs in place.

Use *shape* to obtain the shape of the physical legs.

Parameters **newshapes** – A single new shape or a list of new shape. Alternatively, you can pass 'prune' to get rid of all legs of dimension 1.

Returns Reshaped MPA

Todo: Why is this here? What's wrong with the purne function?

---

reverse()

shape
List of tuples with the dimensions of each tensor leg at each site

singularvals()

Return singular values of `self` for all bipartitions

Returns Iterate over bipartitions with 1, 2, … len(self) - 1 sites on the left hand side. Yields a `np.ndarray` containing singular values for each bipartition.

Note: May decrease the rank (without changing the represented tensor).

size

Returns the number of floating point numbers used to represent the MPArray

```
>>> from .factory import zero
>>> zero(sites=3, ldim=4, rank=3).lt.shape
((1, 4, 3), (3, 4, 3), (3, 4, 1))
>>> zero(sites=3, ldim=4, rank=3).size
60
```

split(*pos*)

Splits the MPA into two by transforming the virtual legs into local legs according to `vleg2leg()`.

Parameters **pos** – Number of the virtual to perform the transformation

Returns (mpa_left, mpa_right)

split_sites(*sites_per_group*)

Split MPA sites into several sites.

The resulting MPA has length `len(self) * sites_per_group` and `self.ndims[i] // sites_per_group` indices on site i.

Parameters **sites_per_group** (*int*) – Split each site in that many sites

Returns An mpa with `sites_per_group` more sites and fewer `ndims`

sum(*axes=None*)

Element-wise sum over physical legs

Parameters **axes** – Physical legs to sum over

axes can have the following values:

- Sequence of length zero: Sum over nothing
- Sequence of (sequences or None): `axes[i]` specifies the physical legs to sum over at site `i`; `None` sums over all physical legs at a site
- Sequence of integers: `axes` specifies the physical legs to sum over at each site
- Single integer: Sum over physical leg `axes` at each site
- `None`: Sum over all physical legs at each site

To not sum over any axes at a certain site, specify the empty sequence for that site.

to_array()

Return MPA as array in local form.

See `mpnum.tools.global_to_local()` for global vs. local form.

Returns ndarray of shape `sum(self.shape, ())`

Note: Full arrays can require much more memory than MPAs. (That's why you are using MPAs, right?)

to_array_global()

Return MPA as array in global form.

See `mpnum.tools.global_to_local()` for global vs. local form.

Returns ndarray of shape `sum(zip(*self.shape, ()))`

See *`to_array()`* for more details.

transpose(*axes=None*)

Transpose (=reverse order of) physical legs on each site

Parameters **axes** – New order of the physical axes. If `None` is passed, we reverse the order of the legs on each site. (default `None`)

```
>>> from .factory import random_mpa
>>> mpa = random_mpa(2, (2, 3, 4), 2)
>>> mpa.shape
((2, 3, 4), (2, 3, 4))
>>> mpa.transpose((2, 0, 1)).shape
((4, 2, 3), (4, 2, 3))
```

vleg2leg(*pos*)

Transforms the virtual leg between site `pos` and `pos + 1` into local legs at those sites. The new leg will be the rightmost one at site `pos` and the leftmost one at site `pos + 1`. The new rank is 1.

Also see *`leg2vleg()`*.

Parameters **pos** – Number of the virtual to perform the transformation

Returns MPA with transformed virtual

Todo: More appropriate naming for this functions?

`mpnum.mparray.`**dot**(*mpa1*, *mpa2*, *axes=(-1, 0)*, *astype=None*)

Compute the matrix product representation of the contraction of a and `b` over the given axes. [*Sch11*, Sec. 4.2]

Parameters

- **mpa2** (*mpa1,*) – Factors as MPArrays
- **axes** – Tuple `(ax1, ax2)` where `ax1` (`ax2`) is a single physical leg number or sequence of physical leg numbers referring to `mpa1` (`mpa2`). The first (second, etc) entries of `ax1` and `ax2` will be contracted. Very similar to the `axes` argument for `numpy.tensordot()`. (default: `(-1, 0)`)

Note: Note that the default value of `axes` is different compared to `numpy.tensordot()`.

Parameters **astype** – Return type. If `None`, use the type of `mpa1`

Returns Dot product of the physical arrays

`mpnum.mparray.`**inject**(*mpa*, *pos*, *num=None*, *inject_ten=None*)

Interleaved chain product of an MPA and a rank 1 MPA

Return the chain product between mpa and `num` copies of the local tensor `inject_ten`, but place the copies of `inject_ten` before site `pos` inside or outside `mpa`. You can also supply `num = None` and a sequence of local tensors. All legs of the local tensors are interpreted as physical legs. Placing the local tensors at the beginning or end of `mpa` using `pos = 0` or `pos = len(mpa)` is also supported, but *`chain()`* is preferred for that as it is a much simpler function.

If `inject_ten` is omitted, use a square identity matrix of size `mpa.shape[pos][0]`. If `pos == len(mpa)`, `mpa.shape[pos - 1][0]` will be used for the size of the matrix.

Parameters

- **`mpa`** – An MPA.
- **`pos`** – Inject sites into the MPA before site `pos`.
- **`num`** – Inject `num` copies. Can be `None`; in this case `inject_ten` must be a sequence of values.
- **`inject_ten`** – Physical tensor to inject (if omitted, an identity matrix will be used; cf. above)

Returns The chain product

`pos` can also be a sequence of positions. In this case, `num` and `inject_ten` must be either sequences or `None`, where `None` is interpreted as `len(pos) * [None]`. As above, if `num[i]` is `None`, then `inject_ten[i]` must be a sequence of values.

`mpnum.mparray.`**`inner`**`(`*mpa1*, *mpa2*`)`

Compute the inner product <*mpa1*|*mpa2*>. Both have to have the same physical dimensions. If these represent a MPS, `inner(...)` corresponds to the canoncial Hilbert space scalar product. If these represent a MPO, `inner(...)` corresponds to the Frobenius scalar product (with Hermitian conjugation in the first argument)

Parameters

- **`mpa1`** – MPArray with same number of physical legs on each site
- **`mpa2`** – MPArray with same physical shape as mpa1

Returns <mpa1|mpa2>

`mpnum.mparray.`**`local_sum`**`(`*mpas*, *embed_tensor=None*, *length=None*, *slices=None*`)`

Embed local MPAs on a linear chain and sum as MPA.

We return the sum over *`embed_slice(length, slices[i], mpas[i], embed_tensor)`* as MPA.

If `slices` is omitted, we use *`regular_slices(length, width, offset)`* with `offset = 1`, `width = len(mpas[0])` and `length = len(mpas) + width - offset`.

If `slices` is omitted or if the slices just described are given, we call `_local_sum_identity()`, which gives a smaller virtual dimension than naive embedding and summing.

Parameters

- **`mpas`** – List of local MPAs.
- **`embed_tensor`** – Defaults to square identity matrix (see `_embed_ltens_identity()` for details)
- **`length`** – Length of the resulting chain, ignored unless slices is given.
- **`slices`** – `slice[i]` specifies the position of `mpas[i]`, optional.

Returns An MPA.

`mpnum.mparray.`**`localouter`**(*a*, *b*)

Computes the tensor product of $a \otimes b$ locally, that is when a and b have the same number of sites, the new local tensors are the tensorproducts of the original ones.

Parameters

- **a** (`MPArray`) – *MPArray*
- **b** (`MPArray`) – *MPArray* of same length as `a`

Returns Tensor product of `a` and `b` in terms of their local tensors

`mpnum.mparray.`**`norm`**(*mpa*)

Computes the norm (Hilbert space norm for MPS, Frobenius norm for MPO) of the matrix product operator. In contrast to `mparray.inner`, this can take advantage of the canonicalization

WARNING This also changes the MPA inplace by normalizing.

Parameters **mpa** – MPArray

Returns l2-norm of that array

`mpnum.mparray.`**`normdist`**(*mpa1*, *mpa2*)

More efficient version of norm(mpa1 - mpa2)

Parameters

- **mpa1** – MPArray
- **mpa2** – MPArray

Returns l2-norm of mpa1 - mpa2

`mpnum.mparray.`**`chain`**(*mpas*, *astype=None*)

Computes the tensor product of MPAs given in `*args` by adding more sites to the array.

Parameters

- **mpas** – Iterable of MPAs in the order as they should appear in the chain
- **astype** – dtype of the returned MPA. If `None`, use the type of the first MPA.

Returns MPA of length `len(args[0]) + ... + len(args[-1])`

Todo: Make this canonicalization aware

Todo: Raise warning when casting complex to real dtype

`mpnum.mparray.`**`partialdot`**(*mpa1*, *mpa2*, *start_at*, *axes=(-1, 0)*)

Partial dot product of two MPAs of inequal length.

The shorter MPA will start on site `start_at`. Local dot products will be carried out on all sites of the shorter MPA. Other sites will remain unmodified.

mpa1 and mpa2 can also have equal length if `start_at == 0`. In this case, we do the same as *`dot()`*.

Parameters

- **mpa2** (*`mpa1,`*) – Factors as MPArrays, length must be inequal.
- **start_at** – The shorter MPA will start on this site.
- **axes** – See `axes` argument to *`dot()`*.

Returns MPA with length of the longer MPA.

`mpnum.mparray.`**`partialtrace`**(*mpa*, *axes=(0, 1)*, *mptype=None*)

Computes the trace or partial trace of an MPA.

This function is most useful for computing traces of an MPO or MPA over given physical legs. For obtaining partial traces (i.e., reduced states) of an MPO, *`mpnum.mpsmpo.reductions_mpo()`* will be more convenient.

By default `(axes=(0, 1))` compute the trace and return the value as length-one MPA with zero physical legs.

For `axes=(m, n)` with integer `m`, trace over the given axes at all sites and return a length-one MPA with zero physical legs. (Use trace() to get the value directly.)

For `axes=(axes1, axes2, ...)` trace over `axesN` at site `N`, with `axesN=(axisN_1, axisN_2)` tracing the given physical legs and `axesN=None` leaving the site invariant. Afterwards, *`prune()`* is called to remove sites with zero physical legs from the result.

Parameters

- **`mpa`** – MPArray
- **`axes`** – Axes for trace, (axis1, axis2) or (axes1, axes2, . . .) with axesN=(axisN_1, axisN_2) or axesN=None.
- **`mptype`** – Which constructor to call with the new local tensors (default: `type(mpa)`)

Returns An MPArray (possibly one site with zero physical legs)

`mpnum.mparray.`**`prune`**(*mpa*, *singletons=False*)

Contract sites with zero (physical) legs.

Parameters

- **`mpa`** (`MPArray`) – *`MPArray`* or iterator over local tensors
- **`singletons`** – If True, also contract sites where all physical legs have size 1 (default: `False`)

Returns An *`MPArray`* (of possibly smaller length)

`mpnum.mparray.`**`regular_slices`**(*length*, *width*, *offset*)

Iterate over regular slices on a linear chain.

Put slices on a linear chain as follows:

```
>>> n = 5
>>> [tuple(range(*s.indices(n))) for s in regular_slices(n, 3, 2)]
[(0, 1, 2), (2, 3, 4)]
>>> n = 7
>>> [tuple(range(*s.indices(n))) for s in regular_slices(n, 3, 2)]
[(0, 1, 2), (2, 3, 4), (4, 5, 6)]
```

The scheme is illustrated by the following figure:

###### `width` #######		
`offset`	`overlap`	`offset`
	####### `width` ######	

Todo: This table needs cell borders in the HTML output (-> CSS) and the tabularcolumns command doesn't work.

Note that the overlap may be larger than, equal to or smaller than zero.

We enforce that the last slice coincides with the end of the chain, i.e. `(length - width) / offset` must be integer. We produce `(length - width) / offset + 1` slices and the i-th slice is `slice(offset * i, offset * i + width)`, with `i` starting at zero.

Parameters

- **`length`** (*`int`*) – The length of the chain.
- **`width`** (*`int`*) – The width of each slice.
- **`offset`** (*`int`*) – Difference between starting positions of successive slices. First slice starts at 0.

Returns Iterator over slices.

`mpnum.mparray.`**`sandwich`**(*mpo*, *mps*, *mps2=None*)

Compute `<mps|MPO|mps>` efficiently

This function computes the same value as `mp.inner(mps, mp.dot(mpo, mps))` in a more efficient way.

The runtime of this method scales with `D**3 * Dp + D**2 * Dp**3` where `D` and `Dp` are the ranks of `mps` and `mpo`. This is more efficient than `mp.inner(mps, mp.dot(mpo, mps))`, whose runtime scales with `D**4 * Dp**3`, and also more efficient than `mp.dot(mps.conj(), mp.dot(mpo, mps)).to_array()`, whose runtime scales with `D**6 * Dp**3`.

If `mps2` is given, `<mps2|MPO|mps>` is computed instead (i.e. `mp.inner(mps2, mp.dot(mpo, mps))`; see also *`dot()`*).

`mpnum.mparray.`**`embed_slice`**(*length*, *slice_*, *mpa*, *embed_tensor=None*)

Embed a local MPA on a linear chain.

Parameters

- **`length`** (*`int`*) – Length of the resulting MPA.
- **`slice`** (*`slice`*) – Specifies the position of `mpa` in the result.
- **`mpa`** (`MPArray`) – MPA of length `slice_.stop - slice_.start`.
- **`embed_tensor`** – Defaults to square identity matrix (see `_embed_ltens_identity()` for details)

Returns MPA of length *length*

`mpnum.mparray.`**`trace`**(*mpa*, *axes=(0, 1)*)

Compute the trace of the given MPA.

If you specify axes (see partialtrace() for details), you must ensure that the result has no physical legs anywhere.

Parameters

- **`mpa`** – MParray
- **`axes`** – Axes for trace, `(axis1, axis2)` or `(axes1, axes2, ...)` with `axesN=(axisN_1, axisN_2)` or `axesN=None`. (default: `(0, 1)`)

Returns A single scalar of type `mpa.dtype` `mpnum.mparray.`**`diag`**`(`*mpa*, *axis=0*`)`

Returns the diagonal elements `mpa[i, i, ..., i]`. If `mpa` has more than one physical dimension, the result is a numpy array with `MPArray` entries, otherwise its a numpy array with floats.

Parameters

- **`mpa`** – *`MPArray`* with shape > `axis`
- **`axis`** – The physical index to take diagonals over

Returns Array containing the diagonal elements (each diagonal element is an *`MPArray`* with the physical dimension reduced by one, note that an *`MPArray`* with dimension 0 is a simple number)

`mpnum.mparray.`**`sumup`**`(`*mpas*, *weights=None*`)`

Returns the sum of the MPArrays in `mpas`. Same as

```
functools.reduce(mp.MPArray.__add__, mpas)
```

but should be faster as we can get rid of intermediate allocations.

