tenpy Documentation

Release 0.5.0

TenPy Team

Dec 19, 2019

CONTENTS

TeNPy (short for 'Tensor Network Python') is a Python library for the simulation of strongly correlated quantum systems with tensor networks.

The philosophy of this library is to get a new balance of a good readability and usability for new-comers, and at the same time powerful algorithms and fast development of new algorithms for experts. For good readability, we include an extensive documentation next to the code, both in Python doc strings and separately as *user guides*, as well as simple example codes and even toy codes, which just demonstrate various algorithms (like TEBD and DMRG) in ~100 lines per file.

CHAPTER

ONE

HOW DO I GET SET UP?

Follow the instructions in the file `doc/INSTALL.rst`, online at https://tenpy.github.io/INSTALL.html. The latest version of the source code can be obtained from https://github.com/tenpy/tenpy.

CHAPTER

TWO

HOW TO READ THE DOCUMENTATION

The **documentation is available online** at https://tenpy.github.io. The documentation is roughly split in two parts: on one hand the full "reference" containing the documentation of all functions, classes, methods, etc., and on the other hand the "user guide" containing some introductions and additional explanations.

The documentation is based on Python's docstrings, and some additional `*.rst` files located in the folder *doc/* of the repository. All documentation is formated as reStructuredText, which means it is quite readable in the source plain text, but can also be converted to other formats. If you like it simple, you can just use intective python *help()*, Python IDEs of your choice or jupyter notebooks, or just read the source. Moreover, the documentation is nightly converted into HTML using Sphinx, and made available online at https://tenpy.github.io/. The big advantages of the (online) HTML documentation are a lot of cross-links between different functions, and even a search function. If you prefer yet another format, you can try to build the documentation yourself, as described in `doc/contributing.rst`, online at https://tenpy.github.io/contributing.html.

CHAPTER

THREE

HELP - I LOOKED AT THE DOCUMENTATION, BUT I DON'T UNDERSTAND HOW . . . ?

We have set up a **community forum** at https://tenpy.johannes-hauschild.de/, where you can post questions and hopefully find answers. Once you got some experience with TeNPy, you might also be able to contribute to the community and answer some questions yourself ;-) We also use this forum for official annoucements, for example when we release a new version.

CHAPTER

FOUR

CITING TENPY

When you use TeNPy for a work published in an academic journal, you can cite this paper to acknowledge the work put into the development of TeNPy. (The license of TeNPy does not force you, however.) For example, you could add the sentence "`Calculations were performed using the TeNPy Library (version X.X.X)\cite{tenpy}.`" in the acknowledgements or in the main text.

The corresponding BibTex Entry would be the following (the `\url{...}` requires `\usepackage{hyperref}` in the LaTeX preamble.):

```
@Article{tenpy,
    title={{Efficient numerical simulations with Tensor Networks: Tensor Network
↪Python (TeNPy)}},
    author={Johannes Hauschild and Frank Pollmann},
    journal={SciPost Phys. Lect. Notes},
    pages={5},
    year={2018},
    publisher={SciPost},
    doi={10.21468/SciPostPhysLectNotes.5},
    url={https://scipost.org/10.21468/SciPostPhysLectNotes.5},
    archiveprefix={arXiv},
    eprint={1805.00055},
    note={Code available from \url{https://github.com/tenpy/tenpy}},
}
```

CHAPTER

FIVE

I FOUND A BUG

You might want to check the github issues, if someone else already reported the same problem. To report a new bug, just open a new issue on github. If you already know how to fix it, you can just create a pull request :) If you are not sure whether your problem is a bug or a feature, you can also ask for help in the TeNPy forum.

CHAPTER

SIX

LICENSE

The code is licensed under GPL-v3.0 given in the file `LICENSE` of the repository, in the online documentation readable at https://tenpy.github.io/license.html.

CHAPTER

SEVEN

CONTENTS

7.1 User Guide

First a short warning: the term 'user guide' might be a bit misleading: this part of the documentation simply covers everything but what is documented directly in the source - the latter can be found in the *Tenpy Reference*.

The first step to use tenpy is to download and install it; simply follow the *Installation instructions*.

After that, take a look at the *Overview* to get started.

7.1.1 Content

Installation instructions

Installation from packages

If you have the conda package manager from anaconda, you can simply download the `environment.yml` file and create a new environment for tenpy with all the required packages:

```
conda env create -f environment.yml
conda activate tenpy
```

This will also install pip. Alternatively, if you only have *pip*, install the required packages with:

```
pip install -r requirements.txt
```

Note: Make sure that the *pip* you call corresponds to the python version you want to use. (e.g. by using `python -m pip` instead of a simple `pip` Also, you might need to use the arguement `--user` to install the packages to your home directory, if you don't have `sudo` rights.

Warning: It might just be a temporary problem, but I found that the *pip* version of numpy is incompatible with the python distribution of anaconda. If you have installed the intelpython or anaconda distribution, use the *conda* packagemanager instead of *pip* for updating the packages whenever possible!

After that, you can **install the latest *stable* TeNPy package** (without downloading the source) from PyPi with:

```
pip install physics-tenpy # note the different package name - 'tenpy' was taken!
```

Note: When the installation fails, don't give up yet. In the minimal version, tenpy requires only pure Python with somewhat up-to-date NumPy and SciPy. See the section *Installation from source* below.

To get the latest development version from the github master branch, you can use:

```
pip install git+git://github.com/tenpy/tenpy.git
```

Finally, if you downloaded the source and want to **modify parts of the source**, you should install tenpy in development version with `-e`:

```
cd $HOME/TeNPy # after downloading the source
pip install --editable .
```

In all cases, you can uninstall tenpy with:

```
pip uninstall physics-tenpy  # note the longer name!
```

Updating to a new version

Before you update, take a look at the *CHANGELOG*, which lists the changes, fixes, and new stuff. Most importantly, it has a section on *backwards incompatible changes* (i.e., changes which may break your existing code) along with information how to fix it. Of course, we try to avoid introducing such incompatible changes, but sometimes, there's no way around them.

How to update depends a little bit on the way you installed TeNPy. Of course, you have always the option to just remove the tenpy files and download the newest version, following the instructions above.

Alternatively, if you used `git clone ...` to download the repository, you can update to the newest version using *Git*. First, briefly check that you didn't change anything you need to keep with `git status`. Then, do a `git pull` to download (and possibly merge) the newest commit from the repository.

Note: If some Cython file (ending in `.pyx`) got renamed/removed (e.g., when updating from v0.3.0 to v0.4.0), you first need to remove the corresponding binary files. You can do so with the command `bash cleanup.sh`.

Furthermore, whenever one of the cython files (ending in `.pyx`) changed, you need to re-compile it. To do that, simply call the command `bash ./compile` again. If you are unsure whether a cython file changed, compiling again doesn't hurt.

To summarize, you need to execute the following bash commands in the repository:

```
# 0) make a backup of the whole folder
git status   # check the output whether you modified some files
git pull
bash ./cleanup.sh  # (confirm with 'y')
bash ./compile.sh
```

Installation from source

Minimal Requirements

This code works with a minimal requirement of pure Python>=3.5 and somewhat recent versions of NumPy and SciPy.

Getting the source

The following instructions are for (some kind of) Linux, and tested on Ubuntu. However, the code itself should work on other operating systems as well (in particular MacOS and Windows).

The offical repository is at https://github.com/tenpy/tenpy.git. To get the latest version of the code, you can clone it with Git using the following commands:

```
git clone https://github.com/tenpy/tenpy.git $HOME/TeNPy
cd $HOME/TeNPy
```

Adjust `$HOME/TeNPy` to the path wherever you want to save the library.

Optionally, if you don't want to contribute, you can checkout the latest stable release:

```
git tag   # this prints the available version tags
git checkout v0.3.0  # or whatever is the lastest stable version
```

Note: In case you don't have Git, you can download the repository as a ZIP archive. You can find it under releases, or the latest development version.

Minimal installation: Including tenpy into PYTHONPATH

The python source is in the directory *tenpy/* of the repository. This folder *tenpy/* should be placed in (one of the folders of) the environment variable PYTHONPATH. On Linux, you can simply do this with the following line in the terminal:

```
export PYTHONPATH=$HOME/TeNPy
```

(If you have already a path in this variable, separate the paths with a colon `:`.) However, if you enter this in the terminal, it will only be temporary for the terminal session where you entered it. To make it permanently, you can add the above line to the file `$HOME/.bashrc`. You might need to restart the terminal session or need to relogin to force a reload of the `~/.bashrc`.

Whenever the path is set, you should be able to use the library from within python:

```
>>> import tenpy
/home/username/TeNPy/tenpy/tools/optimization.py:276: UserWarning: Couldn't load␣
↪compiled cython code. Code will run a bit slower.
  warnings.warn("Couldn't load compiled cython code. Code will run a bit slower.")
>>> tenpy.show_config()
tenpy 0.4.0.dev0+7706003 (not compiled),
git revision 77060034a9fa64d2c7c16b4211e130cf5b6f5272 using
python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0]
numpy 1.16.3, scipy 1.2.1
```

`tenpy.show_config()` prints the current version of the used TeNPy library as well as the versions of the used python, numpy and scipy libraries, which might be different on your computer. It is a good idea to save this data (given as string in `tenpy.version.version_summary` along with your data to allow to reproduce your results exactly.

If you got a similar output as above: congratulations! You can now run the codes :)

MKL and further packages

If you want to run larger simulations, we recommend the use of Intel's MKL. It ships with a Lapack library, and uses optimization for Intel CPUs. Moreover, it uses parallelization of the LAPACK/BLAS routines, which makes execution much faster. As of now, the library itself supports no other way of parallelization.

If you don't have a python version which is built against MKL, we recommend using the anaconda distribution, which ships with Intel MKL, or directly intelpython. Conda has the advantage that it allows to use different environments for different projects. Both are available for Linux, Mac and Windows; note that you don't even need administrator rights to install it on linux. Simply follow the (straight-forward) instructions of the web page for the installation. After a successfull installation, if you run `python` interactively, the first output line should state the python version and contain `Anaconda` or `Intel Corporation`, respectively.

If you have a working conda package manager, you can install the numpy build against mkl with:

```
conda install mkl numpy scipy
```

If you prefer using a separete conda environment, you can also use the following code to install all the recommended packages:

```
conda env create -f environment.yml
conda activate tenpy
```

Note: MKL uses different threads to parallelize various BLAS and LAPACK routines. If you run the code on a cluster, make sure that you specify the number of used cores/threads correctly. By default, MKL uses all the available CPUs, which might be in stark contrast than what you required from the cluster. The easiest way to set the used threads is using the environment variable *MKL_NUM_THREADS* (or *OMP_NUM_THREADS*). For a dynamic change of the used threads, you might want to look at *process*.

Some code uses MatPlotLib for plotting, e.g., to visualize a lattice. However, having matplotlib is not necessary for running any of the algorithms: tenpy does not `import matplotlib` by default. Further optional requirements are listed in the `requirements*.txt` files in the source repository.

Compilation of np_conserved

At the heart of the TeNPy library is the module `tenpy.linalg.np_conseved`, which provides an Array class to exploit the conservation of abelian charges. The data model of python is not ideal for the required book-keeping, thus we have implemented the same np_conserved module in Cython. This allows to compile (and thereby optimize) the corresponding python module, thereby speeding up the execution of the code. While this might give a significant speed-up for code with small matrix dimensions, don't expect the same speed-up in cases where most of the CPU-time is already spent in matrix multiplications (i.e. if the bond dimension of your MPS is huge).

To compile the code, you first need to install Cython

```
conda install cython                    # when using anaconda, or
pip install --upgrade Cython            # when using pip
```

Moreover, you need a C++ compiler. For example, on Ubuntu you can install `sudo apt-get install build_essential`, or on Windows you can download MS Visual Studio 2015. If you use anaconda, you can also use one `conda install -c conda-forge cxx-compiler`.

After that, go to the root directory of TeNPy (`$HOME/TeNPy`) and simply run

```
bash ./compile.sh
```

Note that it is not required to separately download (and install) Intel MKL: the compilation just obtains the includes from numpy. In other words, if your current numpy version uses MKL (as the one provided by anaconda), the compiled TeNPy code will also use it.

After a successful compilation, the warning that TeNPy was not compiled should go away:

```
>>> import tenpy
>>> tenpy.show_config()
tenpy 0.4.0.dev0+b60bad3 (compiled from git rev.␣
↪b60bad3243b7e54f549f4f7c1f074dc55bb54ba3),
git revision b60bad3243b7e54f549f4f7c1f074dc55bb54ba3 using
python 3.7.3 (default, Mar 27 2019, 22:11:17)
[GCC 7.3.0]
numpy 1.16.3, scipy 1.2.1
```

Note: For further optimization options, look at *tenpy.tools.optimization*.

Checking the installation

As a first check of the installation you can try to run (one of) the python files in the *examples/* subfolder; hopefully all of them should run without error.

You can also run the automated testsuite with pytest (`pip install pytest`) to make sure everything works fine:

```
cd $HOME/TeNPy/tests
pytest
```

This should run some tests. In case of errors or failures it gives a detailed traceback and possibly some output of the test. At least the stable releases should run these tests without any failures.

If you can run the examples but not the tests, check whether *pytest* actually uses the correct python version.

The test suite is also run automatically with travis-ci, results can be inspected at here.

Overview

Repository

The root directory of this git repository contains the following folders:

tenpy The actual source code of the library. Every subfolder contains an `__init__.py` file with a summary what the modules in it are good for. (This file is also necessary to mark the folder as part of the python package. Consequently, other subfolders of the git repo should not include a `__init__.py` file.)

toycodes Simple toy codes completely independet of the remaining library (i.e., codes in `tenpy/`). These codes should be quite readable and intend to give a flavor of how (some of) the algorithms work.

examples Some example files demonstrating the usage and interface of the library.

doc A folder containing the documentation: the user guide is contained in the `*.rst` files. The online documentation is autogenerated from these files and the docstrings of the library. This folder contains a make file for building the documentation, run `make help` for the different options. The necessary files for the reference in `doc/reference` can be auto-generated/updated with `make src2html`.

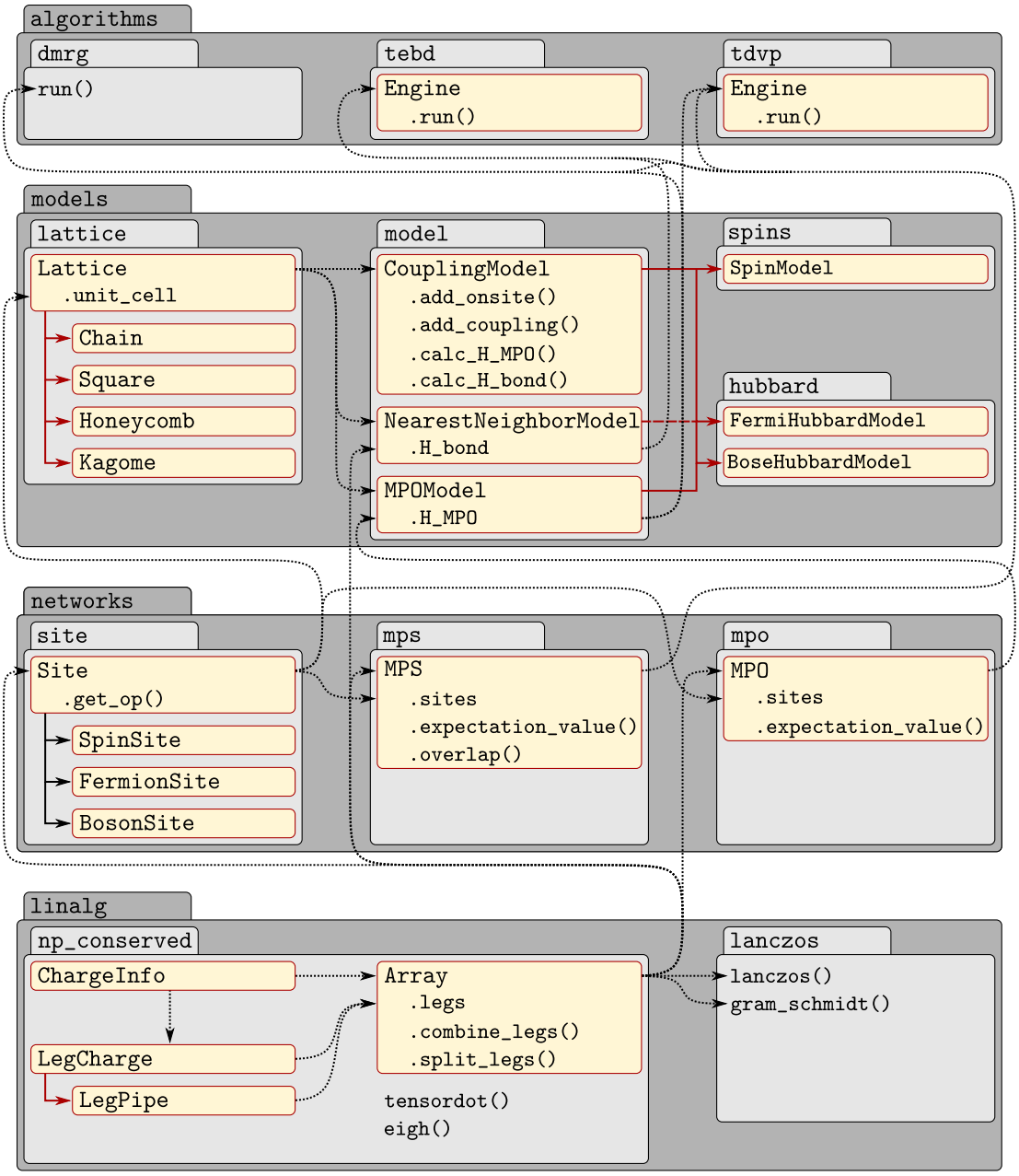
tests Contains files with test routines, to be used with *pytest*. If you are set up correctly and have *pytest* installed, you can run the test suite with `pytest` from within the `tests/` folder.

build This folder is not distributed with the code, but is generated by `setup.py` (or `compile.sh`, respectively). It contains compiled versions of the Cython files, and can be ignored (and even removed without loosing functionality).

Code structure: getting started

There are several layers of abstraction in TeNPy. While there is a certain hierarchy of how the concepts build up on each other, the user can decide to utilize only some of them. A maximal flexibility is provided by an object oriented style based on classes, which can be inherited and adjusted to individual demands.

The following figure gives an overview of the most important modules, classes and functions in TeNPy. Gray backgrounds indicate (sub)modules, yellow backgrounds indicate classes. Red arrows indicate inheritance relations, dashed black arrows indicate a direct use. (The individual models might be derived from the *NearestNeighborModel* depending on the geometry of the lattice.) There is a clear hierarchy from high-level algorithms in the *tenpy.algorithms* module down to basic operations from linear algebra in the *tenpy.linalg* module.



Most basic level: linear algebra

Note: See *Introduction to np_conserved* for more information on defining charges for arrays.

The most basic layer is given by in the `linalg` module, which provides basic features of linear algebra. In particular, the `np_conserved` submodule implements an `Array` class which is used to represent the tensors. The basic interface of `np_conserved` is very similar to that of the NumPy and SciPy libraries. However, the `Array` class implements abelian charge conservation. If no charges are to be used, one can use 'trivial' arrays, as shown in the following example code.

```
"""Basic use of the `Array` class with trivial arrays."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

import tenpy.linalg.np_conserved as npc

M = npc.Array.from_ndarray_trivial([[0., 1.], [1., 0.]])
v = npc.Array.from_ndarray_trivial([2., 4. + 1.j])
v[0] = 3.  # set indiviual entries like in numpy
print("|v> =", v.to_ndarray())
# |v> = [ 3.+0.j  4.+1.j]

M_v = npc.tensordot(M, v, axes=[1, 0])
print("M|v> =", M_v.to_ndarray())
# M|v> = [ 4.+1.j  3.+0.j]
print("<v|M|v> =", npc.inner(v.conj(), M_v, axes='range'))
# <v|M|v> = (24+0j)
```

The number and types of symmetries are specified in a *ChargeInfo* class. An *Array* instance represents a tensor satisfying a charge rule specifying which blocks of it are nonzero. Internally, it stores only the non-zero blocks of the tensor, along with one *LegCharge* instance for each leg, which contains the *charges* and sign *qconj* for each leg. We can combine multiple legs into a single larger *LegPipe*, which is derived from the *LegCharge* and stores all the information necessary to later split the pipe.

The following code explicitly defines the spin-1/2 S^+, S^-, S^z operators and uses them to generate and diagonalize the two-site Hamiltonian $H = \vec{S} \cdot \vec{S}$. It prints the charge values (by default sorted ascending) and the eigenvalues of H.

```
"""Explicit definition of charges and spin-1/2 operators."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

import tenpy.linalg.np_conserved as npc

# consider spin-1/2 with Sz-conservation
chinfo = npc.ChargeInfo([1])  # just a U(1) charge
# charges for up, down state
p_leg = npc.LegCharge.from_qflat(chinfo, [[1], [-1]])
Sz = npc.Array.from_ndarray([[0.5, 0.], [0., -0.5]], [p_leg, p_leg.conj()])
Sp = npc.Array.from_ndarray([[0., 1.], [0., 0.]], [p_leg, p_leg.conj()])
Sm = npc.Array.from_ndarray([[0., 0.], [1., 0.]], [p_leg, p_leg.conj()])

Hxy = 0.5 * (npc.outer(Sp, Sm) + npc.outer(Sm, Sp))
Hz = npc.outer(Sz, Sz)
H = Hxy + Hz
# here, H has 4 legs
H.iset_leg_labels(["s1", "t1", "s2", "t2"])
H = H.combine_legs([["s1", "s2"], ["t1", "t2"]], qconj=[+1, -1])
# here, H has 2 legs
print(H.legs[0].to_qflat().flatten())
# prints [-2  0  0  2]
E, U = npc.eigh(H)  # diagonalize blocks individually
print(E)
# [ 0.25 -0.75  0.25  0.25]
```

Sites for the local Hilbert space and tensor networks

The next basic concept is that of a local Hilbert space, which is represented by a *Site* in TeNPy. This class does not only label the local states and define the charges, but also provides onsite operators. For example, the *SpinHalfSite* provides the S^+, S^-, S^z operators under the names `'Sp', 'Sm', 'Sz'`, defined as *Array* instances similarly as in the code above. Since the most common sites like for example the *SpinSite* (for general spin S=0.5, 1, 1.5,...), *BosonSite* and *FermionSite* are predefined, a user of TeNPy usually does not need to define the local charges and operators explicitly. The total Hilbert space, i.e, the tensor product of the local Hilbert spaces, is then just given by a list of *Site* instances. If desired, different kinds of *Site* can be combined in that list. This list is then given to classes representing tensor networks like the *MPS* and *MPO*. The tensor network classes also use *Array* instances for the tensors of the represented network.

The following example illustrates the initialization of a spin-1/2 site, an *MPS* representing the Neel state, and an *MPO* representing the Heisenberg model by explicitly defining the W tensor.

```
"""Initialization of sites, MPS and MPO."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

from tenpy.networks.site import SpinHalfSite
from tenpy.networks.mps import MPS
from tenpy.networks.mpo import MPO

spin = SpinHalfSite(conserve="Sz")
print(spin.Sz.to_ndarray())
# [[ 0.5  0. ]
#  [ 0.  -0.5]]

N = 6  # number of sites
sites = [spin] * N  # repeat entry of list N times
pstate = ["up", "down"] * (N // 2)  # Neel state
psi = MPS.from_product_state(sites, pstate, bc="finite")
print("<Sz> =", psi.expectation_value("Sz"))
# <Sz> = [ 0.5 -0.5  0.5 -0.5]
print("<Sp_i Sm_j> =", psi.correlation_function("Sp", "Sm"), sep="\n")
# <Sp_i Sm_j> =
# [[1. 0. 0. 0. 0. 0.]
#  [0. 0. 0. 0. 0. 0.]
#  [0. 0. 1. 0. 0. 0.]
#  [0. 0. 0. 0. 0. 0.]
#  [0. 0. 0. 0. 1. 0.]
#  [0. 0. 0. 0. 0. 0.]]

# define an MPO
Id, Sp, Sm, Sz = spin.Id, spin.Sp, spin.Sm, spin.Sz
J, Delta, hz = 1., 1., 0.2
W_bulk = [[Id, Sp, Sm, Sz, -hz * Sz], [None, None, None, None, 0.5 * J * Sm],
          [None, None, None, None, 0.5 * J * Sp], [None, None, None, None, J * Delta␣
↪* Sz],
          [None, None, None, None, Id]]
W_first = [W_bulk[0]]  # first row
W_last = [[row[-1]] for row in W_bulk]  # last column
Ws = [W_first] + [W_bulk] * (N - 2) + [W_last]
H = MPO.from_grids([spin] * N, Ws, bc='finite', IdL=0, IdR=-1)
print("<psi|H|psi> =", H.expectation_value(psi))
# <psi|H|psi> = -1.25
```

Models

Note: See *Introduction to models* for more information on sites and how to define and extend models on your own.

Technically, the explicit definition of an *MPO* is already enough to call an algorithm like DMRG in *dmrg*. However, writing down the W tensors is cumbersome especially for more complicated models. Hence, TeNPy provides another layer of abstraction for the definition of models, which we discuss first. Different kinds of algorithms require different representations of the Hamiltonian. Therefore, the library offers to specify the model abstractly by the individual onsite terms and coupling terms of the Hamiltonian. The following example illustrates this, again for the Heisenberg model.

```
"""Definition of a model: the XXZ chain."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

from tenpy.networks.site import SpinSite
from tenpy.models.lattice import Chain
from tenpy.models.model import CouplingModel, NearestNeighborModel, MPOModel


class XXZChain(CouplingModel, NearestNeighborModel, MPOModel):
    def __init__(self, L=2, S=0.5, J=1., Delta=1., hz=0.):
        spin = SpinSite(S=S, conserve="Sz")
        # the lattice defines the geometry
        lattice = Chain(L, spin, bc="open", bc_MPS="finite")
        CouplingModel.__init__(self, lattice)
        # add terms of the Hamiltonian
        self.add_coupling(J * 0.5, 0, "Sp", 0, "Sm", 1)  # Sp_i Sm_{i+1}
        self.add_coupling(J * 0.5, 0, "Sp", 0, "Sm", -1)  # Sp_i Sm_{i-1}
        self.add_coupling(J * Delta, 0, "Sz", 0, "Sz", 1)
        # (for site dependent prefactors, the strength can be an array)
        self.add_onsite(-hz, 0, "Sz")

        # finish initialization
        # generate MPO for DMRG
        MPOModel.__init__(self, lat, self.calc_H_MPO())
        # generate H_bond for TEBD
        NearestNeighborModel.__init__(self, lat, self.calc_H_bond())
```

While this generates the same MPO as in the previous code, this example can easily be adjusted and generalized, for example to a higher dimensional lattice by just specifying a different lattice. Internally, the MPO is generated using a finite state machine picture. This allows not only to translate more complicated Hamiltonians into their corresponding MPOs, but also to automate the mapping from a higher dimensional lattice to the 1D chain along which the MPS winds. Note that this mapping introduces longer-range couplings, so the model can no longer be defined to be a *NearestNeighborModel* suited for TEBD if another lattice than the *Chain* is to be used. Of course, many commonly studied models are also predefined. For example, the following code initializes the Heisenberg model on a kagome lattice; the spin liquid nature of the ground state of this model is highly debated in the current literature.

```
"""Initialization of the Heisenberg model on a kagome lattice."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

from tenpy.models.spins import SpinModel

model_params = {
    "S": 0.5,  # Spin 1/2
    "lattice": "Kagome",
```

(continues on next page)

(continued from previous page)

```
    "bc_MPS": "infinite",
    "bc_y": "cylinder",
    "Ly": 2,  # defines cylinder circumference
    "conserve": "Sz",  # use Sz conservation
    "Jx": 1.,
    "Jy": 1.,
    "Jz": 1.  # Heisenberg coupling
}
model = SpinModel(model_params)
```

Algorithms

The highest level in TeNPy is given by algorithms like DMRG and TEBD. Using the previous concepts, setting up a simulation running those algorithms is a matter of just a few lines of code. The following example runs a DMRG simulation, see *dmrg*, exemplary for the transverse field Ising model at the critical point.

```
"""Call of (finite) DMRG."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

from tenpy.networks.mps import MPS
from tenpy.models.tf_ising import TFIChain
from tenpy.algorithms import dmrg

N = 16  # number of sites
model = TFIChain({"L": N, "J": 1., "g": 1., "bc_MPS": "finite"})
sites = model.lat.mps_sites()
psi = MPS.from_product_state(sites, ['up'] * N, "finite")
dmrg_params = {"trunc_params": {"chi_max": 100, "svd_min": 1.e-10}, "mixer": True}
info = dmrg.run(psi, model, dmrg_params)
print("E =", info['E'])
# E = -20.01638790048513
print("max. bond dimension =", max(psi.chi))
# max. bond dimension = 27
```

The switch from DMRG to gls{iDMRG} in TeNPy is simply accomplished by a change of the parameter `"bc_MPS"` from `"finite"` to `"infinite"`, both for the model and the state. The returned `E` is then the energy density per site. Due to the translation invariance, one can also evaluate the correlation length, here slightly away from the critical point.

```
"""Call of infinite DMRG."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

from tenpy.networks.mps import MPS
from tenpy.models.tf_ising import TFIChain
from tenpy.algorithms import dmrg

N = 2  # number of sites in unit cell
model = TFIChain({"L": N, "J": 1., "g": 1.1, "bc_MPS": "infinite"})
sites = model.lat.mps_sites()
psi = MPS.from_product_state(sites, ['up'] * N, "infinite")
dmrg_params = {"trunc_params": {"chi_max": 100, "svd_min": 1.e-10}, "mixer": True}
info = dmrg.run(psi, model, dmrg_params)
print("E =", info['E'])
# E = -1.342864022725017
```

(continues on next page)

(continued from previous page)

```
print("max. bond dimension =", max(psi.chi))
# max. bond dimension = 56
print("corr. length =", psi.correlation_length())
# corr. length = 4.915809146764157
```

Running time evolution with TEBD requires an additional loop, during which the desired observables have to be measured. The following code shows this directly for the infinite version of TEBD.

```
"""Call of (infinite) TEBD."""
# Copyright 2019 TeNPy Developers, GNU GPLv3

from tenpy.networks.mps import MPS
from tenpy.models.tf_ising import TFIChain
from tenpy.algorithms import tebd

M = TFIChain({"L": 2, "J": 1., "g": 1.5, "bc_MPS": "infinite"})
psi = MPS.from_product_state(M.lat.mps_sites(), [0] * 2, "infinite")
tebd_params = {
    "order": 2,
    "delta_tau_list": [0.1, 0.001, 1.e-5],
    "max_error_E": 1.e-6,
    "trunc_params": {
        "chi_max": 30,
        "svd_min": 1.e-10
    }
}
eng = tebd.Engine(psi, M, tebd_params)
eng.run_GS()  # imaginary time evolution with TEBD
print("E =", sum(psi.expectation_value(M.H_bond)) / psi.L)
print("final bond dimensions: ", psi.chi)
```

Literature

This is a (by far non-exhaustive) list of some references for the various ideas behind the code, sorted by year and author. They can be cited from the python doc-strings using the format `[Author####]_`.

