mpnum Documentation

Release git

Daniel Suess and Milan Holzäpfel

May 16, 2017

Documentation

CHAPTER 1

A matrix product representation library for Python

mpnum is a Python library providing flexible tools to implement new numerical schemes based on matrix product states (MPS). It is available under the BSD license at

- mpnum source on Github

Introduction

mpnum is a Python library providing flexible tools to implement new numerical schemes based on matrix product states (MPS). It is available under the BSD license at mpnum on Github. So far, mpnum provides:

- basic tools for various matrix product based 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*)
- basic MPA operations: add, multiply, etc; compression (see `compress()`, SVD and variational)
- computing ground states of MPOs (see `mineig()`, which computes smallest eigenvalues and eigenvectors of MPOs)
- flexible tools to implement new schemes based on matrix product representations

Contents

Contributing

Contributions and pull requests for mpnum are very welcome. More information on modifying mpnum is at *Extending mpnum*.

Graphical notation for tensors

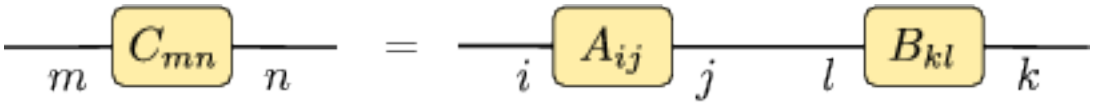
Our graphical notation for tensors is very similar to the graphical notation used by Schollwoeck [*Sch11*, e.g. Figure 38].

Basics

Tensor contractions are much easier to write down using figures. 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.

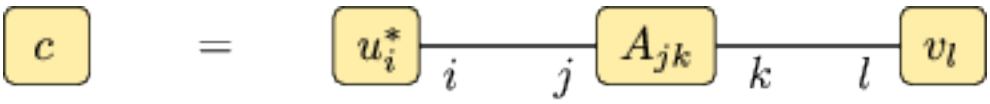
In this simple case, the figure looks more complicated than the formula, but it contains complete information on how all indices of all tensors are connected. To be fair, we should mention the indices in the formula as well:

$$C_{ij} = \sum_j A_{ij} B_{kj}$$

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:

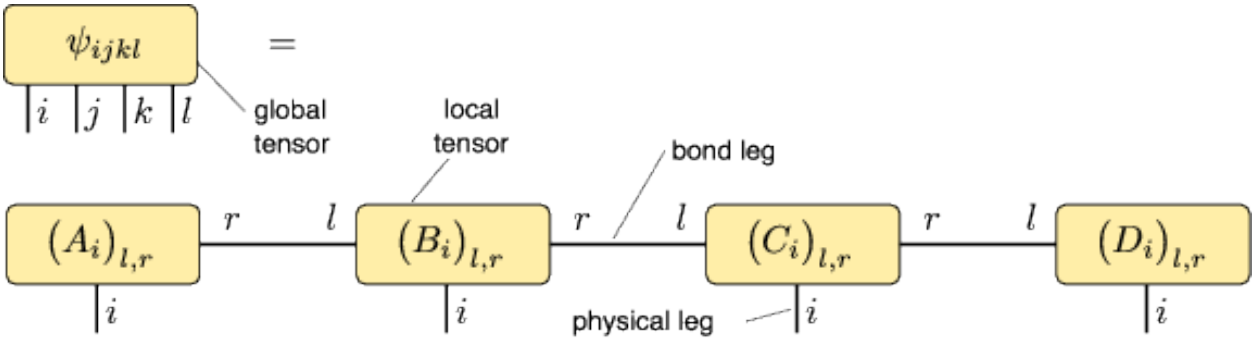


Matrix product states (MPS)

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

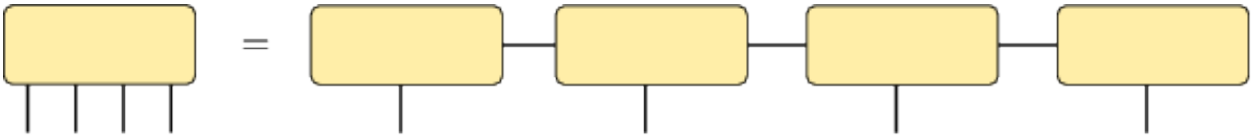
$$\langle ijkl|\psi\rangle = \psi_{ijkl} = A_i B_j C_k D_l$$

where $A_i \in \mathbb{C}^{1\times D}$, $B_j, C_k \in \mathbb{C}^{D\times D}$ 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:

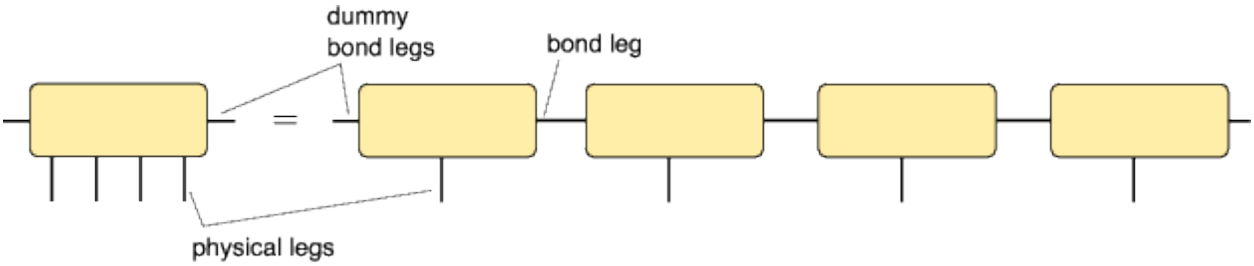


We call ψ a *global tensor* and we call the MPS matrices A_i, B_j etc. which are associated to a certain subsystem *local tensors*.