Parameters **`mpas`** – Iterator over *`MPArray`*

Returns Sum of `mpas`

`mpnum.mparray.`**`full_rank`**`(`*ldims*`)`

Computes a list of maximal ranks for a tensor with given local dimesions

Parameters **`ldims`** – Dimensions of the legs of the tensor per site. Can be either passed as one number per site (`[2, 5, 2]`) or if there are multiple legs per site as a list of tuples similar to *`MPArray.shape`* (e.g. `[(2,), (3, 4), (5,)]`)

Returns Tuple of ranks that are maximal for the local dimensions `ldims`.

```
>>> full_rank([3] * 5)
[3, 9, 9, 3]
>>> full_rank([2] * 8)
[2, 4, 8, 16, 8, 4, 2]
>>> full_rank([(2, 3)] * 4)
[6, 36, 6]
```

1.2.3 mpstruct

Core data structure & routines to manage local tensors

class `mpnum.mpstruct.`**`LocalTensors`**`(`*ltens*, *cform=(None, None)*`)`

Bases: `object`

Core data structure to manage the local tensors of a *`MPArray`*.

The local tensors are kept in `_ltens`, a list of `numpy.ndarray`s such that `_ltens[i]` corresponds to the local tensor of site *i*.

If there are k (non-virtual) indices at site i, the corresponding local tensor is a ndarray with `ndim == k + 2`. The two additional indices of the local tensor correspond to the virtual legs. We reserve the *0*th index of the local tensor for the virtal leg coupling to site $i - 1$ and the last index for the virtual leg coupling to site $i + 1$.

Therefore, if the physical legs at site i have dimensions $d_1, \ldots, d_k$, the corresponding local tensor has shape $(r_{i-1}, d_1, \ldots, d_k, r_i)$. Here, r_{i-1} and r_i denote the rank between sites $(i-1, i)$ and $(i, i+1)$, respectively.

To keep the data structure consistent, we include the left virutal leg of the leftmost local tensor as well as the right virtual leg of the rightmost local tensor as dummy indices of dimension 1.

canonical_form
Tensors which are currently in left/right-canonical form.

Returns tuple `(left, right)` such that

- `self[:left]` are left-normalized
- `self[right:]` are right-normalized.

copy()
Returns a deep copy of the local tensors

shape
List of tuples with the dimensions of each tensor leg at each site

update(*index*, *tens*, *canonicalization=None*)
Update the local tensor at site `index` to the new value `tens`. Checks the rank and shape of the new values to keep the MPA consistent. Therefore, some actions such as changing the rank between two sites require to update both sites at the same time, which can be done by passing in multiple values as arguments.

Parameters

- **index** – Integer/slice. Site index/indices of the local tensor/ tensors to be updated.
- **tens** – New local tensor as `numpy.ndarray`. Alternatively, sequence over multiple ndarrays (in case `index` is a slice).
- **canonicalization** – If `tens` is left-/right-normalized, pass `'left'`/`'right'`, respectively. Otherwise, pass `None` (default `None`). In case `index` is a slice, either pass a sequence of the corresponding values or a single value, which is repeated for each site updated.

1.2.4 factory

Module to create random test instances of matrix product arrays

`mpnum.factory.`**eye**(*sites*, *ldim*)
Returns a MPA representing the identity matrix

Parameters

- **sites** – Number of sites
- **ldim** – Int-like local dimension or iterable of local dimensions

Returns Representation of the identity matrix as MPA

```
>>> I = eye(4, 2)
>>> I.ranks, I.shape
((1, 1, 1), ((2, 2), (2, 2), (2, 2), (2, 2)))
>>> I = eye(3, (3, 4, 5))
>>> I.shape
((3, 3), (4, 4), (5, 5))
```

`mpnum.factory.`**random_local_ham**(*sites*, *ldim=2*, *intlen=2*, *randstate=None*)
Generates a random Hamiltonian on *sites* sites with local dimension *ldim*, which is a sum of local Hamiltonians with interaction length *intlen*.

Parameters

- **sites** – Number of sites
- **ldim** – Local dimension

- **intlen** – Interaction length of the local Hamiltonians

Returns MPA representation of the global Hamiltonian

`mpnum.factory.`**`random_mpa`**(*sites*, *ldim*, *rank*, *randstate=None*, *normalized=False*, *force_rank=False*, *dtype=<class 'numpy.float64'>*)

Returns an MPA with randomly choosen local tensors (real by default)

Parameters

- **sites** – Number of sites
- **ldim** – Physical legs, depending on the type passed:
  - scalar: Single physical leg for each site with given dimension
  - iterable of scalar: Same physical legs for all sites
  - iterable of iterable: Generated MPA will have exactly this as *ndims*
- **rank** – Desired rank, depending on the type passed:
  - scalar: Same rank everywhere
  - iterable of length `sites - 1`: Generated MPA will have exactly this as *ranks*
- **randstate** – numpy.random.RandomState instance or None
- **normalized** – Resulting *mpa* has *mp.norm(mpa) == 1*
- **force_rank** – If True, the rank is exaclty *rank*. Otherwise, it might be reduced if we reach the maximum sensible rank.
- **dtype** – Type of the returned MPA. Currently only `np.float_` and `np.complex_` are implemented (default: `np.float_`, i.e. real values).

Returns Randomly choosen matrix product array

Entries of local tensors are drawn from a normal distribution of unit variance. For complex values, the real and imaginary parts are independent and have unit variance.

```
>>> mpa = random_mpa(4, 2, 10, force_rank=True)
>>> mpa.ranks, mpa.shape
((10, 10, 10), ((2,), (2,), (2,), (2,)))
```

```
>>> mpa = random_mpa(4, (1, 2), 10, force_rank=True)
>>> mpa.ranks, mpa.shape
((10, 10, 10), ((1, 2), (1, 2), (1, 2), (1, 2)))
```

```
>>> mpa = random_mpa(4, [(1, ), (2, 3), (4, 5), (1, )], 10, force_rank=True)
>>> mpa.ranks, mpa.shape
((10, 10, 10), ((1,), (2, 3), (4, 5), (1,)))
```

The following doctest verifies that we do not change how random states are generated, ensuring reproducible results. In addition, it verifies the returned dtype:

```
>>> rng = np.random.RandomState(seed=3208886881)
>>> random_mpa(2, 2, 3, rng).to_array()
array([[-0.7254321 ,  3.44263486],
       [-0.17262967,  2.4505633 ]])
>>> random_mpa(2, 2, 3, rng, dtype=np.complex_).to_array()
array([[-0.53552415+1.39701566j, -2.12128866+0.57913253j],
       [-0.32652114+0.51490923j, -0.32222320-0.32675463j]])
```

`mpnum.factory.`**`random_mpdo`**(*sites*, *ldim*, *rank*, *randstate=<module 'numpy.random' from '/usr/lib/python3/dist-packages/numpy/random/__init__.py'>*)

Returns a randomly choosen matrix product density operator (i.e. positive semidefinite matrix product operator with trace 1).

Parameters

- **`sites`** – Number of sites
- **`ldim`** – Local dimension
- **`rank`** – Rank
- **`randstate`** – numpy.random.RandomState instance

Returns randomly choosen classicaly correlated matrix product density op.

```
>>> rho = random_mpdo(4, 2, 4)
>>> rho.ranks, rho.shape
((4, 4, 4), ((2, 2), (2, 2), (2, 2), (2, 2)))
>>> rho.canonical_form
(0, 4)
```

`mpnum.factory.`**`random_mps`**(*sites*, *ldim*, *rank*, *randstate=None*, *force_rank=False*)

Returns a randomly choosen normalized matrix product state

Parameters

- **`sites`** – Number of sites
- **`ldim`** – Local dimension
- **`rank`** – Rank
- **`randstate`** – numpy.random.RandomState instance or None
- **`force_rank`** – If True, the rank is exaclty *rank*. Otherwise, it might be reduced if we reach the maximum sensible rank.

Returns randomly choosen matrix product (pure) state

```
>>> mps = random_mps(4, 2, 10, force_rank=True)
>>> mps.ranks, mps.shape
((10, 10, 10), ((2,), (2,), (2,), (2,)))
>>> mps.canonical_form
(0, 4)
>>> round(abs(1 - mp.inner(mps, mps)), 10)
0.0
```

`mpnum.factory.`**`random_mpo`**(*sites*, *ldim*, *rank*, *randstate=None*, *hermitian=False*, *normalized=True*, *force_rank=False*)

Returns an hermitian MPO with randomly choosen local tensors

Parameters

- **`sites`** – Number of sites
- **`ldim`** – Local dimension
- **`rank`** – Rank
- **`randstate`** – numpy.random.RandomState instance or None
- **`hermitian`** – Is the operator supposed to be hermitian
- **`normalized`** – Operator should have unit norm

- **force_rank** – If True, the rank is exaclty *rank*. Otherwise, it might be reduced if we reach the maximum sensible rank.

Returns randomly choosen matrix product operator

```
>>> mpo = random_mpo(4, 2, 10, force_rank=True)
>>> mpo.ranks, mpo.shape
((10, 10, 10), ((2, 2), (2, 2), (2, 2), (2, 2)))
>>> mpo.canonical_form
(0, 4)
```

mpnum.factory.**zero**(*sites*, *ldim*, *rank*, *force_rank=False*)

Returns a MPA with localtensors beeing zero (but of given shape)

Parameters

- **sites** – Number of sites
- **ldim** – Depending on the type passed (checked in the following order)
  - iterable of iterable: Detailed list of physical dimensions, retured mpa will have exactly this for mpa.shape
  - iterable of scalar: Same physical dimension for each site
  - scalar: Single physical leg for each site with given dimension
- **rank** – Rank
- **force_rank** – If True, the rank is exaclty *rank*. Otherwise, it might be reduced if we reach the maximum sensible rank.

Returns Representation of the zero-array as MPA

mpnum.factory.**diagonal_mpa**(*entries*, *sites*)

Returns an MPA with `entries` on the diagonal and zeros otherwise.

Parameters **entries** (*numpy.ndarray*) – one-dimensional array

Returns *MPArray* with rank `len(entries)`.

1.2.5 mpsmpo

Matrix Product State (MPS) and Operator (MPO) functions

The *Introduction* also covers the definitions mentioned below.

Definitions

We consider a linear chain of n sites with associated Hilbert spaces mathcal H_k = C^{d_k}, d_k, $k \in [1..n] := \{1, 2, \ldots, n\}$. The set of linear operators $\mathcal{H}_k \to \mathcal{H}_k$ is denoted by $\mathcal{B}_k$. We write $\mathcal{H} = \mathcal{H}_1 \otimes \cdots \otimes \mathcal{H}_n$ and the same for $\mathcal{B}$.

We use the following three representations:

- Matrix product state (MPS): Vector $|\psi\rangle \in \mathcal{H}$
- Matrix product operator (MPO): Operator $M \in \mathcal{B}$
- Locally purified matrix product state (PMPS): Positive semidefinite operator $\rho \in \mathcal{B}$

All objects are represented by n local tensors.

Matrix product state (MPS)

Represent a vector $|\psi\rangle \in \mathcal{H}$ as

$$\langle i_1 \dots i_n|\psi\rangle = A^{(1)}_{i_1} \cdots A^{(n)}_{i_n}, \quad A^{(k)}_{i_k} \in \mathbb{C}^{D_{k-1}\times D_k}, \quad D_0 = 1 = D_n.$$

The k-th local tensor is $T_{l,i,r} = (A^{(k)}_i)_{l,r}$.

The vector $|\psi\rangle$ can be a quantum state, with the density matrix given by $\rho = |\psi\rangle\langle\psi| \in \mathcal{B}$. Reference: E.g. *[Sch11]*.

Matrix product operator (MPO)

Represent an operator $M \in \mathcal{B}$ as

$$\langle i_1 \dots i_n|M|j_1 \dots j_n\rangle = A^{(1)}_{i_1 j_1} \cdots A^{(n)}_{i_n j_n}, \quad A^{(k)}_{i_k j_k} \in \mathbb{C}^{D_{k-1}\times D_k}, \quad D_0 = 1 = D_n.$$

The k-th local tensor is $T_{l,i,j,r} = (A^{(k)}_{ij})_{l,r}$.

This representation can be used to represent a mixed quantum state $\rho = M$, but it is not limited to positive semidefinite M. Reference: E.g. *[Sch11]*.

Locally purified matrix product state (PMPS)

Represent a positive semidefinite operator $\rho \in \mathcal{B}$ as follows: Let $\mathcal{H}'_k = \mathbb{C}^{d'_k}$ with suitable d'_k and $\mathcal{P} = \mathcal{H}_1 \otimes \mathcal{H}'_1 \otimes \cdots \otimes \mathcal{H}_n \otimes \mathcal{H}'_n$. Find $|\Phi\rangle \in \mathcal{P}$ such that

$$\rho = \operatorname{tr}_{\mathcal{H}'_1,\dots,\mathcal{H}'_n}(|\Phi\rangle\langle\Phi|)$$

and represent $|\Phi\rangle$ as

$$\langle i_1 i'_1 \dots i_n i'_n|\Phi\rangle = A^{(1)}_{i_1 i'_1} \cdots A^{(n)}_{i_n i'_n}, \quad A^{(k)}_{i_k j_k} \in \mathbb{C}^{D_{k-1}\times D_k}, \quad D_0 = 1 = D_n.$$

The k-th local tensor is $T_{l,i,i',r} = (A^{(k)}_{ii'})_{l,r}$.

The ancillary dimensions d'_i are not determined by the d_i but depend on the state. E.g. if ρ is pure, one can set all $d_i = 1$. Reference: E.g. [*Cue13*].

Todo: Are derived classes MPO/MPS/PMPS of any help?

Todo: I am not sure the current definition of PMPS is the most elegant for our purposes…

References:

- [Cue13] De las Cuevas, G., Schuch, N., Pérez-García, D., and Cirac, J. I. (2013). "Purifications of multipartite states: limitations and constructive methods". New J. Phys. 15(12), p. 123021. DOI: 10.1088/1367-2630/15/12/123021. arXiv: 1308.1914.

`mpnum.mpsmpo.`**`mps_to_mpo`**(*mps*)

Convert a pure MPS to a mixed state MPO.

Parameters **`mps`** (`MPArray`) – An MPA with one physical leg

Returns An MPO (density matrix as MPA with two physical legs) `mpnum.mpsmpo.`**`mps_to_pmps`**(*mps*)

Convert a pure MPS into a local purification MPS mixed state.

The ancilla legs will have dimension one, not increasing the memory required for the MPS.

Parameters **`mps`** (`MPArray`) – An MPA with one physical leg

Returns An MPA with two physical legs (system and ancilla)

`mpnum.mpsmpo.`**`pmps_dm_to_array`**(*pmps*, *global_=False*)

Convert PMPS to full array representation of the density matrix

The runtime of this method scales with D**3 instead of D**6 where D is the rank and D**6 is the scaling of using *`pmps_to_mpo()`* and `to_array()`. This is useful for obtaining reduced states of a PMPS on non-consecutive sites, as normalizing before using *`pmps_to_mpo()`* may not be sufficient to reduce the rank in that case.

Note: The resulting array will have dimension-1 physical legs removed.