TeNPy related sources

General reading

Further reviews are:

Algorithm developments

Related theory

Two-dimensional systems

Introduction to np_conserved

The basic idea is quickly summarized: By inspecting the Hamiltonian, you can identify symmetries, which correspond to conserved quantities, called **charges**. These charges divide the tensors into different sectors. This can be used to infer for example a block-diagonal structure of certain matrices, which in turn speeds up SVD or diagonalization a lot. Even for more general (non-square-matrix) tensors, charge conservation imposes restrictions which blocks of a tensor can be non-zero. Only those blocks need to be saved, and e.g. tensordot can be speeded up.

This introduction covers our implementation of charges; explaining mathematical details of the underlying symmetry is beyond its scope. We refer you to Ref. [Singh2009] for the general idea, which is more nicely explained for the example of a $U(1)$ symmetry in [Singh2010].

Notations

Lets fix the notation for this introduction and the doc-strings in `np_conserved`.

A `Array` is a multi-dimensional array representing a **tensor** with the entries:

$$T_{a_0,a_1,...a_{rank-1}} \quad \text{with} \quad a_i \in \{0, ..., n_i - 1\}$$

Each **leg** a_i corresponds the a vector space of dimension *n_i*.

An **index** of a leg is a particular value $a_i \in \{0, ..., n_i - 1\}$.

The **rank** is the number of legs, the **shape** is $(n_0, ..., n_{rank-1})$.

We restrict ourselfes to abelian charges with entries in $\mathbb{Z}$ or in $\mathbb{Z}_m$. The nature of a charge is specified by m; we set $m = 1$ for charges corresponding to $\mathbb{Z}$. The number of charges is refered to as **qnumber** as a short hand, and the collection of m for each charge is called **qmod**. The qnumber, qmod and possibly descriptive names of the charges are saved in an instance of `ChargeInfo`.

To each index of each leg, a value of the charge(s) is associated. A **charge block** is a contiguous slice corresponding to the same charge(s) of the leg. A **qindex** is an index in the list of charge blocks for a certain leg. A **charge sector** is for given charge(s) is the set of all qindices of that charge(s). A leg is **blocked** if all charge sectors map one-to-one to qindices. Finally, a leg is **sorted**, if the charges are sorted lexiographically. Note that a *sorted* leg is always *blocked*. We can also speak of the complete array to be **blocked by charges** or **legcharge-sorted**, which means that all of its legs are blocked or sorted, respectively. The charge data for a single leg is collected in the class `LegCharge`. A `LegCharge` has also a flag **qconj**, which tells whether the charges point *inward* (+1) or *outward* (-1). What that means, is explained later in *Which entries of the npc Array can be non-zero?*.

For completeness, let us also summarize also the internal structure of an `Array` here: The array saves only non-zero **blocks**, collected as a list of *np.array* in `self._data`. The qindices necessary to map these blocks to the original leg indices are collected in `self._qdata` An array is said to be **qdata-sorted** if its `self._qdata` is lexiographically sorted. More details on this follow *later*. However, note that you usually shouldn't access *_qdata* and *_data* directly - this is only necessary from within *tensordot*, *svd*, etc. Also, an array has a **total charge**, defining which entries can be non-zero - details in *Which entries of the npc Array can be non-zero?*.

Finally, a **leg pipe** (implemented in `LegPipe`) is used to formally combine multiple legs into one leg. Again, more details follow *later*.

Physical Example

For concreteness, you can think of the Hamiltonian $H = -t\sum_{<i,j>}(c_i^\dagger c_j + H.c.) + Un_in_j$ with $n_i = c_i^\dagger c_i$. This Hamiltonian has the global $U(1)$ gauge symmetry $c_i \to c_i e^{i\phi}$. The corresponding charge is the total number of particles $N = \sum_i n_i$. You would then introduce one charge with $m = 1$.

Note that the total charge is a sum of local terms, living on single sites. Thus, you can infer the charge of a single physical site: it's just the value $q_i = n_i \in \mathbb{N}$ for each of the states.

Note that you can only assign integer charges. Consider for example the spin 1/2 Heisenberg chain. Here, you can naturally identify the magnetization $S^z = \sum_i S_i^z$ as the conserved quantity, with values $S_i^z = \pm\frac{1}{2}$. Obviously, if S^z is conserved, then so is $2S^z$, so you can use the charges $q_i = 2S_i^z \in \{-1, +1\}$ for the *down* and *up* states, respectively. Alternatively, you can also use a shift and define $q_i = S_i^z + \frac{1}{2} \in \{0, 1\}$.

As another example, consider BCS like terms $\sum_k (c_k^\dagger c_{-k}^\dagger + H.c.)$. These terms break the total particle conservation, but they preserve the total parity, i.e., N

In the above examples, we had only a single charge conserved at a time, but you might be lucky and have multiple conserved quantities, e.g. if you have two chains coupled only by interactions. TeNPy is designed to handle the general case of multiple charges. When giving examples, we will restrict to one charge, but everything generalizes to multiple charges.

The different formats for LegCharge

As mentioned above, we assign charges to each index of each leg of a tensor. This can be done in three formats: **qflat**, as **qind** and as **qdict**. Let me explain them with examples, for simplicity considereing only a single charge (the most inner array has one entry for each charge).

qflat form: simply a list of charges for each index. An example:

```
qflat = [[-2], [-1], [-1], [0], [0], [0], [0], [3], [3]]
```

This tells you that the leg has size 9, the charges for are `[-2], [-1], [-1], ..., [3]` for the indices `0, 1, 2, 3,..., 8`. You can identify four *charge blocks* `slice(0, 1), slice(1, 3), slice(3, 7), slice(7, 9)` in this example, which have charges `[-2], [-1], [0], [3]`. In other words, the indices `1, 2` (which are in `slice(1, 3)`) have the same charge value `[-1]`. A *qindex* would just enumerate these blocks as `0, 1, 2, 3`.

qind form: a 1D array *slices* and a 2D array *charges*. This is a more compact version than the *qflat* form: the *slices* give a partition of the indices and the *charges* give the charge values. The same example as above would simply be:

```
slices = [0, 1, 3, 7, 9]
charges = [[-2], [-1], [0], [3]]
```

Note that *slices* includes `0` as first entry and the number of indices (here `9`) as last entries. Thus it has len `block_number + 1`, where `block_number` (given by `block_number`) is the number of charge blocks in the leg, i.e. a *qindex* runs from 0 to `block_number-1`. On the other hand, the 2D array *charges* has shape `(block_number, qnumber)`, where `qnumber` is the number of charges (given by *qnumber*).

In that way, the *qind* form maps an *qindex*, say `qi`, to the indices `slice(slices[qi], slices[qi+1])` and the charge(s) `charges[qi]`.

qdict form: a dictionary in the other direction than qind, taking charge tuples to slices. Again for the same example:

```
{(-2,): slice(0, 1),
 (-1,): slice(1, 3),
 (0,) : slice(3, 7),
 (3,) : slice(7, 9)}
```

Since the keys of a dictionary are unique, this form is only possible if the leg is *completely blocked*.

The `LegCharge` saves the charge data of a leg internally in *qind* form, directly in the attribute *slices* and *charges*. However, it also provides convenient functions for conversion between from and to the *qflat* and *qdict* form.

The above example was nice since all charges were sorted and the charge blocks were 'as large as possible'. This is however not required.

The following example is also a valid *qind* form:

```
slices = [0, 1, 3, 5, 7, 9]
charges = [[-2], [-1], [0], [0], [3]]
```

This leads to the *same qflat* form as the above examples, thus representing the same charges on the leg indices. However, regarding our Arrays, this is quite different, since it diveds the leg into 5 (instead of previously 4) charge blocks. We say the latter example is *not bunched*, while the former one is *bunched*.

To make the different notions of *sorted* and *bunched* clearer, consider the following (valid) examples:

charges	bunched	sorted	blocked
`[[-2], [-1], [0], [1], [3]]`	`True`	`True`	`True`
`[[-2], [-1], [0], [0], [3]]`	`False`	`True`	`False`
`[[-2], [0], [-1], [1], [3]]`	`True`	`False`	`True`
`[[-2], [0], [-1], [0], [3]]`	`True`	`False`	`False`

If a leg is *bunched* and *sorted*, it is automatically *blocked* (but not vice versa). See also *below* for further comments on that.

Which entries of the npc Array can be non-zero?

The reason for the speedup with np_conserved lies in the fact that it saves only the blocks 'compatible' with the charges. But how is this 'compatible' defined?

Assume you have a tensor, call it T, and the `LegCharge` for all of its legs, say $a, b, c, ...$

Remeber that the LegCharge associates to each index of the leg a charge value (for each of the charges, if *qnumber* > 1). Let `a.to_qflat()[ia]` denote the charge(s) of index `ia` for leg `a`, and similar for other legs.

In addition, the LegCharge has a flag `qconj`. This flag **qconj** is only a sign, saved as +1 or -1, specifying whether the charges point 'inward' (+1, default) or 'outward' (-1) of the tensor.

Then, the **total charge of an entry** `T[ia, ib, ic, ...]` of the tensor is defined as:

```
qtotal[ia, ib, ic, ...] = a.to_qflat()[ia] * a.qconj + b.to_qflat()[ib] * b.qconj + c.
↪to_qflat()[ic] * c.qconj + ...  modulo qmod
```

The rule which entries of the a `Array` can be non-zero (i.e., are 'compatible' with the charges), is then very simple:

Rule for non-zero entries

An entry `ia, ib, ic, ...` of a *Array* can only be non-zero, if `qtotal[ia, ib, ic, ...]` matches the *unique* `qtotal` attribute of the class.

In other words, there is a *single* **total charge** `.qtotal` attribute of a *Array*. All indices `ia, ib, ic, ...` for which the above defined `qtotal[ia, ib, ic, ...]` matches this *total charge*, are said to be **compatible with the charges** and can be non-zero. All other indices are **incompatible with the charges** and must be zero.

In case of multiple charges, *qnumber* > 1, is a straigth-forward generalization: an entry can only be non-zero if it is *compatible* with each of the defined charges.

The pesky qconj - contraction as an example

Why did we introduce the `qconj` flag? Remember it's just a sign telling whether the charge points inward or outward. So whats the reasoning?

The short answer is, that LegCharges actually live on bonds (i.e., legs which are to be contracted) rather than individual tensors. Thus, it is convenient to share the LegCharges between different legs and even tensors, and just adjust the sign of the charges with *qconj*.

As an example, consider the contraction of two tensors, $C_{ia,ic} = \sum_{ib} A_{ia,ib} B_{ib,ic}$. For simplicity, say that the total charge of all three tensors is zero. What are the implications of the above rule for non-zero entries? Or rather, how can we ensure that `C` complies with the above rule? An entry `C[ia,ic]` will only be non-zero, if there is an `ib` such that both `A[ia,ib]` and `B[ib,ic]` are non-zero, i.e., both of the following equations are fullfilled:

```
A.qtotal == A.legs[0].to_qflat()[ia] * A.legs[0].qconj + A.legs[1].to_qflat()[ib] * A.
↪legs[1].qconj  modulo qmod
B.qtotal == B.legs[0].to_qflat()[ib] * B.legs[0].qconj + B.legs[1].to_qflat()[ic] * B.
↪legs[1].qconj  modulo qmod
```

(`A.legs[0]` is the *LegCharge* saving the charges of the first leg (with index `ia`) of A.)

For the uncontracted legs, we just keep the charges as they are:

```
C.legs = [A.legs[0], B.legs[1]]
```

It is then straight-forward to check, that the rule is fullfilled for C, if the following condition is met:

```
A.qtotal + B.qtotal - C.qtotal == A.legs[1].to_qflat()[ib] A.b.qconj + B.legs[0].to_
↪qflat()[ib] B.b.qconj  modulo qmod
```

The easiest way to meet this condition is (1) to require that `A.b` and `B.b` share the *same* charges `b.to_qflat()`, but have opposite *qconj*, and (2) to define `C.qtotal = A.qtotal + B.qtotal`. This justifies the introduction of *qconj*: when you define the tensors, you have to define the *LegCharge* for the *b* only once, say for `A.legs[1]`. For `B.legs[0]` you simply use `A.legs[1].conj()` which creates a copy of the LegCharge with shared *slices* and *charges*, but opposite *qconj*. As a more impressive example, all 'physical' legs of an MPS can usually share the same *LegCharge* (up to different `qconj` if the local Hilbert space is the same). This leads to the following convention:

Convention

When an npc algorithm makes tensors which share a bond (either with the input tensors, as for tensordot, or amongst the output tensors, as for SVD), the algorithm is free, but not required, to use the **same** `LegCharge` for the tensors

sharing the bond, *without* making a copy. Thus, if you want to modify a LegCharge, you **must** make a copy first (e.g. by using methods of LegCharge for what you want to acchive).

Assigning charges to non-physical legs

From the above physical examples, it should be clear how you assign charges to physical legs. But what about other legs, e.g, the virtual bond of an MPS (or an MPO)?

The charge of these bonds must be derived by using the 'rule for non-zero entries', as far as they are not arbitrary. As a concrete example, consider an MPS on just two spin 1/2 sites:

```
|
|    x->-  | A |  ->-y->-  | B |  ->-z
|          -----            -----
|            ^               ^
|            |p              |p
```

The two legs `p` are the physical legs and share the same charge, as they both describe the same local Hilbert space. For better distincition, let me label the indices of them by $\uparrow = 0$ and $\downarrow = 1$. As noted above, we can associate the charges 1 ($p = \uparrow$) and -1 ($p = \downarrow$), respectively, so we define:

```
chinfo = npc.ChargeInfo([1], ['2*Sz'])
p  = npc.LegCharge.from_qflat(chinfo, [1, -1], qconj=+1)
```

For the `qconj` signs, we stick to the convention used in our MPS code and indicated by the arrows in above 'picture': physical legs are incoming (`qconj=+1`), and from left to right on the virtual bonds. This is acchieved by using `[p, x, y.conj()]` as *legs* for `A`, and `[p, y, z.conj()]` for `B`, with the default `qconj=+1` for all `p, x, y, z`: `y.conj()` has the same charges as `y`, but opposite `qconj=-1`.

The legs `x` and `z` of an `L=2` MPS, are 'dummy' legs with just one index `0`. The charge on one of them, as well as the total charge of both `A` and `B` is arbitrary (i.e., a gauge freedom), so we make a simple choice: total charge 0 on both arrays, as well as for $x = 0$, `x = npc.LegCharge.from_qflat(chinfo, [0], qconj=+1)`.

The charges on the bonds y and z then depend on the state the MPS represents. Here, we consider a singlet $\psi = (|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle)/\sqrt{2}$ as a simple example. A possible MPS representation is given by:

```
A[up, :, :]   = [[1/2.**0.5, 0]]      B[up, :, :]   = [[0], [-1]]
A[down, :, :] = [[0, 1/2.**0.5]]      B[down, :, :] = [[1], [0]]
```

There are two non-zero entries in `A`, for the indices $(a, x, y) = (\uparrow, 0, 0)$ and $(\downarrow, 0, 1)$. For $(a, x, y) = (\uparrow, 0, 0)$, we want:

```
A.qtotal = 0 = p.to_qflat()[up] * p.qconj + x.to_qflat()[0] * x.qconj + y.conj().to_
↪qflat()[0] * y.conj().qconj
             = 1                * (+1)     + 0               * (+1)     + y.conj().to_
↪qflat()[0] * (-1)
```

This fixes the charge of `y=0` to 1. A similar calculation for $(a, x, y) = (\downarrow, 0, 1)$ yields the charge `-1` for `y=1`. We have thus all the charges of the leg `y` and can define `y = npc.LegCharge.from_qflat(chinfo, [1, -1], qconj=+1)`.

Now take a look at the entries of `B`. For the non-zero entry $(b, y, z) = (\uparrow, 1, 0)$, we want:

```
B.qtotal = 0 = p.to_qflat()[up] * p.qconj + y.to_qflat()[1] * y.qconj + z.conj().to_
↪qflat()[0] * z.conj().qconj
             = 1                * (+1)     + (-1)            * (+1)     + z.conj().to_
↪qflat()[0] * (-1)
```

(continues on next page)

(continued from previous page)

This implies the charge 0 for $z = 0$, thus `z = npc.LegCharge.form_qflat(chinfo, [0], qconj=+1)`. Finally, note that the rule for $(b, y, z) = (\downarrow, 0, 0)$ is automatically fullfilled! This is an implication of the fact that the singlet has a well defined value for $S^z_a + S^z_b$. For other states without fixed magnetization (e.g., $|\uparrow\uparrow\rangle + |\downarrow\downarrow\rangle$) this would not be the case, and we could not use charge conservation.

As an exercise, you can calculate the charge of z in the case that `A.qtotal=5`, `B.qtotal = -1` and charge `2` for `x=0`. The result is -2.

Note: This section is meant be an pedagogical introduction. In you program, you can use the functions *detect_legcharge()* (which does exactly what's described above) or *detect_qtotal()* (if you know all *LegCharges*, but not *qtotal*).

Array creation

Making an new *Array* requires both the tensor entries (data) and charge data.

The default initialization `a = Array(...)` creates an empty Array, where all entries are zero (equivalent to *zeros()*). (Non-zero) data can be provided either as a dense *np.array* to *from_ndarray()*, or by providing a numpy function such as *np.random*, *np.ones* etc. to *from_func()*.

In both cases, the charge data is provided by one *ChargeInfo*, and a *LegCharge* instance for each of the legs.

Note: The charge data instances are not copied, in order to allow it to be shared between different Arrays. Consequently, you *must* make copies of the charge data, if you manipulate it directly. (However, methods like *sort()* do that for you.)

Of course, a new *Array* can also created using the charge data from exisiting Arrays, for examples with *zeros_like()* or creating a (deep or shallow) *copy()*. Further, there are the higher level functions like *tensordot()* or *svd()*, which also return new Arrays.

Further, new Arrays are created by the various functions like *tensordot* or *svd* in *np_conserved*.

Complete blocking of Charges

While the code was designed in such a way that each charge sector has a different charge, the code should still run correctly if multiple charge sectors (for different qindex) correspond to the same charge. In this sense *Array* can act like a sparse array class to selectively store subblocks. Algorithms which need a full blocking should state that explicitly in their doc-strings. (Some functions (like *svd* and *eigh*) require complete blocking internally, but if necessary they just work on a temporary copy returned by `as_completely_blocked()`).

If you expect the tensor to be dense subject to charge constraints (as for MPS), it will be most efficient to fully block by charge, so that work is done on large chunks.

However, if you expect the tensor to be sparser than required by charge (as for an MPO), it may be convenient not to completely block, which forces smaller matrices to be stored, and hence many zeroes to be dropped. Nevertheless, the algorithms were not designed with this in mind, so it is not recommended in general. (If you want to use it, run a benchmark to check whether it is really faster!)

If you haven't created the array yet, you can call *sort()* (with `bunch=True`) on each *LegCharge* which you want to block. This sorts by charges and thus induces a permution of the indices, which is also returned as an 1D array `perm`. For consistency, you have to apply this permutation to your flat data as well.

Alternatively, you can simply call *sort_legcharge()* on an existing *Array*. It calls *sort()* internally on the specified legs and performs the necessary permutations directly to (a copy of) *self*. Yet, you should keep in mind, that the axes are permuted afterwards.

Internal Storage schema of npc Arrays

The actual data of the tensor is stored in `_data`. Rather than keeping a single np.array (which would have many zeros in it), we store only the non-zero sub blocks. So `_data` is a python list of *np.array*'s. The order in which they are stored in the list is not physically meaningful, and so not guaranteed (more on this later). So to figure out where the sub block sits in the tensor, we need the `_qdata` structure (on top of the LegCharges in `legs`).

Consider a rank 3 tensor `T`, with the first leg like:

```
legs[0].slices = np.array([0, 1, 4, ...])
legs[0].charges = np.array([[-2], [1], ...])
```

Each row of *charges* gives the charges for a *charge block* of the leg, with the actual indices of the total tensor determined by the *slices*. The *qindex* simply enumerates the charge blocks of a lex. Picking a qindex (and thus a *charge block*) from each leg, we have a subblock of the tensor.

For each (non-zero) subblock of the tensor, we put a (numpy) ndarray entry in the `_data` list. Since each subblock of the tensor is specified by *rank* qindices, we put a corresponding entry in `_qdata`, which is a 2D array of shape `(#stored_blocks, rank)`. Each row corresponds to a non-zero subblock, and there are rank columns giving the corresponding qindex for each leg.

Example: for a rank 3 tensor we might have:

```
T._data = [t1, t2, t3, t4, ...]
T. _qdata = np.array([[3, 2, 1],
                      [1, 1, 1],
                      [4, 2, 2],
                      [2, 1, 2],
                      ...       ])
```

The third subblock has an ndarray `t3`, and qindices `[4 2 2]` for the three legs.

- To find the position of `t3` in the actual tensor you can use *get_slice()*:

  ```
  T.legs[0].get_slice(4), T.legs[1].get_slice(2), T.legs[2].get_slice(2)
  ```

  The function `leg.get_charges(qi)` simply returns `slice(leg.slices[qi], leg.slices[qi+1])`

- To find the charges of t3, we an use *get_charge()*:

  ```
  T.legs[0].get_charge(2), T.legs[1].get_charge(2), T.legs[2].get_charge(2)
  ```

  The function `leg.get_charge(qi)` simply returns `leg.charges[qi]*leg.qconj`.

Note: Outside of *np_conserved*, you should use the API to access the entries. If you really need to iterate over all blocks of an Array `T`, try `for (block, blockslices, charges, qindices) in T: do_something()`.

The order in which the blocks stored in _data/_qdata is arbitrary (although of course _data and _qdata must be in correspondence). However, for many purposes it is useful to sort them according to some convention. So we include a flag ._qdata_sorted to the array. So, if sorted (with *isort_qdata()*, the _qdata example above goes to

```
_qdata = np.array([[1, 1, 1],
                   [3, 2, 1],
                   [2, 1, 2],
                   [4, 2, 2],
                   ...       ])
```

Note that *np.lexsort* chooses the right-most column to be the dominant key, a convention we follow throughout.

If _qdata_sorted == True, _qdata and _data are guaranteed to be lexsorted. If _qdata_sorted == False, there is no gaurantee. If an algorithm modifies _qdata, it **must** set _qdata_sorted = False (unless it gaurantees it is still sorted). The routine sort_qdata() brings the data to sorted form.

Indexing of an Array

Although it is usually not necessary to access single entries of an *Array*, you can of course do that. In the simplest case, this is something like A[0, 2, 1] for a rank-3 Array A. However, accessing single entries is quite slow and usually not recommended. For small Arrays, it may be convenient to convert them back to flat numpy arrays with *to_ndarray()*.

On top of that very basic indexing, *Array* supports slicing and some kind of advanced indexing, which is however different from the one of numpy arrarys (described here). Unlike numpy arrays, our Array class does not broadcast existing index arrays – this would be terribly slow. Also, *np.newaxis* is not supported, since inserting new axes requires additional information for the charges.

Instead, we allow just indexing of the legs independent of each other, of the form A[i0, i1, ...]. If all indices i0, i1, ... are integers, the single corresponding entry (of type *dtype*) is returned.

However, the individual 'indices' i0 for the individual legs can also be one of what is described in the following list. In that case, a new *Array* with less data (specified by the indices) is returned.

The 'indices' can be:

- an *int*: fix the index of that axis, return array with one less dimension. See also *take_slice()*.
- a slice(None) or :: keep the complete axis
- an Ellipsis or ...: shorthand for slice(None) for missing axes to fix the len
- an 1D bool *ndarray* mask: apply a mask to that axis, see *iproject()*.
- a slice(start, stop, step) or start:stop:step: keep only the indices specified by the slice. This is also implemented with *iproject*.
- an 1D int *ndarray* mask: keep only the indices specified by the array. This is also implemented with *iproject*.

For slices and 1D arrays, additional permuations may be perfomed with the help of *permute()*.

If the number of indices is less than *rank*, the remaining axes remain free, so for a rank 4 Array A, A[i0, i1] == A[i0, i1, ...] == A[i0, i1, :, :].

Note that indexing always **copies** the data – even if *int* contains just slices, in which case numpy would return a view. However, assigning with A[:, [3, 5], 3] = B should work as you would expect.

Warning: Due to numpy's advanced indexing, for 1D integer arrays `a0` and `a1` the following holds

```
A[a0, a1].to_ndarray() == A.to_ndarray()[np.ix_(a0, a1)] != A.to_ndarray()[a0, a1]
```

For a combination of slices and arrays, things get more complicated with numpys advanced indexing. In that case, a simple `np.ix_(...)` doesn't help any more to emulate our version of indexing.

Introduction to combine_legs, split_legs and LegPipes

Often, it is necessary to "combine" multiple legs into one: for example to perfom a SVD, a tensor needs to be viewed as a matrix. For a flat array, this can be done with `np.reshape`, e.g., if `A` has shape `(10, 3, 7)` then `B = np.reshape(A, (30, 7))` will result in a (view of the) array with one less dimension, but a "larger" first leg. By default (`order='C'`), this results in

```
B[i*3 + j , k] == A[i, j, k] for i in range(10) for j in range(3) for k in range(7)
```

While for a np.array, also a reshaping `(10, 3, 7) -> (2, 21, 5)` would be allowed, it does not make sense physically. The only sensible "reshape" operation on an *Array* are

1) to **combine** multiple legs into one **leg pipe** (*LegPipe*) with *combine_legs()*, or
2) to **split** a pipe of previously combined legs with *split_legs()*.

Each leg has a Hilbert space, and a representation of the symmetry on that Hilbert space. Combining legs corresponds to the tensor product operation, and for abelian groups, the corresponding "fusion" of the representation is the simple addition of charge.