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 bonds* of size 1 to our tensors:

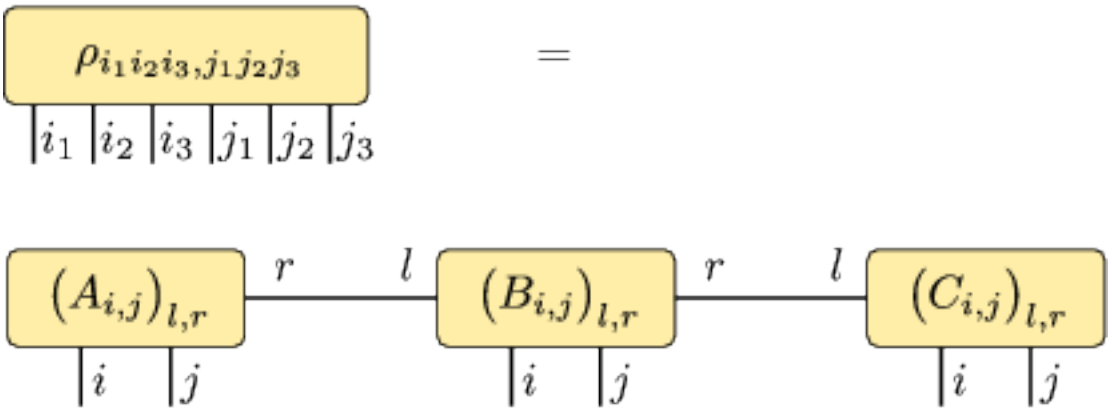


Matrix product operators (MPO)

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

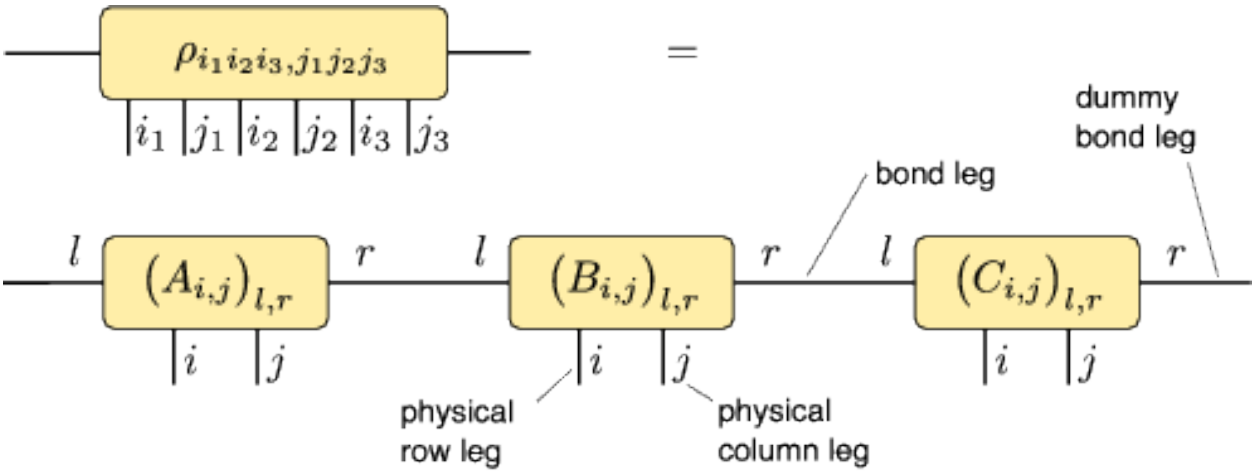
$$\langle i_1 i_2 i_3|\rho|j_1 j_2 j_3\rangle = \rho_{i_1 i_2 i_3, j_1 j_2 j_3} = A_{i_1 j_1} B_{i_2 j_2} C_{i_3 j_3}$$

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:



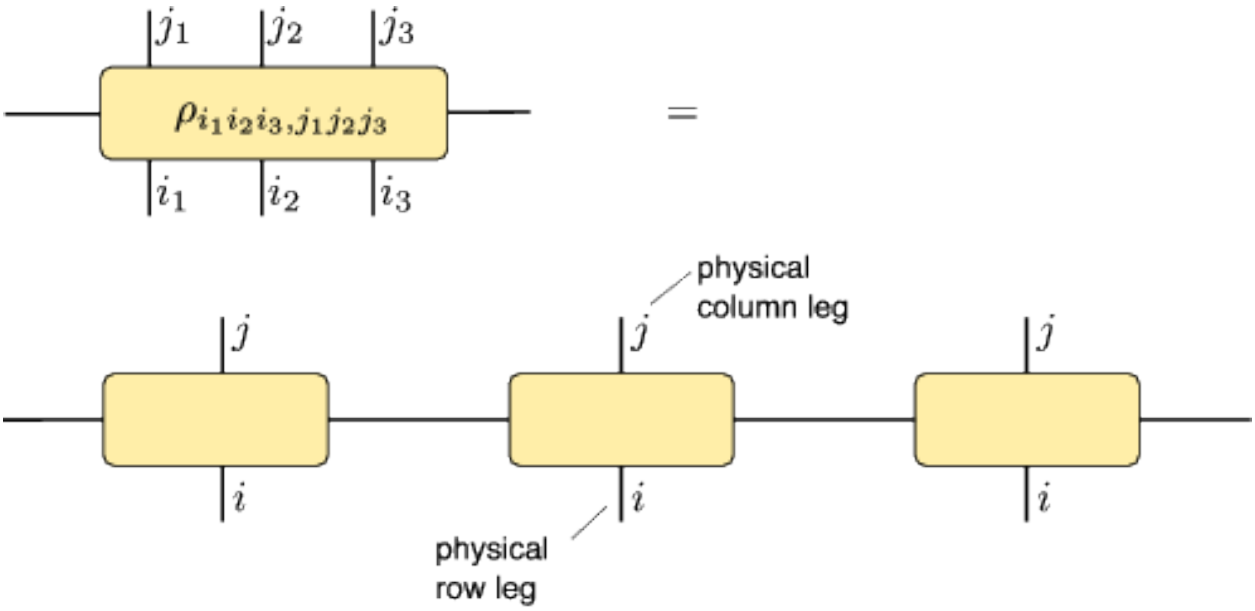
Be aware that the legs of ρ are not in the order $i_1 i_2 i_3 j_1 j_2 j_3$ (called *global order*) which is expected from the expression $\langle i_1 i_2 i_3 | \rho | j_1 j_2 j_3 \rangle$ and which is obtained by a simple reshape of the matrix ρ into a tensor. Instead, the order of the legs of ρ must 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 bonds* with index size 1 on the left and the right of the MPS or MPO chain:

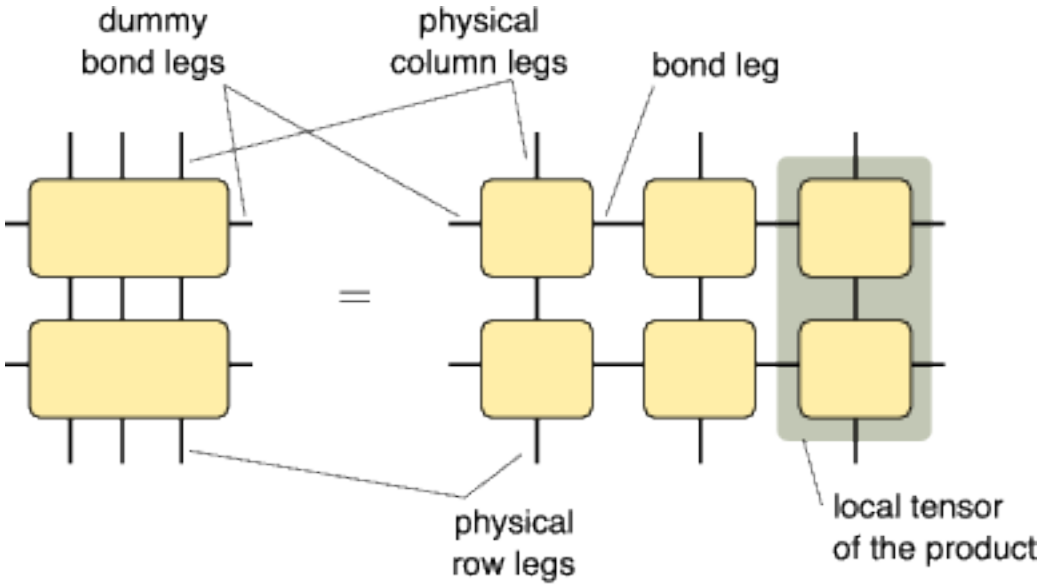


With these dummy bonds, all the tensors in the representation have exactly two bond indices.

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



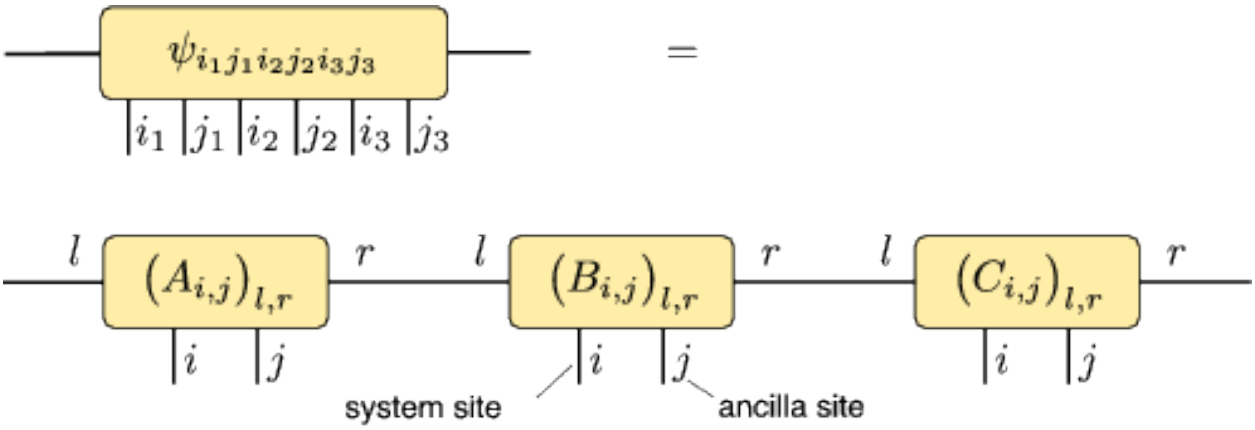
With this arrangement, we can nicely express a product of two MPOs:



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 bond dimension of the result is the product of the bond dimensions of the two individual MPO representations.

Local purification form MPS (PMPS)

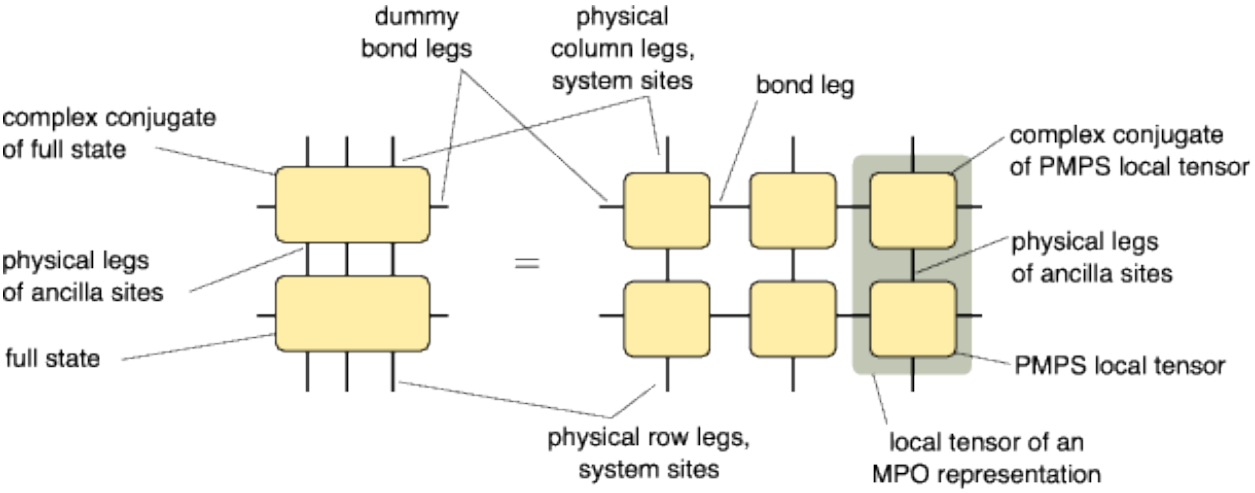
The local purification from matrix product state representation (PMPS or LPMPS) is defined as follows:



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

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

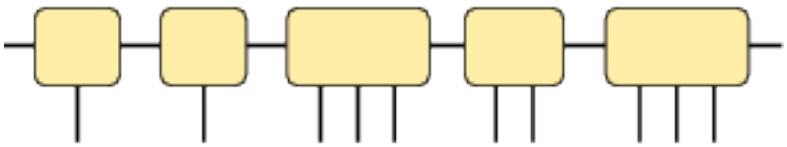
The following figure describes the relation:



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

Matrix product arrays

The library mpnum implements the class `mpnum.mparray.MPArray` which can be used for MPS, MPO, PMPS and other MPS-like representations. `MPArray` is short for *matrix product array* (MPA) and this class provides an MPS with an arbitrary number of physical legs at each site. Each physical leg can also have an arbitrary dimension. A corresponding figure could look like this:



Next steps

The ipython notebook `mpnum_intro.ipynb` in the folder `Notebooks` provides an introduction on how to use `mpnum` for basic MPS, MPO and MPA operations. You can also view mpnum_intro.ipynb on Github.