`mpnum.mpsmpo.`**`pmps_reduction`**(*pmps*, *support*)

Convert a PMPS to a PMPS representation of a local reduced state

Parameters **`support`** – Set of sites to keep

Returns Sites traced out at the beginning or end of the chain are removed using *`reductions_pmps()`* and a suitable normalization. Sites traced out in the middle of the chain are converted to sites with physical dimension 1 and larger ancilla dimension.

`mpnum.mpsmpo.`**`pmps_to_mpo`**(*pmps*)

Convert a local purification MPS to a mixed state MPO.

A mixed state on n sites is represented in local purification MPS form by a MPA with n sites and two physical legs per site. The first physical leg is a 'system' site, while the second physical leg is an 'ancilla' site.

Parameters **`pmps`** (`MPArray`) – An MPA with two physical legs (system and ancilla)

Returns An MPO (density matrix as MPA with two physical legs)

`mpnum.mpsmpo.`**`pmps_to_mps`**(*pmps*)

Convert a PMPS with unit ancilla dimensions to a simple MPS

If all ancilla dimensions of the PMPS are equal to unity, they are removed. Otherwise, an AssertionError is raised.

`mpnum.mpsmpo.`**`reductions_mpo`**(*mpa*, *width=None*, *startsites=None*, *stopsites=None*)

Iterate over MPO partial traces of an MPO

The support of the i-th result is `range(startsites[i], stopsites[i])`.

Parameters

- **`mpa`** (`mpnum.mparray.MPArray`) – An MPO
- **`startsites`** – Defaults to `range(len(mpa) - width + 1)`.
- **`stopsites`** – Defaults to `[ start + width for start in startsites ]`. If specified, we require *startsites* to be given and *width* to be None.
- **`width`** – Number of sites in support of the results. Default *None*. Must be specified if one or both of *startsites* and *stopsites* are not given.

Returns Iterator over partial traces as MPO `mpnum.mpsmpo.`**`reductions_mps_as_mpo`**(*mps*, *width=None*, *startsites=None*, *stopsites=None*)

Iterate over MPO mpdoreduced states of an MPS

width, *startsites* and *stopsites*: See *`reductions_mpo()`*.

Parameters **`mps`** – Pure state as MPS

Returns Iterator over reduced states as MPO

`mpnum.mpsmpo.`**`reductions_mps_as_pmps`**(*mps*, *width=None*, *startsites=None*, *stopsites=None*)

Iterate over PMPS reduced states of an MPS

width, *startsites* and *stopsites*: See *`reductions_mpo()`*.

Parameters **`mps`** – Pure state as MPS

Returns Iterator over reduced states as PMPS

`mpnum.mpsmpo.`**`reductions_pmps`**(*pmps*, *width=None*, *startsites=None*, *stopsites=None*)

Iterate over PMPS partial traces of a PMPS

width, *startsites* and *stopsites*: See *`reductions_mpo()`*.

Parameters **`pmps`** – Mixed state in locally purified MPS representation (PMPS, see *Definitions*)

Returns Iterator over reduced states as PMPS

`mpnum.mpsmpo.`**`reductions`**(*state*, *mode*, ***kwargs*)

Todo: Add docstring

1.2.6 `linalg`

Linear algebra with matrix product arrays

Currently, we support computing extremal eigenvalues and eigenvectors of MPOs.

`mpnum.linalg.`**`eig`**(*mpo*, *num_sweeps*, *var_sites=2*, *startvec=None*, *startvec_rank=None*, *randstate=None*, *eigs=None*)

Iterative search for MPO eigenvalues

Note: This function can return completely inaccurate values. You are responsible for supplying a large enough `startvec_rank` (or `startvec` with large enough rank) and `num_sweeps`.

This function attempts to find eigenvalues by iteratively optimizing $\lambda = \langle\psi|H|\psi\rangle$ where H is the operator supplied in the argument `mpo`. Specifically, we attempt to de- or increase λ by optimizing over several neighbouring local tensors of the MPS $|\psi\rangle$ simultaneously (the number given by `var_sites`).

The algorithm used here is described e.g. in [*Sch11*, Sec. 6.3]. For `var_sites = 1`, it is called “variational MPS ground state search” or “single-site DMRG” [*Sch11*, Sec. 6.3, p. 69]. For `var_sites > 1`, it is called “multi-site DMRG”.

Parameters

- **`mpo`** (`MPArray`) – A matrix product operator (MPA with two physical legs)
- **`num_sweeps`** (*`int`*) – Number of sweeps to do (required)
- **`var_sites`** (*`int`*) – Number of neighbouring sites to be varied simultaneously

- **startvec** – Initial guess for eigenvector (default: random MPS with rank *startvec_rank*)
- **startvec_rank** – Rank of random start vector (required and used only if no start vector is given)
- **randstate** – `numpy.random.RandomState` instance or `None`
- **eigs** – Function which computes one eigenvector of the local eigenvalue problem on `var_sites` sites

Returns eigval, eigvec_mpa

The `eigs` parameter defaults to

```
eigs = functools.partial(scipy.sparse.linalg.eigsh, k=1, tol=1e-6)
```

By default, *eig()* computes the eigenvalue with largest magnitude. To compute e.g. the smallest eigenvalue (sign included), supply `which='SA'` to `eigsh`. For other possible values, refer to the SciPy documentation.

It is recommendable to supply a value for the `tol` parameter of `eigsh()`. Otherwise, `eigsh()` will work at machine precision which is rarely necessary.

Note: One should keep in mind that a variational method (such as the one implemented in this function) can only provide e.g. an upper bound on the lowest eigenvalue of an MPO. Deciding whether a given MPO has an eigenvalue which is smaller than a given threshold has been shown to be NP-hard (in the number of parameters of the MPO representation) *[KGE14]*.

Comments on the implementation, for `var_sites = 1`:

References are to the arXiv version of *[Sch11]* assuming we replace zero-based with one-based indices there.

`Psi^A_{i-1}` and `Psi^B_{i}` are identity matrices because of normalization. (See Fig. 42 on p. 67 and the text; see also Figs. 14 and 15 and pages 28 and 29.)

`mpnum.linalg.`**`eig_sum`**(*mpas*, *num_sweeps*, *var_sites=2*, *startvec=None*, *startvec_rank=None*, *randstate=None*, *eigs=None*)

Iterative search for eigenvalues of a sum of MPOs/MPSs

Try to compute the ground state of the sum of the objects in `mpas`. MPOs are taken as-is. An MPS $|\psi\rangle$ adds $|\psi\rangle\langle\psi|$ to the sum.

This function executes the same algorithm as *eig()* applied to an uncompressed MPO sum of the elements in `mpas`, but it obtains the ingredients for the local optimization steps using less memory and execution time. In particular, this function does not have to convert an MPS in `mpas` to an MPO.

Todo: Add information on how the runtime of *eig()* and *eig_sum()* scale with the the different ranks. For the time being, refer to the benchmark test.

Parameters **mpas** – A sequence of MPOs or MPSs

Remaining parameters and description: See *eig()*.

Algorithm: [*Sch11*, Sec. 6.3]

1.2.7 `povm`

`povm.mppovm`

Matrix-product representation of POVMs

This module provides the following classes:

- *`MPPovm`*: A matrix product representation of a multi-site POVM.

  For example, for a linear chain of *n* qubits this class can represent the POVM of the observable *XX. . . X* with 2^n elements efficiently. It is also possible to sample from the probability distribution of this POVM efficiently.

- *`MPPovmList`*: A list of MP-POVMs.

  This class can be used e.g. to obtain estimated expectation values of the local observable *XX1. . . 1* on two qubits from from samples for the global observables *XX. . . X* and *XXY. . . Y* (cf. below on *Linear combinations of functions of POVM outcomes*).

- The methods *`MPPovm.embed()`*, *`MPPovm.block()`*/*`MPPovmList.block()`*, *`MPPovm.repeat()`*/*`MPPovmList.repeat()`* as well as *`pauli_mpp()`* and *`pauli_mpps()`* allow for convenient construction of MP-POVMs and MP-POVM lists.

Linear combinations of functions of POVM outcomes

In order to perform the just mentioned estimation of probabilities of one POVM from samples of another POVM with possibly larger support, we provide a function which can estimate linear functions of functions of POVM outcomes: Let M a finite index set with real elements $y \in M \subset \mathbb{R}$ such that $\hat{y}$ are the positive semidefinite POVM elements which sum to the identity, $\sum_{y \in M} \hat{y} = 1$. Given a state ρ, the probability mass function (PMF) of the probability distribution given by the POVM and the state can be expressed as $p_y = \operatorname{tr}(\rho\hat{y})$, $y \in M$ or as $p(x) = \sum_{y \in M} \delta(x - y) p_y$. Let further $D = (x_1, \ldots, x_m)$, $x_k \in M$ a set of samples from $p(x)$ and let $f\colon M \to \mathbb{R}$ an arbitrary function of the POVM outcomes. The true value $\langle f \rangle_p = \int f(y)p(y)\mathrm{d}y$ can then be estimated using the sample average $\langle f \rangle_D = \frac{1}{m}\sum_{k=1}^{m} f(x_k)p_{x_k}$. In the same way, a linear combination $f = \sum c_i f_i$ of functions $f_i\colon M \to \mathbb{R}$ of POVM outcomes can be estimated by $\langle f \rangle_D = \sum_i c_i \langle f_i \rangle_D$. Such a linear combination of functions of POVM outcomes can be estimated using *`MPPovm.est_lfun()`*. More technically, the relation $\langle\langle f \rangle_D\rangle_{p_m} = \langle f \rangle_p$ shows that $\langle f \rangle_D$ is an unbiased estimator for the true expectation value $\langle f \rangle_p$; the probability distribution of the dataset D is given by the sampling distribution $p_m(D) = p(x_1) \ldots p(x_m)$.

Estimates of the POVM probabilities p_y can also be expressed as functions of this kind: Consider the function

$$\theta_y(x) = \begin{cases} 1, & x = y, \\ 0, & \text{otherwise.} \end{cases}$$

The true value of this function under $p(x)$ is $\langle \theta_y \rangle_p = p_y$ and the sample average $\langle \theta_y \rangle_D$ provides an estimator for p_y. In order to estimate probabilities of one POVM from samples for another POVM, such a function can be used: E.g. to estimate the probability of the $(+1, +1)$ outcome of the POVM *XX1. . . 1*, we can define a function which is equal to 1 if the outcome of the POVM *XX. . . X* on the first two sites is equal to $(+1, +1)$ and zero otherwise. The sample average of this function over samples for the latter POVM *XX. . . X* will estimate the desired probability. This approach is implemented in *`MPPovm.est_pmf_from()`*. If samples from more than one POVM are available for estimating a given probability, a weighted average of estimators can be used as implemented in *`MPPovm.est_pmf_from_mpps()`*; the list of MP-POVMs for which samples are available is passed as an *`MPPovmList`* instance. Finally, the function *`MPPovmList.est_lfun_from()`* allows estimation of a linear combination of probabilities from different POVMs using samples of a second list of MP-POVMs. This function also estimates the variance of the estimate. In order to perform the two estimation procedures, for each probability, we construct an estimator from a weighted average of functions of outcomes of different POVMs, as has been explained above. For more simple settings, *`MPPovmList.est_lfun()`* is also available.

True values of the functions just mentioned can be obtained from `MPPovm.lfun()`, `MPPovmList.lfun()` and `MPPovmList.lfun_from()`. All functions return both the true expectation value and the variance of the expectation value.

The variance of the (true) expectation value $\langle f \rangle_p$ of a function $f \colon M \to \mathbb{R}$ is given by $\mathrm{var}_p(f) = \mathrm{cov}_p(f, f)$ with $\mathrm{cov}_p(f, g) = \langle fg \rangle_p - \langle f \rangle_p \langle g \rangle_p$. The variance of the estimate $\langle f \rangle_D$ is given by $\mathrm{var}_{p_m}(\langle f \rangle_D) = \frac{1}{m} \mathrm{var}_p(f)$ where $p_m(D)$ is the sampling distribution from above. An unbiased estimator for the covariance $\mathrm{cov}_p(f, g)$ is given by $\frac{m}{m-1} \mathrm{cov}_D(f, g)$ where the sample covariance $\mathrm{cov}_D(f, g)$ is defined in terms of sample averages in the usual way, $\mathrm{cov}_D(f, g) = \langle fg \rangle_D - \langle f \rangle_D \langle g \rangle_D$. This estimator is used by `MPPovm.est_lfun()`.

Todo: Explain the details of the variance estimation, in particular the difference between the variances returned from `MPPovmList.lfun()` and `MPPovmList.lfun_from()`. Check the mean square error.

Add a good references explaining all facts mentioned above and for further reading.

Document the runtime and memory cost of the functions.

Class and function reference

class `mpnum.povm.mppovm.`**`MPPovm`**(**args*, ***kwargs*)

Bases: `mpnum.mparray.MPArray`

MPArray representation of multipartite POVM

There are two different ways to write down a POVM in matrix product form

1. **As a list of matrix product operators, where each entry corresponds to** a single POVM element
2. As a matrix proudct array with 3 physical legs:

   [POVM index, column index, row index]

   that is, the first physical leg of the MPArray corresponds to the index of the POVM element. This representation is especially helpful for computing expectation values with MPSs/MPDOs.

Here, we choose the second.

Todo: This class should provide a function which returns expectation values as full array. (Even though computing expectation values using the POVM struture brings advantages, we usually need the result as full array.) This function should also replace small negative probabilities by zero and canonicalize the sum of all probabilities to unity (if the deviation is non-zero but small). The same checks should also be implemented in localpovm.POVM.

Todo: Right now we use this class for multi-site POVMs with elements obtained from every possible combination of the elements of single-site POVMs: The POVM index is split across all sites. Explore whether and how this concept can also be useful in other cases.

`block`(*nr_sites*)

Embed an MP-POVM on local blocks

The returned `MPPovmList` will contain *self* embedded at every possible position on *len(self)* neighbouring sites in a chain of length *nr_sites*. The remaining sites are not measured (`self.embed()`).

self must a have a uniform local Hilbert space dimension.

Parameters **`nr_sites`** – Number of sites of the resulting MP-POVMs

`block_pmfs_as_array`(*state*, *mode*, *asarray=False*, *eps=1e-10*, ***redarg*)

Todo: Add docstring

`elements`

Returns an iterator over all POVM elements. The result is the i-th POVM element in MPO form.

It would be nice to call this method *__iter__*, but this breaks *mp.dot(mppovm, …)*. In addition, *next(iter(mppovm))* would not be equal to *mppovm[0]*.

`embed`(*nr_sites*, *startsite*, *local_dim*)

Embed MP-POVM into larger system

Applying the resulting embedded MP-POVM to a state *rho* gives the same result as applying the original MP-POVM *self* on the reduced state of sites *range(startsite, startsite + len(self))* of *rho*.