Fusion is not a lossless process, so if we ever want to split the combined leg, we need some additional data to tell us how to reverse the tensor product. This data is saved in the class *LegPipe*, derived from the *LegCharge* and used as new *leg*. Details of the information contained in a LegPipe are given in the class doc string.

The rough usage idea is as follows:

1) You can call *combine_legs()* without supplying any LegPipes, *combine_legs* will then make them for you.

   Nevertheless, if you plan to perform the combination over and over again on sets of legs you know to be identical [with same charges etc, up to an overall -1 in *qconj* on all incoming and outgoing Legs] you might make a LegPipe anyway to save on the overhead of computing it each time.

2) In any way, the resulting Array will have a *LegPipe* as a LegCharge on the combined leg. Thus, it – and all tensors inheriting the leg (e.g. the results of *svd*, *tensordot* etc.) – will have the information how to split the *LegPipe* back to the original legs.

3) Once you performed the necessary operations, you can call `split_legs()`. This uses the information saved in the *LegPipe* to split the legs, recovering the original legs.

For a LegPipe, `conj`()` changes `qconj` for the outgoing pipe *and* the incoming legs. If you need a *LegPipe* with the same incoming `qconj`, use *outer_conj()*.

Leg labeling

It's convenient to name the legs of a tensor: for instance, we can name legs 0, 1, 2 to be `'a', 'b', 'c'`: T_{i_a,i_b,i_c}. That way we don't have to remember the ordering! Under tensordot, we can then call

```
U = npc.tensordot(S, T, axes = [ [...],  ['b'] ] )
```

without having to remember where exactly `'b'` is. Obviously `U` should then inherit the name of its legs from the uncontracted legs of *S* and *T*. So here is how it works:

- Labels can *only* be strings. The labels should not include the characters `.` or `?`. Internally, the labels are stored as dict `a.labels = {label:  leg_position, ...}`. Not all legs need a label.
- To set the labels, call

  ```
  A.set_labels(['a', 'b', None, 'c', ... ])
  ```

  which will set up the labeling `{'a':  0, 'b':  1, 'c':  3 ...}`.
- (Where implemented) the specification of axes can use either the labels **or** the index positions. For instance, the call `tensordot(A, B, [ ['a', 2, 'c'], [...]])` will interpret `'a'` and `'c'` as labels (calling *`get_leg_indices()`* to find their positions using the dict) and 2 as 'the 2nd leg'. That's why we require labels to be strings!
- **Labels will be intelligently inherited through the various operations of *np_conserved*.**
  - Under *transpose*, labels are permuted.
  - Under *tensordot*, labels are inherited from uncontracted legs. If there is a collision, both labels are dropped.
  - Under *combine_legs*, labels get concatenated with a `.` delimiter and sourrounded by brackets. Example: let `a.labels = {'a':  1, 'b':  2, 'c':  3}`. Then if `b = a.combine_legs([[0, 1], [2]])`, it will have `b.labels = {'(a.b)':  0, '(c)':  1}`. If some sub-leg of a combined leg isn't named, then a `'?#'` label is inserted (with `#` the leg index), e.g., `'a.?0.c'`.
  - Under *split_legs*, the labels are split using the delimiters (and the `'?#'` are dropped).
  - Under *conj*, *iconj*: take `'a' -> 'a*'`, `'a*' -> 'a'`, and `'(a.(b*.c))' -> '(a*.(b.c*))'`
  - Under *svd*, the outer labels are inherited, and inner labels can be optionally passed.
  - Under *pinv*, the labels are transposed.

See also

- The module *`tenpy.linalg.np_conserved`* should contain all the API needed from the point of view of the algorithms. It contians the fundamental *`Array`* class and functions for working with them (creating and manipulating).
- The module *`tenpy.linalg.charges`* contains implementations for the charge structure, for example the classes *`ChargeInfo`*, *`LegCharge`*, and *`LegPipe`*. As noted above, the 'public' API is imported to (and accessible from) *`np_conserved`*.

A full example code for spin-1/2

Below follows a full example demonstrating the creation and contraction of Arrays. (It's the file *a_np_conserved.py* in the examples folder of the tenpy source.)

```
"""An example code to demonstrate the usage of :class:`~tenpy.linalg.np_conserved.
↪Array`.

This example includes the following steps:
1) create Arrays for an Neel MPS
2) create an MPO representing the nearest-neighbour AFM Heisenberg Hamiltonian
3) define 'environments' left and right
4) contract MPS and MPO to calculate the energy
5) extract two-site hamiltonian ``H2`` from the MPO
6) calculate ``exp(-1.j*dt*H2)`` by diagonalization of H2
7) apply ``exp(H2)`` to two sites of the MPS and truncate with svd

Note that this example uses only np_conserved, but no other modules.
Compare it to the example `b_mps.py`,
which does the same steps using a few predefined classes like MPS and MPO.
"""
# Copyright 2018-2019 TeNPy Developers, GNU GPLv3

import tenpy.linalg.np_conserved as npc
import numpy as np

# model parameters
Jxx, Jz = 1., 1.
L = 20
dt = 0.1
cutoff = 1.e-10
print("Jxx={Jxx}, Jz={Jz}, L={L:d}".format(Jxx=Jxx, Jz=Jz, L=L))

print("1) create Arrays for an Neel MPS")

#  vL ->--B-->- vR
#         |
#         ^
#         |
#         p

# create a ChargeInfo to specify the nature of the charge
chinfo = npc.ChargeInfo([1], ['2*Sz'])  # the second argument is just a descriptive␣
↪name

# create LegCharges on physical leg and even/odd bonds
p_leg = npc.LegCharge.from_qflat(chinfo, [[1], [-1]])  # charges for up, down
v_leg_even = npc.LegCharge.from_qflat(chinfo, [[0]])
v_leg_odd = npc.LegCharge.from_qflat(chinfo, [[1]])

B_even = npc.zeros([v_leg_even, v_leg_odd.conj(), p_leg])
B_odd = npc.zeros([v_leg_odd, v_leg_even.conj(), p_leg])
B_even[0, 0, 0] = 1.  # up
B_odd[0, 0, 1] = 1.  # down

for B in [B_even, B_odd]:
    B.iset_leg_labels(['vL', 'vR', 'p'])  # virtual left/right, physical
```

(continues on next page)

(continued from previous page)

```
Bs = [B_even, B_odd] * (L // 2) + [B_even] * (L % 2)  # (right-canonical)
Ss = [np.ones(1)] * L  # Ss[i] are singular values between Bs[i-1] and Bs[i]

# Side remark:
# An MPS is expected to have non-zero entries everywhere compatible with the charges.
# In general, we recommend to use `sort_legcharge` (or `as_completely_blocked`)
# to ensure complete blocking. (But the code will also work, if you don't do it.)
# The drawback is that this might introduce permutations in the indices of single␣
↪legs,
# which you have to keep in mind when converting dense numpy arrays to and from npc.
↪Arrays.

print("2) create an MPO representing the AFM Heisenberg Hamiltonian")

#         p*
#         |
#         ^
#         |
#  wL ->--W-->- wR
#         |
#         ^
#         |
#         p

# create physical spin-1/2 operators Sz, S+, S-
Sz = npc.Array.from_ndarray([[0.5, 0.], [0., -0.5]], [p_leg, p_leg.conj()])
Sp = npc.Array.from_ndarray([[0., 1.], [0., 0.]], [p_leg, p_leg.conj()])
Sm = npc.Array.from_ndarray([[0., 0.], [1., 0.]], [p_leg, p_leg.conj()])
Id = npc.eye_like(Sz)  # identity
for op in [Sz, Sp, Sm, Id]:
    op.iset_leg_labels(['p', 'p*'])  # physical in, physical out

mpo_leg = npc.LegCharge.from_qflat(chinfo, [[0], [2], [-2], [0], [0]])

W_grid = [[Id,   Sp,   Sm,   Sz,   None          ],
          [None, None, None, None, 0.5 * Jxx * Sm],
          [None, None, None, None, 0.5 * Jxx * Sp],
          [None, None, None, None, Jz * Sz       ],
          [None, None, None, None, Id            ]]  # yapf:disable

W = npc.grid_outer(W_grid, [mpo_leg, mpo_leg.conj()])
W.iset_leg_labels(['wL', 'wR', 'p', 'p*'])  # wL/wR = virtual left/right of the MPO
Ws = [W] * L

print("3) define 'environments' left and right")

#  .---->- vR     vL ->----.
#  |                       |
#  envL->- wR     wL ->-envR
#  |                       |
#  .---->- vR*    vL*->----.

envL = npc.zeros([W.get_leg('wL').conj(), Bs[0].get_leg('vL').conj(), Bs[0].get_leg(
↪'vL')])
envL.iset_leg_labels(['wR', 'vR', 'vR*'])
envL[0, :, :] = npc.diag(1., envL.legs[1])
```

(continues on next page)

(continued from previous page)

```
envR = npc.zeros([W.get_leg('wR').conj(), Bs[-1].get_leg('vR').conj(), Bs[-1].get_leg(
↪'vR')])
envR.iset_leg_labels(['wL', 'vL', 'vL*'])
envR[-1, :, :] = npc.diag(1., envR.legs[1])

print("4) contract MPS and MPO to calculate the energy <psi|H|psi>")
contr = envL
for i in range(L):
    # contr labels: wR, vR, vR*
    contr = npc.tensordot(contr, Bs[i], axes=('vR', 'vL'))
    # wR, vR*, vR, p
    contr = npc.tensordot(contr, Ws[i], axes=(['p', 'wR'], ['p*', 'wL']))
    # vR*, vR, wR, p
    contr = npc.tensordot(contr, Bs[i].conj(), axes=(['p', 'vR*'], ['p*', 'vL*']))
    # vR, wR, vR*
    # note that the order of the legs changed, but that's no problem with labels:
    # the arrays are automatically transposed as necessary
E = npc.inner(contr, envR, axes=(['vR', 'wR', 'vR*'], ['vL', 'wL', 'vL*']))
print("E =", E)

print("5) calculate two-site hamiltonian ``H2`` from the MPO")
# label left, right physical legs with p, q
W0 = W.replace_labels(['p', 'p*'], ['p0', 'p0*'])
W1 = W.replace_labels(['p', 'p*'], ['p1', 'p1*'])
H2 = npc.tensordot(W0, W1, axes=('wR', 'wL')).itranspose(['wL', 'wR', 'p0', 'p1', 'p0*
↪', 'p1*'])
H2 = H2[0, -1]  # (If H has single-site terms, it's not that simple anymore)
print("H2 labels:", H2.get_leg_labels())

print("6) calculate exp(H2) by diagonalization of H2")
# diagonalization requires to view H2 as a matrix
H2 = H2.combine_legs([('p0', 'p1'), ('p0*', 'p1*')], qconj=[+1, -1])
print("labels after combine_legs:", H2.get_leg_labels())
E2, U2 = npc.eigh(H2)
print("Eigenvalues of H2:", E2)
U_expE2 = U2.scale_axis(np.exp(-1.j * dt * E2), axis=1)  # scale_axis ~= apply an␣
↪diagonal matrix
exp_H2 = npc.tensordot(U_expE2, U2.conj(), axes=(1, 1))
exp_H2.iset_leg_labels(H2.get_leg_labels())
exp_H2 = exp_H2.split_legs()  # by default split all legs which are `LegPipe`
# (this restores the originial labels ['p0', 'p1', 'p0*', 'p1*'] of `H2` in `exp_H2`)

print("7) apply exp(H2) to even/odd bonds of the MPS and truncate with svd")
# (this implements one time step of first order TEBD)
for even_odd in [0, 1]:
    for i in range(even_odd, L - 1, 2):
        B_L = Bs[i].scale_axis(Ss[i], 'vL').ireplace_label('p', 'p0')
        B_R = Bs[i + 1].replace_label('p', 'p1')
        theta = npc.tensordot(B_L, B_R, axes=('vR', 'vL'))
        theta = npc.tensordot(exp_H2, theta, axes=(['p0*', 'p1*'], ['p0', 'p1']))
        # view as matrix for SVD
        theta = theta.combine_legs([('vL', 'p0'), ('p1', 'vR')], new_axes=[0, 1],␣
↪qconj=[+1, -1])
        # now theta has labels '(vL.p0)', '(p1.vR)'
        U, S, V = npc.svd(theta, inner_labels=['vR', 'vL'])
        # truncate
        keep = S > cutoff
```

(continues on next page)

(continued from previous page)

```
        S = S[keep]
        invsq = np.linalg.norm(S)
        Ss[i + 1] = S / invsq
        U = U.iscale_axis(S / invsq, 'vR')
        Bs[i] = U.split_legs('(vL.p0)').iscale_axis(Ss[i]**(-1), 'vL').ireplace_label(
↪'p0', 'p')
        Bs[i + 1] = V.split_legs('(p1.vR)').ireplace_label('p1', 'p')
print("finished")
```

Introduction to models

What is a model?

Abstractly, a **model** stands for some physical (quantum) system to be described. For tensor networks algorithms, the model is usually specified as a Hamiltonian written in terms of second quantization. For example, let us consider a spin-1/2 Heisenberg model described by the Hamiltonian

$$H = J \sum_i S_i^x S_{i+1}^x + S_i^y S_{i+1}^y + S_i^z S_{i+1}^z$$

Note that a few things are defined more or less implicitly.

- The local Hilbert space: it consists of Spin-1/2 degrees of freedom with the usual spin-1/2 operators S^x, S^y, S^z.
- The geometric (lattice) strucuture: above, we spoke of a 1D "chain".
- The boundary conditions: do we have open or periodic boundary conditions? The "chain" suggests open boundaries, which are in most cases preferable for MPS-based methods.
- The range of i: How many sites do we consider (for a 2D system: in each direction)?

Obviously, these things need to be specified in TeNPy in one way or another, if we want to define a model.

Ultimately, our goal is to run some algorithm. Each algorithm requires the model and Hamiltonian to be specified in a particular form. We have one class for each such required form. For example *dmrg* requires an *MPOModel*, which contains the Hamiltonian written as an *MPO*. On the other hand, if we want to evolve a state with *tebd* we need a *NearestNeighborModel*, in which the Hamiltonian is written in terms of two-site bond-terms to allow a Suzuki-Trotter decomposition of the time-evolution operator.

Implmenting you own model ultimatley means to get an instance of *MPOModel* or *NearestNeighborModel*. The predefined classes in the other modules under *models* are subclasses of at least one of those, you will see examples later down below.

The Hilbert space

The **local Hilbert** space is represented by a *Site* (read its doc-string!). In particular, the *Site* contains the local *LegCharge* and hence the meaning of each basis state needs to be defined. Beside that, the site contains the local operators - those give the real meaning to the local basis. Having the local operators in the site is very convenient, because it makes them available by name for example when you want to calculate expectation values. The most common sites (e.g. for spins, spin-less or spin-full fermions, or bosons) are predefined in the module *tenpy.networks.site*, but if necessary you can easily extend them by adding further local operators or completely write your own subclasses of *Site*.

The full Hilbert space is a tensor product of the local Hilbert space on each site.

Note: The *LegCharge* of all involved sites need to have a common `ChargeInfo` in order to allow the contraction of tensors acting on the various sites. This can be ensured with the function *multi_sites_combine_charges()*.

An example where *multi_sites_combine_charges()* is needed would be a coupling of different types of sites, e.g., when a tight binding chain of fermions is coupled to some local spin degrees of freedom. Another use case of this function would be a model with a $U(1)$ symmetry involving only half the sites, say $\sum_{i=0}^{L/2} n_{2i}$.

Note: If you don't know about the charges and *np_conserved* yet, but want to get started with models right away, you can set `conserve=None` in the existing sites or use `leg = tenpy.linalg.np_conserved.LegCharge.from_trivial(d)` for an implementation of your custom site, where *d* is the dimension of the local Hilbert space. Alternatively, you can find some introduction to the charges in the *Introduction to np_conserved*.

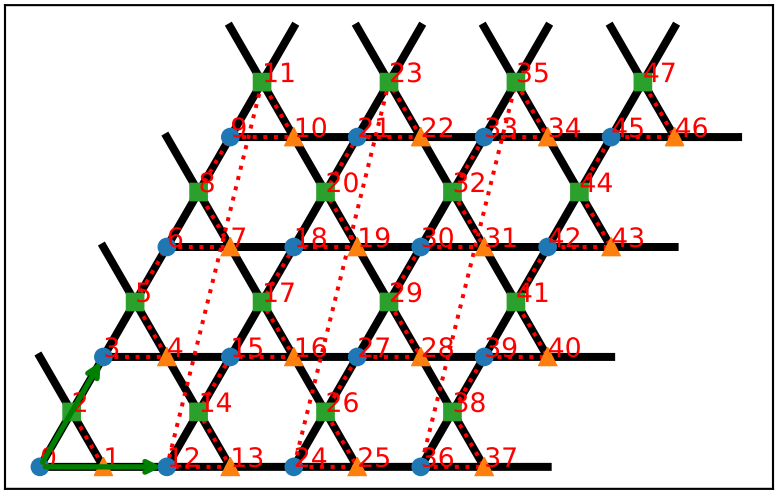
The geometry : lattices

The geometry is usually given by some kind of **lattice** structure how the sites are arranged, e.g. implicitly with the sum over nearest neighbours $\sum_{<i,j>}$. In TeNPy, this is specified by a *Lattice* class, which contains a unit cell of a few *Site* which are shifted periodically by its basis vectors to form a regular lattice. Again, we have pre-defined some basic lattices like a *Chain*, two chains coupled as a *Ladder* or 2D lattices like the *Square*, *Honeycomb* and *Kagome* lattices; but you are also free to define your own generalizations. (More details on that can be found in the doc-string of *Lattice*, read it!)

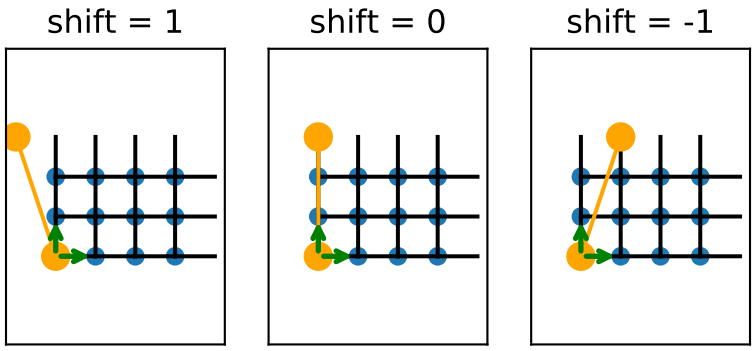
Visualization of the lattice can help a lot to understand which sites are connected by what couplings. The methods `plot_...` of the *Lattice* can do a good job for a quick illustration. We include a small image in the documation of each of the lattices. For example, the following small script can generate the image of the Kagome lattice shown below:

```
import matplotlib.pyplot as plt
from tenpy.models.lattice import Kagome

ax = plt.gca()
lat = Kagome(4, 4, None, bc='periodic')
lat.plot_coupling(ax, lat.nearest_neighbors, linewidth=3.)
lat.plot_order(ax=ax, linestyle=':')
lat.plot_sites()
lat.plot_basis(ax, color='g', linewidth=2.)
ax.set_aspect('equal')
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
plt.show()
```



The lattice contains also the **boundary conditions** *bc* in each direction. It can be one of the usual `'open'` or `'periodic'` in each direcetion. Instead of just saying "periodic", you can also specify a *shift* (except in the first direction). This is easiest to understand at its standard usecase: DMRG on a infinite cylinder. Going around the cylinder, you have a degree of freedom which sites to connect. The orange markers in the following figures illustrates sites identified for a Square lattice with `bc=['periodic', shift]` (see `plot_bc_shift()`):



Note that the "cylinder" axis (and direction for k_x) is perpendicular to the orange line connecting these sites. The line where the cylinder is "cut open" therefore winds around the the cylinder for a non-zero *shift* (or more complicated lattices without perpendicular basis).

MPS based algorithms like DMRG always work on purely 1D systems. Even if our model "lives" on a 2D lattice, these algorithms require to map it onto a 1D chain (probably at the cost of longer-range interactions). This mapping is also done in by the lattice, as it defines an **order** (*`order`*) of the sites. The methods *`mps2lat_idx()`* and *`lat2mps_idx()`* map indices of the MPS to and from indices of the lattice. If you obtained and array with expectation values for a given MPS, you can use *`mps2lat_values()`* to map it to lattice indices, thereby reverting the ordering.

Performing this mapping of the Hamiltonain from a 2D lattice to a 1D chain by hand can be a tideous process.

Therefore, we have automated this mapping in TeNPy as explained in the next section. (Nevertheless it's a good exercise you should do at least once in your life to understand how it works!)

Note: A suitable order is critical for the efficiency of MPS-based algorithms. On one hand, different orderings can lead to different MPO bond-dimensions, with direct impact on the complexity scaling. On the other hand, it influences how much entanglement needs to go through each bonds of the underlying MPS, e.g., the ground strate to be found in DMRG, and therefore influences the required MPS bond dimensions. For the latter reason, the "optimal" ordering can not be known a priori and might even depend on your coupling parameters (and the phase you are in). In the end, you can just try different orderings and see which one works best.

Implementing you own model

When you want to simulate a model not provided in *models*, you need to implement your own model class, lets call it `MyNewModel`. The idea is that you define a new subclass of one or multiple of the model base classes. For example, when you plan to do DMRG, you have to provide an MPO in a `MPOModel`, so your model class should look like this:

```
class MyNewModel(MPOModel):
    """General strucutre for a model suitable for DMRG.

    Here is a good place to document the represented Hamiltonian and parameters.

    In the models of TeNPy, we usually take a single dictionary `model_params`
    containing all parameters, and read values out with ``tenpy.tools.params.get_
↪parameter(...)``,
    The model needs to provide default values if the parameters was not specified.
    """
    def __init__(self, model_params):
        # some code here to read out model parameters and generate H_MPO
        lattice = somehow_generate_lattice(model_params)
        H_MPO = somehow_generate_MPO(lattice, model_params)
        # initialize MPOModel
        MPOModel.__init__(self, lattice, H_MPO)
```

TEBD requires another representation of H in terms of bond terms *H_bond* given to a `NearestNeighborModel`, so in this case it would look so like this instead:

```
class MyNewModel2(NearestNeighborModel):
    """General strucutre for a model suitable for TEBD."""
    def __init__(self, model_params):
        # some code here to read out model parameters and generate H_bond
        lattice = somehow_generate_lattice(model_params)
        H_bond = somehow_generate_H_bond(lattice, model_params)
        # initialize MPOModel
        NearestNeighborModel.__init__(self, lattice, H_bond)
```

Of course, the difficult part in these examples is to generate the `H_MPO` and `H_bond`. Moreover, it's quite annoying to write every model multiple times, just because we need different representations of the same Hamiltonian. Luckily, there is a way out in TeNPy: the CouplingModel!

The easy way to new models: the (Multi)CouplingModel

The *CouplingModel* provides a general, quite abstract way to specify a Hamiltonian of two-site couplings on a given lattice. Once initialized, its methods `add_onsite()` and *add_coupling()* allow to add onsite and coupling terms repeated over the different unit cells of the lattice. In that way, it basically allows a straight-forward translation of the Hamiltonian given as a math forumla $H = \sum_i A_i B_{i+dx} + ...$ with onsite operators *A*, *B*,... into a model class.

The general structure for a new model based on the *CouplingModel* is then:

```
class MyNewModel3(CouplingModel,MPOModel,NearestNeighborModel):
    def __init__(self, ...):
        ...  # follow the basic steps explained below
```

In the initialization method `__init__(self, ...)` of this class you can then follow these basic steps:

0. Read out the parameters.
1. Given the parameters, determine the charges to be conserved. Initialize the *LegCharge* of the local sites accordingly.
2. Define (additional) local operators needed.
3. Initialize the needed *Site*.

   Note: Using pre-defined sites like the *SpinHalfSite* is recommended and can replace steps 1-3.

4. Initialize the lattice (or if you got the lattice as a parameter, set the sites in the unit cell).
5. Initialize the *CouplingModel* with `CouplingModel.__init__(self, lat)`.
6. Use `add_onsite()` and *add_coupling()* to add all terms of the Hamiltonian. Here, the `nearest_neighbors` of the lattice (and its friends for next nearest neighbors) can come in handy, for example:

   ```
   self.add_onsite(-np.asarray(h), 0, 'Sz')
   for u1, u2, dx in self.lat.nearest_neighbors:
       self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
   ```

   Note: The method *add_coupling()* adds the coupling only in one direction, i.e. not switching *i* and *j* in a $\sum_{\langle i,j \rangle}$. If you have terms like $c_i^\dagger c_j$ in your Hamiltonian, you *need* to add it in both directions to get a hermitian Hamiltonian! Simply add another line with switched, conjugated operatores, switched (*u1*, *u2*), and negative *dx*, for example when using the *SpinHalfFermionSite*:

   ```
   self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)
   self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.
   # ('Cdd' is h.c. of 'Cd', and 'Cu' is h.c. of 'Cdu'!)
   ```

   See also the other examples in *add_coupling()*.

   Note that the *strength* arguments of these functions can be (numpy) arrays for site-dependent couplings. If you need to add or multipliy some parameters of the model for the *strength* of certain terms, it is recommended use `np.asarray` beforehand – in that way lists will also work fine.

7. Finally, if you derived from the *MPOModel*, you can call *calc_H_MPO()* to build the MPO and use it for the initialization as `MPOModel.__init__(self, lat, self.calc_H_MPO())`.

8. Similarly, if you derived from the *NearestNeighborModel*, you can call *calc_H_MPO()* to initialze it as `NearestNeighborModel.__init__(self, lat, self.calc_H_bond())`. Calling `self.calc_H_bond()` will fail for models which are not nearest-neighbors (with respect to the MPS ordering), so you should only subclass the *NearestNeighborModel* if the lattice is a simple *Chain*.

The *CouplingModel* works for Hamiltonians which are a sum of terms involving at most two sites. The generalization *MultiCouplingModel* can be used for Hamlitonians with coupling terms acting on more than 2 sites at once. Follow the exact same steps in the initialization, and just use the *add_multi_coupling()* instead or in addition to the *add_coupling()*. A prototypical example is the exactly solvable *ToricCode*.