Extending mpnum

This section contains information for anyone who wishes to modify mpnum. Contributions and pull requests for mpnum are very welcome.

Contents

Code style

All warnings reported by flake8 should be fixed:

```
python -m flake8 .
```

Slightly more than 80 characters can sometimes be tolerated if reformatting would be cumbersome.

Automated unit tests

After any change to mpnum, it should be verified that automated tests succeed.

A short set of tests takes only 15 seconds and is invoked with one of

```
python -m pytest
python setup.py test
```

An intermediate set of tests takes about 2 minutes to run, is executed automatically on Travis and is invoked with one of

```
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 with pytest. Additions to mpnum should always be accompanied by unit tests.

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`.

Building the documentation

The HTML documentation uses Sphinx. On Linux, it 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.

mpnum reference

Contents

mpnum: A matrix-product-representation library for Python

- `mpnum.mparray`: Basic matrix product array (MPA) routines and compression
- `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 ground states (smallest eigenvalue and eigenvector) of MPOs
- `mpnum.povm`: 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

Basic matrix product array functionality

Module containing routines for dealing with general matrix product arrays.

References:

- [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.

class mpnum.mparray.**MPArray**(*ltens*)
Bases: object

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

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

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

By convention, the 0th and last dimension of the local tensors are reserved for the auxillary legs.

Todo

As it is now, e.g. `MPArray.__imul__()` modifies items from *self._ltens*. This requires e.g. `outer()` 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?

Todo

If we enable all special members (e.g. *__len__*) to be shown, we get things like *__dict__* with very long contents. Therefore, special members are hidden at the moment, but we should show the interesting one.

__init__(*ltens*)

Parameters **ltens** (*LocalTensors*) – local tensors as instance of *mpstruct.LocalTensors*

T
Transpose (=reverse order of) physical legs

_adapt_to(*target*, *num_sweeps*, *var_sites*)
Iteratively minimize the l2 distance between *self* and *target*. This is especially important for variational compression, where *self* is the initial guess and target the MPA to be compressed.

Parameters **target** – MPS to compress; i.e. MPA with only one physical leg per site

Other parameters and references: See *MPArray.compress()*.

_compress_svd(*bdim=None*, *relerr=None*, *direction=None*, *normalize=True*, *svdfunc=<function truncated_svd>*)
Compress *self* using SVD [*Sch11*, Sec. 4.5.1]

Parameters: See *MPArray.compress()*.

_compress_svd_l(*bdim*, *relerr*, *svdfunc*)
Compresses the MPA in place from right to left using SVD; yields a right-canonical state

See *MPArray.compress()* for parameters

_compress_svd_r(*bdim*, *relerr*, *svdfunc*)
Compresses the MPA in place from left to right using SVD; yields a left-canonical state

See *MPArray.compress()* for parameters

_compression_var(*startmpa=None*, *bdim=None*, *randstate=<module 'numpy.random' from '/usr/lib/python3/dist-packages/numpy/random/__init__.py'>*, *num_sweeps=5*, *var_sites=1*)
Return a compression from variational compression [*Sch11*, Sec. 4.5.2]

Parameters and return value: See *MPArray.compression()*.

_lnormalize(*to_site*)
Left-normalizes all local tensors _ltens[:to_site] in place

Parameters **to_site** – Index of the site up to which normalization is to be performed

_rnormalize(*to_site*)
Right-normalizes all local tensors _ltens[to_site:] in place

Parameters **to_site** – Index of the site up to which normalization is to be performed

adj()
Hermitian adjoint

bdim
Largest bond dimension across the chain

bdims
Tuple of bond dimensions

bleg2pleg(*pos*)
Transforms the bond leg between site *pos* and *pos + 1* into physical 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 bond dimension is 1.

Also see *pleg2bleg()*.

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

Returns read-only MPA with transformed bond

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

- **bdim** – Maximal bond dimension of the result. Default *None*.
- **relerr** – Maximal fraction of discarded singular values. Default *0*. If both bdim and relerr are given, the smaller resulting bond dimension is used.
- **direction** – *right* (sweep from left to right), *left* (inverse) or *None* (choose depending on normalization). Default *None*.
- **normalize** – 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)

Parameters for 'var':

Parameters

- **startmpa** – Start vector, also fixes the bond dimension of the result. Default: Random, with same norm as self.
- **bdim** – Maximal bond dimension for the result. Either *startmpa* or *bdim* is required.
- **randstate** – *numpy.random.RandomState* instance used for random start vector. Default: *numpy.random*.
- **num_sweeps** – Maximum number of sweeps to do. Default 5.
- **var_sites** – Number of sites to modify simultaneously. Default 1.

Increasing *var_sites* makes it less likely to get stuck in a local minimum.

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 *MPArray.compress()*.

Returns *(compressed_mpa, iprod)* where *iprod* is the inner product returned by *MPArray.compress()*.

conj()
Complex conjugate

copy()
Makes a deep copy of the MPA

dims
Tuple of shapes for the local tensors

dtype
Returns the dtype that should be returned by to_array

dump(*target*)
Serializes MPArray to `h5py.Group`. Recover using *MPArray.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*, *plegs=None*, *has_bond=False*)
Create MPA from array in local form.

See *mpnum._tools.global_to_local()* for global vs. local form.

Computes the (exact) representation of *array* as MPA with open boundary conditions, i.e. bond dimension 1 at the boundary. This is done by factoring the off the left and the "physical" legs from the rest of the tensor by a 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 *plegs* physical legs at each location and has array.ndim // plegs number of sites.

has_bond = True allows to treat a part of the linear chain of an MPA as MPA as well. The bond dimension 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. It is used in linalg._mineig_minimize_locally().

Parameters

- **array** (*np.ndarray*) – Array representation with global structure array[(i1), ..., (iN)], i.e. the legs which are factorized into the same factor are already adjacent. (For me details see `_tools.global_to_local()`)
- **plegs** – Number of physical legs per site (default array.ndim) or iterable over number of physical legs
- **has_bond** (*bool*) – True if array already has indices for the left and right bond

classmethod from_array_global(*array*, *plegs=None*, *has_bond=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_bond=True* is not supported yet.

classmethod from_kron (*factors*)
Returns the (exact) representation of an n-fold Kronecker (tensor) product as MPA with bond dimensions 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_phys (*pind*, *astype=None*)
Fix values for first physical leg

Parameters **pind** – 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.plegs[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 plegs

legs
Tuple of total number of legs per site

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

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

lt

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

normalize (*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-normalization. The right- (left- resp.) most local tensor is not normalized since this can be done by simply calculating its norm (instead of using SVD).

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

Left-/Right- normalize:	Do Nothing	To normalize 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*

pad_bdim(*bdim=None*, *force_bdim=False*)
Increase bond dimension by padding with zeros

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

Parameters

- **bdim** – Increase bond dimension to this value, use *self.bdim* if *None*
- **force_bdim** – Use full bond dimension even at the beginning and end of the MPS (generally not useful)

Returns MPA representation of the same array with increased bond dimension

paxis_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.paxis_iter(0) is equivalent to:

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

FIXME The previous code is not highlighted because `A[(k, :)]` is invalid syntax. Example of working highlighting:

```
(x**2 for x in range(...))
```

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

Returns Iterator over MPArray

pdims
Tuple of physical dimensions

pleg2bleg(*pos*)
Performs the inverse operation to *bleg2pleg()*.

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

Returns read-only MPA with transformed bond

plegs
Tuple of number of physical legs per site

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

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

Use self.pdims to obtain the shapes of the physical legs.

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

Returns Reshaped MPA

reverse()

singularvals()
Return singular values 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 bond dimension (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, bdim=3).dims
((1, 4, 3), (3, 4, 3), (3, 4, 1))
>>> zero(sites=3, ldim=4, bdim=3).size
60
```

split(*pos*)
Splits the MPA into two by transforming the bond legs into physical legs

Parameters **pos** – Number of the bond 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.plegs[i] // sites_per_group physical legs on site i. The physical legs on before splitting must be in local form.

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

Returns An mpa with sites_per_group more sites and fewer plegs

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.pdims, ())`

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.pdims, ()))`

See *to_array()* for more details.

transpose(*axes=None*)
Transpose physical legs

Parameters **axes** – New order of the physical axes (default *None* = reverse the order).

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

mpnum.mparray.**dot**(*mpa1*, *mpa2*, *axes=(-1, 0)*, *astype=None*)
Compute the matrix product representation of a.b over the given (physical) 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 *np.tensordot()*, but the default value is different.
- **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 outer product of an MPA and a bond dimension 1 MPA

Return the outer 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 *outer()* is preferred for that as it is a much simpler function.

If *inject_ten* is omitted, use a square identity matrix of size *mpa.pdims[pos][0]*. If *pos* = *len(mpa)*, *mpa.pdims[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 outer 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 bond 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.**louter**(*a*, *b*)

Computes the tensorproduct 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 normalization

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.**outer**(*mpas*, *astype=None*)
Performs the tensor product of MPAs given in **args*

Parameters

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

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

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.

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) – MPA or iterator over local tensors
- **singletons** – If True, also contract sites where all physical legs have size 1

Returns An MPA of 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

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

If *mps2* is given, <mps2|MPO|mps> is computed instead.

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.

By default, just compute the trace.

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.

Returns A single scalar (int/float/complex, depending on mpa)

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 pdims > `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 physical 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.

Parameters **mpas** – Iterator over MPArrays

Returns Sum of *mpas*

Random test matrix product arrays

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.bdims, I.pdims
((1, 1, 1), ((2, 2), (2, 2), (2, 2), (2, 2)))
>>> I = eye(3, (3, 4, 5))
>>> I.pdims
((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*, *bdim*, *randstate=None*, *normalized=False*, *force_bdim=False*, *dtype=<class 'numpy.float64'>*)

Returns a MPA with randomly choosen local tensors

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.pdims
  - iterable of scalar: Same physical dimension for each site
  - scalar: Single physical leg for each site with given dimension
- **bdim** – Bond dimension
- **randn** – Function used to generate random local tensors
- **randstate** – numpy.random.RandomState instance or None
- **normalized** – Resulting *mpa* has *mp.norm(mpa) == 1*
- **force_bdim** – If True, the bond dimension is exaclty *bdim*. Otherwise, it might be reduced if we reach the maximum sensible bond dimension for a bond.
- **dtype** – Whicht type the returned array should have. Currently only *np.real_* and *np.complex_* is implemented (default: complex)

Returns randomly choosen matrix product array

```
>>> mpa = random_mpa(4, 2, 10, force_bdim=True)
>>> mpa.bdims, mpa.pdims
((10, 10, 10), ((2,), (2,), (2,), (2,)))
```

```
>>> mpa = random_mpa(4, (1, 2), 10, force_bdim=True)
>>> mpa.bdims, mpa.pdims
((10, 10, 10), ((1, 2), (1, 2), (1, 2), (1, 2)))
```

```
>>> mpa = random_mpa(4, [(1, ), (2, 3), (4, 5), (1, )], 10, force_bdim=True)
>>> mpa.bdims, mpa.pdims
((10, 10, 10), ((1,), (2, 3), (4, 5), (1,)))
```

mpnum.factory.**random_mpdo**(*sites*, *ldim*, *bdim*, *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
- **bdim** – Bond dimension
- **randstate** – numpy.random.RandomState instance

Returns randomly choosen classicaly correlated matrix product density op.

```
>>> rho = random_mpdo(4, 2, 4)
>>> rho.bdims, rho.pdims
((4, 4, 4), ((2, 2), (2, 2), (2, 2), (2, 2)))
>>> rho.normal_form
(0, 4)
```

mpnum.factory.**random_mps**(*sites*, *ldim*, *bdim*, *randstate=None*, *force_bdim=False*)

Returns a randomly choosen normalized matrix product state

Parameters

- **sites** – Number of sites
- **ldim** – Local dimension
- **bdim** – Bond dimension
- **randstate** – numpy.random.RandomState instance or None
- **force_bdim** – If True, the bond dimension is exaclty *bdim*. Otherwise, it might be reduced if we reach the maximum sensible bond dimension for a bond.