Parameters

- **`nr_sites`** – Number of sites of the resulting MP-POVM
- **`startsite`** – Position of the first site of *self* in the resulting MP-POVM
- **`local_dim`** – Local dimension of sites to be added

Returns MP-POVM with *self* on sites *range(startsite, startsite + len(self))* and `MPPovm.eye()` elsewhere

`est_lfun`(*coeff*, *funs*, *samples*, *weights=None*, *eps=1e-10*)

Estimate a linear combination of functions of POVM outcomes

This function estimates the function with exact value given by `MPPovm.lfun()`; see there for description of the parameters *coeff* and *funs*.

Parameters

- **`samples`** (`np.ndarray`) – A shape *(n_samples, len(self.nsoutdims))* with samples from *self*
- **`weights`** – A length *n_samples* array for weighted samples. You can submit counts by passing them as weights. The number of samples used in average and variance estimation is determined by *weights.sum()* if *weights* is given.

Returns *(est, var)*: Estimated value and estimated variance of the estimated value. For details, see *Linear combinations of functions of POVM outcomes*.

`est_pmf`(*samples*, *normalize=True*, *eps=1e-10*)

Estimate probability mass function from samples

Parameters

- **`samples`** (`np.ndarray`) – *(n_samples, len(self.nsoutdims))* array of samples
- **`normalize`** (`bool`) – True: Return normalized probability estimates (default). False: Return integer outcome counts.

Returns Estimated probabilities as ndarray *est_pmf* with shape *self.nsoutdims*

*n_samples * est_pmf[i1, …, ik]* provides the number of occurences of outcome *(i1, …, ik)* in *samples*.

`est_pmf_from`(*other*, *samples*, *eps=1e-10*)
Estimate PMF from samples of another MPPovm *other*

If *other* does not provide information on all elements in *self*, we require that the elements in *self* for which information is provided sum to a multiple of the identity.

Example: If we consider the MPPovm `MPPovm.from_local_povm(x, n)` for given local POVMs *x*, it is possible to obtain counts for the Pauli X part of `x = pauli_povm()` from samples for `x = x_povm()`; this is also true if the latter is supported on a larger part of the chain.

Parameters

- **`other`** (`MPPovm`) – Another MPPovm
- **`samples`** (*`np.ndarray`*) – *(n_samples, len(other.nsoutdims))* array of samples for *other*

Returns *(est_pmf, n_samples_used)*. *est_pmf*: Shape *self.nsoutdims* ndarray of normalized probability estimates; the sum over the available probability estimates is equal to the fraction of the identity obtained by summing the corresponding POVM elements. *n_samples_used*: Number of samples which have contributed to the PMF estimate.

`est_pmf_from_mpps`(*other*, *samples*, *eps=1e-10*)
Estimate probability mass function from MPPovmList samples

Parameters

- **`other`** (`MPPovmList`) – An *`MPPovmList`* instance
- **`samples`** – Iterable of samples (e.g. from `MPPovmList.samples()`)

Returns *(p_est, n_samples_used)*, both are shape *self.nsoutdims* ndarrays. *p_est* provides estimated probabilities and *n_samples_used* provides the effective number of samples used for each probability.

`expectations`(*mpa*, *mode='auto'*)
Computes the exp. values of the POVM elements with given state

Parameters

- **`mpa`** – State given as MPDO, MPS, or PMPS
- **`mode`** – In which form *mpa* is given. Possible values: 'mpdo', 'pmps', 'mps', or 'auto'. If 'auto' is passed, we choose between 'mps' or 'mpdo' depending on the number of physical legs

Returns Iterator over the expectation values, the n-th element is the expectation value correponding to the reduced state on sites [n,. . . ,n + len(self) - 1]

`classmethod eye`(*local_dims*)
Construct MP-POVM with no output or measurement

Corresponds to taking the partial trace of the quantum state and a shorter MP-POVM.

Parameters **`local_dims`** – Iterable of local dimensions

`classmethod from_local_povm`(*lelems*, *width*)
Generates a product POVM on *width* sites.

Parameters

- **`lelems`** – POVM elements as an iterator over all local elements (i.e. an iterator over numpy arrays representing the latter)
- **`width`** (*`int`*) – Number of sites the POVM lives on

Returns `MPPovm` which is a product POVM of the *lelems*

hdims
Local Hilbert space dimensions

lfun (*coeff*, *funs*, *state*, *mode='auto'*, *eps=1e-10*)
Evaluate a linear combination of functions of POVM outcomes

Parameters

- **coeff** (`np.ndarray`) – A length *n_funs* array with the coefficients of the linear combination. If *None*, return the estimated values of the individual functions and the estimated covariance matrix of the estimates.
- **funs** (`np.ndarray`) – A length *n_funs* sequence of functions. If *None*, the estimated function will be a linear function of the POVM probabilities.

For further information, see also *Linear combinations of functions of POVM outcomes*.

The parameters *state* and *mode* are passed to `MPPovm.pmf()`.

Returns *(value, var)*: Expectation value and variance of the expectation value

match_elems (*other*, *exclude_dup=()*, *eps=1e-10*)
Find POVM elements in *other* which have information on *self*

We find all POVM sites in *self* which have only one possible outcome. We discard these outputs in *other* and afterwards check *other* and *self* for any common POVM elements.

Parameters

- **other** – Another MPPovm
- **exclude_dup** – Sequence which can include *'self'* or *'other'* (or both) to assert that there are no linearly dependent pairs of elements in *self* or *other*.
- **eps** – Threshould for values which should be treated as zero

Returns (*matches*, *prefactors*)

matches[i_1, . . . , i_k, j_1, . . . , j_k] specifies whether outcome *(i_1, . . . , i_k)* of *self* has the same POVM element as the partial outcome *(j_1, . . . , j_k)* of *other*; outcomes are specified only on the sites mentioned in *sites* such that *k = len(sites)*.

prefactors[i_1, . . . , i_k, j_1, . . . , j_k] specifies how samples from *other* have to be weighted to correspond to samples for *self*.

nsoutdims
Non-singleton outcome dimensions (dimension larger one)

nsoutpos
Sites with non-singleton outcome dimension (dimension larger one)

outdims
Outcome dimensions

pack_samples (*samples*, *dtype=None*)
Pack samples into one integer per sample

Store one sample in a single integer instead of a list of integers with length *len(self.nsoutdims)*. Example:

```
>>> p = pauli_mpp(nr_sites=2, local_dim=2)
>>> p.outdims
(6, 6)
```

```
>>> p.pack_samples(np.array([[0, 1], [1, 0], [1, 2], [5, 5]]))
array([ 1,  6,  8, 35])
```

pmf (*state*, *mode='auto'*)

Compute the POVM's probability mass function for *state*

If you want to compute the probabilities for reduced states of *state*, you can use `MPPovm.expectations()` instead of this function.

Parameters

- **state** (`mp.MPArray`) – A quantum state as MPA. Must have the same length as *self*.
- **mode** – *'mps'*, *'mpdo'* or *'pmps'*. See `MPPovm.expectations()`.

Returns Probabilities as MPArray

pmf_as_array (*state*, *mode='auto'*, *eps=1e-10*, *impl='auto'*)

Compute the POVM's PMF for *state* as full array

Parameters: See `MPPovm.pmf()`.

Parameters impl – *'auto'*, *'default'*, *'pmps-symm'* or *'pmps-ltr'*. *'auto'* will use *'pmps-symm'* for mode *'pmps'* and *'default'* otherwise.

Returns PMF as shape *self.nsoutdims* ndarray

The resulting (real or complex) probabilities *pmf* are passed through `project_pmf(pmf, eps, eps)` before being returned.

pmfs_as_array (*states*, *mode*, *asarray=False*, *eps=1e-10*)

Todo: Add docstring

probability_map

Map that takes a raveled MPDO to the POVM probabilities

You can use `MPPovm.expectations()` or `MPPovm.pmf()` as convenient wrappers around this map.

If *rho* is a matrix product density operator (MPDO), then

produces the POVM probabilities as MPA (similar to `mpnum.povm.localpovm.POVM.probability_map()`).

repeat (*nr_sites*)

Construct a longer MP-POVM by repetition

The resulting POVM will have length *nr_sites*. If *nr_sites* is not an integer multiple of *len(self)*, *self* must factorize (have leg dimension one) at the position where it will be cut. For example, consider the tensor product MP-POVM of Pauli X and Pauli Y. Calling *repeat(nr_sites=5)* will construct the tensor product POVM XYXYX:

```
>>> import mpnum as mp
>>> import mpnum.povm as mpp
>>> x, y = (mpp.MPPovm.from_local_povm(lp(3), 1) for lp in
...          (mpp.x_povm, mpp.y_povm))
>>> xy = mp.chain([x, y])
>>> xyxyx = mp.chain([x, y, x, y, x])
>>> mp.norm(xyxyx - xy.repeat(5)) <= 1e-10
True
```

sample (*rng*, *state*, *n_samples*, *method='cond'*, *n_group=1*, *mode='auto'*, *pack=False*, *eps=1e-10*)
Random sample from *self* on a quantum state

Parameters

- **state** (`mp.MPArray`) – A quantum state as MPA (see *mode*)
- **n_samples** – Number of samples to create
- **method** – Sampling method (*'cond'* or *'direct'*, see below)
- **n_group** – Number of sites to sample at a time in conditional sampling.
- **mode** – Passed to `MPPovm.expectations()`
- **eps** – Threshold for small values to be treated as zero.

Two different sampling methods are available:

- Direct sampling (*method='direct'*): Compute probabilities for all outcomes and sample from the full probability distribution. Usually faster than conditional sampling for measurements on a small number of sites. Requires memory linear in the number of possible outcomes.
- Conditional sampling (*method='cond'*): Sample outcomes on all sites by sampling from conditional outcome probabilities on at most *n_group* sites at a time. Requires memory linear in the number of outcomes on *n_group* sites. Useful for measurements which act on large parts of a system (e.g. Pauli X on each spin).

Returns ndarray *samples* with shape *(n_samples, len(self.nsoutdims))*

The *i*-th sample is given by *samples[i, :]*. *samples[i, j]* is the outcome for the *j*-th non-singleton output dimension of *self*.

unpack_samples (*samples*)
Unpack samples into several integers per sample

Inverse of `MPPovm.pack_samples()`. Example:

```
>>> p = pauli_mpp(nr_sites=2, local_dim=2)
>>> p.outdims
(6, 6)
>>> p.unpack_samples(np.array([0, 6, 7, 12]))
array([[0, 0],
       [1, 0],
       [1, 1],
       [2, 0]], dtype=uint8)
```

class mpnum.povm.mppovm.**MPPovmList** (*mppseq*)
Bases: `object`

A list of `Matrix Product POVMs`

This class allows you to

- Conveniently obtain samples and estimated or exact probabilities for a list of `MPPovms`
- Estimate probabilities from samples for a different MPPovmList
- Estimate linear functions of probabilities of an MPPovmList from samples for a different MPPovmList

__init__ (*mppseq*)
Construct a MPPovmList

Parameters **mppseq** – An iterable of `MPPovm` objects

All MPPovms must have the same number of sites.

block (*nr_sites*)
Embed MP-POVMs on local blocks

This function calls `MPPovm.block(nr_sites)()` for each MP-POVM in the list. Embedded MP-POVMs at the same position appear consecutively in the returned list:

```
>>> import mpnum as mp
>>> import mpnum.povm as mpp
>>> ldim = 3
>>> x, y = (mpp.MPPovm.from_local_povm(lp(ldim), 1) for lp in
...         (mpp.x_povm, mpp.y_povm))
>>> e = mpp.MPPovm.eye([ldim])
>>> xx = mp.chain([x, x])
>>> xy = mp.chain([x, y])
>>> mppl = mpp.MPPovmList((xx, xy))
>>> xxe = mp.chain([x, x, e])
>>> xye = mp.chain([x, y, e])
>>> exx = mp.chain([e, x, x])
>>> exy = mp.chain([e, x, y])
>>> expect = (xxe, xye, exx, exy)
>>> [abs(mp.norm(a - b)) <= 1e-10
...  for a, b in zip(mppl.block(3).mpps, expect)]
[True, True, True, True]
```

block_pmfs_as_array (*state*, *mode*, *asarray=False*, *eps=1e-10*, ***redarg*)

Todo: Add docstring

est_lfun (*coeff*, *funs*, *samples*, *weights=None*, *eps=1e-10*)
Estimate a linear combination of functions of POVM outcomes

Parameters

- **coeff** – Iterable of coefficient lists
- **funs** – Iterable of function lists
- **samples** – Iterable of samples
- **weights** – Iterable of weight lists or *None*

The *i*-th item from these parameters is passed to `MPPovm.est_lfun()` on *self.mpps[i].est_lfun*.

Returns (*est*, *var*): Estimated value *est* and estimated variance *var* of the estimate *est*

est_lfun_from (*other*, *coeff*, *samples*, *eps=1e-10*)
Estimate a linear function from samples for another MPPovmList

The function to estimate is a linear function of the probabilities of *self* and it is specified by *coeff*. Its true expectation value and variance are returned by `MPPovmList.lfun_from()`. First, an estimator is constructed using `MPPovmList._lfun_estimator()` and this estimator is passed to `MPPovm.est_lfun()` to obtain the estimate. See *Linear combinations of functions of POVM outcomes* for more details.

Parameters

- **other** (`MPPovmList`) – Another MP-POVM list

- **coeff** – A sequence of shape *self.mpps[i].nsoutdims* coefficients which specify the function to estimate
- **samples** – A sequence of samples for *other*

Returns *(est, var)*: Estimated value and estimated variance of the estimated value. Return *(np.nan, np.nan)* if *other* is not sufficient to estimate the function.

est_pmf (*samples*, *normalized=True*, *eps=1e-10*)
Estimate PMF from samples

Returns an iterator over results from `MPPovm.est_pmf()` (see there).

est_pmf_from (*other*, *samples*, *eps=1e-10*)
Estimate PMF from samples of another MPPovmList

Parameters

- **other** (`MPPovmList`) – A different MPPovmList
- **samples** – Samples from *other*

Returns Iterator over *(p_est, n_samples_used)* from `MPPovm.est_pmf_from_mpps()`.

lfun (*coeff*, *funs*, *state*, *mode='auto'*, *eps=1e-10*)
Evaluate a linear combination of functions of POVM outcomes

coeff[i] and *funs[i]* are passed to `MPPovm.lfun()` on *self.mpps[i]*. *funs* = *None* is treated as *[None] * len(self.mpps)*. *state* and *mode* are passed to `MPPovm.pmf()`.

Returns *(value, var)*: Expectation value and variance of the expectation value

lfun_from (*other*, *coeff*, *state*, *mode='auto'*, *other_weights=None*, *eps=1e-10*)
Evaluate a linear combination of POVM probabilities

This function computes the same expectation value as `MPPovmList.lfun()` if supplied with *funs* = *None*, but it computes the variance for a different estimation procedure: It uses weighted averages of POVM probabilities from *other* to obtain the necessary POVM probabilities for *self* (the same is done in `MPPovmList.est_lfun_from()`).

The parameter *coeff* is explained in `MPPovmList.est_lfun_from()`. *state* and *mode* are passed to `MPPovm.pmf()`.