The code of the module *tenpy.models.xxz_chain* is included below as an illustrative example how to implemnet a Model. The implementation of the *XXZChain* directly follows the steps outline above. The *XXZChain2* implements the very same model, but based on the *CouplingMPOModel* explained in the next section.

```
"""Prototypical example of a 1D quantum model: the spin-1/2 XXZ chain.

The XXZ chain is contained in the more general :class:`~tenpy.models.spins.SpinChain`;
↪ the idea of
this module is more to serve as a pedagogical example for a model.
"""
# Copyright 2018-2019 TeNPy Developers, GNU GPLv3

import numpy as np

from .lattice import Site, Chain
from .model import CouplingModel, NearestNeighborModel, MPOModel, CouplingMPOModel
from ..linalg import np_conserved as npc
from ..tools.params import get_parameter, unused_parameters
from ..networks.site import SpinHalfSite  # if you want to use the predefined site

__all__ = ['XXZChain', 'XXZChain2']


class XXZChain(CouplingModel, NearestNeighborModel, MPOModel):
    r"""Spin-1/2 XXZ chain with Sz conservation.

    The Hamiltonian reads:

    .. math ::
        H = \sum_i \mathtt{Jxx}/2 (S^{+}_i S^{-}_{i+1} + S^{-}_i S^{+}_{i+1})
                 + \mathtt{Jz} S^z_i S^z_{i+1} \\
            - \sum_i \mathtt{hz} S^z_i

    All parameters are collected in a single dictionary `model_params` and read out␣
↪with
    :func:`~tenpy.tools.params.get_parameter`.

    Parameters
    ----------
    L : int
        Length of the chain.
    Jxx, Jz, hz : float | array
        Couplings as defined for the Hamiltonian above.
    bc_MPS : {'finite' | 'infinte'}
        MPS boundary conditions. Coupling boundary conditions are chosen␣
↪appropriately.
    """
    def __init__(self, model_params):
```

(continues on next page)

(continued from previous page)

```
        # 0) read out/set default parameters
        name = "XXZChain"
        L = get_parameter(model_params, 'L', 2, name)
        Jxx = get_parameter(model_params, 'Jxx', 1., name, asarray=True)
        Jz = get_parameter(model_params, 'Jz', 1., name, True)
        hz = get_parameter(model_params, 'hz', 0., name, True)
        bc_MPS = get_parameter(model_params, 'bc_MPS', 'finite', name)
        unused_parameters(model_params, name)  # checks for mistyped parameters
        # 1-3):
        USE_PREDEFINED_SITE = False
        if not USE_PREDEFINED_SITE:
            # 1) charges of the physical leg. The only time that we actually define␣
↪charges!
            leg = npc.LegCharge.from_qflat(npc.ChargeInfo([1], ['2*Sz']), [1, -1])
            # 2) onsite operators
            Sp = [[0., 1.], [0., 0.]]
            Sm = [[0., 0.], [1., 0.]]
            Sz = [[0.5, 0.], [0., -0.5]]
            # (Can't define Sx and Sy as onsite operators: they are incompatible with␣
↪Sz charges.)
            # 3) local physical site
            site = Site(leg, ['up', 'down'], Sp=Sp, Sm=Sm, Sz=Sz)
        else:
            # there is a site for spin-1/2 defined in TeNPy, so just we can just use␣
↪it
            # replacing steps 1-3)
            site = SpinHalfSite(conserve='Sz')
        # 4) lattice
        bc = 'periodic' if bc_MPS == 'infinite' else 'open'
        lat = Chain(L, site, bc=bc, bc_MPS=bc_MPS)
        # 5) initialize CouplingModel
        CouplingModel.__init__(self, lat)
        # 6) add terms of the Hamiltonian
        # (u is always 0 as we have only one site in the unit cell)
        self.add_onsite(-hz, 0, 'Sz')
        self.add_coupling(Jxx * 0.5, 0, 'Sp', 0, 'Sm', 1)
        self.add_coupling(np.conj(Jxx * 0.5), 0, 'Sp', 0, 'Sm', -1)  # h.c.
        self.add_coupling(Jz, 0, 'Sz', 0, 'Sz', 1)
        # 7) initialize H_MPO
        MPOModel.__init__(self, lat, self.calc_H_MPO())
        # 8) initialize H_bond (the order of 7/8 doesn't matter)
        NearestNeighborModel.__init__(self, lat, self.calc_H_bond())


class XXZChain2(CouplingMPOModel, NearestNeighborModel):
    """Another implementation of the Spin-1/2 XXZ chain with Sz conservation.

    This implementation takes the same parameters as the :class:`XXZChain`, but is␣
↪implemented
    based on the :class:`~tenpy.models.model.CouplingMPOModel`.
    """
    def __init__(self, model_params):
        model_params.setdefault('lattice', "Chain")
        CouplingMPOModel.__init__(self, model_params)

    def init_sites(self, model_params):
        return SpinHalfSite(conserve='Sz')  # use predefined Site
```

(continues on next page)

(continued from previous page)

```
    def init_terms(self, model_params):
        # read out parameters
        Jxx = get_parameter(model_params, 'Jxx', 1., self.name, True)
        Jz = get_parameter(model_params, 'Jz', 1., self.name, True)
        hz = get_parameter(model_params, 'hz', 0., self.name, True)
        # add terms
        for u in range(len(self.lat.unit_cell)):
            self.add_onsite(-hz, u, 'Sz')
        for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
            self.add_coupling(Jxx * 0.5, u1, 'Sp', u2, 'Sm', dx)
            self.add_coupling(np.conj(Jxx * 0.5), u2, 'Sp', u1, 'Sm', -dx)  # h.c.
            self.add_coupling(Jz, u1, 'Sz', u2, 'Sz', dx)
```

The easy easy way: the CouplingMPOModel

Since many of the basic steps above are always the same, we don't need to repeat them all the time. So we have yet another class helping to structure the initialization of models: the *CouplingMPOModel*. The general structure of the class is like this:

```
class CouplingMPOModel(CouplingModel,MPOModel):
    def __init__(self, model_param):
        # ... follow the basic steps 1-8 using the methods
        lat = self.init_lattice(self, model_param)  # for step 4
        # ...
        self.init_terms(self, model_param) # for step 6
        # ...

    def init_sites(self, model_param):
        # You should overwrite this

    def init_lattice(self, model_param):
        sites = self.init_sites(self, model_param) # for steps 1-3
        # initialize an arbitrary pre-defined lattice
        # using model_params['lattice']

    def init_terms(self, model_param):
        # does nothing.
        # You should overwrite this
```

The *XXZChain2* included above illustrates, how it can be used. You need to implement steps 1-3) by overwriting the method *init_sites()* Step 4) is performed in the method *init_lattice()*, which initializes arbitrary 1D or 2D lattices; by default a simple 1D chain. If your model only works for specific lattices, you can overwrite this method in your own class. Step 6) should be done by overwriting the method *init_terms()*. Steps 5,7,8 and calls to the *init_...* methods for the other steps are done automatically if you just call the `CouplingMPOModel.__init__(self, model_param)`.

The *XXZChain* and *XXZChain2* work only with the *Chain* as lattice, since they are derived from the *NearestNeighborModel*. This allows to use them for TEBD in 1D (yeah!), but we can't get the MPO for DMRG on a e.g. a *Square* lattice cylinder - although it's intuitively clear, what the hamiltonian there should be: just put the nearest-neighbor coupling on each bond of the 2D lattice.

It's not possible to generalize a *NearestNeighborModel* to an arbitrary lattice where it's no longer nearest Neighbors in the MPS sense, but we can go the other way around: first write the model on an arbitrary 2D lattice and then restrict it to a 1D chain to make it a *NearestNeighborModel*.

Let me illustrate this with another standard example model: the transverse field Ising model, imlemented in the module *tenpy.models.tf_ising* included below. The *TFIModel* works for arbitrary 1D or 2D lattices. The *TFIChain* is then taking the exact same model making a *NearestNeighborModel*, which only works for the 1D chain.

```
"""Prototypical example of a quantum model: the transverse field Ising model.

Like the :class:`~tenpy.models.xxz_chain.XXZChain`, the transverse field ising chain
:class:`TFIChain` is contained in the more general :class:`~tenpy.models.spins.
↪SpinChain`;
the idea is more to serve as a pedagogical example for a 'model'.

We choose the field along z to allow to conserve the parity, if desired.
"""
# Copyright 2018-2019 TeNPy Developers, GNU GPLv3

from .model import CouplingMPOModel, NearestNeighborModel
from ..tools.params import get_parameter
from ..networks.site import SpinHalfSite

__all__ = ['TFIModel', 'TFIChain']


class TFIModel(CouplingMPOModel):
    r"""Transverse field Ising model on a general lattice.

    The Hamiltonian reads:

    .. math ::
        H = - \sum_{\langle i,j\rangle, i < j} \mathtt{J} \sigma^x_i \sigma^x_{j}
            - \sum_{i} \mathtt{g} \sigma^z_i

    Here, :math:`\langle i,j \rangle, i< j` denotes nearest neighbor pairs, each pair␣
↪appearing
    exactly once.
    All parameters are collected in a single dictionary `model_params` and read out␣
↪with
    :func:`~tenpy.tools.params.get_parameter`.

    Parameters
    ----------
    conserve : None | 'parity'
        What should be conserved. See :class:`~tenpy.networks.Site.SpinHalfSite`.
    J, g : float | array
        Couplings as defined for the Hamiltonian above.
    lattice : str | :class:`~tenpy.models.lattice.Lattice`
        Instance of a lattice class for the underlaying geometry.
        Alternatively a string being the name of one of the Lattices defined in
        :mod:`~tenpy.models.lattice`, e.g. ``"Chain", "Square", "HoneyComb", ...``.
    bc_MPS : {'finite' | 'infinte'}
        MPS boundary conditions along the x-direction.
        For 'infinite' boundary conditions, repeat the unit cell in x-direction.
        Coupling boundary conditions in x-direction are chosen accordingly.
        Only used if `lattice` is a string.
    order : string
        Ordering of the sites in the MPS, e.g. 'default', 'snake';
        see :meth:`~tenpy.models.lattice.Lattice.ordering`.
        Only used if `lattice` is a string.
```

(continues on next page)

(continued from previous page)

```
    L : int
        Lenght of the lattice.
        Only used if `lattice` is the name of a 1D Lattice.
    Lx, Ly : int
        Length of the lattice in x- and y-direction.
        Only used if `lattice` is the name of a 2D Lattice.
    bc_y : 'ladder' | 'cylinder'
        Boundary conditions in y-direction.
        Only used if `lattice` is the name of a 2D Lattice.
    """
    def __init__(self, model_params):
        CouplingMPOModel.__init__(self, model_params)

    def init_sites(self, model_params):
        conserve = get_parameter(model_params, 'conserve', 'parity', self.name)
        assert conserve != 'Sz'
        if conserve == 'best':
            conserve = 'parity'
            if self.verbose >= 1.:
                print(self.name + ": set conserve to", conserve)
        site = SpinHalfSite(conserve=conserve)
        return site

    def init_terms(self, model_params):
        J = get_parameter(model_params, 'J', 1., self.name, True)
        g = get_parameter(model_params, 'g', 1., self.name, True)
        for u in range(len(self.lat.unit_cell)):
            self.add_onsite(-g, u, 'Sigmaz')
        for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
            self.add_coupling(-J, u1, 'Sigmax', u2, 'Sigmax', dx)
        # done


class TFIChain(TFIModel, NearestNeighborModel):
    """The :class:`TFIModel` on a Chain, suitable for TEBD.

    See the :class:`TFIModel` for the documentation of parameters.
    """
    def __init__(self, model_params):
        model_params.setdefault('lattice', "Chain")
        CouplingMPOModel.__init__(self, model_params)
```

Some final remarks

- Needless to say that we have also various predefined models under *tenpy.models*.
- Of course, an MPO is all you need to initialize a *MPOModel* to be used for DMRG; you don't have to use the *CouplingModel* or *CouplingMPOModel*. For example an exponentially decaying long-range interactions are not supported by the coupling model but straight-forward to include to an MPO, as demonstrated in the example `examples/mpo_exponentially_decaying.py`.
- If the model of your interest contains Fermions, you should read the *Fermions and the Jordan-Wigner transformation*.
- We suggest writing the model to take a single parameter dicitionary for the initialization, which is to be read out inside the class with *get_parameter()*. Read the doc-string of this function for more details on why this

is a good idea. The `CouplingMPOModel.__init__(...)` calls *`unused_parameters()`*, helping to avoid typos in the specified parameters.

- When you write a model and want to include a test that it can be at least constructed, take a look at `tests/test_model.py`.

Fermions and the Jordan-Wigner transformation

The Jordan-Wigner tranformation maps fermionic creation- and annihilation operators to (bosonic) spin-operators.

Spinless fermions in 1D

Let's start by explicitly writing down the transformation. With the Pauli matrices $\sigma_j^{x,y,z}$ and $\sigma_j^{\pm} = (\sigma_j^x \pm \mathrm{i}\sigma_j^y)/2$ on each site, we can map

$$\begin{aligned} n_j &\leftrightarrow (\sigma_j^z + 1)/2 \\ c_j &\leftrightarrow (-1)^{\sum_{l<j} n_l} \sigma_j^- \\ c_j^\dagger &\leftrightarrow (-1)^{\sum_{l<j} n_l} \sigma_j^+ \end{aligned}$$

The n_l in the second and third row are defined in terms of Pauli matrices according to the first row. We do not interpret the Pauli matrices as spin-1/2; they have nothing to do with the spin in the spin-full case. If you really want to interpret them physically, you might better think of them as hard-core bosons ($b_j = \sigma_j^-, b_j^\dagger = \sigma_j^+$), with a spin of the fermions mapping to a spin of the hard-core bosons.

Note that this transformation maps the fermionic operators c_j and $c_j^\dagger$ to *global* operators; although they carry an index *j* indicating a site, they actually act on all sites `l <= j`! Thus, clearly the operators `C` and `Cd` defined in the *`FermionSite`* do *not* directly correspond to c_j and $c_j^\dagger$. The part $(-1)^{\sum_{l<j} n_l}$ is called Jordan-Wigner string and in the *`FermionSite`* is given by the local operator $JW := (-1)^{n_l}$ acting all sites `l < j`. Since this important, let me stress it again:

Warning: The fermionic operator c_j (and similar $c_j^\dagger$) maps to a *global* operator consisting of the Jordan-Wigner string built by the local operator `JW` on sites `l < j` *and* the local operator `C` (or `Cd`, respectively) on site `j`.

On the sites itself, the onsite operators `C` and `Cd` in the *`FermionSite`* fulfill the correct anti-commutation relation, without the need to include `JW` strings. The `JW` string is necessary to ensure the anti-commutation for operators acting on different sites.

Written in terms of *onsite* operators defined in the *`FermionSite`*, with the *i*-th entry entry in the list acting on site *i*, the relations are thus:

```
["JW", ..., "JW", "C",  "Id", ..., "Id"]   # for the annihilation operator
["JW", ..., "JW", "Cd", "Id", ..., "Id"]   # for the creation operator
```

Note that `"JW"` squares to the identity, `"JW JW" == "Id"`, which is the reason that the Jordan-wigner string completely cancels in $n_j = c_j^\dagger c_j$. In the above notation, this can be written as:

```
["JW", ..., "JW", "Cd",  "Id", ..., "Id"] * ["JW", ..., "JW", "C", "Id", ..., "Id"]
== ["JW JW", ..., "JW JW", "Cd C",  "Id Id", ..., "Id Id"]      # by definition of␣
↪the tensorproduct
== ["Id",    ..., "Id",    "N",     "Id",    ..., "Id"]         # by definition of␣
↪the local operators
# ("X Y" stands for the local operators X and Y applied on the same site. We assume␣
↪that the "Cd" and "C" on the first line act on the same site.)
```

For a pair of operators acting on different sites, `JW` strings have to be included for every site between the operators. For example, taking `i < j`, $c_i^\dagger c_j \leftrightarrow \sigma_i^+ (-1)^{\sum_{i<=l<j} n_l} \sigma_j^-$. More explicitly, for `j = i+2` we get:

```
["JW", ..., "JW", "Cd", "Id", "Id", "Id", ..., "Id"] * ["JW", ..., "JW", "JW", "JW",
↪"C", "Id", ..., "Id"]
== ["JW JW", ..., "JW JW", "Cd JW",  "Id JW", "Id C", ..., "Id"]
== ["Id",    ..., "Id",    "Cd JW",  "JW",    "C",    ..., "Id"]
```

In other words, the Jordan-Wigner string appears only in the range `i <= l < j`, i.e. between the two sites *and* on the smaller/left one of them. (You can easily generalize this rule to cases with more than two c or $c^\dagger$.)

This last line (as well as the last line of the previous example) can be rewritten by changing the order of the operators `Cd JW` to `"JW Cd" == - "Cd"`. (This is valid because either site `i` is occupied, yielding a minus sign from the `JW`, or it is empty, yielding a 0 from the `Cd`.)

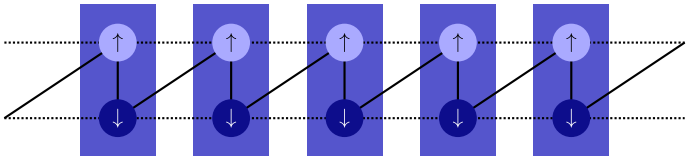
This is also the case for `j < i`, say `j = i-2`: $c_i^\dagger c_j \leftrightarrow (-1)^{\sum_{j<=l<i} n_l} \sigma_i^+ \sigma_j^-$. As shown in the following, the `JW` again appears on the left site, but this time acting *after* `C`:

```
["JW", ..., "JW", "JW", "JW", "Cd", "Id", ..., "Id"] * ["JW", ..., "JW", "C", "Id",
↪"Id", "Id", ..., "Id"]
== ["JW JW", ..., "JW JW", "JW C",  "JW", "Cd Id", ..., "Id"]
== ["Id",    ..., "Id",    "JW C",  "JW", "Cd",    ..., "Id"]
```

Higher dimensions

For an MPO or MPS, you always have to define an ordering of all your sites. This ordering effectifely maps the higher-dimensional lattice to a 1D chain, usually at the expence of long-range hopping/interactions. With this mapping, the Jordan-Wigner transformation generalizes to higher dimensions in a straight-forward way.

Spinful fermions



As illustrated in the above picture, you can think of spin-1/2 fermions on a chain as spinless fermions living on a ladder (and analogous mappings for higher dimensional lattices). Each rung (a blue box in the picture) forms a *SpinHalfFermionSite* which is composed of two *FermionSite* (the circles in the picture) for spin-up and spin-down. The mapping of the spin-1/2 fermions onto the ladder induces an ordering of the spins, as the final result must again be a one-dimensional chain, now containing both spin species. The solid line indicates the convention for the ordering, the dashed lines indicate spin-preserving hopping $c_{s,i}^\dagger c_{s,i+1} + h.c.$ and visualize the ladder structure. More generally, each species of fermions appearing in your model gets a separate label, and its Jordan-Wigner string includes the signs $(-1)^{n_l}$ of *all* species of fermions to the 'left' of it (in the sense of the ordering indicated by the solid line in the picture).

In the case of spin-1/2 fermions labeled by $\uparrow$ and $\downarrow$ on each *site*, the complete mapping is given (where j and l are

indices of the *FermionSite*):

$$
\begin{aligned}
n_{\uparrow,j} &\leftrightarrow (\sigma^z_{\uparrow,j}+1)/2 \\
n_{\downarrow,j} &\leftrightarrow (\sigma^z_{\downarrow,j}+1)/2 \\
c_{\uparrow,j} &\leftrightarrow (-1)^{\sum_{l<j} n_{\uparrow,l}+n_{\downarrow,l}} \sigma^-_{\uparrow,j} \\
c^\dagger_{\uparrow,j} &\leftrightarrow (-1)^{\sum_{l<j} n_{\uparrow,l}+n_{\downarrow,l}} \sigma^+_{\uparrow,j} \\
c_{\downarrow,j} &\leftrightarrow (-1)^{\sum_{l<j} n_{\uparrow,l}+n_{\downarrow,l}} (-1)^{n_{\uparrow,j}} \sigma^-_{\downarrow,j} \\
c^\dagger_{\downarrow,j} &\leftrightarrow (-1)^{\sum_{l<j} n_{\uparrow,l}+n_{\downarrow,l}} (-1)^{n_{\uparrow,j}} \sigma^+_{\downarrow,j}
\end{aligned}
$$

In each of the above mappings the operators on the right hand sides commute; we can rewrite $(-1)^{\sum_{l<j} n_{\uparrow,l}+n_{\downarrow,l}} = \prod_{l<j}(-1)^{n_{\uparrow,l}}(-1)^{n_{\downarrow,l}}$, which resembles the actual structure in the code more closely. The parts of the operator acting in the same box of the picture, i.e. which have the same index *j* or *l*, are the 'onsite' operators in the *SpinHalfFermionSite*: for example `JW` on site *j* is given by $(-1)^{n_{\uparrow,j}}(-1)^{n_{\downarrow,j}}$, `Cu` is just the $\sigma^-_{\uparrow,j}$, `Cdu` is $\sigma^+_{\uparrow,j}$, `Cd` is $(-1)^{n_{\uparrow,j}}\sigma^-_{\downarrow,j}$. and `Cdd` is $(-1)^{n_{\uparrow,j}}\sigma^+_{\downarrow,j}$. Note the asymmetry regarding the spin in the definition of the onsite operators: the spin-down operators include Jordan-Wigner signs for the spin-up fermions on the same site. This asymetry stems from the ordering convention introduced by the solid line in the picture, according to which the spin-up site is "left" of the spin-down site. With the above definition, the operators within the same *SpinHalfFermionSite* fulfill the expected commutation relations, for example `"Cu Cdd" == - "Cdd Cu"`, but again the `JW` on sites left of the operator pair is crucial to get the correct commutation relations globally.

Warning: Again, the fermionic operators $c_{\downarrow,j}, c^\dagger_{\downarrow,j}, c_{\downarrow,j}, c^\dagger_{\downarrow,j}$ correspond to *global* operators consisting of the Jordan-Wigner string built by the local operator `JW` on sites `l < j` *and* the local operators `'Cu', 'Cdu', 'Cd', 'Cdd'` on site `j`.

Written explicitly in terms of onsite operators defined in the `FermionSite`, with the *j*-th entry entry in the list acting on site *j*, the relations are:

```
["JW", ..., "JW", "Cu",  "Id", ..., "Id"]    # for the annihilation operator spin-up
["JW", ..., "JW", "Cd",  "Id", ..., "Id"]    # for the annihilation operator spin-down
["JW", ..., "JW", "Cdu",  "Id", ..., "Id"]   # for the creation operator spin-up
["JW", ..., "JW", "Cdd",  "Id", ..., "Id"]   # for the creation operator spin-down
```

As you can see, the asymmetry regaring the spins in the definition of the local onsite operators `"Cu", "Cd", "Cdu", "Cdd"` lead to a symmetric definition in the global sense. If you look at the definitions very closely, you can see that in terms like `["Id", "Cd JW", "JW", "Cd"]` the Jordan-Wigner sign $(-1)^{n_{\uparrow,2}}$ appears twice (namely once in the definition of `"Cd"` and once in the `"JW"` on site 2) and could in principle be canceled, however in favor of a simplified handling in the code we do not recommend you to cancel it. Similar, within a spinless *FermionSite*, one can simplify `"Cd JW" == "Cd"` and `"JW C" == "C"`, but these relations do *not* hold in the *SpinHalfSite*, and for consistency we recommend to explicitly keep the `"JW"` operator string even in nearest-neighbor models where it is not strictly necessary.

How to handle Jordan-Wigner strings in practice

There are only a few pitfalls where you have to keep the mapping in mind: When **building a model**, you map the physical fermionic operators to the usual spin/bosonic operators. The algorithms don't care about the mapping, they just use the given Hamiltonian, be it given as MPO for DMRG or as nearest neighbor couplings for TEBD. Only when you do a **measurement** (e.g. by calculating an expectation value or a correlation function), you have to reverse this mapping. Be aware that in certain cases, e.g. when calculating the entanglement entropy on a certain bond, you cannot reverse this mapping (in a straightforward way), and thus your results might depend on how you defined the Jordan-Wigner string.

Whatever you do, you should first think about if (and how much of) the Jordan-Wigner string cancels. For example for many of the onsite operators (like the particle number operator `N` or the spin operators in the *SpinHalfFermionSite*) the Jordan-Wigner string cancels completely and you can just ignore it both in onsite-terms and couplings. To check, whether the Jordan-Wigner string cancels for a given operator, take a look at `need_JW_string` and *op_needs_JW()*. In case of operators acting on different sites, you typically have a Jordan-Wigner string inbetween (e.g. for the $c_i^\dagger c_j$ examples described above and below) or no Jordan-Wigner strings at all (e.g. for density-density interactions $n_i n_j$). In fact, the case that the Jordan Wigner string on the left of the first non-trivial operator does not cancel is currently not supported for models and expectation values, as it usually doesn't appear in practice.

When **building a model** with the *CouplingModel*, *onsite* terms for which the Jordan-Wigner string cancels can be added directly. Care has to be taken when adding *couplings* with *add_coupling()*. When you need a Jordan-Wigner string inbetween the operators, set the optional arguments `op_string='JW', str_on_first=True`. Then, the function automatically takes care of the Jordan-Wigner string in the correct way, adding it on the left operator. With the default arguments, it is checked automatically whether the model

Obviously, you should be careful about the convention which of the two coupling terms is applied first (in a physical sense as an operator acting on a state), as this corresponds to a sign. We follow the convention that the operator given as argument *op2* is applied first, independent of wheter it ends up left or right in the MPS ordering sense.

As a concrete example, let us specify a hopping $\sum_{\langle i,j \rangle} (c_i^\dagger c_j + h.c.) = \sum_{\langle i,j \rangle} (c_i^\dagger c_j + c_j^\dagger c_i)$ in a 1D chain of *FermionSite* with *add_coupling()*:

```
add_coupling(strength, 0, 'Cd', 0, 'C', 1, 'JW', True)
add_coupling(strength, 0, 'Cd', 0, 'C', -1, 'JW', True)
# (without the last 2 arguments, add_coupling checks for necessary JW strings␣
↪automatically)
```

Slightly more complicated, to specify the hopping $\sum_{\langle i,j \rangle, s} (c_{s,i}^\dagger c_{s,j} + h.c.)$ in the Fermi-Hubbard model on a 2D square lattice, we would need more terms:

```
for (dx, dy) in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
    add_coupling(strength, 0, 'Cdu', 0, 'Cu', (dx, dy), 'JW', True)
    add_coupling(strength, 0, 'Cdd', 0, 'Cd', (dx, dy), 'JW', True)
```

If you want to build a model directly as an MPO or with nearest-neighbor bonds only, *you* have to care about how to handle the Jordan-Wigner string correctly.

The most important functions for doing **measurements** are probably *expectation_value()* and *correlation_function()*. Again, if all the Jordan-Wigner strings cancel, you don't have to worry about them at all, e.g. for many onsite operators or correlation functions involving only number operators. If you measure operators involving multiple sites with *expectation_value*, take care to include the Jordan-Wigner string correctly while building these operators.

The *correlation_function()* supports a Jordan-Wigner string in between the two operators to be measured. As for *add_coupling()*, you should set the optional arguments `op_string='JW', str_on_first=True` in that case. Functions like *expectation_value_term()* also care about the Jordan Wigner string (if specified in the documentation).

Contributing

The code is maintained in a git repository, the official repository is on github. You're welcome to contribute and submit pull requests on github. If you're unsure how or what to do, you can ask for help in the community forum. If you want to become a member of the developer team, just ask ;-)

To keep consistency, we ask you to comply with the following guidelines for contributions:

- Use a code style based on **PEP 8**. The git repo includes a config file `.style.yapf` for the python package yapf. *yapf* is a tool to auto-format code, e.g., by the command `yapf -i some/file` (-i for "in place"). We run yapf on a regular basis on the github master branch. If your branch diverged, it might help to run yapf before merging.

Note: Since no tool is perfect, you can format some regions of code manually and enclose them with the special comments `# yapf: disable` and `# yapf: enable`.

- Every function/class/module should be documented by its doc-string (c.f. **PEP 257**), additional documentation is in `doc/`. The documentation uses *reStructuredText*. If you're new to *reStructuredText*, read this introduction. We use the *numpydoc* extension to sphinx, so please read and follow these Instructions for the doc strings. In addition, you can take a look at the following example file. Helpful hints on top of that:

```
r"""<- this r makes me a raw string, thus '\' has no special meaning.
Otherwise you would need to escape backslashes, e.g. in math formulas.

You can include cross references to classes, methods, functions, modules like
:class:`~tenpy.linalg.np_conserved.Array`, :meth:`~tenpy.linalg.np_conserved.
↪Array.to_ndarray`,
:func:`tenpy.tools.math.toiterable`, :mod:`tenpy.linalg.np_conserved`.
The ~ in the beginning makes only the last part of the name appear in the␣
↪generated documentation.
Documents of the userguide can be referenced with :doc:`/intro_npc` even from␣
↪inside the doc-strings.
You can also cross-link to other documentations, e.g. :class:`numpy.ndarray`,␣
↪:func`scipy.linalg.svd` and :mod: will work.

Moreover, you can link to github issues, arXiv papers, dois, and topics in the␣
↪community forum with
e.g. :issue:`5`, :arxiv:`1805.00055`, :doi:`10.1000/1` and :forum:`3`.


Write inline formulas as :math:`H |\Psi\rangle = E |\Psi\rangle` or displayed␣
↪equations as
.. math ::

   e^{i\pi} + 1 = 0

In doc-strings, math can only be used in the Notes section.
To refer to variables within math, use `\mathtt{varname}`.

.. todo ::

   This block can describe things which need to be done and is automatically␣
↪included in a section of :doc:`todo`.
"""
```

- Use relative imports within TeNPy. Example:

```
from ..linalg import np_conserved as npc
```

- Use the python package pytest for testing. Run it simply with `pytest` in *tests/*. You should make sure that all tests run through, before you `git push` back into the public repo. Long-running tests are marked with the attribute *slow*; for a quick check you can also run `pytest -m "not slow"`.
- Reversely, if you write new functions, please also include suitable tests!
- During development, you might introduce `# TODO` comments. But also try to remove them again later! If you're not 100% sure that you will remove it soon, please add a doc-string with a `.. todo ::` block, such that we can keep track of it as explained in the previous point.

  Unfinished functions should `raise NotImplementedError()`.
- if you want to try out new things in temporary files: any folder named `playground` is ignored by *git*.

Thank You for helping with the development!

Bulding the documentation

You can use Sphinx to generate the full documentation in various formats (including HTML or PDF) yourself, as described in the following. First, install Sphinx and the extension numpydoc with:

```
pip install --upgrade sphinx numpydoc
```

Afterwards, simply go to the folder *doc/* and run the following command:

```
make html
```

This should generate the html documentation in the folder *doc/sphinx_build/html*. Open this folder (or to be precise: the file *index.html* in it) in your webbroser and enjoy this and other documentation beautifully rendered, with cross links, math formulas and even a search function. Other output formats are available as other make targets, e.g., `make latexpdf`.