Returns randomly choosen matrix product (pure) state

```
>>> mps = random_mps(4, 2, 10, force_bdim=True)
>>> mps.bdims, mps.pdims
((10, 10, 10), ((2,), (2,), (2,), (2,)))
>>> mps.normal_form
(0, 4)
>>> round(abs(1 - mp.inner(mps, mps)), 10)
0.0
```

mpnum.factory.**random_mpo**(*sites*, *ldim*, *bdim*, *randstate=None*, *hermitian=False*, *normalized=True*, *force_bdim=False*)

Returns an hermitian MPO with randomly choosen local tensors

Parameters

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

- **randstate** – numpy.random.RandomState instance or None
- **hermitian** – Is the operator supposed to be hermitian
- **normalized** – Operator should have unit norm
- **force_bdim** – If True, the bond dimension is exaclty *bdim*. Otherwise, it might be reduced if we reach the maximum sensible bond dimension for a bond.

Returns randomly choosen matrix product operator

```
>>> mpo = random_mpo(4, 2, 10, force_bdim=True)
>>> mpo.bdims, mpo.pdims
((10, 10, 10), ((2, 2), (2, 2), (2, 2), (2, 2)))
>>> mpo.normal_form
(0, 4)
```

mpnum.factory.**zero**(*sites*, *ldim*, *bdim*, *force_bdim=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.pdims
  - iterable of scalar: Same physical dimension for each site
  - scalar: Single physical leg for each site with given dimension
- **bdim** – Bond dimension
- **force_bdim** – If True, the bond dimension is exaclty *bdim*. Otherwise, it might be reduced if we reach the maximum sensible bond dimension for a bond.

Returns Representation of the zero-array as MPA

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

@todo: Docstring for diagonal_mpa.

Parameters **entries** – @todo

Returns @todo

Matrix product states and operators

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

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

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 bond 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 bond dimension 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

Eigenvalues and eigenvectors (DMRG)

Linear algebra with matrix product arrays

Currently, we support computing ground states (i.e. minimal eigenvalue and eigenvector).

mpnum.linalg.**mineig**(*mpo*, *startvec=None*, *startvec_bonddim=None*, *randstate=None*, *max_num_sweeps=5*, *eigs_opts=None*, *minimize_sites=1*)
Iterative search for smallest eigenvalue and eigenvector of an MPO.

Algorithm: [*Sch11*, Sec. 6.3]

Parameters

- **mpo** (MPArray) – A matrix product operator (MPA with two physical legs)
- **startvec** – initial guess for eigenvector (default random MPS with bond *startvec_bonddim*)

- **startvec_bonddim** – Bond dimension of random start vector if no start vector is given. (default: Use the bond dimension of *mpo*)
- **randstate** – numpy.random.RandomState instance or None
- **max_num_sweeps** – Maximum number of sweeps to do (default 5)
- **eigs_opts** – kwargs for *scipy.sparse.linalg.eigs()*. If you supple *which*, you will probably not obtain the minimal eigenvalue. *k* different from one is not supported at the moment.
- **minimize_sites** (*int*) – Number of connected sites minimization should be performed on (default 1)

Returns mineigval, mineigval_eigvec_mpa

We minimize the eigenvalue by obtaining the minimal eigenvalue of an operator supported on 'minimize_sites' many sites. For minimize_sites=1, this is called "variational MPS ground state search" or "single-site DMRG" [*Sch11*, Sec. 6.3, p. 69]. For minimize_sites>1, this is called "multi-site DMRG".

Comments on the implementation, for minimize_sites=1:

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

leftvecs[i] is L_{i-1} rightvecs[i] is R_{i} | See Fig. 38 and Eq. (191) on p. 62. mpo[i] is W_{i} / eigvec[i] is M_{i} This is just the MPS matrix.

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.**mineig_sum**(*mpas*, *startvec=None*, *startvec_bonddim=None*, *randstate=None*, *max_num_sweeps=5*, *eigs_opts=None*, *minimize_sites=1*)

Iterative search for smallest eigenvalue+vector of a sum

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

This function executes exactly the same algorithm as *mineig()* 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 *mineig()* and *mineig_sum()* with the the different bond dimensions.

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

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

Algorithm: [*Sch11*, Sec. 6.3]

Local POVMs

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

Matrix-product POVMs

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 $p_y = \mathrm{tr}(\rho\hat{y})$, $y \in M$ 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 normalize 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.

_elemsum_identity(*support*, *given*, *eps*)
Check whether a given subset of POVM elements sums to a multiple of the identity

Parameters

- **support** (*np.ndarray*) – List of sites on which POVM elements are selected by *given*
- **given** (*np.ndarray*) – Whether a POVM element with a given index should be included (bool array)

A POVM element specified by the compound index *(i_1, ..., i_n)* with *n = len(self)* is included if *given[i_(support[0]), ..., i_(support[k])]* is *True*.

Returns If the POVM elements sum to a fraction of the identity, return the fraction. Otherwise return *None*.

_fill_outcome_mpa_holes(*support*, *outcome_mpa*)
Fill holes in an MPA on some of the outcome physical legs

The dot product of *outcome_mpa* and *self* provides a sum over some or all elements of the POVM. The way sites are added to *outcome_mpa* implements the selection rule described in `self._elemsum_identity()`.