You can supply the array *other_weights* to determine the weighted average used when a probability in a POVM in *self* can be estimated from probabilities in multiple different POVMs in *other*.

Returns *(value, var)*: Expectation value and variance of the expectation value. Return *(np.nan, np.nan)* if *other* is not sufficient to estimate the function.

pack_samples (*samples*)
Pack samples into one integer per sample

Returns Iterator over output from `MPPovm.pack_samples()`

pmf (*state*, *mode='auto'*)
Compute the probability mass functions of all MP-POVMs

Parameters

- **state** – A quantum state as MPA
- **mode** – Passed to `MPPovm.expectations()`

Returns Iterator over probabilities as MPArrays

pmf_as_array(*state*, *mode='auto'*, *eps=1e-10*)

Compute the PMF of all MP-POVMs as full arrays

Parameters: See `MPPovmList.pmf()`. Sanity checks: See `MPPovm.pmf_as_array()`.

Returns Iterator over probabilities as ndarrays

pmfs_as_array(*states*, *mode*, *asarray=False*, *eps=1e-10*)

Todo: Add docstring

repeat(*nr_sites*)

Construct longer MP-POVMs by repeating each MP-POVM

This function calls `MPPovm.repeat(nr_sites)` for each MP-POVM in the list.

For example, `pauli_mpps()` for *local_dim* > *3* (i.e. without Z) and two sites returns POVMs for the four tensor product observables XX, XY, YX and YY:

```
>>> import mpnum as mp
>>> import mpnum.povm as mpp
>>> block_sites = 2
>>> ldim = 3
>>> x, y = (mpp.MPPovm.from_local_povm(lp(ldim), 1) for lp in
...         (mpp.x_povm, mpp.y_povm))
>>> pauli = mpp.pauli_mpps(block_sites, ldim)
>>> expect = (
...     mp.chain((x, x)),
...     mp.chain((x, y)),
...     mp.chain((y, x)),
...     mp.chain((y, y)),
... )
>>> [abs(mp.norm(a - b)) <= 1e-10 for a, b in zip(pauli.mpps, expect)]
[True, True, True, True]
```

Calling *repeat(5)* then returns the following `MPPovmList`:

```
>>> expect = (
...     mp.chain((x, x, x, x, x)),
...     mp.chain((x, y, x, y, x)),
...     mp.chain((y, x, y, x, y)),
...     mp.chain((y, y, y, y, y)),
... )
>>> [abs(mp.norm(a - b)) <= 1e-10
...  for a, b in zip(pauli.repeat(5).mpps, expect)]
[True, True, True, True]
```

sample(*rng*, *state*, *n_samples*, *method*, *n_group=1*, *mode='auto'*, *pack=False*, *eps=1e-10*)

Random sample from all MP-POVMs on a quantum state

Parameters: See `MPPovm.sample()`.

Return value: Iterable of return values from `MPPovm.sample()`.

unpack_samples(*samples*)

Unpack samples into several integers per sample

Returns Iterator over output from `MPPovm.unpack_samples()`

`mpnum.povm.mppovm.`**`pauli_mpp`**(*nr_sites*, *local_dim*)

Pauli POVM tensor product as MP-POVM

The resulting MP-POVM will contain all tensor products of the elements of the local Pauli POVM from `mpp.pauli_povm()`.

Parameters

- **`nr_sites`** (*int*) – Number of sites of the returned MP-POVM
- **`local_dim`** (*int*) – Local dimension

Return type *MPPovm*

For example, for two qubits the *(1, 3)* measurement outcome is *minus X* on the first and *minus Y* on the second qubit:

```
>>> nr_sites = 2
>>> local_dim = 2
>>> pauli = pauli_mpp(nr_sites, local_dim)
>>> xy = np.kron([1, -1], [1, -1j]) / 2
>>> xyproj = np.outer(xy, xy.conj())
>>> proj = pauli.get([1, 3], astype=mp.MPArray) \
...             .to_array_global().reshape((4, 4))
>>> abs(proj - xyproj / 3**nr_sites).max() <= 1e-10
True
```

The prefactor *1 / 3**nr_sites* arises because X, Y and Z are in a single POVM.

`mpnum.povm.mppovm.`**`pauli_mpps`**(*nr_sites*, *local_dim*)

Pauli POVM tensor product as MP-POVM list

The returned `MPPovmList` contains all tensor products of the single-site X, Y (and Z if *local_dim == 2*) POVMs:

```
>>> import mpnum as mp
>>> import mpnum.povm as mpp
>>> block_sites = 2
>>> ldim = 3
>>> x, y = (mpp.MPPovm.from_local_povm(lp(ldim), 1) for lp in
...           (mpp.x_povm, mpp.y_povm))
>>> pauli = mpp.pauli_mpps(block_sites, ldim)
>>> expect = (
...     mp.chain((x, x)),
...     mp.chain((x, y)),
...     mp.chain((y, x)),
...     mp.chain((y, y)),
... )
>>> [abs(mp.norm(a - b)) <= 1e-10 for a, b in zip(pauli.mpps, expect)]
[True, True, True, True]
```

Parameters

- **`nr_sites`** (*int*) – Number of sites of the returned MP-POVMs
- **`local_dim`** (*int*) – Local dimension

Return type *MPPovmList*

povm.localpovm

An informationally complete d-level POVM.

The POVM simplifies to measuring Paulis matrices in the case of qubits.

class `mpnum.povm.localpovm.`**`POVM`**(*elements*, *info_complete=False*, *pinv=<function pinv>*)

Bases: `object`

Represent a Positive Operator-Valued Measure (POVM).

classmethod from_vectors(*vecs*, *info_complete=False*)

Generates a POVM consisting of rank 1 projectors based on the corresponding vectors.

Parameters

- **vecs** – Iterable of np.ndarray with ndim=1 representing the vectors for the POVM
- **info_complete** – Is the POVM informationally complete (default False)

Returns

informationally_complete

linear_inversion_map

Map that reconstructs a density matrix with linear inversion.

Linear inversion is performed by taking the Moore–Penrose pseudoinverse of self.probability_map.

probability_map

Map that takes a raveled density matrix to the POVM probabilities

The following two return the same:

```
probab = np.array([ np.trace(np.dot(elem, rho)) for elem in a_povm ])
probab = np.dot(a_povm.probability_map, rho.ravel())
```

`mpnum.povm.localpovm.`**`concat`**(*povms*, *weights*, *info_complete=False*)

Combines the POVMs given in *povms* according the weights given to a new POVM.

Parameters

- **povms** – Iterable of POVM
- **weights** – Iterable of real numbers, should sum up to one
- **info_complete** – Is the resulting POVM informationally complete

Returns POVM

`mpnum.povm.localpovm.`**`pauli_parts`**(*dim*)

The POVMs used by `pauli_povm()` as a list

For *dim* > *3*, `x_povm()` and `y_povm()` are returned. For *dim* = *2*, `z_povm()` is included as well.

Parameters **dim** – Dimension of the system

Returns Tuple of `POVMs`

`mpnum.povm.localpovm.`**`pauli_povm`**(*dim*)

An informationally complete d-level POVM that simplifies to measuring Pauli matrices in the case d=2.

Parameters **dim** – Dimension of the system

Returns `POVM` with (generalized) Pauli measurments `mpnum.povm.localpovm.`**`x_povm`**`(`*dim*`)`

The X POVM simplifies to measuring Pauli X eigenvectors for dim=2.

Parameters **`dim`** – Dimension of the system

Returns POVM with generalized X measurments

`mpnum.povm.localpovm.`**`y_povm`**`(`*dim*`)`

The Y POVM simplifies to measuring Pauli Y eigenvectors for dim=2.

Parameters **`dim`** – Dimension of the system

Returns POVM with generalized Y measurments

`mpnum.povm.localpovm.`**`z_povm`**`(`*dim*`)`

The Z POVM simplifies to measuring Pauli Z eigenvectors for dim=2.

Parameters **`dim`** – Dimension of the system

Returns POVM with generalized Z measurments

1.2.8 `special`

Optimized functions

Module contains some specialiced versions of some functions from mparray. They are tuned for speed with special applications in mind

`mpnum.special.`**`inner_prod_mps`**`(`*mpa1*, *mpa2*`)`

Same as `mparray.inner()`, but assumes that *mpa1* is a product MPS

Parameters

- **`mpa1`** – MPArray with one leg per site and rank 1
- **`mpa2`** – MPArray with same shape as mpa1 but arbitrary rank

Returns <mpa1|mpa2>

`mpnum.special.`**`sumup`**`(`*mpas*, *rank*, *weights=None*, *svdfunc=<function truncated_svd>*`)`

Same as `mparray.sumup()` with a consequent compression, but with in-place svd compression. Also, we use a sparse-matrix format for the intermediate local tensors of the sum. Therefore, the memory footprint scales only linearly in the number of summands (instead of quadratically).

Right now, only the sum of product tensors is supported.

Parameters

- **`mpas`** – Iterator over MPArrays
- **`rank`** – Rank of the final result.
- **`weights`** – Iterator of same length as mpas containing weights for computing weighted sum (default: None)
- **`svdfunc`** – Function implementing the truncated svd, for required signature see `truncated_svd()`.

Returns Sum of *mpas* with max. rank *rank*

Possible values for `svdfunc` include:

- `truncated_svd()`: Almost no speedup compared to the standard sumup and compression, since it computes the full SVD

- `scipy.sparse.linalg.svds()`: Only computes the necessary singular values/vectors, but slow if *rank* is not small enough
- *`mpnum.utils.extmath.randomized_svd()`*: Randomized truncated SVD, fast and efficient, but only approximation.

1.2.9 `utils`

`utils.array_transforms`

Helper functions for transforming arrays

`mpnum.utils.array_transforms.`**`global_to_local`**(*array*, *sites*, *left_skip=0*, *right_skip=0*)

Converts a general *sites*-local array with fixed number p of physical legs per site from the global form

A[i_1,. . . , i_N, j_1,. . . , j_N, . . .]

(i.e. grouped by physical legs) to the local form

A[i_1, j_1, . . . , i_2, j_2, . . .]

(i.e. grouped by site).

Parameters

- **`array`** (*`np.ndarray`*) – Array with ndim, such that ndim % sites = 0
- **`sites`** (*`int`*) – Number of distinct sites
- **`left_skip`** (*`int`*) – Ignore that many axes on the left
- **`right_skip`** (*`int`*) – Ignore that many axes on the right

Returns Array with same ndim as array, but reshaped

```
>>> global_to_local(np.zeros((1, 2, 3, 4, 5, 6)), 3).shape
(1, 4, 2, 5, 3, 6)
>>> global_to_local(np.zeros((1, 2, 3, 4, 5, 6)), 2).shape
(1, 3, 5, 2, 4, 6)
```

`mpnum.utils.array_transforms.`**`local_to_global`**(*array*, *sites*, *left_skip=0*, *right_skip=0*)

Inverse of local_to_global

Parameters

- **`array`** (*`np.ndarray`*) – Array with ndim, such that ndim % sites = 0
- **`sites`** (*`int`*) – Number of distinct sites
- **`left_skip`** (*`int`*) – Ignore that many axes on the left
- **`right_skip`** (*`int`*) – Ignore that many axes on the right

Returns Array with same ndim as array, but reshaped

```
>>> ltg, gtl = local_to_global, global_to_local
>>> ltg(gtl(np.zeros((1, 2, 3, 4, 5, 6)), 3), 3).shape
(1, 2, 3, 4, 5, 6)
>>> ltg(gtl(np.zeros((1, 2, 3, 4, 5, 6)), 2), 2).shape
(1, 2, 3, 4, 5, 6)
```

Transform all or only the inner axes:

```
>>> ltg = local_to_global
>>> ltg(np.zeros((1, 2, 3, 4, 5, 6)), 3).shape
(1, 3, 5, 2, 4, 6)
>>> ltg(np.zeros((1, 2, 3, 4, 5, 6)), 2, left_skip=1, right_skip=1).shape
(1, 2, 4, 3, 5, 6)
```

`utils.extmath`

Additional math functions for dealing with dense arrays

`mpnum.utils.extmath.`**`block_diag`**(*summands*, *axes=(0, 1)*)

Block-diagonal sum for n-dimensional arrays.

Perform something like a block diagonal sum (if len(axes) == 2) along the specified axes. All other axes must have identical sizes.

Parameters **`axes`** – Along these axes, perform a block-diagonal sum. Can be negative.

```
>>> a = np.arange(8).reshape((2, 2, 2))
>>> b = np.arange(8, 16).reshape((2, 2, 2))
>>> a
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> b
array([[[ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15]]])
>>> block_diag((a, b), axes=(1, -1))
array([[[ 0,  1,  0,  0],
        [ 2,  3,  0,  0],
        [ 0,  0,  8,  9],
        [ 0,  0, 10, 11]],

       [[ 4,  5,  0,  0],
        [ 6,  7,  0,  0],
        [ 0,  0, 12, 13],
        [ 0,  0, 14, 15]]])
```

`mpnum.utils.extmath.`**`matdot`**(*A*, *B*, *axes=((-1,), (0,))*)

np.tensordot with sane defaults for matrix multiplication

`mpnum.utils.extmath.`**`mkron`**(**args*)

np.kron() with an arbitrary number of n >= 1 arguments

`mpnum.utils.extmath.`**`partial_trace`**(*array*, *traceout*)

Return the partial trace of an array over the sites given in traceout.

Parameters

- **`array`** (*`np.ndarray`*) – Array in global form (see `global_to_local()` above) with exactly 2 legs per site
- **`traceout`** – List of sites to trace out, must be in _ascending_ order

Returns Partial trace over input array

`mpnum.utils.extmath.`**`truncated_svd`**`(`*A*, *k*`)`

Compute the truncated SVD of the matrix *A* i.e. the *k* largest singular values as well as the corresponding singular vectors. It might return less singular values/vectors, if one dimension of *A* is smaller than *k*.

In the background it performs a full SVD. Therefore, it might be inefficient when *k* is much smaller than the dimensions of *A*.

Parameters

- **`A`** – A real or complex matrix
- **`k`** – Number of singular values/vectors to compute

Returns u, s, v, where u: left-singular vectors s: singular values in descending order v: right-singular vectors

`mpnum.utils.extmath.`**`randomized_svd`**`(`*M*, *n_components*, *n_oversamples=10*, *n_iter='auto'*, *piter_normalizer='auto'*, *transpose='auto'*, *randstate=<module 'numpy.random' from '/usr/lib/python3/dist-packages/numpy/random/__init__.py'>*`)`

Computes a truncated randomized SVD. Uses the same convention as `scipy.sparse.linalg.svds()`. However, we guarantee to return the singular values in descending order.