Note: Building the documentation with sphinx requires loading the modules. Thus make sure that the folder *tenpy* is included in your `$PYTHONPATH`, as described in *doc/INSTALL.rst*.

To-Do list

Primary goals for the coming release

- finish documentation and tests on existing stuff

Concrete things to be fixed in different files

- The MPO class has no function for expectation value with MPS
- Since we switched to python 3 completely, there's no need to subclass 'object' anymore.
- npc.Array: comparison with ==, pickle, hashable?
- MPS class: group_sites, split_sites, pad
- MPS class: probability_per_charge, charge_variance
- MPS class: string correlation function

To be done at some point for the next releases

- remove this file: use GitHub issues instead
- overview and usage introduction to the overall library
- trace: allow multiple axes to be traced over; optimize
- Summary of defined classes/functions at the beginning of a module in the reference
- Inconsistency: *NearestNeighborModel.H_bond* with `bc_MPS='infinite'` has bonds `[(L, 0), (0, 1), ...]`, but *expectation_value()* takes two-site operators on bonds `[(0, 1), (1, 2), ...(L,0)]`.

Wish-list

- logging mechanism?
- Johannes Motruk: extend simulation class: save standard variables like entropy, energy, etc?
- Ruben: extend MPS TransferMatrix class
- Jakob: function for Arrays: Perfrom trace over multiple pairs of legs at once. Tracing one after the other calculates unnecessary "off-diagonal" elements.

Auto-generated To-Do list

The following list is auto-generated by sphinx, extracting `.. todo ::` blocks from doc-strings of the code.

Todo: Write UserGuide!!!

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/algorithms/dmrg.py:do
of tenpy.algorithms.dmrg, line 30.)

Todo: Rebuild TDVP engine as subclasses of sweep Do testing

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/algorithms/mps_sweep
of tenpy.algorithms.mps_sweeps, line 18.)

Todo:

- **implement or wrap netcon.m, a function to find optimal contractionn sequences** (arXiv:1304.6112)
- improve helpfulness of Warnings
- _do_trace: trace over all pairs of legs at once. need the corresponding npc function first.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/algorithms/network_co of tenpy.algorithms.network_contractor, line 8.)

Todo: This is still a beta version, use with care. The interface might still change.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/algorithms/tdvp.py:doc of tenpy.algorithms.tdvp, line 12.)

Todo: long-term: Much of the code is similar as in DMRG. To avoid too much duplicated code, we should have a general way to sweep through an MPS and updated one or two sites, used in both cases.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/algorithms/tdvp.py:doc of tenpy.algorithms.tdvp, line 16.)

Todo: -add further terms (e.g. c^dagger c^dagger + h.c.) to the Hamiltonian.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/fermions_spinl of tenpy.models.fermions_spinless, line 3.)

Todo: WARNING: These models are still under development and not yet tested for correctness. Use at your own risk! Replicate known results to confirm models work correctly. Long term: implement different lattices. Long term: implement variable hopping strengths Jx, Jy.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/hofstadter.py:d of tenpy.models.hofstadter, line 3.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs of tenpy.models.lattice.Honeycomb.mps2lat_values, line 69.)

Todo:

- this doesn't fully work yet...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs of tenpy.models.lattice.IrregularLattice, line 4.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs of tenpy.models.lattice.IrregularLattice.mps2lat_values, line 69.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs
of tenpy.models.lattice.Kagome.mps2lat_values, line 69.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs
of tenpy.models.lattice.Ladder.mps2lat_values, line 69.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs
of tenpy.models.lattice.Lattice.mps2lat_values, line 69.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/lattice.py:docs
of tenpy.models.lattice.TrivialLattice.mps2lat_values, line 69.)

Todo: implement MPO for time evolution...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/model.py:docs
of tenpy.models.model.MPOModel, line 8.)

Todo: make sure this function is used for expectation values...

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/models/toric_code.py:
of tenpy.models.toric_code.DualSquare.mps2lat_values, line 69.)

Todo: might be useful to add a "cleanup" function which removes operators cancelling each other and/or unused states. Or better use a 'compress' of the MPO?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/networks/mpo.py:docs
of tenpy.networks.mpo.MPOGraph, line 18.)

Todo: Make more general: it should be possible to specify states as strings.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/networks/mps.py:docs
of tenpy.networks.mps.build_initial_state, line 14.)

Todo: One can also look at the canonical ensembles by defining the conserved quantities differently, see Barthel (2016), arXiv:1607.01696 for details. Idea: usual charges on p, trivial charges on q; fix total charge to desired value. I think it should suffice to implement another *from_infiniteT*.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/networks/purification_
of tenpy.networks.purification_mps, line 104.)

Todo: Check if Jordan-Wigner strings for 4x4 operators are correct.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/tenpy/checkouts/v0.5.0/tenpy/networks/site.py:docst
of tenpy.networks.site.SpinHalfFermionSite, line 62.)

CHANGELOG

All notable changes to the project will be documented in this file. The project adheres semantic versioning

[0.5.0] - 2019-12-18

Backwards incompatible changes

- **Major** rewriting of the DMRG Engines, see issue #39 and issue #85 for details. The *EngineCombine* and *EngineFracture* have been combined into a single *TwoSiteDMRGEngine* with an The *run* function works as before. In case you have directly used the *EngineCombine* or *EngineFracture*, you should update your code and use the `TwoSiteEngine` instead.
- Moved `init_LP` and `init_RP` method from *MPS* into *MPSEnvironment* and *MPOEnvironment*.

Changed

- Addition/subtraction of *Array*: check whether the both arrays have the same labels in differnt order, and in that case raise a warning that we will transpose in the future.
- Made *tenpy.linalg.np_conserved.Array.get_block()* public (previously `tenpy.linalg.np_conserved.Array._get_block`).
- *groundstate()* now returns a tuple `(E0, psi0)` instead of just `psi0`. Moreover, the argument *charge_sector* was added.
- Simplification in the *Lattice*: Instead of having separate arguments/attributes/functions for `'nearest_neighbors'`, `'next_nearest_neighbors'`, `'next_next_nearest_neighbors'` and possibly (Honeycomb) even `'fourth_nearest_neighbors'`, `'fifth_nearest_neighbors'`, collect them in a dictionary called *pairs*. Old call structures still allowed, but deprecated.
- issue #94: Array addition and *inner()* should reflect the order of the labels, if they coincided. Will change the default behaviour in the future, raising *FutureWarning* for now.
- **Default parameter** for DMRG params: increased precision by setting *P_tol_min* down to the maximum of `1.e-30`, `lanczos_params['svd_min']**2 * P_tol_to_trunc`, `lanczos_params['trunc_cut']**2 * P_tol_to_trunc` by default.

Added

- *tenpy.algorithms.mps_sweeps* with the *Sweep* class and *EffectiveH* to be a *OneSiteH* or *TwoSiteH*.
- Single-Site DMRG with the `SingleSiteDMRG`.
- Example function in `examples/c_tebd.py` how to run TEBD with a model originally having next-nearest neighbors.
- *increase_L()* to allow increasing the unit cell of an MPS.
- Additional option `order='folded'` for the *Chain*.
- *tenpy.algorithms.exact_diag.ExactDiag.from_H_mpo()* wrapper as replacement for *tenpy.networks.mpo.MPO.get_full_hamiltonian()* and *tenpy.networks.mpo.MPO.get_grouped_mpo()*. The latter are now deprecated.
- Argument *max_size* to limit the matrix dimension in *ExactDiag*.
- *tenpy.linalg.sparse.FlatLinearOperator.from_guess_with_pipe()* to allow quickly converting *matvec* functions acting on multi-dimensional arrays to a *FlatLinearOperator* by combining the legs into a LegPipe.
- *tenpy.tools.math.speigsh()* for hermitian variant of *speigs()*
- Allow for arguments `'LA', 'SA'` in *argsort()*.
- *tenpy.linalg.lanczos.lanczos_arpack()* as possiple replacement of the self-implemented *lanczos* function.
- *tenpy.algorithms.dmrg.full_diag_effH()* as another replacement of *lanczos()*.
- The new DMRG parameter `'diag_method'` allows to select a method for the diagonalization of the effective Hamiltonian. See *tenpy.algorithms.dmrg.DMRGEngine.diag()* for details.
- dtype attribute in *EffectiveH*.
- *tenpy.linalg.charges.LegCharge.get_qindex_of_charges()* to allow selecting a block of an Array from the charges.
- *tenpy.algorithms.mps_sweeps.EffectiveH.to_matrix* to allow contracting an *EffectiveH* to a matrix, as well as metadata `tenpy.linalg.sparse.NpcLinearOperator.acts_on` and `tenpy.algorithms.mps_sweeps.EffectiveH.N`.
- argument *only_physical_legs* in *tenpy.networks.mps.MPS.get_total_charge()*

Fixed

- MPO *expectation_value()* did not work for finite systems.
- Calling *compute_K()* repeatedly with default parameters but on states with different *chi* would use the *chi* of the very first call for the truncation parameters.
- allow `MPSEnvironment` and `MPOEnvironment` to have MPS/MPO with different length
- *group_sites()* didn't work correctly in some situations.
- *matvec_to_array()* returned the transposed of A.
- *tenpy.networks.mps.MPS.from_full()* messed up the form of the first array.

- issue #95: blowup of errors in DMRG with *update_env* > *0*. Turns out to be a problem in the precision of the truncation error: *TruncationError.eps* was set to 0 if it would be smaller than machine precision. To fix it, I added *from_S()*.

[0.4.1] - 2019-08-14

Backwards incompatible changes

- Switch the sign of the *BoseHubbardModel* and *FermiHubbardModel* to hopping and chemical potential having negative prefactors. Of course, the same adjustment happens in the *BoseHubbardChain* and *FermiHubbardChain*.
- moved *BoseHubbardModel* and *BoseHubbardChain* as well as *FermiHubbardModel* and *FermiHubbardChain* into the new module *tenpy.models.hubbard*.
- Change arguments of *coupling_term_handle_JW()* and *multi_coupling_term_handle_JW()* to use *strength* and *sites* instead of *op_needs_JW*.
- Only accept valid identifiers as operator names in *add_op()*.

Changed

- *grid_concat()* allows for `None` entries (representing zero blocks).
- *from_full()* allows for 'segment' boundary conditions.
- *apply_local_op()* allows for n-site operators.

Added

- *max_range* attribute in *MPO* and *MPOGraph*.
- *is_hermitian()*
- Nearest-neighbor interaction in `BoseHubbardModel`
- *multiply_op_names()* to replace `' '.join(op_names)` and allow explicit compression/multiplication.
- *order_combine_term()* to group operators together.
- *dagger()* of MPO's (and to implement that also *flip_charges_qconj()*).
- `has_label()` to check if a label exists
- *qr_li()* and *rq_li()*
- Addition of MPOs
- 3 additional examples for chern insulators in `examples/chern_insulators/`.
- *FermionicHaldaneModel* and *BosonicHaldaneModel*.
- *from_MPOModel()* for initializing nearest-neighbor models after grouping sites.

Fixed

- issue #36: long-range couplings could give IndexError.
- issue #42: Onsite-terms in *FermiHubbardModel* were wrong for lattices with non-trivial unit cell.
- Missing a factor 0.5 in *GUE()*.
- Allow *TermList* to have terms with multiple operators acting on the same site.
- Allow MPS indices outside unit cell in *mps2lat_idx()* and *lat2mps_idx()*.
- `expectation_value()` did not work for n-site operators.

[0.4.0] - 2019-04-28

Backwards incompatible changes

- The argument order of *tenpy.models.lattice.Lattice* could be a tuple `(priority, snake_winding)` before. This is no longer valid and needs to be replaced by `("standard", snake_winding, priority)`.
- Moved the boundary conditions *bc_coupling* from the *tenpy.models.model.CouplingModel* into the *tenpy.models.lattice.Lattice* (as *bc*). Using the parameter *bc_coupling* will raise a FutureWarning, one should set the boundary conditions directly in the lattice.
- Added parameter *permute* (True by default) in *tenpy.networks.mps.MPS.from_product_state()* and *tenpy.networks.mps.MPS.from_Bflat()*. The resulting state will therefore be independent of the "conserve" parameter of the Sites - unlike before, where the meaning of the p_state argument might have changed.
- Generalize and rename `tenpy.networks.site.DoubleSite` to *tenpy.networks.site.GroupedSite*, to allow for an arbitrary number of sites to be grouped. Arguments `site0, site1, label0, label1` of the __init__ can be replaced with `[site0, site1], [label0, label1]` and `op0, op1` of the *kronecker_product* with `[op0, op1]`; this will recover the functionality of the *DoubleSite*.
- Restructured callstructure of Mixer in DMRG, allowing an implementation of other mixers. To enable the mixer, set the DMRG parameter `"mixer"` to `True` or `'DensityMatrixMixer'` instead of just `'Mixer'`.
- The interaction parameter in the `tenpy.models.bose_hubbbard_chain.BoseHubbardModel` (and `tenpy.models.bose_hubbbard_chain.BoseHubbardChain`) did not correspond to $U/2N(N-1)$ as claimed in the Hamiltonian, but to UN^2. The correcting factor 1/2 and change in the chemical potential have been fixed.
- Major restructuring of *tenpy.linalg.np_conserved* and *tenpy.linalg.charges*. This should not break backwards-compatibility, but if you compiled the cython files, you **need** to remove the old binaries in the source directory. Using `bash cleanup.sh` might be helpful to do that, but also remove other files within the repository, so be careful and make a backup beforehand to be on the save side. Afterwards recompile with `bash compile.sh`.
- Changed structure of `tenpy.models.model.CouplingModel.onsite_terms` and `tenpy.models.model.CouplingModel.coupling_terms`: Each of them is now a dictionary with category strings as keys and the newly introduced *tenpy.networks.terms.OnsiteTerms* and *tenpy.networks.terms.CouplingTerms* as values.
- *tenpy.models.model.CouplingModel.calc_H_onsite()* is deprecated in favor of new methods.
- Argument *raise_op2_left* of *tenpy.models.model.CouplingModel.add_coupling()* is deprecated.

Added

- *tenpy.networks.mps.MPS.canonical_form_infinite()*.
- *tenpy.networks.mps.MPS.expectation_value_term()*, *tenpy.networks.mps.MPS.expectation_value_terms_sum()* and *tenpy.networks.mps.MPS.expectation_value_multi_sites()* for expectation values of terms.
- *tenpy.networks.mpo.MPO.expectation_value()* for an MPO.
- *tenpy.linalg.np_conserved.Array.extend()* and *tenpy.linalg.charges.LegCharge.extend()*, allowing to extend an Array with zeros.
- DMRG parameter `'orthogonal_to'` allows to calculate excited states for finite systems.
- possibility to change the number of charges after creating LegCharges/Arrays.
- more general way to specify the order of sites in a *tenpy.models.lattice.Lattice*.
- new *tenpy.models.lattice.Triangular*, *tenpy.models.lattice.Honeycomb* and *tenpy.models.lattice.Kagome* lattice
- a way to specify nearest neighbor couplings in a *Lattice*, along with methods to count the number of nearest neighbors for sites in the bulk, and a way to plot them (*plot_coupling()* and friends)
- *tenpy.networks.mpo.MPO.from_grids()* to generate the MPO from a grid.
- *tenpy.models.model.MultiCouplingModel* for couplings involving more than 2 sites.
- request #8: Allow shift in boundary conditions of *CouplingModel*.
- Allow to use state labels in *tenpy.networks.mps.MPS.from_product_state()*.
- *tenpy.models.model.CouplingMPOModel* structuring the default initialization of most models.
- Allow to force periodic boundary conditions for finite MPS in the `CouplingMPOModel`. This is not recommended, though.
- *tenpy.models.model.NearestNeighborModel.calc_H_MPO_from_bond()* and *tenpy.models.model.MPOModel.calc_H_bond_from_MPO()* for conversion of H_bond into H_MPO and vice versa.
- *tenpy.algorithms.tebd.RandomUnitaryEvolution* for random unitary circuits
- Allow documentation links to github issues, arXiv, papers by doi and the forum with e.g. `:issue:`5`, :arxiv:`1805.00055`, :doi:`10.21468/SciPostPhysLectNotes.5`, :forum:`3``
- *tenpy.models.model.CouplingModel.coupling_strength_add_ext_flux()* for adding hoppings with external flux.
- `tenpy.models.model.CouplingModel.plot_coupling_terms()` to visualize the added coupling terms.
- *tenpy.networks.terms.OnsiteTerms*, *tenpy.networks.terms.CouplingTerms*, `tenpy.networks.terms.MultiCouplingTerm` containing the of terms for the *CouplingModel* and *MultiCouplingModel*. This allowed to add the *category* argument to *add_onsite*, *add_coupling* and *add_multi_coupling*.
- *tenpy.networks.terms.TermList* as another (more human readable) representation of terms with conversion from and to the other `*Term` classes.
- `tenpy.networks.mps.MPS.init_LP()` and `tenpy.networks.mps.MPS.init_RP()` to initialize left and right parts of an Environment.

- *tenpy.networks.mpo.MPOGraph.from_terms()* and *tenpy.networks.mpo.MPOGraph.from_term_list()*.
- argument *charge_sector* in *tenpy.networks.mps.MPS.correlation_length()*.

Changed

- moved toycodes from the folder `examples/` to a new folder `toycodes/` to separate them clearly.
- **major remodelling of the internals of *tenpy.linalg.np_conserved* and *tenpy.linalg.charges*.**
  - Introduced the new module `tenpy/linalg/_npc_helper.pyx` which contains all the Cython code, and gets imported by
  - *Array* now rejects addition/subtraction with other types
  - *Array* now rejects multiplication/division with non-scalar types
  - By default, make deep copies of npc Arrays.
- Restructured lanczos into a class, added time evolution calculating `exp(A*dt)|psi0>`
- Warning for poorly conditioned Lanczos; to overcome this enable the new parameter *reortho*.
- Simplified call strucutre of *extend()*, and *extend()*.
- Restructured *tenpy.algorithms.dmrg*:
  - *run()* is now just a wrapper around the new `run()`, `run(psi, model, pars)` is roughly equivalent to `eng = EngineCombine(psi, model, pars); eng.run()`.
  - Added `init_env()` and `reset_stats()` to allow a simple restart of DMRG with slightly different parameters, e.g. for tuning Hamiltonian parameters.
  - Call *canonical_form()* for infinite systems if the final state is not in canonical form.
- Changed **default values** for some parameters:
  - set `trunc_params['chi_max'] = 100`. Not setting a *chi_max* at all will lead to memory problems. Disable `DMRG_params['chi_list'] = None` by default to avoid conflicting settings.
  - reduce to `mixer_params['amplitude'] = 1.e-5`. A too strong mixer screws DMRG up pretty bad.
  - increase `Lanczos_params['N_cache'] = N_max` (i.e., keep all states)
  - set `DMRG_params['P_tol_to_trunc'] = 0.05` and provide reasonable ..._min and ..._max values.
  - increased (default) DMRG accuracy by setting `DMRG_params['max_E_err'] = 1.e-8` and `DMRG_params['max_S_err'] = 1.e-5`.
  - don't check the (absolute) energy for convergence in Lanczos.
  - set `DMRG_params['norm_tol'] = 1.e-5` to check whether the final state is in canonical form.
- Verbosity of *get_parameter()* reduced: Print parameters only for verbosity >=1. and default values only for verbosity >= 2.
- Don't print the energy during real-time TEBD evolution - it's preserved up to truncation errors.
- Renamed the *SquareLattice* class to *tenpy.models.lattice.Square* for better consistency.
- auto-determine whether Jordan-Wigner strings are necessary in *add_coupling()*.

- The way the labels of npc Arrays are stored internally changed to a simple list with None entries. There is a deprecated propery setter yielding a dictionary with the labels.
- renamed *first_LP* and *last_RP* arguments of *MPSEnvironment* and *MPOEnvironment* to *init_LP* and *init_RP*.
- Testing: insetad of the (outdated) nose, we now use *pytest <https://pytest.org>* for testing.

Fixed

- issue #22: **Serious bug** in *tenpy.linalg.np_conserved.inner()*: if `do_conj=True` is used with non-zero `qtotal`, it returned 0. instead of non-zero values.
- avoid error in *tenpy.networks.mps.MPS.apply_local_op()*
- Don't carry around total charge when using DMRG with a mixer
- Corrected couplings of the FermionicHubbardChain
- issue #2: memory leak in cython parts when using intelpython/anaconda
- issue #4: incompatible data types.
- issue #6: the CouplingModel generated wrong Couplings in some cases
- issue #19: Convergence of energy was slow for infinite systems with `N_sweeps_check=1`
- more reasonable traceback in case of wrong labels
- wrong dtype of npc.Array when adding/subtracting/... arrays of different data types
- could get wrong H_bond for completely decoupled chains.
- SVD could return outer indices with different axes
- *tenpy.networks.mps.MPS.overlap()* works now for MPS with different total charge (e.g. after `psi.apply_local_op(i, 'Sp')`).
- skip existing graph edges in MPOGraph.add() when building up terms without the strength part.

Removed

- Attribute *chinfo* of *Lattice*.

[0.3.0] - 2018-02-19

This is the first version published on github.

Added

- Cython modules for np_conserved and charges, which can optionally be compiled for speed-ups
- tools.optimization for dynamical optimization
- Various models.
- More predefined lattice sites.
- Example toy-codes.
- Network contractor for general networks

Changed

- Switch to python3

Removed

- Python 2 support.

[0.2.0] - 2017-02-24

- Compatible with python2 and python3 (using the 2to3 tool).
- Development version.
- Includes TEBD and DMRG.

Changes compared to previous TeNPy

This library is based on a previous (closed source) version developed mainly by Frank Pollmann, Michael P. Zaletel and Roger S. K. Mong. While allmost all files are completely rewritten and not backwards compatible, the overall structure is similar. In the following, we list only the most important changes.

Global Changes

- syntax style based on PEP8. Use `$>yapf -r -i ./` to ensure consitent formatting over the whole project. Special comments `# yapf: disable` and `# yapf: enable` can be used for manual formatting of some regions in code.
- Following PEP8, we distinguish between 'private' functions, indicated by names starting with an underscore and to be used only within the library, and the public API. The puplic API should be backwards-compatible with different releases, while private functions might change at any time.
- all modules are in the folder `tenpy` to avoid name conflicts with other libraries.
- withing the library, relative imports are used, e.g., `from ..tools.math import (toiterable, tonparray)` Exception: the files in *tests/* and *examples/* run as `__main__` and can't use relative imports

  Files outside of the library (and in *tests/*, *examples/*) should use absolute imports, e.g. `import tenpy.algorithms.tebd`
- renamed *tenpy/mps/* to *tenpy/networks*, since it containes various tensor networks.

- added *Site* describing the local physical sites by providing the physical LegCharge and onsite operators.

np_conserved

- pure python, no need to compile!
- in module *tenpy.linalg* instead of algorithms/linalg.
- moved functionality for charges to *charges*
- Introduced the classes *ChargeInfo* (basically the old q_number, and mod_q) and *LegCharge* (the old qind, qconj).
- Introduced the class *LegPipe* to replace the old leg_pipe. It is derived from LegCharge and used as a leg in the *array* class. Thus any inherited array (after tensordot etc still has all the necessary information to split the legs. (The legs are shared between different arrays, so it's saved only once in memory)
- Enhanced indexing of the array class to support slices and 1D index arrays along certain axes
- more functions, e.g. *grid_outer()*

TEBD

- Introduced TruncationError for easy handling of total truncation error.
- some truncation parameters are renamed and may have a different meaning, e.g. *svd_max* -> *svd_min* has no 'log' in the definition.

DMRG

- separate Lanczos module in *tenpy/linalg/*. Strangely, the old version orthoganalized against the complex conjugates of *orthogonal_to* (contrary to it's doc string!) (and thus calculated 'theta_o' as bra, not ket).
- cleaned up, provide prototypes for DMRG engine and mixer.

Tools

- added *tenpy.tools.misc*, which contains 'random stuff' from old tools.math like to_iterable and to_array (renamed to follow PEP8, documented)
- moved stuff for fitting to *tenpy.tools.fit*
- enhanced *tenpy.tools.string.vert_join()* for nice formatting
- moved (parts of) old *cluster/omp.py* to *tenpy.tools.process*
- added *tenpy.tools.params* for a simplified handling of parameter/arguments for models and/or algorithms. Similar as the old *models.model.set_var*, but use it also for algorithms. Also, it may modify the given dictionary.

TeNPy developer team

```
The following people are part of the TeNPy developer team.
The full list of contributors can be obtained from the git repository with ``git␣
↪shortlog -sn``.

Johannes Hauschild        tenpy@johannes-hauschild.de
Frank Pollmann
Michael P. Zaletel
Maximilian Schulz
Leon Schoonderwoerd
Kévin Hémery
Gunnar Moeller
Jakob Unfried
Yu-Chin Tzeng

Further, the code is based on an earlier version of the library, mainly developed by
Frank Pollmann, Michael P. Zaletel and Roger S. K. Mong.
```

License

The source code documented here is published under a GPL v3 license, which we include below.

```
                    GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.
```

(continues on next page)

(continued from previous page)

```
  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.
```

(continues on next page)

(continued from previous page)

```
  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.

  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
```

(continues on next page)

(continued from previous page)

```
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.
```

(continues on next page)

(continued from previous page)

```
  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.

    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
```

(continues on next page)

(continued from previous page)

```
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
```

(continues on next page)

(continued from previous page)

```
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or
```

(continues on next page)

(continued from previous page)

```
    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.

  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
```

(continues on next page)

(continued from previous page)

```
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
```

(continues on next page)

(continued from previous page)

```
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
```

(continues on next page)

(continued from previous page)

```
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
```

(continues on next page)

(continued from previous page)

```
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
```

(continues on next page)

(continued from previous page)

```
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.
```

7.2 Tenpy Reference

TeNPy - a Python library for Tensor Network Algorithms

TeNPy is a library for algorithms working with tensor networks, e.g., matrix product states and -operators, designed to study the physics of strongly correlated quantum systems. The code is intended to be accessible for newcommers and yet powerful enough for day-to-day research.

Submodules

algorithms	A collection of algorithms such as TEBD and DMRG.
linalg	Linear-algebra tools for tensor networks.
models	Definition of the various models.
networks	Definitions of tensor networks like MPS and MPO.
tools	A collection of tools: mostly short yet quite useful functions.
version	Access to version of this library.

7.2.1 algorithms

- full name: tenpy.algorithms
- parent module: *tenpy*
- type: module

Module description

A collection of algorithms such as TEBD and DMRG.

Submodules

truncation	Truncation of Schmidt values.
dmrg	Density Matrix Renormalization Group (DMRG).
mps_sweeps	'Sweep' algorithm and effective Hamiltonians.
tebd	Time evolving block decimation (TEBD).
tdvp	Time Dependant Variational Principle (TDVP) with MPS (finite version only).
purification_tebd	Time evolving block decimation (TEBD) for MPS of purification.
network_contractor	Network Contractor.
exact_diag	Full diagonalization (ED) of the Hamiltonian.

truncation

- full name: tenpy.algorithms.truncation
- parent module: *tenpy.algorithms*
- type: module

Classes

TruncationError([eps, ov])	Class representing a truncation error.

TruncationError

- full name: tenpy.algorithms.truncation.TruncationError
- parent module: *tenpy.algorithms.truncation*
- type: class

class tenpy.algorithms.truncation.**TruncationError**(*eps=0.0*, *ov=1.0*)

Bases: object

Class representing a truncation error.

The default initialization represents "no truncation".

Warning: For imaginary time evolution, this is *not* the error you are interested in!

Parameters

eps, ov [float] See below.

Examples

```
>>> TE = TruncationError()
>>> TE += tebd.time_evolution(...)  # add `eps`, multiply `ov`
```

Attributes

ov_err Error `1.-ov` of the overlap with the correct state.

eps [float] The total sum of all discared Schmidt values squared. Note that if you keep singular values up to 1.e-14 (= a bit more than machine precision for 64bit floats), *eps* is on the order of 1.e-28 (due to the square)!

ov [float] A lower bound for the overlap $|\langle\psi_{trunc}|\psi_{correct}\rangle|^2$ (assuming normalization of both states). This is probably the quantity you are actually interested in. Takes into account the factor 2 explained in the section on Errors in the *TEBD Wikipedia article <https://en.wikipedia.org/wiki/Time-evolving_block_decimation>*.