Parameters

- **support** (*np.ndarray*) – List of sites where *outcome_mpa* lives
- **outcome_mpa** (*mp.MPArray*) – An MPA with physical legs in agreement with *self.outdims* with some sites omitted

Returns An MPA with physical legs given by *self.outdims*

_mppl_lfun_estimator(*est_coeff*, *est_funs*, *other*, *n_samples*, *coeff*, *eps*)
Compute the estimator used by `MPPovmList.estfun_from()`

Used by *MPPovmList._lfun_estimator()*.

est_coeff[i] and *est_funs[i]* will specify an estimator in the format used by *MPPovm.lfun()* on *other.mpps[i]*. This function adds the coefficients and functions necessary to estimate the linear function of *self* probabilities specified by *coeff*.

Parameters

- **est_coeff** – Output parameter, tuple of lists
- **est_funs** – Output parameter, tuple of lists
- **other** (*MPPovmList*) – An MP-POVM list
- **n_samples** – *n_samples[i]* specifies the number of samples available for *other.mpps[i]*. They are used for a weighted average if a given *self* probability can be estimated from more than one MP-POVMs in *other*.
- **coeff** – A linear function of *self* probabilities is specified by *coeff*

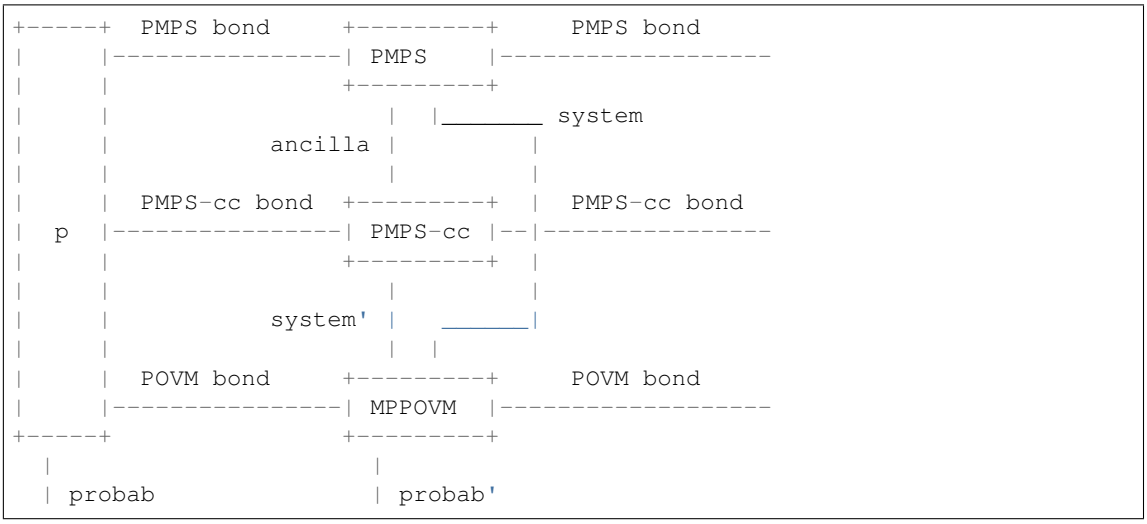
Returns *n_samples*: A shape *self.nsoutdims* array which specifies how many samples very available for each probability.

Note: Output is also added to the parameters *est_coeff* and *est_fun*.

_pmf_as_array_pmps_ltr(*pmps*, *partial=False*)
PMF-as-array fast path for PMPS

Called automatically by *pmf_as_array()*.

This function contracts the following tensor network from top to bottom and from left to right along the chain:



_pmf_as_array_pmps_symm(*state*)
PMF-as-array fast path for PMPS (uses less memory)

This function contracts the same tensor network as `self._pmf_as_array_pmps_ltr()`, but it starts at both ends of the chain and proceeds to a certain position in the middle of the chain. We choose the position such that the maximal size of all intermediate results is minimal. This might also minimize runtime in some cases.

_sample_cond(*rng*, *state*, *mode*, *n_samples*, *n_group*, *out*, *eps*)
Sample using conditional probabilities (call `self.sample()`)

static _sample_cond_single(*rng*, *marginal_pmf*, *n_group*, *out*, *eps*)
Single sample from conditional probab. (call `self.sample()`)

_sample_direct(*rng*, *state*, *mode*, *n_samples*, *out*, *eps*)
Sample from full pmfilities (call `self.sample()`)

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 corresponding 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 *check_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

```
mp.dot(a_povm.probability_map, rho.ravel())
```

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 bond 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.outer([x, y])
>>> xyxyx = mp.outer([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.

_lfun_estimator(*other*, *coeff*, *n_samples*, *eps*)

Compute the estimator used by *MPPovmList.est_lfun_from()*

Parameters: See *MPPovmList.est_lfun_from()* for *other* and *coeff*. See *MPPovm._mppl_lfun_estimator()* for *n_samples*.

Returns *(n_sam, est_coeff, 'est_funs)*: *est_coeff[i]* and *est_funs[i]* specify an estimator in the format used by *MPPovm.est_lfun()* on *other.mpps[i]*. *n_sam* is a shape *self.nsoutdims*

array providing the number of samples available for each probability of *self*; zero indicates that a probability cannot be estimated.

This method aggregates the results from *MPPovm._mppl_lfun_estimator()* on each *self.mpps[i]*.

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.outer([x, x])
>>> xy = mp.outer([x, y])
>>> mppl = mpp.MPPovmList((xx, xy))
>>> xxe = mp.outer([x, x, e])
>>> xye = mp.outer([x, y, e])
>>> exx = mp.outer([e, x, x])
>>> exy = mp.outer([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.outer((x, x)),
...     mp.outer((x, y)),
...     mp.outer((y, x)),
...     mp.outer((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.outer((x, x, x, x, x)),
...     mp.outer((x, y, x, y, x)),
...     mp.outer((y, x, y, x, y)),
...     mp.outer((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_phys([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.outer((x, x)),
...     mp.outer((x, y)),
...     mp.outer((y, x)),
...     mp.outer((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*

Internal utility functions

General tools

General helper functions for dealing with arrays (esp. for quantum mechanics)

mpnum._tools.**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._tools.**check_nonneg_trunc**(*values*, *imag_eps=1e-10*, *real_eps=1e-10*, *real_trunc=0.0*)

Check that values are real and non-negative

Parameters

- **values** (*np.ndarray*) – An ndarray of complex or real values (or a single value). *values* is modified in-place unless *values* is complex. A single value is also accepted.
- **imag_eps** (*float*) – Raise an error if imaginary parts with modulus larger than *imag_eps* are present.
- **real_eps** (*float*) – Raise an error if real parts smaller than -*real_eps* are present. Replace all remaining negative values by zero.
- **real_trunc** (*float*) – Replace positive real values smaller than or equal to *real_trunc* by zero.