Parameters

- **`M`** – The input data matrix, can be any type that can be converted into a `scipy.linalg.LinarOperator`, e.g. `numpy.ndarray`, or a sparse matrix.
- **`n_components`** (*`int`*) – Number of singular values and vectors to extract.
- **`n_oversamples`** (*`int`*) – Additional number of random vectors to sample the range of *M* so as to ensure proper conditioning. The total number of random vectors used to find the range of M is `n_components + n_oversamples`. Smaller number can improve speed but can negatively impact the quality of approximation of singular vectors and singular values. (default 10)
- **`n_iter`** – Number of power iterations. It can be used to deal with very noisy problems. When `'auto'`, it is set to 4, unless `n_components` is small (`< .1 * min(X.shape)`). Then, `n_iter` is set to 7. This improves precision with few components. (default `'auto'`)
- **`piter_normalizer`** (*`str`*) – `'auto'` (default), `'QR'`, `'LU'`, `'none'`. Whether the power iterations are normalized with step-by-step QR factorization (the slowest but most accurate), `'none'` (the fastest but numerically unstable when *n_iter* is large, e.g. typically 5 or larger), or `'LU'` factorization (numerically stable but can lose slightly in accuracy). The 'auto' mode applies no normalization if `n_iter <= 2` and switches to LU otherwise.
- **`transpose`** – `True`, `False` or `'auto'` Whether the algorithm should be applied to `M.T` instead of `M`. The result should approximately be the same. The `'auto'` mode will trigger the transposition if `M.shape[1] > M.shape[0]` since then the computational overhead in the randomized SVD is generally smaller. (default `'auto'`).
- **`randstate`** – An instance of `numpy.random.RandomState` (default is `np.random`))

Notes

This algorithm finds a (usually very good) approximate truncated singular value decomposition using randomization to speed up the computations. It is particularly fast on large matrices on which you wish to extract only a small number of components. In order to obtain further speed up, `n_iter` can be set <=2 (at the cost of loss of precision).

References

- Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions Halko, et al., 2009 http://arxiv.org/abs/arXiv:0909.4061
- A randomized algorithm for the decomposition of matrices Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert
- An implementation of a randomized algorithm for principal component analysis A. Szlam et al. 2014

`utils.physics`

Code related to physical models

Contents:

- Hamiltonian and analytic ground state energy of the cyclic XY model

References:

`mpnum.utils.physics.`**`cXY_E0`**`(`*nr_sites*, *gamma*`)`

Ground state energy of the cyclic XY model

Parameters

- **`nr_sites`** – Number of spin one-half sites
- **`gamma`** – Asymmetry parameter

Returns Exact energy of the ground state

This function is implemented for `nr_sites` which is an odd multiple of two. In this case, the ground state energy of the XY model is given by (Eqs. (A-12), (2.20) of *[LSM61]*)

$$E_0 = -\frac{1}{2}\sum_{l=0}^{N-1}\Lambda_{k(l)}$$

with (Eqs. (2.18b), (2.18c))

$$\Lambda_k^2 = 1 - (1-\gamma^2)[\sin(k)]^2, \quad k(l) = \frac{2\pi}{N}\left(l - \frac{N}{2}\right)$$

and $\Lambda_k \geq 0$.

`mpnum.utils.physics.`**`cXY_local_terms`**`(`*nr_sites*, *gamma*`)`

Local terms of the cyclic XY model (MPOs)

Parameters

- **`nr_sites`** – Number of spin one-half sites
- **`gamma`** – Asymmetry parameter

Returns List `terms` of length `nr_sites` (MPOs)

The term `terms[i]` acts on spins `(i, i + 1)` and spin `nr_sites` is the same as the first spin.

The Hamiltonian of the cyclic XY model is given by [LSM61, Eq. 2.1]:

$$H_\gamma = \sum_{i=1}^{N} (1+\gamma) S_i^x S_{i+1}^x + (1-\gamma) S_i^y S_{i+1}^y$$

with $S_{N+1}^j = S_1^j$. The function `cXY_E0()` returns the exact ground state energy of this Hamiltonian.

`mpnum.utils.physics.`**`mpo_cH`**`(`*terms*`)`

Construct an MPO cyclic nearest-neighbour Hamiltonian

Parameters **`terms`** – List of nearst-neighbour terms (MPOs, see return value of `cXY_local_terms()`)

Returns The Hamiltonian as MPO

Note: It may not be advisable to call `mp.MPArray.canonicalize()` on a Hamiltonian, e.g.:

```
>>> mpoH = mpo_cH(cXY_local_terms(nr_sites=100, gamma=0))
>>> abs1 = max(abs(lt).max() for lt in mpoH.lt)
>>> mpoH.canonicalize()
>>> abs2 = np.round(max(abs(lt).max() for lt in mpoH.lt), -3)
>>> print('{:.3f}  {:.2e}'.format(abs1, abs2))
1.000  2.79e+15
```

The Hamiltonian generally has a large Frobenius norm because local terms are embedded with identity matrices. This causes large tensor entries of canonicalization which will eventually overflow the numerical maximum (the overflow happens somewhere between 2000 and 3000 sites in this example). One could embed local terms with Frobenius-normalized identity matrices instead, but this would make the eigenvalues of H exponentially (in `nr_sites`) small. This would eventually cause numerical underflows.

`mpnum.utils.physics.`**`sparse_cH`**`(`*terms*, *ldim=2*`)`

Construct a sparse cyclic nearest-neighbour Hamiltonian

Parameters

- **`terms`** – List of nearst-neighbour terms (square array or MPO, see return value of `cXY_local_terms()`)
- **`ldim`** – Local dimension

Returns The Hamiltonian as sparse matrix

1.2.10 Todo list (autogenerated)

Todo: single site MPAs – what is left?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray, line 3.)

Todo: Local tensor ownership – see MPArray class comment

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray, line 4.)

Todo: Possible optimization:

- replace integer-for loops with iterator (not obviously possible everwhere)
- replace internal structure as list of arrays with lazy generator of arrays (might not be possible, since we often iterate both ways!)
- more in place operations for addition, subtraction, multiplication

---

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray, line 5.)

Todo: Replace all occurences of self._ltens with self[...] or similar & benchmark. This will allow easier transition to lazy evaluation of local tensors

---

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray, line 12.)

Todo: As it is now, e.g. `__imul__()` modifies items from `self._ltens`. This requires e.g. `chain()` to take copies of the local tensors. The data model seems to be that an *MPArray* instance owns its local tensors and everyone else, including each new *MPArray* instance, must take copies. Is this correct?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray.MPArray, line 18.)

Todo: More appropriate naming for this functions?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray.MPArray.leg2vleg, line 6.)

Todo: Why is this here? What's wrong with the purne function?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray.MPArray.reshape, line 10.)

Todo: More appropriate naming for this functions?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray.MPArray.vleg2leg, line 11.)

Todo: Make this canonicalization aware

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstri
of mpnum.mparray.chain, line 10.)

Todo: Raise warning when casting complex to real dtype

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstrin of mpnum.mparray.chain, line 11.)

Todo: This table needs cell borders in the HTML output (-> CSS) and the tabularcolumns command doesn't work.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mparray.py:docstrin of mpnum.mparray.regular_slices, line 24.)

Todo: Are derived classes MPO/MPS/PMPS of any help?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mpsmpo.py:docstri of mpnum.mpsmpo, line 99.)

Todo: I am not sure the current definition of PMPS is the most elegant for our purposes…

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mpsmpo.py:docstri of mpnum.mpsmpo, line 101.)

Todo: Add docstring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/mpsmpo.py:docstri of mpnum.mpsmpo.reductions, line 1.)

Todo: Add information on how the runtime of `eig()` and `eig_sum()` scale with the the different ranks. For the time being, refer to the benchmark test.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/linalg.py:docstring of mpnum.linalg.eig_sum, line 13.)

Todo: Explain the details of the variance estimation, in particular the difference between the variances returned from `MPPovmList.lfun()` and `MPPovmList.lfun_from()`. Check the mean square error.

Add a good references explaining all facts mentioned above and for further reading.

Document the runtime and memory cost of the functions.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py: of mpnum.povm.mppovm, line 116.)

Todo: This class should provide a function which returns expectation values as full array. (Even though computing expectation values using the POVM struture brings advantages, we usually need the result as full array.) This function should also replace small negative probabilities by zero and canonicalize the sum of all probabilities to unity (if the deviation is non-zero but small). The same checks should also be implemented in localpovm.POVM.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py:
of mpnum.povm.mppovm.MPPovm, line 19.)

Todo: Right now we use this class for multi-site POVMs with elements obtained from every possible combination of the elements of single-site POVMs: The POVM index is split across all sites. Explore whether and how this concept can also be useful in other cases.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py:
of mpnum.povm.mppovm.MPPovm, line 28.)

Todo: Add docstring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py:
of mpnum.povm.mppovm.MPPovm.block_pmfs_as_array, line 1.)

Todo: Add docstring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py:
of mpnum.povm.mppovm.MPPovm.pmfs_as_array, line 1.)

Todo: Add docstring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py:
of mpnum.povm.mppovm.MPPovmList.block_pmfs_as_array, line 1.)

Todo: Add docstring

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/mpnum/povm/mppovm.py:
of mpnum.povm.mppovm.MPPovmList.pmfs_as_array, line 1.)

Todo: Reference to Schollwoeck not working anymore.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/mpnum/checkouts/latest/docs/mpnum.rst, line 118.)

Todo: Reference to Schollwoeck not working anymore.

Note: `make livehtml` (based on sphinx-autobuild) does not rebuild this list.

1.3 Introductory Notebook to mpnum

mpnum implements *matrix product arrays* (MPA), which are efficient parameterizations of certain multi-partite arrays. Special cases of the MPA structure, which are omnipresent in many-body quantum physics, are *matrix product states*

(MPS) and *matrix product operators* (MPO) with one and two array indices per site, respectively. In the applied math community, matrix product states are also known as *tensor trains* (TT).

The main class implementing an MPA with arbitrary number of array indices (or "physical legs") is `mpnum.MPArray`.

```
In [1]: import numpy as np
        import numpy.linalg as la

        import mpnum as mp
```

1.3.1 MPA and MPS basics

A convenient example to deal with is a random MPA. First, we create a fixed seed, then a random MPA:

```
In [2]: rng = np.random.RandomState(seed=42)
        mpa = mp.random_mpa(sites=4, ldim=2, rank=3, randstate=rng, normalized=True)
```

The MPA is an instance of the MPArray class:

```
In [3]: mpa
```

```
Out[3]: <mpnum.mparray.MPArray at 0x7f65672eaa90>
```

Number of sites:

```
In [4]: len(mpa)
```

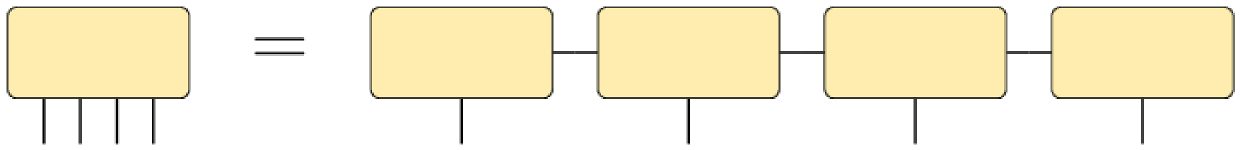
```
Out[4]: 4
```

Number of physical legs at each site (=number of array indices at each site):

```
In [5]: mpa.ndims
```

```
Out[5]: (1, 1, 1, 1)
```

Because the MPA has one physical leg per site, we have created a matrix product *state* (i.e. a tensor train). In the graphical notation, this MPS looks like this



Note that `mpnum` internally stores the local tensors of the matrix product representation on the right hand side. We see below how to obtain the "dense" tensor from an `MPArray`. Dimension of each physical leg:

```
In [6]: mpa.shape
```

```
Out[6]: ((2,), (2,), (2,), (2,))
```

Note that the number and dimension of the physical legs at each site can differ (altough this is rarely used in practice).

Representation ranks (aka compression ranks) between each pair of sites:

```
In [7]: mpa.ranks
```

```
Out[7]: (2, 3, 2)
```

In physics, the representation ranks are usually called the *bond dimensions* of the representation.

Dummy bonds before and after the chain are omitted in `mpa.ranks`. (Currently, mpnum only implements open boundary conditions.)

Above, we have specified `normalized=True`. Therefore, we have created an MPA with ℓ_2-norm 1. In case the MPA does not represent a vector but has more physical legs, it is nonetheless treated as a vector. Hence, for operators mp.norm implements the Frobenius norm.

```
In [8]: mp.norm(mpa)
```

```
Out[8]: 1.0000000000000002
```

Convert to a dense array, which should be used with care due because the memory used increases exponentially with the number of sites:

```
In [9]: arr = mpa.to_array()
        arr.shape
```

```
Out[9]: (2, 2, 2, 2)
```

The resulting full array has one index for each physical leg.

Now convert the full array back to an MPA:

```
In [10]: mpa2 = mp.MPArray.from_array(arr)
         len(mpa2)
```

```
Out[10]: 1
```

We have obtained an MPA with length 1. This is not what we expected. The reason is that by default, all legs are placed on a single site (also notice the difference between `mpa2.shape` here and `mpa.shape` from above):

```
In [11]: mpa2.shape
```

```
Out[11]: ((2, 2, 2, 2),)
```

```
In [12]: mpa.shape
```

```
Out[12]: ((2,), (2,), (2,), (2,))
```

We obtain the desired result by specifying the number of legs per site we want:

```
In [13]: mpa2 = mp.MPArray.from_array(arr, ndims=1)
         len(mpa2)
```

```
Out[13]: 4
```

Finally, we can compute the norm distance between the two MPAs. (Again, the Frobenius norm is used.)

```
In [14]: mp.norm(mpa - mpa2)
```

```
Out[14]: 7.2998268912398721e-16
```

Since this is an often used operation and allows for additional optimization (not implemented currently), it is advisable to use the specific mp.normdist for this:

```
In [15]: mp.normdist(mpa, mpa2)
```

```
Out[15]: 7.2998268912398721e-16
```

Sums, differences and scalar multiplication of MPAs is done with the normal operators:

```
In [16]: mp.norm(3 * mpa)
```

```
Out[16]: 3.0000000000000009
```

```
In [17]: mp.norm(mpa + 0.5 * mpa)
```

```
Out[17]: 1.5000000000000011
```

```
In [18]: mp.norm(mpa - 1.5 * mpa)
```

```
Out[18]: 0.50000000000000133
```

Multiplication with a scalar leaves the bond dimension unchanged:

```
In [19]: mpa.ranks
```

```
Out[19]: (2, 3, 2)
```

```
In [20]: (3 * mpa).ranks
```

```
Out[20]: (2, 3, 2)
```

The bond dimensions of a sum (or difference) are given by the sums of the bond dimensions:

```
In [21]: mpa2 = mp.random_mpa(sites=4, ldim=2, rank=2, randstate=rng)
         mpa2.ranks
```

```
Out[21]: (2, 2, 2)
```

```
In [22]: (mpa + mpa2).ranks
```