Methods

copy(self)	Return a copy of self.
from_S(S_discarded[, norm_old])	Construct TruncationError from discarded singular values.
from_norm(norm_new[, norm_old])	Construct TruncationError from norm after and before the truncation.

copy (*self*)
Return a copy of self.

classmethod from_norm (*norm_new*, *norm_old=1.0*)
Construct TruncationError from norm after and before the truncation.

Parameters

norm_new [float] Norm of Schmidt values kept, $\sqrt{\sum_{akept} \lambda_a^2}$ (before re-normalization).

norm_old [float] Norm of all Schmidt values before truncation, $\sqrt{\sum_a \lambda_a^2}$.

classmethod from_S (*S_discarded*, *norm_old=None*)
Construct TruncationError from discarded singular values.

Parameters

S_discarded [1D numpy array] The singular values discarded.

norm_old [float] Norm of all Schmidt values before truncation, $\sqrt{\sum_a \lambda_a^2}$. Default (`None`) is 1.

property ov_err
Error `1.-ov` of the overlap with the correct state.

Functions

svd_theta(theta, trunc_par[, qtotal_LR, ...])	Performs SVD of a matrix *theta* (= the wavefunction) and truncates it.
truncate(S, trunc_par)	Given a Schmidt spectrum *S*, determine which values to keep.

svd_theta

- full name: tenpy.algorithms.truncation.svd_theta
- parent module: *tenpy.algorithms.truncation*
- type: function

tenpy.algorithms.truncation.**svd_theta**(*theta, trunc_par, qtotal_LR=[None, None], inner_labels=['vR', 'vL']*)

Performs SVD of a matrix *theta* (= the wavefunction) and truncates it.

Perform a singular value decomposition (SVD) with *svd()* and truncates with *truncate()*. The result is an approximation `theta ~= tensordot(U.scale_axis(S*renormalization, 1), VH, axes=1)`

Parameters

theta [*Array*, shape `(M, N)`] The matrix, on which the singular value decomposition (SVD) is performed. Usually, *theta* represents the wavefunction, such that the SVD is a Schmidt decomposition.

trunc_par [dict] truncation parameters as described in *truncate()*.

qtotalLR [(charges, charges)] The total charges for the returned *U* and *VH*.

inner_labels [(string, string)] Labels for the *U* and *VH* on the newly-created bond.

Returns

U [*Array*] Matrix with left singular vectors as columns. Shape `(M, M)` or `(M, K)` depending on *full_matrices*.

S [1D ndarray] The singluar values of the array. If no *cutoff* is given, it has lenght `min(M, N)`. Normalized to `np.linalg.norm(S)==1`.

VH [*Array*] Matrix with right singular vectors as rows. Shape `(N, N)` or `(K, N)` depending on *full_matrices*.

err [*TruncationError*] The truncation error introduced.

renormalization [float] Factor, by which S was renormalized.

truncate

- full name: tenpy.algorithms.truncation.truncate
- parent module: *tenpy.algorithms.truncation*
- type: function

tenpy.algorithms.truncation.**truncate**(*S*, *trunc_par*)
Given a Schmidt spectrum *S*, determine which values to keep.

Parameters

S [1D array] Schmidt values (as returned by an SVD), not necessarily sorted. Should be normalized to `np.sum(S*S) == 1.`.

trunc_par: dict Parameters giving constraints for the truncation. If a constraint can not be fullfilled (without violating a previous one), it is ignored. A value `None` indicates that the constraint should be ignored.

key	type	constraint
chi_max	int	Keep at most *chi_max* Schmidt values.
chi_min	int	Keep at least *chi_min* Schmidt values.
symmetry_tol	float	Don't cut between Schmidt values with `|log(S[i]/S[j])| < log(symmetry_tol)` (i.e. either keep either both *i* and *j* or none). This is useful to prevent discarding (nearly) degenerate pairs in case of symmetries.
svd_min	float	Discard all small Schmidt values `S[i] < svd_min`.
trunc_cut	float	Discard all small Schmidt values as long as `sum_{i discarded} S[i]**2 <= trunc_cut**2`.

Returns

mask [1D bool array] Index mask, True for indices which should be kept.

norm_new [float] The norm of the truncated Schmidt values, `np.linalg.norm(S[mask])`. Useful for re-normalization.

err [*TruncationError*] The error of the represented state which is introduced due to the truncation.

Module description

Truncation of Schmidt values.

Often, it is necessary to truncate the number of states on a virtual bond of an MPS, keeping only the state with the largest Schmidt values. The function *truncate()* picks exactly those from a given Schmidt spectrum λ_a, depending on some parameters explained in the doc-string of the function.

Further, we provide *TruncationError* for a simple way to keep track of the total truncation error.

The SVD on a virtual bond of an MPS actually gives a Schmidt decomposition $|\psi\rangle = \sum_a \lambda_a |L_a\rangle |R_a\rangle$ where $|L_a\rangle$ and $|R_a\rangle$ form orthonormal bases of the parts left and right of the virtual bond. Let us assume that the state is properly normalized, $\langle\psi|\psi\rangle = \sum_a \lambda_a^2 = 1$. Assume that the singular values are ordered descending, and that we keep the first χ_c of the initially χ Schmidt values.

Then we decompose the untruncated state as $|\psi\rangle = \sqrt{1-\epsilon}|\psi_{tr}\rangle + \sqrt{\epsilon}|\psi_{tr}^{\perp}\rangle$ where $|\psi_{tr}\rangle = \frac{1}{\sqrt{1-\epsilon}} \sum_{a<\chi_c} \lambda_a |L_a\rangle|R_a\rangle$ is the truncated state kept (normalized to 1), $|\psi_{tr}^{\perp}\rangle = \frac{1}{\sqrt{\epsilon}} \sum_{a>=\chi_c} \lambda_a |L_a\rangle|R_a\rangle$ is the discarded part (orthogonal to the kept part) and the *truncation error of a single truncation* is defined as $\epsilon = 1 - |\langle\psi|\psi_{tr}\rangle|^2 = \sum_{a>=\chi_c} \lambda_a^2$.

Warning: For imaginary time evolution (e.g. with TEBD), you try to project out the ground state. Then, looking at the truncation error defined in this module does *not* give you any information how good the found state coincides with the actual ground state! (Instead, the returned truncation error depends on the overlap with the initial state, which is arbitrary > 0)

Warning: This module takes only track of the errors coming from the truncation of Schmidt values. There might be other sources of error as well, for example TEBD has also an discretisation error depending on the chosen time step.

dmrg

- full name: tenpy.algorithms.dmrg
- parent module: *tenpy.algorithms*
- type: module

Classes

DMRGEngine(psi, model, engine_params)	Generic 'Engine' for the single-site DMRG algorithm.
DensityMatrixMixer(mixer_params)	Mixer based on density matrices.
EngineCombine(psi, model, DMRG_params)	Engine which combines legs into pipes as far as possible.
EngineFracture(psi, model, DMRG_params)	Engine which keeps the legs separate.
Mixer(mixer_params)	Base class of a general Mixer.
SingleSiteDMRGEngine(psi, model, engine_params)	'Engine' for the single-site DMRG algorithm.
SingleSiteMixer(mixer_params)	Mixer for single-site DMRG.
TwoSiteDMRGEngine(psi, model, engine_params)	'Engine' for the two-site DMRG algorithm.
TwoSiteMixer(mixer_params)	Mixer for two-site DMRG.

DMRGEngine

- full name: tenpy.algorithms.dmrg.DMRGEngine
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**DMRGEngine**(*psi*, *model*, *engine_params*)

Bases: *tenpy.algorithms.mps_sweeps.Sweep*

Generic 'Engine' for the single-site DMRG algorithm.

This engine is implemented as a subclass of *Sweep*. It contains all methods that are generic between *SingleSiteDMRGEngine* and *TwoSiteDMRGEngine*.

Parameters

psi [*MPS*] Initial guess for the ground state, which is to be optimized in-place.

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state.

engine_params [dict] Further optional parameters. These are usually algorithm-specific, and thus should be described in subclasses.

Attributes

EffectiveH [class type] Class for the effective Hamiltonian (i.e., a subclass of *EffectiveH*. Has a *length* class attribute which specifies the number of sites updated at once (e.g., whether we do single-site vs. two-site DMRG).

chi_list [dict | None] A dictionary to gradually increase the *chi_max* parameter of *trunc_params*. The key defines starting from which sweep *chi_max* is set to the value, e.g. {0: 50, 20: 100} uses chi_max=50 for the first 20 sweeps and chi_max=100 afterwards. Overwrites *trunc_params['chi_list']*`. By default (None) this feature is disabled.

eff_H [*EffectiveH*] Effective two-site Hamiltonian.

mixer [*Mixer* | None] If None, no mixer is used (anymore), otherwise the mixer instance.

shelve [bool] If a simulation runs out of time (*time.time() - start_time > max_seconds*), the run will terminate with *shelve = True*.

sweeps [int] The number of sweeps already performed. (Useful for re-start).

time0 [float] Time marker for the start of the run.

update_stats [dict] A dictionary with detailed statistics of the convergence at local update-level. For each key in the following table, the dictionary contains a list where one value is added each time DMRGEngine.update_bond() is called.

key	description
i0	An update was performed on sites i0, i0+1.
age	The number of physical sites involved in the simulation.
E_total	The total energy before truncation.
N_lanczos	Dimension of the Krylov space used in the lanczos diagonalization.
time	Wallclock time evolved since time0 (in seconds).
ov_change	1 - abs(<theta_guess|theta_diag>), where |theta_guess> is the initial guess for the wave function and |theta_diag> is the *untruncated* wave function returned by *diag()*.

sweep_stats [dict] A dictionary with detailed statistics at the sweep level. For each key in the following table, the dictionary contains a list where one value is added each time Engine.sweep() is called (with optimize=True).

key	description
sweep	Number of sweeps (excluding environment sweeps) performed so far.
N_updates	Number of updates (including environment sweeps) performed so far.
E	The energy *before* truncation (as calculated by Lanczos).
S	Maximum entanglement entropy.
time	Wallclock time evolved since `time0` (in seconds).
max_trunc_err	The maximum truncation error in the last sweep
max_E_trunc	Maximum change or Energy due to truncation in the last sweep.
max_chi	Maximum bond dimension used.
norm_err	Error of canonical form `np.linalg.norm(psi.norm_test())`.

Methods

diag(self, theta_guess)	Diagonalize the effective Hamiltonian represented by self.
environment_sweeps(self, N_sweeps)	Perform *N_sweeps* sweeps without optimization to update the environment.
get_sweep_schedule(self)	Define the schedule of the sweep.
init_env(self[, model])	(Re-)initialize the environment.
mixer_activate(self)	Set *self.mixer* to the class specified by *engine_params['mixer']*.
mixer_cleanup(self)	Cleanup the effects of a mixer.
plot_sweep_stats(self[, axes, xaxis, yaxis, ...])	Plot `sweep_stats` to display the convergence with the sweeps.
plot_update_stats(self, axes[, xaxis, ...])	Plot `update_stats` to display the convergence during the sweeps.
post_update_local(self, update_data[, ...])	Perform post-update actions.
prepare_update(self)	Prepare everything algorithm-specific to perform a local update.
reset_stats(self)	Reset the statistics, useful if you want to start a new sweep run.
run(self)	Run the DMRG simulation to find the ground state.
sweep(self[, optimize, meas_E_trunc])	One 'sweep' of a sweeper algorithm.
update_local(self, theta, **kwargs)	Perform algorithm-specific local update.

run (*self*)
Run the DMRG simulation to find the ground state.

Returns

E [float] The energy of the resulting ground state MPS.

psi [*MPS*] The MPS representing the ground state after the simluation, i.e. just a reference to `psi`.

reset_stats (*self*)
Reset the statistics, useful if you want to start a new sweep run.

post_update_local (*self*, *update_data*, *meas_E_trunc=False*)
Perform post-update actions.

Compute truncation energy, remove *LP*/*RP* that are no longer needed and collect statistics.

Parameters

update_data [dict] Data computed during the local update, as described in the following list.

meas_E_trunc [bool, optional] Wheter to measure the energy after truncation.

diag(*self*, *theta_guess*)

Diagonalize the effective Hamiltonian represented by self.

The method used depends on the DMRG parameter *diag_method*.

diag_method	Function, comment
'lanczos'	*lanczos()* Default, the Lanczos implementation of TeNPy
'arpack'	*lanczos_arpack()* Based on `scipy.linalg.sparse.eigsh()`. Slower than 'lanczos', since it needs to convert the npc arrays to numpy arrays during *each* matvec, and possibly does many more iterations.
'ED_block'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize the block in the charge sector of the initial state. Preserves the charge sector of the explicitly conserved charges. However, if you don't preserve a charge explicitly, it can break it. For example if you use a `SpinChain({'conserve': 'parity'})`, it could change the total "Sz", but not the parity of 'Sz'.
'ED_all'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize it completely. Allows to change the charge sector *even for explicitly conserved charges*. For example if you use a `SpinChain({'conserve': 'Sz'})`, it **can** change the total "Sz".

Parameters

theta_guess [*Array*] Initial guess for the ground state of the effective Hamiltonian.

Returns

E0 [float] Energy of the found ground state.

theta [*Array*] Ground state of the effective Hamiltonian.

N [int] Number of Lanczos iterations used. `-1` if unknown.

ov_change [float] Change in the wave function `1. - abs(<theta_guess|theta_diag>)`

plot_update_stats(*self*, *axes*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)

Plot `update_stats` to display the convergence during the sweeps.

Parameters

axes [`matplotlib.axes.Axes`] The axes to plot into. Defaults to `matplotlib.pyplot.gca()`

xaxis [`'N_updates' | 'sweep' |` keys of `update_stats`] Key of `update_stats` to be used for the x-axis of the plots. `'N_updates'` is just enumerating the number of bond updates, and `'sweep'` corresponds to the sweep number (including environment sweeps).

yaxis [`'E' |` keys of `update_stats`] Key of `update_stats` to be used for the y-axisof the plots. For 'E', use the energy (per site for infinite systems).

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot `abs((y-y_exact)/y_exact)` on a log-scale yaxis.

****kwargs :** Further keyword arguments given to `axes.plot(...)`.

plot_sweep_stats(*self*, *axes=None*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot `sweep_stats` to display the convergence with the sweeps.

Parameters

axes [`matplotlib.axes.Axes`] The axes to plot into. Defaults to `matplotlib.pyplot.gca()`

xaxis, yaxis [key of `sweep_stats`] Key of `sweep_stats` to be used for the x-axis and y-axis of the plots.

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot `abs((y-y_exact)/y_exact)` on a log-scale yaxis.

****kwargs :** Further keyword arguments given to `axes.plot(...)`.

environment_sweeps(*self*, *N_sweeps*)
Perform *N_sweeps* sweeps without optimization to update the environment.

Parameters

N_sweeps [int] Number of sweeps to run without optimization

get_sweep_schedule(*self*)
Define the schedule of the sweep.

One 'sweep' is a full sequence from the leftmost site to the right and back. Only those *LP* and *RP* that can be used later should be updated.

Returns

schedule [iterable of (int, bool, (bool, bool))] Schedule for the sweep. Each entry is `(i0, move_right, (update_LP, update_RP))`, where *i0* is the leftmost of the `self.EffectiveH.length` sites to be updated in *update_local()*, *move_right* indicates whether the next *i0* in the schedule is rigth (*True*) of the current one, and *update_LP*, *update_RP* indicate whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory.

init_env(*self*, *model=None*)
(Re-)initialize the environment.

This function is useful to (re-)start a Sweep with a slightly different model or different (engine) parameters. Note that we assume that we still have the same *psi*. Calls *reset_stats()*.

Parameters

model [`MPOModel`] The model representing the Hamiltonian for which we want to find the ground state. If `None`, keep the model used before.

Raises

ValueError If the engine is re-initialized with a new model, which legs are incompatible with those of hte old model.

mixer_activate(*self*)
Set *self.mixer* to the class specified by *engine_params['mixer']*.

It is expected that different algorithms have differen ways of implementing mixers (with different defaults). Thus, this is algorithm-specific.

mixer_cleanup(*self*)
Cleanup the effects of a mixer.

A *sweep()* with an enabled *Mixer* leaves the MPS *psi* with 2D arrays in *S*. To recover the originial form, this function simply performs one sweep with disabled mixer.

prepare_update(*self*)
Prepare everything algorithm-specific to perform a local update.

sweep(*self*, *optimize=True*, *meas_E_trunc=False*)
One 'sweep' of a sweeper algorithm.

Iteratate over the bond which is optimized, to the right and then back to the left to the starting point. If optimize=False, don't actually diagonalize the effective hamiltonian, but only update the environment.

Parameters

optimize [bool, optional] Whether we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool, optional] Whether to measure truncation energies.

Returns

max_trunc_err [float] Maximal truncation error introduced.

max_E_trunc [None | float] None if meas_E_trunc is False, else the maximal change of the energy due to the truncation.

update_local(*self*, *theta*, ***kwargs*)
Perform algorithm-specific local update.

DensityMatrixMixer

- full name: tenpy.algorithms.dmrg.DensityMatrixMixer
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**DensityMatrixMixer**(*mixer_params*)
Bases: *tenpy.algorithms.dmrg.Mixer*

Mixer based on density matrices.

This mixer constructs density matrices as described in the original paper [White2005].

Methods

get_xL(self, wL_leg, Id_L, Id_R)	Generate the coupling of the MPO legs for the reduced density matrix.
get_xR(self, wR_leg, Id_L, Id_R)	Generate the coupling of the MPO legs for the reduced density matrix.
mix_rho_L(self, engine, theta, i0, mix_enabled)	Calculated mixed reduced density matrix for left site.
mix_rho_R(self, engine, theta, i0, mix_enabled)	Calculated mixed reduced density matrix for left site.
perturb_svd(self, engine, theta, i0, ...)	Mix extra terms to theta and perform an SVD.
update_amplitude(self, sweeps)	Update the amplitude, possibly disable the mixer.

perturb_svd(*self*, *engine*, *theta*, *i0*, *update_LP*, *update_RP*)

Mix extra terms to theta and perform an SVD.

We calculate the left and right reduced density using the mixer (which might include applications of H). These density matrices are diagonalized and truncated such that we effectively perform a svd for the case `mixer.amplitude=0`.

Parameters

engine [*SingleSiteDMRGEngine* | *TwoSiteDMRGEngine*] The DMRG engine calling the mixer.

theta [*Array*] The optimized wave function, prepared for svd.

i0 [int] Site index; *theta* lives on `i0, i0+1`.

update_LP [bool] Whether to calculate the next `env.LP[i0+1]`.

update_RP [bool] Whether to calculate the next `env.RP[i0]`.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

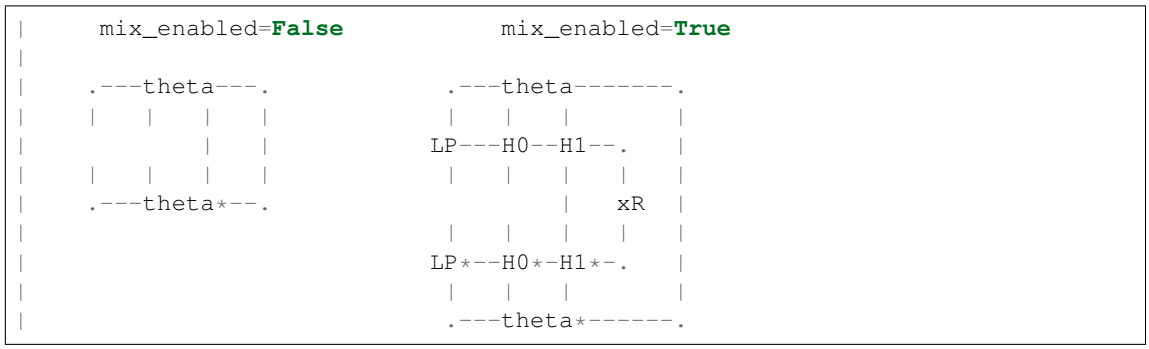
S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(p1.vR)'`.

err [*TruncationError*] The truncation error introduced.

mix_rho_L(*self*, *engine*, *theta*, *i0*, *mix_enabled*)
Calculated mixed reduced density matrix for left site.

Pictorially:



Parameters

engine [Engine] The DMRG engine calling the mixer.

theta [*Array*] Ground state of the effective Hamiltonian, prepared for svd.

i0 [int] Site index; *theta* lives on `i0, i0+1`.

mix_enabled [bool] Whether we should perturb the density matrix.

Returns

rho_L [*Array*] A (hermitian) square array with labels `'(vL.p0)', '(vL*.p0*)'`, Mainly the reduced density matrix of the left part, but with some additional mixing.

mix_rho_R(*self*, *engine*, *theta*, *i0*, *mix_enabled*)
Calculated mixed reduced density matrix for left site.

Pictorially:

```
|     mix_enabled=False           mix_enabled=True
|
|    .---theta---.           .------theta---.
|    |   |   |   |           |      |   |   |
|    |   |                   |   .--H0--H1--RP
|    |   |   |   |           |   |  |   |   |
|    .---theta*--.           |  wL  |
|                            |   |  |   |   |
|                            |   .--H0*-H1*-RP*
|                            |      |   |   |
|                            .------theta*--.
```

Parameters

engine [Engine] The DMRG engine calling the mixer.

theta [*Array*] Ground state of the effective Hamiltonian, prepared for svd.

i0 [int] Site index; *theta* lives on `i0, i0+1`.

mix_enabled [bool] Whether we should perturb the density matrix.

Returns

rho_R [*Array*] A (hermitian) square array with labels `'(p1.vR)', '(p1*.vR*)'`. Mainly the reduced density matrix of the right part, but with some additional mixing.

get_xR(*self*, *wR_leg*, *Id_L*, *Id_R*)
Generate the coupling of the MPO legs for the reduced density matrix.

Parameters

wR_leg [*LegCharge*] LegCharge to be connected to.

IdL [int | None] Index within the leg for which the MPO has only identities to the left.

IdR [int | None] Index within the leg for which the MPO has only identities to the right.

Returns

mixed_xR [*Array*] Connection of the MPOs on the right for the reduced density matrix *rhoL*. Labels `('wL', 'wL*')`.

add_separate_Id [bool] If Id_L is None, we can't include the identity into *mixed_xR*, so it has to be added directly in *mix_rho_L()*.

get_xL(*self*, *wL_leg*, *Id_L*, *Id_R*)
Generate the coupling of the MPO legs for the reduced density matrix.

Parameters

wL_leg [*LegCharge*] LegCharge to be connected to.

Id_L [int | None] Index within the leg for which the MPO has only identities to the left.

Id_R [int | None] Index within the leg for which the MPO has only identities to the right.

Returns

mixed_xL [*Array*] Connection of the MPOs on the left for the reduced density matrix *rhoR*. Labels `('wR', 'wR*')`.

add_separate_Id [bool] If Id_R is `None`, we can't include the identity into *mixed_xL*, so it has to be added directly in *mix_rho_R()*.

update_amplitude(*self*, *sweeps*)
Update the amplitude, possibly disable the mixer.

Parameters

sweeps [int] The number of performed sweeps, to check if we need to disable the mixer.

Returns

mixer [*Mixer* | None] Returns *self* if we should continue mixing, or `None`, if the mixer should be disabled.

EngineCombine

- full name: tenpy.algorithms.dmrg.EngineCombine
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**EngineCombine**(*psi*, *model*, *DMRG_params*)
Bases: *tenpy.algorithms.dmrg.TwoSiteDMRGEngine*

Engine which combines legs into pipes as far as possible.

This engine combines the virtual and physical leg for the left site and right site into pipes. This reduces the overhead of calculating charge combinations in the contractions, but one `matvec()` is formally more expensive, $O(2d^3\chi^3 D)$.

Methods

diag(self, theta_guess)	Diagonalize the effective Hamiltonian represented by self.
environment_sweeps(self, N_sweeps)	Perform *N_sweeps* sweeps without optimization to update the environment.
get_sweep_schedule(self)	Define the schedule of the sweep.
init_env(self[, model])	(Re-)initialize the environment.
mixed_svd(self, theta)	Get (truncated) *B* from the new theta (as returned by diag).
mixer_activate(self)	Set *self.mixer* to the class specified by *engine_params['mixer']*.
mixer_cleanup(self)	Cleanup the effects of a mixer.
plot_sweep_stats(self[, axes, xaxis, yaxis, ...])	Plot `sweep_stats` to display the convergence with the sweeps.
plot_update_stats(self, axes[, xaxis, ...])	Plot `update_stats` to display the convergence during the sweeps.
post_update_local(self, update_data[, ...])	Perform post-update actions.
prepare_svd(self, theta)	Transform theta into matrix for svd.
prepare_update(self)	Prepare *self* to represent the effective Hamiltonian on sites `(i0, i0+1)`.
reset_stats(self)	Reset the statistics, useful if you want to start a new sweep run.

Continued on next page

Table 9 – continued from previous page

run(self)	Run the DMRG simulation to find the ground state.
set_B(self, U, S, VH)	Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.
sweep(self[, optimize, meas_E_trunc])	One 'sweep' of a sweeper algorithm.
update_LP(self, U)	Update left part of the environment.
update_RP(self, VH)	Update right part of the environment.
update_local(self, theta[, optimize, ...])	Perform bond-update on the sites `(i0, i0+1)`.

diag (*self*, *theta_guess*)

Diagonalize the effective Hamiltonian represented by self.

The method used depends on the DMRG parameter *diag_method*.

diag_method	Function, comment
'lanczos'	*lanczos()* Default, the Lanczos implementation of TeNPy
'arpack'	*lanczos_arpack()* Based on `scipy.linalg.sparse.eigsh()`. Slower than 'lanczos', since it needs to convert the npc arrays to numpy arrays during *each* matvec, and possibly does many more iterations.
'ED_block'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize the block in the charge sector of the initial state. Preserves the charge sector of the explicitly conserved charges. However, if you don't preserve a charge explicitly, it can break it. For example if you use a `SpinChain({'conserve': 'parity'})`, it could change the total "Sz", but not the parity of 'Sz'.
'ED_all'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize it completely. Allows to change the charge sector *even for explicitly conserved charges*. For example if you use a `SpinChain({'conserve': 'Sz'})`, it **can** change the total "Sz".

Parameters

theta_guess [*Array*] Initial guess for the ground state of the effective Hamiltonian.

Returns

E0 [float] Energy of the found ground state.

theta [*Array*] Ground state of the effective Hamiltonian.

N [int] Number of Lanczos iterations used. `-1` if unknown.

ov_change [float] Change in the wave function `1. - abs(<theta_guess|theta_diag>)`

environment_sweeps (*self*, *N_sweeps*)

Perform *N_sweeps* sweeps without optimization to update the environment.

Parameters

N_sweeps [int] Number of sweeps to run without optimization

get_sweep_schedule (*self*)

Define the schedule of the sweep.

One 'sweep' is a full sequence from the leftmost site to the right and back. Only those *LP* and *RP* that can be used later should be updated.

Returns

schedule [iterable of (int, bool, (bool, bool))] Schedule for the sweep. Each entry is `(i0, move_right, (update_LP, update_RP))`, where *i0* is the leftmost of the `self.EffectiveH.length` sites to be updated in *update_local()*, *move_right* indicates whether the next *i0* in the schedule is rigth (*True*) of the current one, and *update_LP*, *update_RP* indicate whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory.

init_env(*self*, *model=None*)

(Re-)initialize the environment.

This function is useful to (re-)start a Sweep with a slightly different model or different (engine) parameters. Note that we assume that we still have the same *psi*. Calls *reset_stats()*.

Parameters

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state. If None, keep the model used before.

Raises

ValueError If the engine is re-initialized with a new model, which legs are incompatible with those of hte old model.

mixed_svd(*self*, *theta*)

Get (truncated) *B* from the new theta (as returned by diag).

The goal is to split theta and truncate it:

```
|   -- theta --   ==>    -- U -- S --  VH -
|      |   |                |          |
```

Without a mixer, this is done by a simple svd and truncation of Schmidt values.

With a mixer, the state is perturbed before the SVD. The details of the perturbation are defined by the *Mixer* class.

Note that the returned *S* is a general (not diagonal) matrix, with labels `'vL', 'vR'`.

Parameters

theta [*Array*] The optimized wave function, prepared for svd.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix with labels `'vL', 'vR'`; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(p1.vR)'`.

err [*TruncationError*] The truncation error introduced.

mixer_activate(*self*)

Set *self.mixer* to the class specified by *engine_params['mixer']*.

mixer_cleanup(*self*)

Cleanup the effects of a mixer.