Returns An ndarray of real values (or a single real value).

If *values* is an array with complex type, a new array is returned. If *values* is an array with real type, it is modified in-place and returned.

mpnum._tools.**check_pmf**(*values*, *imag_eps=1e-10*, *real_eps=1e-10*, *real_trunc=0.0*)
Check that values are real probabilities

See *check_nonneg_trunc()* for parameters and return value. In addition, we check that *abs(values.sum() - 1.0)* is smaller than or equal to *real_eps* and divide *values* by *values.sum()* afterwards.

mpnum._tools.**compression_svd**(*array*, *bdim*, *direction='right'*, *retproj=False*)
Re-implement MPArray.compress('svd') but on the level of the full array representation, i.e. it truncates the Schmidt-decompostion on each bipartition sequentially.

Parameters

- **mpa** – Array to compress
- **bdim** – Compress to this bond dimension
- **direction** – 'right' means sweep from left to right, 'left' vice versa
- **retproj** – Besides the compressed array, also return the projectors on the appropriate eigenspaces

Returns Result as numpy.ndarray

mpnum._tools.**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._tools.**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)
```

mpnum._tools.**matdot**(*A*, *B*, *axes=((-1,), (0,))*)
np.tensordot with sane defaults for matrix multiplication

mpnum._tools.**mkron**(**args*)
np.kron() with an arbitrary number of n >= 1 arguments

mpnum._tools.**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._tools.**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 v: right-singular vectors

mpnum._tools.**verify_real_nonnegative**(*values*, *zero_tol=1e-06*, *zero_cutoff=None*)
Deprecated; use *check_nonneg_trunc()* instead

NumPy ndarray with named axes

A numpy.ndarray with axis names

Access as mpnum.named_ndarray.

class mpnum._named_ndarray.**named_ndarray**(*array*, *axisnames*)
Bases: object

Associate names to the axes of a ndarray.

Property axisnames The names of the axes.

All methods which return arrays return named_ndarray instances.

Method axispos(axisname) Return the position of the named axis

Method rename(translate) Rename axes

Method conj() Return the complex conjugate array

Method to_array(name_order) Return a ndarray with axis order specified by name_order.

Method tensordot(other, axes) numpy.tensordot() with axis names instead of axis indices

axisnames
The names of the array

axispos(*axisname*)
Return the position of an axis.

conj()
Complex conjugate as named_ndarray.

rename(*translate*)
Rename axes.

An error will be raised if the resulting list of names contains duplicates.

Parameters **translate** – List of (old_name, new_name) axis name pairs.

tensordot(*other*, *axes*)
Compute tensor dot product along named axes.

An error will be raised if the remaining axes of self and other contain duplicate names.

Parameters

- **other** – Another named_ndarray instance
- **axes** – List of axis name pairs (self_name, other_name) to be contracted

Returns Result as named_ndarray

to_array(*name_order*)
Convert to a normal ndarray with given axes ordering.

Parameters **name_order** – Order of axes in the array

Unit test utilities

Auxiliary functions useful for writing tests

mpnum._testing.**_assert_lcanonical**(*ltens*, *msg=''*)

mpnum._testing.**_assert_rcanonical**(*ltens*, *msg=''*)

mpnum._testing.**assert_correct_normalization**(*lt*, *lnormal_target=None*, *rnormal_target=None*)
Verify that normalization info in *lt* is correct

We check that *lt* is at least as normalized as specified by the information. *lt* being "more normalized" than the information specifies is admissible and not treated as an error.

If *[lr]normal_target* are not None, verify that normalization info is exactly equal to the given values.

mpnum._testing.**assert_mpa_almost_equal**(*mpa1*, *mpa2*, *full=False*, ***kwargs*)
Verify that two MPAs are almost equal

mpnum._testing.**assert_mpa_identical**(*mpa1*, *mpa2*, *decimal=inf*)
Verify that two MPAs are complety identical

mpnum._testing.**mpo_to_global**(*mpo*)
Convert mpo to dense global array

Todo

Use *mpa.to_array_global()* instead.

Todo list (autogenerated)

Todo

As it is now, e.g. `MPArray.__imul__()` modifies items from *self._ltens*. This requires e.g. `outer()` 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 docstring of mpnum.mparray.MPArray, line 18.)

Todo

If we enable all special members (e.g. *__len__*) to be shown, we get things like *__dict__* with very long contents. Therefore, special members are hidden at the moment, but we should show the interesting one.

(The original entry is located in docstring of mpnum.mparray.MPArray, line 26.)

Todo

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

(The original entry is located in docstring of mpnum.mparray.regular_slices, line 24.)

Todo

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

(The original entry is located in docstring 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 docstring of mpnum.mpsmpo, line 101.)

Todo

Add docstring

(The original entry is located in docstring of mpnum.mpsmpo.reductions, line 1.)

Todo

Add information on how the runtime of *mineig()* and *mineig_sum()* with the the different bond dimensions.

(The original entry is located in docstring of mpnum.linalg.mineig_sum, line 15.)

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 docstring 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 normalize 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 docstring 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 docstring of mpnum.povm.mppovm.MPPovm, line 28.)

Todo

Add docstring

(The original entry is located in docstring of mpnum.povm.mppovm.MPPovm.block_pmfs_as_array, line 1.)

Todo

Add docstring

(The original entry is located in docstring of mpnum.povm.mppovm.MPPovm.pmfs_as_array, line 1.)

Todo

Add docstring

(The original entry is located in docstring of mpnum.povm.mppovm.MPPovmList.block_pmfs_as_array, line 1.)

Todo

Add docstring

(The original entry is located in docstring of mpnum.povm.mppovm.MPPovmList.pmfs_as_array, line 1.)

Todo

Use *mpa.to_array_global()* instead.

(The original entry is located in docstring of mpnum._testing.mpo_to_global, line 3.)

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

CHAPTER 2

Indices and tables

- genindex
- modindex

Python Module Index

m

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

Z