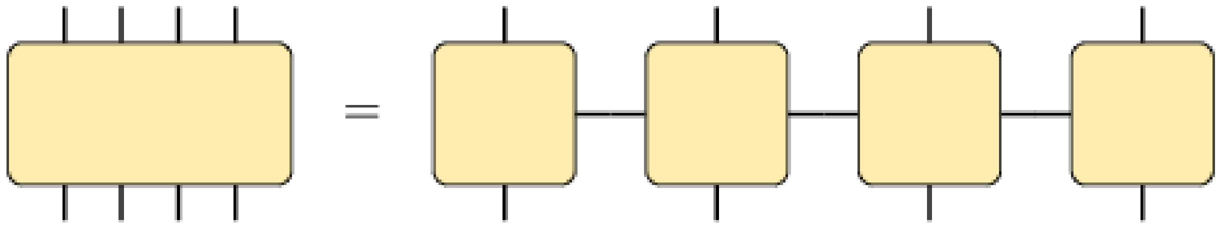
```
Out[22]: (4, 5, 4)
```

1.3.2 MPO basics

First, we create a random MPA with two physical legs per site:

```
In [23]: mpo = mp.random_mpa(sites=4, ldim=(3, 2), rank=3, randstate=rng, normalized=True)
```

In graphical notation, `mpo` looks like this



It's basic properties are:

```
In [24]: [len(mpo), mpo.ndims, mpo.ranks]
```

```
Out[24]: [4, (2, 2, 2, 2), (3, 3, 3)]
```

Each site has two physical legs, one with dimension 3 and one with dimension 2. This corresponds to a non-square full array.

```
In [25]: mpo.shape
```

```
Out[25]: ((3, 2), (3, 2), (3, 2), (3, 2))
```

Now convert the mpo to a full array:

```
In [26]: mpo_arr = mpo.to_array()
         mpo_arr.shape
```

```
Out[26]: (3, 2, 3, 2, 3, 2, 3, 2)
```

We refer to this arangement of axes as *local form*, since indices which correspond to the same site are neighboring. This is a natural form for the MPO representation. However, for some operations it is necessary to have row and column indices grouped together – we refer to this as *global form*:

```
In [27]: from mpnum.utils.array_transforms import local_to_global

         mpo_arr = mpo.to_array()
         mpo_arr = local_to_global(mpo_arr, sites=len(mpo))
         mpo_arr.shape
```

```
Out[27]: (3, 3, 3, 3, 2, 2, 2, 2)
```

This gives the expected result. Note that it is crucial to specify the correct number of sites, otherwise we do not get what we want:

```
In [28]: mpo_arr = mpo.to_array()
         mpo_arr = local_to_global(mpo_arr, sites=2)
         mpo_arr.shape
```

```
Out[28]: (3, 3, 2, 2, 3, 3, 2, 2)
```

As an alternative, there is the following shorthand:

```
In [29]: mpo_arr = mpo.to_array_global()
         mpo_arr.shape
```

```
Out[29]: (3, 3, 3, 3, 2, 2, 2, 2)
```

An array in global form can be converted into matrix-product form with the following API:

```
In [30]: mpo2 = mp.MPArray.from_array_global(mpo_arr, ndims=2)
         mp.normdist(mpo, mpo2)
```

```
Out[30]: 1.0881840590136613e-15
```

1.3.3 MPO-MPS product and arbitrary MPA-MPA products

We can now compute the matrix-vector product of `mpa` from above (which is an MPS) and `mpo`.

```
In [31]: mpa.shape
```

```
Out[31]: ((2,), (2,), (2,), (2,))
```

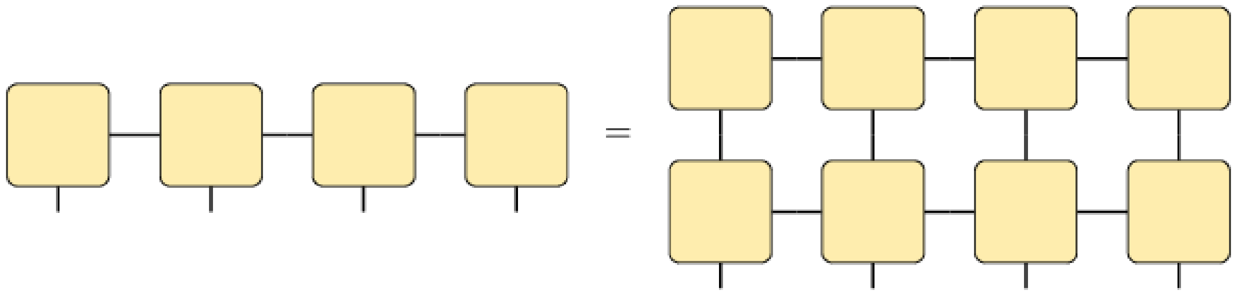
```
In [32]: mpo.shape
```

```
Out[32]: ((3, 2), (3, 2), (3, 2), (3, 2))
```

```
In [33]: prod = mp.dot(mpo, mpa, axes=(-1, 0))
         prod.shape
```

```
Out[33]: ((3,), (3,), (3,), (3,))
```

The result is a new MPS, with local dimension changed by `mpo` and looks like this:



The `axes` argument is optional and defaults to `axes=(-1, 0)` – i.e. contracting, at each site, the last pyhsical index of the first factor with the first physical index of the second factor. More specifically, the `axes` argument specifies which *physical legs* should be contracted: `axes[0]` specifies the physical in the first argument, and `axes[1]` specifies the physical leg in the second argument. This means that the same product can be achieved with

```
In [34]: prod2 = mp.dot(mpa, mpo, axes=(0, 1))
         mp.normdist(prod, prod2)
```

```
Out[34]: 1.7794893594008944e-16
```

Note that in any case, the ranks of the output of `mp.dot` are the *products* of the original ranks:

```
In [35]: mpo.ranks, mpa.ranks, prod.ranks
```

```
Out[35]: ((3, 3, 3), (2, 3, 2), (6, 9, 6))
```

Now we compute the same product using the full arrays `arr` and `mpo_arr`:

```
In [36]: arr_vec = arr.ravel()
         mpo_arr = mpo.to_array_global()
         mpo_arr_matrix = mpo_arr.reshape((81, 16))
         prod3_vec = np.dot(mpo_arr_matrix, arr_vec)
         prod3_vec.shape
```

```
Out[36]: (81,)
```

As you can see, we need to reshape the result `prod3_vec` before we can convert it back to an MPA:

```
In [37]: prod3_arr = prod3_vec.reshape((3, 3, 3, 3))
         prod3 = mp.MPArray.from_array(prod3_arr, ndims=1)
         prod3.shape
```

```
Out[37]: ((3,), (3,), (3,), (3,))
```

Now we can compare the two results:

```
In [38]: mp.normdist(prod, prod3)
```

```
Out[38]: 2.0433926816958574e-16
```

We can also compare by converting `prod` to a full array:

```
In [39]: prod_arr = prod.to_array()
         la.norm((prod3_arr - prod_arr).reshape(81))
```

```
Out[39]: 1.0434960119970279e-16
```

1.3.4 Converting full operators to MPOs

While MPO algorithms avoid using full operators in general, we will need to convert a term acting on only two sites to an MPO in order to continue with MPO operations; i.e. we will need to convert a full array to an MPO.

First, we define a full operator:

```
In [40]: CZ = np.array([[ 1.,  0.,  0.,  0.],
                        [ 0.,  1.,  0.,  0.],
                        [ 0.,  0.,  1.,  0.],
                        [ 0.,  0.,  0., -1.]])
```

This operator is the so-called *controlled Z* gate: Apply `Z` on the second qubit if the first qubit is in state `e2`.

To convert it to an MPO, we have to reshape:

```
In [41]: CZ_arr = CZ.reshape((2, 2, 2, 2))
```

Now we can create an MPO, being careful to specify the correct number of legs per site:

```
In [42]: CZ_mpo = mp.MPArray.from_array_global(CZ_arr, ndims=2)
```

To test it, we apply the operator to the state which has both qubits in state `e2`:

```
In [43]: vec = np.kron([0, 1], [0, 1])
         vec
```

```
Out[43]: array([0, 0, 0, 1])
```

Reshape and convert to an MPS:

```
In [44]: vec_arr = vec.reshape([2, 2])
         mps = mp.MPArray.from_array(vec_arr, ndims=1)
```

Now we can compute the matrix-vector product:

```
In [45]: out = mp.dot(CZ_mpo, mps)
         out.to_array().ravel()
```

```
Out[45]: array([ 0.,  0.,  0., -1.])
```

The output is as expected: We have acquired a minus sign.

We have to be careful to use `from_array_global` and not `from_array` for `CZ_mpo`, because the `CZ_arr` is in *global form*. Here, all physical legs have the same dimension, so we can use `from_array` without error:

```
In [46]: CZ_mpo2 = mp.MPArray.from_array(CZ_arr, ndims=2)
```

However, the result is not what we want:

```
In [47]: out2 = mp.dot(CZ_mpo2, mps)
         out2.to_array().ravel()
```

```
Out[47]: array([ 1.,  0.,  0., -1.])
```

The reason is easy to see: We have applied the following matrix to our state:

```
In [48]: CZ_mpo2.to_array_global().reshape(4, 4)
```

```
Out[48]: array([[ 1.,  0.,  0.,  1.],
               [ 0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.],
               [ 1.,  0.,  0., -1.]])
```

Keep in mind that we have to use `to_array_global` before the reshape. Using `to_array` would not provide us the matrix which we have applied to the state with `mp.dot`. Instead, it will exactly return the input:

```
In [49]: CZ_mpo2.to_array().reshape(4, 4)
```

```
Out[49]: array([[ 1.,  0.,  0.,  0.],
               [ 0.,  1.,  0.,  0.],
               [ 0.,  0.,  1.,  0.],
               [ 0.,  0.,  0., -1.]])
```

Again, `from_array_global` is just the shorthand for the following:

```
In [50]: from mpnum.utils.array_transforms import global_to_local

         CZ_mpo3 = mp.MPArray.from_array(global_to_local(CZ_arr, sites=2), ndims=2)

         mp.normdist(CZ_mpo, CZ_mpo3)
```

```
Out[50]: 1.5700924586837752e-16
```

As you can see, in the explicit version you must submit both the correct number of sites and the correct number of physical legs per site. Therefore, the function `MPArray.from_array_global` simplifies the conversion.

1.3.5 Creating MPAs from Kronecker products

It is a frequent task to create an MPS which represents the product state of $|0\rangle$ on each qubit. If the chain is very long, we cannot create the full array with `np.kron` and use `MPArray.from_array` afterwards because the array would be too large.

In the following, we describe how to efficiently construct an MPA representation of a Kronecker product of vectors. The same methods can be used to efficiently construct MPA representations of Kronecker products of operators or tensors with three or more indices.

First, we need the state on a single site:

```
In [51]: e1 = np.array([1, 0])
         e1
```

```
Out[51]: array([1, 0])
```

Then we can use `from_kron` to directly create an MPS representation of the Kronecker product:

```
In [52]: mps = mp.MPArray.from_kron([e1, e1, e1])
         mps.to_array().ravel()
```

```
Out[52]: array([1, 0, 0, 0, 0, 0, 0, 0])
```

This works well for large numbers of sites because the needed memory scales linearly with the number of sites:

```
In [53]: mps = mp.MPArray.from_kron([e1] * 2000)
         len(mps)
```

```
Out[53]: 2000
```

An even more pythonic solution is the use of iterators in this example:

```
In [54]: from itertools import repeat

         mps = mp.MPArray.from_kron(repeat(e1, 2000))
         len(mps)
```

```
Out[54]: 2000
```

Do not call `.to_array()` on this state!

The bond dimension of the state is 1, because it is a product state:

```
In [55]: np.array(mps.ranks)  # Convert to an array for nicer display
```

```
Out[55]: array([1, 1, 1, ..., 1, 1, 1])
```

We can also create a single-site MPS:

```
In [56]: mps1 = mp.MPArray.from_array(e1, ndims=1)
         len(mps1)
```

```
Out[56]: 1
```

After that, we can use `mp.chain` to create Kronecker products of the MPS directly:

```
In [57]: mps = mp.chain([mps1, mps1, mps1])
         len(mps)
```

```
Out[57]: 3
```

It returns the same result as before:

```
In [58]: mps.to_array().ravel()
```

```
Out[58]: array([1, 0, 0, 0, 0, 0, 0, 0])
```

We can also use `mp.chain` on the three-site MPS:

```
In [59]: mps = mp.chain([mps] * 100)
         len(mps)
```

```
Out[59]: 300
```

Note that `mp.chain` interprets the factors in the tensor product as distinct sites. Hence, the factors do not need to be of the same length or even have the same number of indices. In contrast, there is also `mp.localouter`, which computes the tensor product of MPArrays with the same number of sites:

```
In [60]: mps = mp.chain([mps1] * 4)
         len(mps), mps.shape,
```

```
Out[60]: (4, ((2,), (2,), (2,), (2,)))
```

```
In [61]: rho = mp.localouter(mps.conj(), mps)
         len(rho), rho.shape

Out[61]: (4, ((2, 2), (2, 2), (2, 2), (2, 2)))
```

1.3.6 Compression

A typical matrix product based numerical algorithm performs many additions or multiplications of MPAs. As mentioned above, both operations increase the rank. If we let the bond dimension grow, the amount of memory we need grows with the number of operations we perform. To avoid this problem, we have to find an MPA with a smaller rank which is a good approximation to the original MPA.

We start by creating an MPO representation of the identity matrix on 6 sites with local dimension 3:

```
In [62]: op = mp.eye(sites=6, ldim=3)

In [63]: op.shape

Out[63]: ((3, 3), (3, 3), (3, 3), (3, 3), (3, 3), (3, 3))
```

As it is a tensor product operator, it has rank 1:

```
In [64]: op.ranks

Out[64]: (1, 1, 1, 1, 1)
```

However, addition increases the rank:

```
In [65]: op2 = op + op + op
         op2.ranks

Out[65]: (3, 3, 3, 3, 3)
```

Matrix multiplication multiplies the individual ranks:

```
In [66]: op3 = mp.dot(op2, op2)
         op3.ranks

Out[66]: (9, 9, 9, 9, 9)
```

(NB: `compress` or `compression` below can call `canonicalize` on the MPA, which in turn could already reduce the rank to 1 in case the rank can be compressed without error. Keep that in mind.)