A *sweep()* with an enabled *Mixer* leaves the MPS *psi* with 2D arrays in *S*. To recover the originial form, this function simply performs one sweep with disabled mixer.

plot_sweep_stats(*self*, *axes=None*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)

Plot `sweep_stats` to display the convergence with the sweeps.

Parameters

axes [matplotlib.axes.Axes] The axes to plot into. Defaults to matplotlib.pyplot.gca()

xaxis, yaxis [key of sweep_stats] Key of sweep_stats to be used for the x-axis and y-axis of the plots.

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot abs((y-y_exact)/y_exact) on a log-scale yaxis.

****kwargs :** Further keyword arguments given to axes.plot(...).

plot_update_stats(*self*, *axes*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot update_stats to display the convergence during the sweeps.

Parameters

axes [matplotlib.axes.Axes] The axes to plot into. Defaults to matplotlib.pyplot.gca()

xaxis ['N_updates' | 'sweep' | keys of update_stats] Key of update_stats to be used for the x-axis of the plots. 'N_updates' is just enumerating the number of bond updates, and 'sweep' corresponds to the sweep number (including environment sweeps).

yaxis ['E' | keys of update_stats] Key of update_stats to be used for the y-axisof the plots. For 'E', use the energy (per site for infinite systems).

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot abs((y-y_exact)/y_exact) on a log-scale yaxis.

****kwargs :** Further keyword arguments given to axes.plot(...).

post_update_local(*self*, *update_data*, *meas_E_trunc=False*)
Perform post-update actions.

Compute truncation energy, remove *LP*/*RP* that are no longer needed and collect statistics.

Parameters

update_data [dict] Data computed during the local update, as described in the following list.

meas_E_trunc [bool, optional] Wheter to measure the energy after truncation.

prepare_svd(*self*, *theta*)
Transform theta into matrix for svd.

prepare_update(*self*)
Prepare *self* to represent the effective Hamiltonian on sites (i0, i0+1).

Returns

theta [*Array*] Current best guess for the ground state, which is to be optimized. Labels 'vL', 'p0', 'vR', 'p1'.

reset_stats(*self*)
Reset the statistics, useful if you want to start a new sweep run.

run(*self*)
Run the DMRG simulation to find the ground state.

Returns

E [float] The energy of the resulting ground state MPS.

psi [*MPS*] The MPS representing the ground state after the simluation, i.e. just a reference to `psi`.

set_B(*self*, *U*, *S*, *VH*)
Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.

Parameters

U, VH [*Array*] Left and Right-canonical matrices as returned by the SVD.

S [1D array | 2D *Array*] The middle part returned by the SVD, `theta = U S VH`. Without a mixer just the singular values, with enabled *mixer* a 2D array.

sweep(*self*, *optimize=True*, *meas_E_trunc=False*)
One 'sweep' of a sweeper algorithm.

Iteratate over the bond which is optimized, to the right and then back to the left to the starting point. If optimize=False, don't actually diagonalize the effective hamiltonian, but only update the environment.

Parameters

optimize [bool, optional] Whether we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool, optional] Whether to measure truncation energies.

Returns

max_trunc_err [float] Maximal truncation error introduced.

max_E_trunc [`None` | float] `None` if meas_E_trunc is False, else the maximal change of the energy due to the truncation.

update_LP(*self*, *U*)
Update left part of the environment.

We always update the environment at site i0 + 1: this environment then contains the site where we just performed a local update (when sweeping right).

Parameters

U [*Array*] The U as returned by the SVD, with combined legs, labels `'vL.p0', 'vR'`.

update_RP(*self*, *VH*)
Update right part of the environment.

We always update the environment at site i0: this environment then contains the site where we just performed a local update (when sweeping left).

Parameters

VH [*Array*] The VH as returned by SVD, with combined legs, labels `'vL', '(vR.p1)'`.

update_local(*self*, *theta*, *optimize=True*, *meas_E_trunc=False*)
Perform bond-update on the sites `(i0, i0+1)`.

Parameters

theta [*Array*] Initial guess for the ground state of the effective Hamiltonian.

optimize [bool] Wheter we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool] Wheter to measure the energy after truncation.

Returns

update_data [dict] Data computed during the local update, as described in the following:

E0 [float] Total energy, obtained *before* truncation (if `optimize=True`), or *after* truncation (if `optimize=False`) (but never `None`).

N [int] Dimension of the Krylov space used for optimization in the lanczos algorithm. 0 if `optimize=False`.

age [int] Current size of the DMRG simulation: number of physical sites involved into the contraction.

U, VH: ***Array*** *U* and *VH* returned by *mixed_svd()*.

ov_change: float Change in the wave function `1. - abs(<theta_guess|theta>)` induced by *diag()*, *not* including the truncation!

EngineFracture

- full name: tenpy.algorithms.dmrg.EngineFracture
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**EngineFracture**(*psi*, *model*, *DMRG_params*)

Bases: *tenpy.algorithms.dmrg.TwoSiteDMRGEngine*

Engine which keeps the legs separate.

Due to a different contraction order in `matvec()`, this engine might be faster than *EngineCombine*, at least for large physical dimensions and if the MPO is sparse. One `matvec()` is $O(2\chi^3 d^2 W + 2\chi^2 d^3 W^2)$.

Methods

diag(self, theta_guess)	Diagonalize the effective Hamiltonian represented by self.
environment_sweeps(self, N_sweeps)	Perform *N_sweeps* sweeps without optimization to update the environment.
get_sweep_schedule(self)	Define the schedule of the sweep.
init_env(self[, model])	(Re-)initialize the environment.
mixed_svd(self, theta)	Get (truncated) *B* from the new theta (as returned by diag).
mixer_activate(self)	Set *self.mixer* to the class specified by *engine_params['mixer']*.
mixer_cleanup(self)	Cleanup the effects of a mixer.
plot_sweep_stats(self[, axes, xaxis, yaxis, ...])	Plot `sweep_stats` to display the convergence with the sweeps.
plot_update_stats(self, axes[, xaxis, ...])	Plot `update_stats` to display the convergence during the sweeps.
post_update_local(self, update_data[, ...])	Perform post-update actions.
prepare_svd(self, theta)	Transform theta into matrix for svd.
prepare_update(self)	Prepare *self* to represent the effective Hamiltonian on sites `(i0, i0+1)`.

Continued on next page

Table 10 – continued from previous page

reset_stats(self)	Reset the statistics, useful if you want to start a new sweep run.
run(self)	Run the DMRG simulation to find the ground state.
set_B(self, U, S, VH)	Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.
sweep(self[, optimize, meas_E_trunc])	One 'sweep' of a sweeper algorithm.
update_LP(self, U)	Update left part of the environment.
update_RP(self, VH)	Update right part of the environment.
update_local(self, theta[, optimize, ...])	Perform bond-update on the sites `(i0, i0+1)`.

diag(*self*, *theta_guess*)
Diagonalize the effective Hamiltonian represented by self.

The method used depends on the DMRG parameter *diag_method*.

diag_method	Function, comment
'lanczos'	*lanczos()* Default, the Lanczos implementation of TeNPy
'arpack'	*lanczos_arpack()* Based on `scipy.linalg.sparse.eigsh()`. Slower than 'lanczos', since it needs to convert the npc arrays to numpy arrays during *each* matvec, and possibly does many more iterations.
'ED_block'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize the block in the charge sector of the initial state. Preserves the charge sector of the explicitly conserved charges. However, if you don't preserve a charge explicitly, it can break it. For example if you use a `SpinChain({'conserve': 'parity'})`, it could change the total "Sz", but not the parity of 'Sz'.
'ED_all'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize it completely. Allows to change the charge sector *even for explicitly conserved charges*. For example if you use a `SpinChain({'conserve': 'Sz'})`, it **can** change the total "Sz".

Parameters

theta_guess [*Array*] Initial guess for the ground state of the effective Hamiltonian.

Returns

E0 [float] Energy of the found ground state.

theta [*Array*] Ground state of the effective Hamiltonian.

N [int] Number of Lanczos iterations used. `-1` if unknown.

ov_change [float] Change in the wave function `1. - abs(<theta_guess|theta_diag>)`

environment_sweeps(*self*, *N_sweeps*)
Perform *N_sweeps* sweeps without optimization to update the environment.

Parameters

N_sweeps [int] Number of sweeps to run without optimization

get_sweep_schedule(*self*)
Define the schedule of the sweep.

One 'sweep' is a full sequence from the leftmost site to the right and back. Only those *LP* and *RP* that can be used later should be updated.

Returns

schedule [iterable of (int, bool, (bool, bool))] Schedule for the sweep. Each entry is `(i0, move_right, (update_LP, update_RP))`, where *i0* is the leftmost of the `self.EffectiveH.length` sites to be updated in *update_local()*, *move_right* indicates whether the next *i0* in the schedule is rigth (*True*) of the current one, and *update_LP*, *update_RP* indicate whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory.

init_env(*self*, *model=None*)

(Re-)initialize the environment.

This function is useful to (re-)start a Sweep with a slightly different model or different (engine) parameters. Note that we assume that we still have the same *psi*. Calls *reset_stats()*.

Parameters

model [`MPOModel`] The model representing the Hamiltonian for which we want to find the ground state. If `None`, keep the model used before.

Raises

ValueError If the engine is re-initialized with a new model, which legs are incompatible with those of hte old model.

mixed_svd(*self*, *theta*)

Get (truncated) *B* from the new theta (as returned by diag).

The goal is to split theta and truncate it:

```
|   -- theta --   ==>    -- U -- S --  VH -
|      |   |                |          |
```

Without a mixer, this is done by a simple svd and truncation of Schmidt values.

With a mixer, the state is perturbed before the SVD. The details of the perturbation are defined by the *Mixer* class.

Note that the returned *S* is a general (not diagonal) matrix, with labels `'vL', 'vR'`.

Parameters

theta [*Array*] The optimized wave function, prepared for svd.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix with labels `'vL', 'vR'`; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(p1.vR)'`.

err [*TruncationError*] The truncation error introduced.

mixer_activate(*self*)

Set *self.mixer* to the class specified by *engine_params['mixer']*.

mixer_cleanup(*self*)

Cleanup the effects of a mixer.

A *sweep()* with an enabled *Mixer* leaves the MPS *psi* with 2D arrays in *S*. To recover the originial form, this function simply performs one sweep with disabled mixer.

plot_sweep_stats(*self*, *axes=None*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot `sweep_stats` to display the convergence with the sweeps.

Parameters

axes [`matplotlib.axes.Axes`] The axes to plot into. Defaults to `matplotlib.pyplot.gca()`

xaxis, yaxis [key of `sweep_stats`] Key of `sweep_stats` to be used for the x-axis and y-axis of the plots.

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot `abs((y-y_exact)/y_exact)` on a log-scale yaxis.

****kwargs :** Further keyword arguments given to `axes.plot(...)`.

plot_update_stats(*self*, *axes*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot `update_stats` to display the convergence during the sweeps.

Parameters

axes [`matplotlib.axes.Axes`] The axes to plot into. Defaults to `matplotlib.pyplot.gca()`

xaxis [`'N_updates' | 'sweep' |` keys of `update_stats`] Key of `update_stats` to be used for the x-axis of the plots. `'N_updates'` is just enumerating the number of bond updates, and `'sweep'` corresponds to the sweep number (including environment sweeps).

yaxis [`'E'` | keys of `update_stats`] Key of `update_stats` to be used for the y-axisof the plots. For 'E', use the energy (per site for infinite systems).

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot `abs((y-y_exact)/y_exact)` on a log-scale yaxis.

****kwargs :** Further keyword arguments given to `axes.plot(...)`.

post_update_local(*self*, *update_data*, *meas_E_trunc=False*)
Perform post-update actions.

Compute truncation energy, remove *LP*/*RP* that are no longer needed and collect statistics.

Parameters

update_data [dict] Data computed during the local update, as described in the following list.

meas_E_trunc [bool, optional] Wheter to measure the energy after truncation.

prepare_svd(*self*, *theta*)
Transform theta into matrix for svd.

prepare_update(*self*)
Prepare *self* to represent the effective Hamiltonian on sites `(i0, i0+1)`.

Returns

theta [*Array*] Current best guess for the ground state, which is to be optimized. Labels `'vL', 'p0', 'vR', 'p1'`.

reset_stats(*self*)
Reset the statistics, useful if you want to start a new sweep run.

run(*self*)
Run the DMRG simulation to find the ground state.

Returns

E [float] The energy of the resulting ground state MPS.

psi [*MPS*] The MPS representing the ground state after the simluation, i.e. just a reference to `psi`.

set_B (*self*, *U*, *S*, *VH*)
Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.

Parameters

U, VH [*Array*] Left and Right-canonical matrices as returned by the SVD.

S [1D array | 2D *Array*] The middle part returned by the SVD, `theta = U S VH`. Without a mixer just the singular values, with enabled *mixer* a 2D array.

sweep (*self*, *optimize=True*, *meas_E_trunc=False*)
One 'sweep' of a sweeper algorithm.

Iteratate over the bond which is optimized, to the right and then back to the left to the starting point. If optimize=False, don't actually diagonalize the effective hamiltonian, but only update the environment.

Parameters

optimize [bool, optional] Whether we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool, optional] Whether to measure truncation energies.

Returns

max_trunc_err [float] Maximal truncation error introduced.

max_E_trunc [`None` | float] `None` if meas_E_trunc is False, else the maximal change of the energy due to the truncation.

update_LP (*self*, *U*)
Update left part of the environment.

We always update the environment at site i0 + 1: this environment then contains the site where we just performed a local update (when sweeping right).

Parameters

U [*Array*] The U as returned by the SVD, with combined legs, labels `'vL.p0', 'vR'`.

update_RP (*self*, *VH*)
Update right part of the environment.

We always update the environment at site i0: this environment then contains the site where we just performed a local update (when sweeping left).

Parameters

VH [*Array*] The VH as returned by SVD, with combined legs, labels `'vL', '(vR.p1)'`.

update_local (*self*, *theta*, *optimize=True*, *meas_E_trunc=False*)
Perform bond-update on the sites `(i0, i0+1)`.

Parameters

theta [*Array*] Initial guess for the ground state of the effective Hamiltonian.

optimize [bool] Wheter we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool] Wheter to measure the energy after truncation.

Returns

update_data [dict] Data computed during the local update, as described in the following:

E0 [float] Total energy, obtained *before* truncation (if `optimize=True`), or *after* truncation (if `optimize=False`) (but never `None`).

N [int] Dimension of the Krylov space used for optimization in the lanczos algorithm. 0 if `optimize=False`.

age [int] Current size of the DMRG simulation: number of physical sites involved into the contraction.

U, VH: ***Array*** *U* and *VH* returned by *mixed_svd()*.

ov_change: float Change in the wave function `1. - abs(<theta_guess|theta>)` induced by *diag()*, *not* including the truncation!

Mixer

- full name: tenpy.algorithms.dmrg.Mixer
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**Mixer**(*mixer_params*)

Bases: object

Base class of a general Mixer.

Since DMRG performs only local updates of the state, it can get stuck in "local minima", in particular if the Hamiltonain is long-range – which is the case if one maps a 2D system ("infinite cylinder") to 1D – or if one wants to do single-site updates (currently not implemented in TeNPy). The idea of the mixer is to perturb the state with the terms of the Hamiltonian which have contributions in both the "left" and "right" side of the system. In that way, it adds fluctuation of the quantum numbers and non-zero contributions of the long-range terms - leading to a significantly improved convergence of DMRG.

The strength of the perturbation is given by the *amplitude* of the mixer. A good strategy is to choose an initially significant amplitude and let it decay until the perturbation becomes completely irrelevant and the mixer gets disabled.

This original idea of the mixer was introduced in [White2005]. [Hubig2015] discusses the mixer and provides an improved version.

Parameters

env [*MPOEnvironment*] Environment for contraction `<psi|H|psi>` for later

mixer_params [dict] Optional parameters as described in the following table. Use `verbose>0` to print the used parameters during runtime.

key	type	description
amplitude	float	Initial strength of the mixer. (Should be << 1.)
decay	float	To slowly turn off the mixer, we divide *amplitude* by *decay* after each sweep. (Should be >= 1.)
disable_after	int	We disable the mixer completely after this number of sweeps.

Attributes

amplitude [float] Current amplitude for mixing.

decay [float] Factor by which *amplitude* is divided after each sweep.

disable_after [int] The number of sweeps after which the mixer should be disabled.

verbose [int] Level of output vebosity.

Methods

perturb_svd(self, engine, theta, i0, ...)	Perturb the wave function and perform an SVD with truncation.
update_amplitude(self, sweeps)	Update the amplitude, possibly disable the mixer.

update_amplitude (*self*, *sweeps*)
Update the amplitude, possibly disable the mixer.

Parameters

sweeps [int] The number of performed sweeps, to check if we need to disable the mixer.

Returns

mixer [*Mixer* | None] Returns *self* if we should continue mixing, or `None`, if the mixer should be disabled.

perturb_svd (*self*, *engine*, *theta*, *i0*, *update_LP*, *update_RP*)
Perturb the wave function and perform an SVD with truncation.

Parameters

engine [`Engine`] The DMRG engine calling the mixer.

theta [*Array*] The optimized wave function, prepared for svd.

i0 [int] Site index; *theta* lives on `i0, i0+1`.

update_LP [bool] Whether to calculate the next `env.LP[i0+1]`.

update_RP [bool] Whether to calculate the next `env.RP[i0]`.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(vR.p1)'`.

err [*TruncationError*] The truncation error introduced.

SingleSiteDMRGEngine

- full name: tenpy.algorithms.dmrg.SingleSiteDMRGEngine
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**SingleSiteDMRGEngine**(*psi*, *model*, *engine_params*)
Bases: *tenpy.algorithms.dmrg.DMRGEngine*

'Engine' for the single-site DMRG algorithm.

Parameters

psi [*MPS*] Initial guess for the ground state, which is to be optimized in-place.

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state.

engine_params [dict] Further optional parameters. These are usually algorithm-specific, and thus should be described in subclasses.

Attributes

EffectiveH [class type] Class for the effective Hamiltonian (i.e., a subclass of *EffectiveH*. Has a *length* class attribute which specifies the number of sites updated at once (e.g., whether we do single-site vs. two-site DMRG).

chi_list [dict | None] A dictionary to gradually increase the *chi_max* parameter of *trunc_params*. The key defines starting from which sweep *chi_max* is set to the value, e.g. {0: 50, 20: 100} uses chi_max=50 for the first 20 sweeps and chi_max=100 afterwards. Overwrites *trunc_params['chi_list']*`. By default (None) this feature is disabled.

eff_H [*EffectiveH*] Effective two-site Hamiltonian.

mixer [*Mixer* | None] If None, no mixer is used (anymore), otherwise the mixer instance.

shelve [bool] If a simulation runs out of time (*time.time() - start_time > max_seconds*), the run will terminate with *shelve = True*.

sweeps [int] The number of sweeps already performed. (Useful for re-start).

time0 [float] Time marker for the start of the run.

update_stats [dict] A dictionary with detailed statistics of the convergence. For each key in the following table, the dictionary contains a list where one value is added each time Engine.update_bond() is called.

key	description
i0	An update was performed on sites i0, i0+1.
age	The number of physical sites involved in the simulation.
E_total	The total energy before truncation.
N_lanczos	Dimension of the Krylov space used in the lanczos diagonalization.
time	Wallclock time evolved since time0 (in seconds).

sweep_stats [dict] A dictionary with detailed statistics of the convergence. For each key in the following table, the dictionary contains a list where one value is added each time Engine.sweep() is called (with optimize=True).

key	description
sweep	Number of sweeps performed so far.
E	The energy *before* truncation (as calculated by Lanczos).
S	Maximum entanglement entropy.
time	Wallclock time evolved since `time0` (in seconds).
max_trunc_err	The maximum truncation error in the last sweep
max_E_trunc	Maximum change or Energy due to truncation in the last sweep.
max_chi	Maximum bond dimension used.
norm_err	Error of canonical form `np.linalg.norm(psi.norm_test())`.

Methods

diag(self, theta_guess)	Diagonalize the effective Hamiltonian represented by self.
environment_sweeps(self, N_sweeps)	Perform *N_sweeps* sweeps without optimization to update the environment.
get_sweep_schedule(self)	Define the schedule of the sweep.
init_env(self[, model])	(Re-)initialize the environment.
mixed_svd(self, theta, next_B)	Get (truncated) *B* from the new theta (as returned by diag).
mixer_activate(self)	Set *self.mixer* to the class specified by *engine_params['mixer']*.
mixer_cleanup(self)	Cleanup the effects of a mixer.
plot_sweep_stats(self[, axes, xaxis, yaxis, ...])	Plot `sweep_stats` to display the convergence with the sweeps.
plot_update_stats(self, axes[, xaxis, ...])	Plot `update_stats` to display the convergence during the sweeps.
post_update_local(self, update_data[, ...])	Perform post-update actions.
prepare_svd(self, theta)	Transform theta into matrix for svd.
prepare_update(self)	Prepare *self* to represent the effective Hamiltonian on site `i0`.
reset_stats(self)	Reset the statistics, useful if you want to start a new sweep run.
run(self)	Run the DMRG simulation to find the ground state.
set_B(self, U, S, VH)	Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.
sweep(self[, optimize, meas_E_trunc])	One 'sweep' of a sweeper algorithm.
update_LP(self, U)	Update left part of the environment.
update_RP(self, VH)	Update right part of the environment.
update_local(self, theta[, optimize, ...])	Perform site-update on the site `i0`.

prepare_update(*self*)
Prepare *self* to represent the effective Hamiltonian on site `i0`.

Returns

theta [*Array*] Current best guess for the ground state, which is to be optimized. Labels `'vL', 'p0', 'vR'`, or combined versions of it (if *self.combine*).

update_local(*self*, *theta*, *optimize=True*, *meas_E_trunc=False*)
Perform site-update on the site `i0`.

Parameters

theta [*Array*] Initial guess for the ground state of the effective Hamiltonian.

optimize [bool] Wheter we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool] Wheter to measure the energy after truncation.

Returns

update_data [dict] Data computed during the local update, as described in the following:

E0 [float] Total energy, obtained *before* truncation (if `optimize=True`), or *after* truncation (if `optimize=False`) (but never `None`).

N [int] Dimension of the Krylov space used for optimization in the lanczos algorithm. 0 if `optimize=False`.

age [int] Current size of the DMRG simulation: number of physical sites involved into the contraction.

U, VH: ***Array*** *U* and *VH* returned by *mixed_svd()*.

ov_change: float Change in the wave function `1. - abs(<theta_guess|theta>)` induced by *diag()*, *not* including the truncation!

prepare_svd(*self*, *theta*)

Transform theta into matrix for svd.

In contrast with the 2-site engine, the matrix here depends on the direction we move, as we need *'p'* to point away from the direction we are going in.

mixed_svd(*self*, *theta*, *next_B*)

Get (truncated) *B* from the new theta (as returned by diag).

The goal is to split theta and truncate it. For a move to the right:

```
|   -- theta -- next_B --    ==>    -- U -- S -- VH -- next_B --
|        |      |                       |               |
```

For a move to the left:

```
|   -- next_B -- theta -- ==>    -- next_B -- U -- S -- VH --
|      |          |                  |                  |
```

The *VH* for right-move or *U* for left-move is absorebed into the *next_B*.

Without a mixer, this is done by a simple svd and truncation of Schmidt values of theta followed by the absorption of VH/U.

With a mixer, the state is perturbed before the SVD. The details of the perturbation are defined by the *Mixer* class.

Parameters

theta [*Array*] The optimized wave function, prepared for svd with *prepare_svd()*, i.e. with combined legs.

nextB [*Array*] MPS tensor at the site that will be visited next.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix with labels `'vL', 'vR'`; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(p0.vR)'`.

err [*TruncationError*] The truncation error introduced.

set_B(*self*, *U*, *S*, *VH*)
Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.

Parameters

U, VH [*Array*] Left and Right-canonical matrices as returned by the SVD.

S [1D array | 2D *Array*] The middle part returned by the SVD, `theta = U S VH`. Without a mixer just the singular values, with enabled *mixer* a 2D array.

mixer_activate(*self*)
Set *self.mixer* to the class specified by *engine_params['mixer']*.

update_LP(*self*, *U*)
Update left part of the environment.

The site at which to update the environment depends on the direction of the sweep. If we are sweeping right, update the invironment at *i0+1*. If we are sweeping left, update the environment at *i0*

Parameters

U [*Array*] The U as returned by SVD, with combined legs, labels `'(vL.p0)', 'vR'` if self.move_right, else `'vL', '(p0.vR)'`.

update_RP(*self*, *VH*)
Update right part of the environment.

The site at which to update the environment depends on the direction of the sweep. If we are sweeping right, update the invironment at *i0*. If we are sweeping left, update the environment at *i0-1*

Parameters

VH [*Array*] The VH as returned by SVD, with combined legs, labels `'(vL.p0)', 'vR'` if self.move_right, else `'vL', '(p0.vR)'`.

diag(*self*, *theta_guess*)
Diagonalize the effective Hamiltonian represented by self.

The method used depends on the DMRG parameter *diag_method*.

diag_method	Function, comment
'lanczos'	*lanczos()* Default, the Lanczos implementation of TeNPy
'arpack'	*lanczos_arpack()* Based on `scipy.linalg.sparse.eigsh()`. Slower than 'lanczos', since it needs to convert the npc arrays to numpy arrays during *each* matvec, and possibly does many more iterations.
'ED_block'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize the block in the charge sector of the initial state. Preserves the charge sector of the explicitly conserved charges. However, if you don't preserve a charge explicitly, it can break it. For example if you use a `SpinChain({'conserve': 'parity'})`, it could change the total "Sz", but not the parity of 'Sz'.
'ED_all'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize it completely. Allows to change the charge sector *even for explicitly conserved charges*. For example if you use a `SpinChain({'conserve': 'Sz'})`, it **can** change the total "Sz".

Parameters

theta_guess [*Array*] Initial guess for the ground state of the effective Hamiltonian.

Returns

E0 [float] Energy of the found ground state.

theta [*Array*] Ground state of the effective Hamiltonian.

N [int] Number of Lanczos iterations used. `-1` if unknown.

ov_change [float] Change in the wave function `1. - abs(<theta_guess|theta_diag>)`

environment_sweeps(*self*, *N_sweeps*)
Perform *N_sweeps* sweeps without optimization to update the environment.

Parameters

N_sweeps [int] Number of sweeps to run without optimization

get_sweep_schedule(*self*)
Define the schedule of the sweep.

One 'sweep' is a full sequence from the leftmost site to the right and back. Only those *LP* and *RP* that can be used later should be updated.

Returns

schedule [iterable of (int, bool, (bool, bool))] Schedule for the sweep. Each entry is `(i0, move_right, (update_LP, update_RP))`, where *i0* is the leftmost of the `self.EffectiveH.length` sites to be updated in *update_local()*, *move_right* indicates whether the next *i0* in the schedule is rigth (*True*) of the current one, and *update_LP*, *update_RP* indicate whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory.

init_env(*self*, *model=None*)
(Re-)initialize the environment.

This function is useful to (re-)start a Sweep with a slightly different model or different (engine) parameters. Note that we assume that we still have the same *psi*. Calls *reset_stats()*.

Parameters

model [`MPOModel`] The model representing the Hamiltonian for which we want to find the ground state. If `None`, keep the model used before.

Raises

ValueError If the engine is re-initialized with a new model, which legs are incompatible with those of hte old model.

mixer_cleanup(*self*)
Cleanup the effects of a mixer.

A *sweep()* with an enabled *Mixer* leaves the MPS *psi* with 2D arrays in *S*. To recover the originial form, this function simply performs one sweep with disabled mixer.

plot_sweep_stats(*self*, *axes=None*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot `sweep_stats` to display the convergence with the sweeps.

Parameters

axes [matplotlib.axes.Axes] The axes to plot into. Defaults to matplotlib.pyplot.gca()

xaxis, yaxis [key of sweep_stats] Key of sweep_stats to be used for the x-axis and y-axis of the plots.

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot abs((y-y_exact)/y_exact) on a log-scale yaxis.

****kwargs :** Further keyword arguments given to axes.plot(...).

plot_update_stats(*self*, *axes*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot update_stats to display the convergence during the sweeps.

Parameters

axes [matplotlib.axes.Axes] The axes to plot into. Defaults to matplotlib.pyplot.gca()

xaxis ['N_updates' | 'sweep' | keys of update_stats] Key of update_stats to be used for the x-axis of the plots. 'N_updates' is just enumerating the number of bond updates, and 'sweep' corresponds to the sweep number (including environment sweeps).

yaxis ['E' | keys of update_stats] Key of update_stats to be used for the y-axisof the plots. For 'E', use the energy (per site for infinite systems).