Keep in mind that the operator represented by `op3` is still the identity operator, i.e. a tensor product operator. This means that we expect to find a good approximation with low rank easily. Finding such an approximation is called *compression* and is achieved as follows:

```
In [67]: op3 /= mp.norm(op3.copy())  # normalize to make overlap meaningful
         copy = op3.copy()
         overlap = copy.compress(method='svd', rank=1)
         copy.ranks

Out[67]: (1, 1, 1, 1, 1)
```

Calling `compress` on an MPA replaces the MPA in place with a version with smaller bond dimension. Overlap gives the absolute value of the (Hilbert-Schmidt) inner product between the original state and the output:

```
In [68]: overlap

Out[68]: 0.99999999999999911
```

Instead of in-place compression, we can also obtain a compressed copy:

```
In [69]: compr, overlap = op3.compression(method='svd', rank=2)
         overlap, compr.ranks, op3.ranks

Out[69]: (0.99999999999999911, (2, 2, 2, 2, 2), (9, 9, 9, 9, 9))
```

SVD compression can also be told to meet a certain truncation error (see the documentation of `mp.MPArray.compress` for details).

```
In [70]: compr, overlap = op3.compression(method='svd', relerr=1e-6)
         overlap, compr.ranks, op3.ranks
```

```
Out[70]: (0.99999999999999911, (1, 1, 1, 1, 1), (9, 9, 9, 9, 9))
```

We can also use variational compression instead of SVD compression:

```
In [71]: compr, overlap = op3.compression(method='var', rank=2, num_sweeps=10, var_sites=2)
         # Convert overlap from numpy array with shape () to float for nicer display:
         overlap = overlap.flat[0]
         complex(overlap), compr.ranks, op3.ranks
```

```
Out[71]: ((1+0j), (2, 2, 2, 2, 2), (9, 9, 9, 9, 9))
```

As a reminder, it is always advisable to check whether the overlap between the input state and the compression is large enough. In an involved algorithm, it can be useful to store the compression error at each invocation of compression.

1.3.7 MPO sum of local terms

A frequent task is to compute the MPO representation of a local Hamiltonian, i.e. of an operator of the form

$$H = \sum_{i=1}^{n-1} h_{i,i+1}$$

$$where : math : `h_{i,i+1}`actsonlyonsites : math : `i`and$$

$i + 1$. This means that $h_{i,i+1} = \nVdash_{i-1} \otimes h'_{i,i+1} \otimes \nVdash_{n-w+1}$ where $\nVdash_k$ is the identity matrix on k sites and $w = 2$ is the width of $h'_{i,i+1}$.

We show how to obtain an MPO representation of such a Hamiltonian. First of all, we need to define the local terms. For simplicity, we choose $h'_{i,i+1} = \sigma_Z \otimes \sigma_Z$ independently of i.

```
In [72]: zeros = np.zeros((2, 2))
         zeros
```

```
Out[72]: array([[ 0.,  0.],
                [ 0.,  0.]])
```

```
In [73]: idm = np.eye(2)
         idm
```

```
Out[73]: array([[ 1.,  0.],
                [ 0.,  1.]])
```

```
In [74]: # Create a float array instead of an int array to avoid problems later
         Z = np.diag([1., -1])
         Z
```

```
Out[74]: array([[ 1.,  0.],
                [ 0., -1.]])
```

```
In [75]: h = np.kron(Z, Z)
         h
```

```
Out[75]: array([[ 1.,  0.,  0.,  0.],
                [ 0., -1.,  0., -0.],
                [ 0.,  0., -1., -0.],
                [ 0., -0., -0.,  1.]])
```

First, we have to convert the local term `h` to an MPO:

```
In [76]: h_arr = h.reshape((2, 2, 2, 2))
         h_mpo = mp.MPArray.from_array_global(h_arr, ndims=2)
         h_mpo.ranks

Out[76]: (4,)
```

h_mpo has rank 4 even though h is a tensor product. This is far from optimal. We improve things as follows: (We could also compress `h_mpo`.)

```
In [77]: h_mpo = mp.MPArray.from_kron([Z, Z])
         h_mpo.ranks

Out[77]: (1,)
```

The most simple way is to implement the formula from above with MPOs: First we compute the $h_{i,i+1}$ from the $h'_{i,i+1}$:

```
In [78]: width = 2
         sites = 6
         local_terms = []

         for startpos in range(sites - width + 1):
             left = [mp.MPArray.from_kron([idm] * startpos)] if startpos > 0 else []
             right = [mp.MPArray.from_kron([idm] * (sites - width - startpos))] \
                 if sites - width - startpos > 0 else []
             h_at_startpos = mp.chain(left + [h_mpo] + right)
             local_terms.append(h_at_startpos)

         local_terms

Out[78]: [<mpnum.mparray.MPArray at 0x7f6563b08588>,
          <mpnum.mparray.MPArray at 0x7f6563b084e0>,
          <mpnum.mparray.MPArray at 0x7f6563b086a0>,
          <mpnum.mparray.MPArray at 0x7f6563b08710>,
          <mpnum.mparray.MPArray at 0x7f6563b08630>]
```

Next, we compute the sum of all the local terms and check the bond dimension of the result:

```
In [79]: H = local_terms[0]

         for local_term in local_terms[1:]:
             H += local_term

         H.ranks

Out[79]: (5, 5, 5, 5, 5)
```

The ranks are explained by the ranks of the local terms:

```
In [80]: [local_term.ranks for local_term in local_terms]

Out[80]: [(1, 1, 1, 1, 1),
          (1, 1, 1, 1, 1),
          (1, 1, 1, 1, 1),
          (1, 1, 1, 1, 1),
          (1, 1, 1, 1, 1)]
```

We just have to add the ranks at each position.

`mpnum` provides a function which constructs `H` from `h_mpo`, with an output MPO with smaller rank by taking into account the trivial action on some sites:

```
In [81]: H2 = mp.local_sum([h_mpo] * (sites - width + 1))
         H2.ranks

Out[81]: (2, 3, 3, 3, 2)
```

Without additional arguments, `mp.local_sum()` just adds the local terms with the first term starting on site 0, the second on site 1 and so on. In addition, the length of the chain is chosen such that the last site of the chain coincides with the last site of the last local term. Other constructions can be obtained by prodividing additional arguments.

We can check that the two Hamiltonians are equal:

```
In [82]: mp.normdist(H, H2)
```

```
Out[82]: 6.4354640488548389e-15
```

Of course, this means that we could just compress `H`:

```
In [83]: H_comp, overlap = H.compression(method='svd', rank=3)
         overlap / mp.norm(H)**2
```

```
Out[83]: 0.99999999999999889
```

```
In [84]: H_comp.ranks
```

```
Out[84]: (3, 3, 3, 3, 3)
```

We can also check the minimal bond dimension which can be achieved with SVD compression with small error:

```
In [85]: H_comp, overlap = H.compression(method='svd', relerr=1e-6)
         overlap / mp.norm(H)**2
```

```
Out[85]: 0.99999999999999933
```

```
In [86]: H_comp.ranks
```

```
Out[86]: (2, 3, 3, 3, 2)
```

1.3.8 MPS, MPOs and PMPS

We can represent vectors (e.g. pure quantum states) as MPS, we can represent arbitrary matrices as MPO and we can represent positive semidefinite matrices as purifying matrix product states (PMPS). For mixed quantum states, we can thus choose between the MPO and PMPS representations.

As mentioned in the introduction, MPS and MPOs are handled as MPAs with one and two physical legs per site. In addition, PMPS are handled as MPAs with two physical legs per site, where the first leg is the "system" site and the second leg is the corresponding "ancilla" site.

From MPS and PMPS representations, we can easily obtain MPO representations. `mpnum` provides routines for this:

```
In [87]: mps = mp.random_mpa(sites=5, ldim=2, rank=3, normalized=True)
         mps_mpo = mp.mps_to_mpo(mps)
         mps_mpo.ranks
```

```
Out[87]: (4, 9, 9, 4)
```

As expected, the rank of `mps_mpo` is the square of the rank of `mps`.

Now we create a PMPS with system site dimension 2 and ancilla site dimension 3:

```
In [88]: pmps = mp.random_mpa(sites=5, ldim=(2, 3), rank=3, normalized=True)
         pmps.shape
```

```
Out[88]: ((2, 3), (2, 3), (2, 3), (2, 3), (2, 3))
```

```
In [89]: pmps_mpo = mp.pmps_to_mpo(pmps)
         pmps_mpo.ranks
```

```
Out[89]: (9, 9, 9, 9)
```

Again, the rank is squared, as expected. We can verify that the first physical leg of each site of `pmps` is indeed the system site by checking the shape of `pmps_mpo`:

```
In [90]: pmps_mpo.shape
Out[90]: ((2, 2), (2, 2), (2, 2), (2, 2), (2, 2))
```

1.3.9 Local reduced states

For state tomography applications, we frequently need the local reduced states of an MPS, MPO or PMPS. We provide the following functions for this task:

- `mp.reductions_mps_as_pmps()`: Input: MPS, output: local reductions as PMPS
- `mp.reductions_mps_as_mpo()`: Input: MPS, output: local reductions as MPO
- `mp.reductions_pmps()`: Input: PMPS, output: Local reductions as PMPS
- `mp.reductions_mpo()`: Input: MPO, output: Local reductions as MPO

The arguments of all functions are similar, e.g.:

```
In [91]: width = 3
         startsites = range(len(pmps) - width + 1)
         for startsite, red in zip(startsites, mp.reductions_pmps(pmps, width, startsites)):
             print('Reduction starting on site', startsite)
             print('bdims:', red.ranks)
             red_mpo = mp.pmps_to_mpo(red)
             print('trace:', mp.trace(red_mpo))
             print()
```

```
Reduction starting on site 0
bdims: (3, 3)
trace: 1.0

Reduction starting on site 1
bdims: (3, 3)
trace: 1.0

Reduction starting on site 2
bdims: (3, 3)
trace: 1.0
```

Because `pmps` was a normalized state, the trace of the reduced states is close to 1.

You can omit the `startsites` argument: The default behaviour is the first reductions starting on site 0, the second on site 1, and so on (which is just what we have requested). The functions for reduced states can also compute different constructions by providing different arguments not described here.

1.4 Development & Contributing

This section contains information for anyone who wishes to contribute to mpnum. Contributions and pull requests for mpnum are very welcome.

Contents

1.4.1 Code style

All contributions should be formated according to the PEP8 standard. Slightly more than 80 characters can sometimes be tolerated if increased line width increases readability.

1.4.2 Unit tests

After any change to mpnum, it should be verified that the test suite runs without any errors. For any new functionality, please provide suitable unit tests. Also, if you find a bug, consider adding a test that detects the bug before fixing it.

A short set of tests takes less than 30 seconds and is invoked with one of

```
python -m pytest
python setup.py test
```

Note that the second command also installs the dependencies for tests if they are not present. However, since this command ignores wheel files for the dependencies, it tries to install *h5py* from source on many systems. This is not trivial and might take some time since it builds the HDF5 binaries from scratch. A better way is to install binaries for the test dependencies via running the following command from the *mpnum* source code root directory

```
pip install -r requirements.txt
```

An intermediate set of tests, which takes about 2 minutes to run, is executed automatically for every commit on GitHub via Travis continuous integration. It can be run locally via

```
python -m pytest -m "not verylong"
bash tests/travis.sh
```

A long set of tests takes about 30 minutes and is invoked with

```
python -m pytest -m 1
```

Unit tests are implemented using pytest. Every addition to mpnum should be accompanied by corresponding unit tests. Make sure to use the right pytest-mark for each test. The intermediate and long running tests should be marked with the ‘long’ and ‘verylong’ pytest mark, respectively.

1.4.3 Test coverage

Code not covered by unit tests can be detected with pytest-cov. A HTML coverage report can be generated using

```
python -m pytest --cov-report term --cov-report html --cov=mpnum
```

Afterwards, the HTML coverage report is available in `htmlcov/index.html`.

1.4.4 Benchmark tests

In addition to unit tests, there are benchmark tests which measure the runtime of certain functions. To run all benchmark tests, run

```
python -m pytest -m benchmark
```

1.4.5 Building the documentation

The HTML documentation uses Sphinx. Building the documentation requires the RTD theme:

```
conda install sphinx_rtd_theme  # or
pip install sphinx_rtd_theme
```

On Linux/MacOS, the documentation can be built with a simple

```
make -C docs html
```

or

```
cd docs; make html
```

After the build, the HTML documentation is available at `docs/_build/html/index.html`.

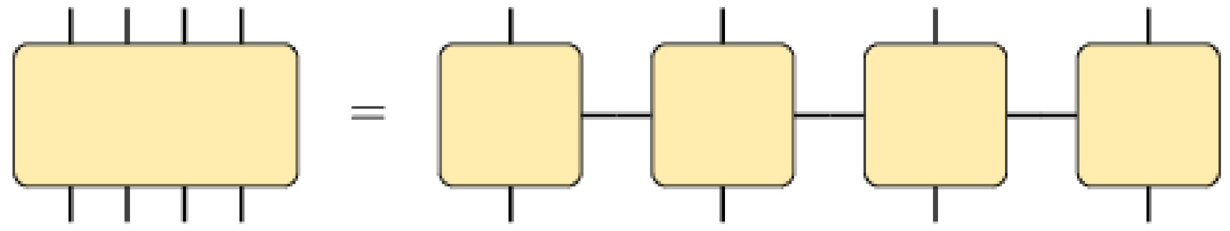
sphinx-autobuild can be used to rebuild HTML documentation automatically anytime a source file is changed:

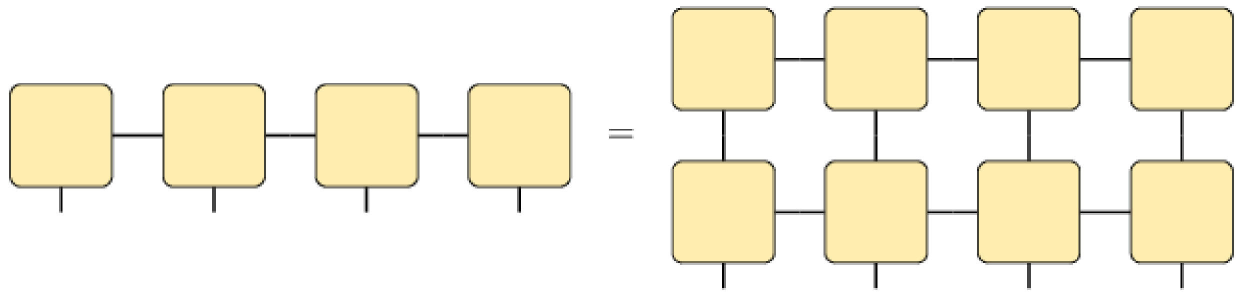
```
pip install sphinx-autobuild
make -C docs livehtml
```

On Windows, `docs/make.bat` may be useful. For more information, see the Sphinx tutorial.

1.5 Gallery

This page contains images needed for the example notebook. In the future (when sphinx_rtd_theme v.0.2.5 is available on PyPi), this page will not be visible anymore globally.





CHAPTER 2

Indices and tables

- genindex
- modindex

Bibliography

[Sch11] Schollwöck, U. (2011). The density-matrix renormalization group in the age of matrix product states. Ann. Phys. 326(1), pp. 96–192. DOI: 10.1016/j.aop.2010.09.012. arXiv:1008.3477.

[KGE14] Kliesch, Gross and Eisert (2014). Matrix-product operators and states: NP-hardness and undecidability. Phys. Rev. Lett. 113, 160503. DOI: 10.1103/PhysRevLett.113.160503. arXiv:1404.4466.

[LSM61] Lieb, Schultz and Mattis (1961). Two soluble models of an antiferromagnetic chain.

Python Module Index

Index