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot abs((y-y_exact)/y_exact) on a log-scale yaxis.

****kwargs :** Further keyword arguments given to axes.plot(...).

post_update_local(*self*, *update_data*, *meas_E_trunc=False*)
Perform post-update actions.

Compute truncation energy, remove *LP*/*RP* that are no longer needed and collect statistics.

Parameters

update_data [dict] Data computed during the local update, as described in the following list.

meas_E_trunc [bool, optional] Wheter to measure the energy after truncation.

reset_stats(*self*)
Reset the statistics, useful if you want to start a new sweep run.

run(*self*)
Run the DMRG simulation to find the ground state.

Returns

E [float] The energy of the resulting ground state MPS.

psi [*MPS*] The MPS representing the ground state after the simluation, i.e. just a reference to psi.

sweep(*self*, *optimize=True*, *meas_E_trunc=False*)
One 'sweep' of a sweeper algorithm.

Iteratate over the bond which is optimized, to the right and then back to the left to the starting point. If optimize=False, don't actually diagonalize the effective hamiltonian, but only update the environment.

Parameters

optimize [bool, optional] Whether we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool, optional] Whether to measure truncation energies.

Returns

max_trunc_err [float] Maximal truncation error introduced.

max_E_trunc [`None` | float] `None` if meas_E_trunc is False, else the maximal change of the energy due to the truncation.

SingleSiteMixer

- full name: tenpy.algorithms.dmrg.SingleSiteMixer
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**SingleSiteMixer**(*mixer_params*)
Bases: *tenpy.algorithms.dmrg.Mixer*

Mixer for single-site DMRG.

Performs a subspace expansion following [Hubig2015].

Methods

perturb_svd(self, engine, theta, i0, ...)	Mix extra terms to theta and perform an SVD.
subspace_expand(self, engine, theta, i0, ...)	Expand the MPS subspace, to allow the bond dimension to increase.
update_amplitude(self, sweeps)	Update the amplitude, possibly disable the mixer.

perturb_svd(*self*, *engine*, *theta*, *i0*, *move_right*, *next_B*)
Mix extra terms to theta and perform an SVD.

We calculate the left and right reduced density matrix using the mixer (which might include applications of H). These density matrices are diagonalized and truncated such that we effectively perform a svd for the case `mixer.amplitude=0`.

Parameters

engine [`Engine`] The DMRG engine calling the mixer.

theta [*Array*] The optimized wave function, prepared for svd.

i0 [int] The site index where *theta* lives.

move_right [bool] Whether we move to the right (`True`) or left (`False`).

next_B [*Array*] The subspace expansion requires to change the tensor on the next site as well. If *move_right*, it should correspond to `engine.psi.get_B(i0+1, form='B')`. If not *move_right*, it should correspond to `engine.psi.get_B(i0-1, form='A')`.

Returns

U [*Array*] Left-canonical part of *tensordot(theta, next_B)*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray] (Perturbed) singular values on the new bond (between *theta* and *next_B*).

VH [*Array*] Right-canonical part of *tensordot(theta, next_B)*. Labels `'vL', '(p1.vR)'`.

err [*TruncationError*] The truncation error introduced.

subspace_expand(*self*, *engine*, *theta*, *i0*, *move_right*, *next_B*)
Expand the MPS subspace, to allow the bond dimension to increase.

This is the subspace expansion following [Hubig2015].

Parameters

engine [*SingleSiteDMRGEngine* | *TwoSiteDMRGEngine*] 'Engine' for the DMRG algorithm

theta [*Array*] Optimized guess for the ground state of the effective local Hamiltonian.

i0 [int] Site index at which the local update has taken place.

move_right [bool] Whether the next *i0* of the sweep will be right or left of the current one.

next_B [*Array*] The subspace expansion requires to change the tensor on the next site as well. If *move_right*, it should correspond to `engine.psi.get_B(i0+1, form='B')`. If not *move_right*, it should correspond to `engine.psi.get_B(i0-1, form='A')`.

Returns

theta : Local MPS tensor at site *i0* after subspace expansion.

next_B : MPS tensor at site *i0+1* or *i0-1* (depending on sweep direction) after subspace expansion.

update_amplitude(*self*, *sweeps*)
Update the amplitude, possibly disable the mixer.

Parameters

sweeps [int] The number of performed sweeps, to check if we need to disable the mixer.

Returns

mixer [*Mixer* | None] Returns *self* if we should continue mixing, or `None`, if the mixer should be disabled.

TwoSiteDMRGEngine

- full name: tenpy.algorithms.dmrg.TwoSiteDMRGEngine
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**TwoSiteDMRGEngine**(*psi*, *model*, *engine_params*)
Bases: *tenpy.algorithms.dmrg.DMRGEngine*

'Engine' for the two-site DMRG algorithm.

Parameters

psi [*MPS*] Initial guess for the ground state, which is to be optimized in-place.

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state.

engine_params [dict] Further optional parameters. These are usually algorithm-specific, and thus should be described in subclasses.

Attributes

EffectiveH [class type] Class for the effective Hamiltonian (i.e., a subclass of *EffectiveH*. Has a *length* class attribute which specifies the number of sites updated at once (e.g., whether we do single-site vs. two-site DMRG).

chi_list [dict | None] A dictionary to gradually increase the *chi_max* parameter of *trunc_params*. The key defines starting from which sweep *chi_max* is set to the value, e.g. {0: 50, 20: 100} uses chi_max=50 for the first 20 sweeps and chi_max=100 afterwards. Overwrites *trunc_params['chi_list']*`. By default (None) this feature is disabled.

eff_H [*EffectiveH*] Effective two-site Hamiltonian.

mixer [*Mixer* | None] If None, no mixer is used (anymore), otherwise the mixer instance.

shelve [bool] If a simulation runs out of time (*time.time() - start_time* > *max_seconds*), the run will terminate with *shelve* = *True*.

sweeps [int] The number of sweeps already performed. (Useful for re-start).

time0 [float] Time marker for the start of the run.

update_stats [dict] A dictionary with detailed statistics of the convergence. For each key in the following table, the dictionary contains a list where one value is added each time Engine.update_bond() is called.

key	description
i0	An update was performed on sites i0, i0+1.
age	The number of physical sites involved in the simulation.
E_total	The total energy before truncation.
N_lanczos	Dimension of the Krylov space used in the lanczos diagonalization.
time	Wallclock time evolved since time0 (in seconds).

sweep_stats [dict] A dictionary with detailed statistics of the convergence. For each key in the following table, the dictionary contains a list where one value is added each time Engine.sweep() is called (with optimize=True).

key	description
sweep	Number of sweeps performed so far.
E	The energy *before* truncation (as calculated by Lanczos).
S	Maximum entanglement entropy.
time	Wallclock time evolved since time0 (in seconds).
max_trunc_err	The maximum truncation error in the last sweep
max_E_trunc	Maximum change or Energy due to truncation in the last sweep.
max_chi	Maximum bond dimension used.
norm_err	Error of canonical form np.linalg.norm(psi.norm_test()).

Methods

diag(self, theta_guess)	Diagonalize the effective Hamiltonian represented by self.
environment_sweeps(self, N_sweeps)	Perform *N_sweeps* sweeps without optimization to update the environment.
get_sweep_schedule(self)	Define the schedule of the sweep.
init_env(self[, model])	(Re-)initialize the environment.
mixed_svd(self, theta)	Get (truncated) *B* from the new theta (as returned by diag).
mixer_activate(self)	Set *self.mixer* to the class specified by *engine_params['mixer']*.
mixer_cleanup(self)	Cleanup the effects of a mixer.
plot_sweep_stats(self[, axes, xaxis, yaxis, ...])	Plot `sweep_stats` to display the convergence with the sweeps.
plot_update_stats(self, axes[, xaxis, ...])	Plot `update_stats` to display the convergence during the sweeps.
post_update_local(self, update_data[, ...])	Perform post-update actions.
prepare_svd(self, theta)	Transform theta into matrix for svd.
prepare_update(self)	Prepare *self* to represent the effective Hamiltonian on sites `(i0, i0+1)`.
reset_stats(self)	Reset the statistics, useful if you want to start a new sweep run.
run(self)	Run the DMRG simulation to find the ground state.
set_B(self, U, S, VH)	Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.
sweep(self[, optimize, meas_E_trunc])	One 'sweep' of a sweeper algorithm.
update_LP(self, U)	Update left part of the environment.
update_RP(self, VH)	Update right part of the environment.
update_local(self, theta[, optimize, ...])	Perform bond-update on the sites `(i0, i0+1)`.

prepare_update(*self*)
Prepare *self* to represent the effective Hamiltonian on sites `(i0, i0+1)`.

Returns

theta [*Array*] Current best guess for the ground state, which is to be optimized. Labels `'vL', 'p0', 'vR', 'p1'`.

update_local(*self*, *theta*, *optimize=True*, *meas_E_trunc=False*)
Perform bond-update on the sites `(i0, i0+1)`.

Parameters

theta [*Array*] Initial guess for the ground state of the effective Hamiltonian.

optimize [bool] Wheter we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool] Wheter to measure the energy after truncation.

Returns

update_data [dict] Data computed during the local update, as described in the following:

E0 [float] Total energy, obtained *before* truncation (if `optimize=True`), or *after* truncation (if `optimize=False`) (but never `None`).

N [int] Dimension of the Krylov space used for optimization in the lanczos algorithm. 0 if `optimize=False`.

age [int] Current size of the DMRG simulation: number of physical sites involved into the contraction.

U, VH: ***Array*** *U* and *VH* returned by *mixed_svd()*.

ov_change: float Change in the wave function `1. - abs(<theta_guess|theta>)` induced by *diag()*, *not* including the truncation!

prepare_svd(*self*, *theta*)
Transform theta into matrix for svd.

mixed_svd(*self*, *theta*)
Get (truncated) *B* from the new theta (as returned by diag).

The goal is to split theta and truncate it:

```
|   -- theta --   ==>    -- U -- S --  VH -
|      |   |               |          |
```

Without a mixer, this is done by a simple svd and truncation of Schmidt values.

With a mixer, the state is perturbed before the SVD. The details of the perturbation are defined by the *Mixer* class.

Note that the returned *S* is a general (not diagonal) matrix, with labels `'vL', 'vR'`.

Parameters

theta [*Array*] The optimized wave function, prepared for svd.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix with labels `'vL', 'vR'`; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(p1.vR)'`.

err [*TruncationError*] The truncation error introduced.

set_B(*self*, *U*, *S*, *VH*)
Update the MPS with the `U, S, VH` returned by *self.mixed_svd*.

Parameters

U, VH [*Array*] Left and Right-canonical matrices as returned by the SVD.

S [1D array | 2D *Array*] The middle part returned by the SVD, `theta = U S VH`. Without a mixer just the singular values, with enabled *mixer* a 2D array.

mixer_activate(*self*)
Set *self.mixer* to the class specified by *engine_params['mixer']*.

update_LP(*self*, *U*)
Update left part of the environment.

We always update the environment at site i0 + 1: this environment then contains the site where we just performed a local update (when sweeping right).

Parameters

U [*Array*] The U as returned by the SVD, with combined legs, labels `'vL.p0', 'vR'`.

update_RP (*self*, *VH*)
Update right part of the environment.

We always update the environment at site i0: this environment then contains the site where we just performed a local update (when sweeping left).

Parameters

VH [*Array*] The VH as returned by SVD, with combined legs, labels `'vL', '(vR.p1)'`.

diag (*self*, *theta_guess*)
Diagonalize the effective Hamiltonian represented by self.

The method used depends on the DMRG parameter *diag_method*.

diag_method	Function, comment
'lanczos'	*lanczos()* Default, the Lanczos implementation of TeNPy
'arpack'	*lanczos_arpack()* Based on `scipy.linalg.sparse.eigsh()`. Slower than 'lanczos', since it needs to convert the npc arrays to numpy arrays during *each* matvec, and possibly does many more iterations.
'ED_block'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize the block in the charge sector of the initial state. Preserves the charge sector of the explicitly conserved charges. However, if you don't preserve a charge explicitly, it can break it. For example if you use a `SpinChain({'conserve': 'parity'})`, it could change the total "Sz", but not the parity of 'Sz'.
'ED_all'	*full_diag_effH()* Contract the effective Hamiltonian to a (large!) matrix and diagonalize it completely. Allows to change the charge sector *even for explicitly conserved charges*. For example if you use a `SpinChain({'conserve': 'Sz'})`, it **can** change the total "Sz".

Parameters

theta_guess [*Array*] Initial guess for the ground state of the effective Hamiltonian.

Returns

E0 [float] Energy of the found ground state.

theta [*Array*] Ground state of the effective Hamiltonian.

N [int] Number of Lanczos iterations used. `-1` if unknown.

ov_change [float] Change in the wave function `1. - abs(<theta_guess|theta_diag>)`

environment_sweeps (*self*, *N_sweeps*)
Perform *N_sweeps* sweeps without optimization to update the environment.

Parameters

N_sweeps [int] Number of sweeps to run without optimization

get_sweep_schedule (*self*)
Define the schedule of the sweep.

One 'sweep' is a full sequence from the leftmost site to the right and back. Only those *LP* and *RP* that can be used later should be updated.

Returns

schedule [iterable of (int, bool, (bool, bool))] Schedule for the sweep. Each entry is `(i0, move_right, (update_LP, update_RP))`, where *i0* is the leftmost of the `self.EffectiveH.length` sites to be updated in *update_local()*, *move_right* indicates whether the next *i0* in the schedule is rigth (*True*) of the current one, and *update_LP*, *update_RP* indicate whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory.

init_env(*self*, *model=None*)
(Re-)initialize the environment.

This function is useful to (re-)start a Sweep with a slightly different model or different (engine) parameters. Note that we assume that we still have the same *psi*. Calls *reset_stats()*.

Parameters

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state. If None, keep the model used before.

Raises

ValueError If the engine is re-initialized with a new model, which legs are incompatible with those of hte old model.

mixer_cleanup(*self*)
Cleanup the effects of a mixer.

A *sweep()* with an enabled *Mixer* leaves the MPS *psi* with 2D arrays in *S*. To recover the originial form, this function simply performs one sweep with disabled mixer.

plot_sweep_stats(*self*, *axes=None*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot `sweep_stats` to display the convergence with the sweeps.

Parameters

axes [matplotlib.axes.Axes] The axes to plot into. Defaults to `matplotlib.pyplot.gca()`

xaxis, yaxis [key of `sweep_stats`] Key of `sweep_stats` to be used for the x-axis and y-axis of the plots.

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot `abs((y-y_exact)/y_exact)` on a log-scale yaxis.

****kwargs :** Further keyword arguments given to `axes.plot(...)`.

plot_update_stats(*self*, *axes*, *xaxis='time'*, *yaxis='E'*, *y_exact=None*, ***kwargs*)
Plot `update_stats` to display the convergence during the sweeps.

Parameters

axes [matplotlib.axes.Axes] The axes to plot into. Defaults to `matplotlib.pyplot.gca()`

xaxis [`'N_updates' | 'sweep' |` keys of `update_stats`] Key of `update_stats` to be used for the x-axis of the plots. `'N_updates'` is just enumerating the number of bond updates, and `'sweep'` corresponds to the sweep number (including environment sweeps).

yaxis [`'E' |` keys of `update_stats`] Key of `update_stats` to be used for the y-axisof the plots. For 'E', use the energy (per site for infinite systems).

y_exact [float] Exact value for the quantity on the y-axis for comparison. If given, plot `abs((y-y_exact)/y_exact)` on a log-scale yaxis.

****kwargs :** Further keyword arguments given to `axes.plot(...)`.

post_update_local(*self*, *update_data*, *meas_E_trunc=False*)
Perform post-update actions.

Compute truncation energy, remove *LP*/*RP* that are no longer needed and collect statistics.

Parameters

update_data [dict] Data computed during the local update, as described in the following list.

meas_E_trunc [bool, optional] Wheter to measure the energy after truncation.

reset_stats(*self*)
Reset the statistics, useful if you want to start a new sweep run.

run(*self*)
Run the DMRG simulation to find the ground state.

Returns

E [float] The energy of the resulting ground state MPS.

psi [*MPS*] The MPS representing the ground state after the simluation, i.e. just a reference to `psi`.

sweep(*self*, *optimize=True*, *meas_E_trunc=False*)
One 'sweep' of a sweeper algorithm.

Iteratate over the bond which is optimized, to the right and then back to the left to the starting point. If optimize=False, don't actually diagonalize the effective hamiltonian, but only update the environment.

Parameters

optimize [bool, optional] Whether we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool, optional] Whether to measure truncation energies.

Returns

max_trunc_err [float] Maximal truncation error introduced.

max_E_trunc [`None` | float] `None` if meas_E_trunc is False, else the maximal change of the energy due to the truncation.

TwoSiteMixer

- full name: tenpy.algorithms.dmrg.TwoSiteMixer
- parent module: *tenpy.algorithms.dmrg*
- type: class

class tenpy.algorithms.dmrg.**TwoSiteMixer**(*mixer_params*)
Bases: *tenpy.algorithms.dmrg.SingleSiteMixer*

Mixer for two-site DMRG.

This is the two-site version of the mixer described in [Hubig2015]. Equivalent to the *DensityMatrixMixer*, but never construct the full density matrix.

Methods

perturb_svd(self, engine, theta, i0, move_right)	Mix extra terms to theta and perform an SVD.
subspace_expand(self, engine, theta, i0, ...)	Expand the MPS subspace, to allow the bond dimension to increase.
update_amplitude(self, sweeps)	Update the amplitude, possibly disable the mixer.

perturb_svd(*self*, *engine*, *theta*, *i0*, *move_right*)
Mix extra terms to theta and perform an SVD.

Parameters

engine [`Engine`] The DMRG engine calling the mixer.

theta [*Array*] The optimized wave function, prepared for svd.

i0 [int] Site index; *theta* lives on `i0, i0+1`.

update_LP [bool] Whether to calculate the next `env.LP[i0+1]`.

update_RP [bool] Whether to calculate the next `env.RP[i0]`.

Returns

U [*Array*] Left-canonical part of *theta*. Labels `'(vL.p0)', 'vR'`.

S [1D ndarray | 2D *Array*] Without mixer just the singluar values of the array; with mixer it might be a general matrix; see comment above.

VH [*Array*] Right-canonical part of *theta*. Labels `'vL', '(vR.p1)'`.

err [*TruncationError*] The truncation error introduced.

subspace_expand(*self*, *engine*, *theta*, *i0*, *move_right*, *next_B*)
Expand the MPS subspace, to allow the bond dimension to increase.

This is the subspace expansion following [Hubig2015].

Parameters

engine [*SingleSiteDMRGEngine* | *TwoSiteDMRGEngine*] 'Engine' for the DMRG algorithm

theta [*Array*] Optimized guess for the ground state of the effective local Hamiltonian.

i0 [int] Site index at which the local update has taken place.

move_right [bool] Whether the next *i0* of the sweep will be right or left of the current one.

next_B [*Array*] The subspace expansion requires to change the tensor on the next site as well. If *move_right*, it should correspond to `engine.psi.get_B(i0+1, form='B')`. If not *move_right*, it should correspond to `engine.psi.get_B(i0-1, form='A')`.

Returns

theta : Local MPS tensor at site *i0* after subspace expansion.

next_B : MPS tensor at site *i0+1* or *i0-1* (depending on sweep direction) after subspace expansion.

update_amplitude(*self*, *sweeps*)
Update the amplitude, possibly disable the mixer.

Parameters

sweeps [int] The number of performed sweeps, to check if we need to disable the mixer.

Returns

mixer [*Mixer* | None] Returns *self* if we should continue mixing, or `None`, if the mixer should be disabled.

Functions

chi_list(chi_max[, dchi, nsweeps])	Compute a 'ramping-up' chi_list.
full_diag_effH(effH, theta_guess[, keep_sector])	Perform an exact diagonalization of *effH*.
run(psi, model, DMRG_params)	Run the DMRG algorithm to find the ground state of the given model.

chi_list

- full name: tenpy.algorithms.dmrg.chi_list
- parent module: *tenpy.algorithms.dmrg*
- type: function

tenpy.algorithms.dmrg.**chi_list**(*chi_max*, *dchi=20*, *nsweeps=20*)
Compute a 'ramping-up' chi_list.

The resulting chi_list allows to increases *chi* by *dchi* every *nsweeps* sweeps up to a given maximal *chi_max*.

Parameters

chi_max [int] Final value for the bond dimension.

dchi :int Step size how to increase chi

nsweeps [int] Step size for sweeps

Returns

chi_list [dict] To be used as *chi_list* parameter for DMRG, see *run()*. Keys increase by *nsweeps*, values by *dchi*, until a maximum of *chi_max* is reached.

full_diag_effH

- full name: tenpy.algorithms.dmrg.full_diag_effH
- parent module: *tenpy.algorithms.dmrg*
- type: function

tenpy.algorithms.dmrg.**full_diag_effH**(*effH*, *theta_guess*, *keep_sector=True*)
Perform an exact diagonalization of *effH*.

This function offers an alternative to *lanczos()*.

Parameters

effH [*EffectiveH*] The effective Hamiltonian.

theta_guess [*Array*] Current guess to select the charge sector. Labels as specified by `effH.acts_on`.

run

- full name: tenpy.algorithms.dmrg.run
- parent module: *tenpy.algorithms.dmrg*
- type: function

tenpy.algorithms.dmrg.**run**(*psi*, *model*, *DMRG_params*)

Run the DMRG algorithm to find the ground state of the given model.

Parameters

psi [*MPS*] Initial guess for the ground state, which is to be optimized in-place.

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state.

DMRG_params [dict] Further optional parameters as described in the following table. Use verbose>0 to print the used parameters during runtime.

key	type	description
LP	npc.Array	Initial left-most *LP* and right-most *RP* ('left/right part')
RP		of the environment. By default (None) generate trivial, see *MPOEnvironment* for details.
LP_age	int	The 'age' (i.e. number of physical sites invovled into the
RP_age		contraction) of the left-most *LP* and right-most *RP* of the environment.
mixer	str | class | bool	Chooses the *Mixer* to be used. A string stands for one of the mixers defined in this module, a
mixer_params	dict	Non-default initialization arguments of the mixer. Options may be custom to the specified mixe
orthogonal_to	list of MPS	List of other matrix produc states to orthogonalize against. Works only for finite systems. This
combine	bool	Whether to combine legs into pipes. This combines the virtual and physical leg for the left site
trunc_params	dict	Truncation parameters as described in *truncate()*
chi_list	dict | None	A dictionary to gradually increase the *chi_max* parameter of *trunc_params*. The key defines sta
lanczos_params	dict	Lanczos parameters as described in *lanczos()*
diag_method	str	Method to be used for diagonalzation, default 'lanczos'. For possible arguments see *DMRG*
N_sweeps_check	int	Number of sweeps to perform between checking convergence criteria and giving a status updat
sweep_0	int	The number of sweeps already performed. (Useful for re-start).
start_env	int	Number of initial sweeps performed without bond optimizaiton to initialize the environment.
update_env	int	Number of sweeps without bond optimizaiton to update the environment for infinite boundary c
norm_tol	float	After the DMRG run, update the environment with at most *norm_tol_iter* sweeps until np.lin
norm_tol_iter	float	Perform at most *norm_tol_iter*`*`*update_env* sweeps to converge the norm error below *norm_to*
max_sweeps	int	Maximum number of sweeps to be performed.
min_sweeps	int	Minimum number of sweeps to be performed. Defaults to 1.5*N_sweeps_check.
max_E_err	float	Convergence if the change of the energy in each step satisfies -Delta E / max(|E|, 1)
max_S_err	float	Convergence if the relative change of the entropy in each step satisfies |Delta S|/S < ma
max_hours	float	If the DMRG took longer (measured in wall-clock time), 'shelve' the simulation, i.e. stop and r
P_tol_to_trunc	float	It's reasonable to choose the Lanczos convergence criteria
P_tol_max		'P_tol' not many magnitudes lower than the current
P_tol_min		truncation error. Therefore, if *P_tol_to_trunc* is not None, we update *P_tol* of *lanczos_params*
E_tol_to_trunc	float	It's reasonable to choose the Lanczos convergence criteria
E_tol_max		'E_tol' not many magnitudes lower than the current
E_tol_min		truncation error. Therefore, if *E_tol_to_trunc* is not None, we update *E_tol* of *lanczos_params*
active_sites	int	The number of active sites to be used by DMRG. If set to 1, *SingleSiteDMRGEngine* is us

Returns

info [dict] A dictionary with keys 'E', 'shelve', 'bond_statistics',

```
'sweep_statistics'
```

Module description

Density Matrix Renormalization Group (DMRG).

Although it was originally not formulated with tensor networks, the DMRG algorithm (invented by Steven White in 1992 [White1992]) opened the whole field with its enormous success in finding ground states in 1D.

We implement DMRG in the modern formulation of matrix product states [Schollwoeck2011], both for finite systems (`'finite'` or `'segment'` boundary conditions) and in the thermodynamic limit (`'infinite'` b.c.).

The function *run()* - well - runs one DMRG simulation. Internally, it generates an instance of an `Sweep`. This class implements the common functionality like defining a *sweep*, but leaves the details of the contractions to be performed to the derived classes.

Currently, there are two derived classes implementing the contractions: *SingleSiteDMRGEngine* and *TwoSiteDMRGEngine*. They differ (as their name implies) in the number of sites which are optimized simultaneously. They should both give the same results (up to rounding errors). However, if started from a product state, *SingleSiteDMRGEngine* depends critically on the use of a *Mixer*, while *TwoSiteDMRGEngine* is in principle more computationally expensive to run and has occasionally displayed some convergence issues.. Which one is preffered in the end is not obvious a priori and might depend on the used model. Just try both of them.

A *Mixer* should be used initially to avoid that the algorithm gets stuck in local energy minima, and then slowly turned off in the end. For *SingleSiteDMRGEngine*, using a mixer is crucial, as the one-site algorithm cannot increase the MPS bond dimension by itself.

Todo: Write UserGuide!!!

mps_sweeps

- full name: tenpy.algorithms.mps_sweeps
- parent module: *tenpy.algorithms*
- type: module

Classes

EffectiveH(env, i0[, combine, move_right, ...])	Prototype class for local effective Hamiltonians used in sweep algorithms.
OneSiteH(env, i0[, combine, move_right, ...])	Class defining the one-site effective Hamiltonian for Lanczos.
Sweep(psi, model, engine_params)	Prototype class for a 'sweeping' algorithm.
TwoSiteH(env, i0[, combine, move_right, ...])	Class defining the two-site effective Hamiltonian for Lanczos.

EffectiveH

- full name: tenpy.algorithms.mps_sweeps.EffectiveH
- parent module: *tenpy.algorithms.mps_sweeps*
- type: class

class tenpy.algorithms.mps_sweeps.**EffectiveH**(*env*, *i0*, *combine=False*, *move_right=True*, *ortho_to_envs=[]*)

Bases: *tenpy.linalg.sparse.NpcLinearOperator*

Prototype class for local effective Hamiltonians used in sweep algorithms.

As an example, the local effective Hamiltonian for a two-site (DMRG) algorithm looks like:

```
|         .---       ---.
|         |   |   |   |
|        LP----H0--H1---RP
|         |   |   |   |
|         .---       ---.
```

where `H0` and `H1` are MPO tensors.

Parameters

env [*MPOEnvironment*] Environment for contraction `<psi|H|psi>`.

i0 [int] Index of the active site if length=1, or of the left-most active site if length>1.

combine [bool, optional] Whether to combine legs into pipes as far as possible. This reduces the overhead of calculating charge combinations in the contractions.

move_right [bool, optional] Whether the sweeping algorithm that calls for an *EffectiveH* is moving to the right.

ortho_to_envs [list of *MPSEnvironment*] List of environments `<psi|psi_ortho>`, where *psi_ortho* is an MPS to orthogonalize against. See *matvec_theta_ortho()* for more details. We implement this by effectively sending `H -> (1 - sum_o |theta_o><theta_o|) H (1 - sum_o |theta_o><theta_o|)`, where `|theta_o>` is `|psi_o>` projected into the appropriate basis (in which *self* is given).

Attributes

length [int] Number of (MPS) sites the effective hamiltonian covers. NB: Class attribute.

dtype [np.dtype] The data type of the involved arrays.

N [int] Contracting *self* with `as_matrix()` will result in an *N`x`N* matrix .

acts_on [list of str] Labels of the state on which *self* acts. NB: class attribute. Overwritten by normal attribute, if *combine*.

theta_ortho [list of *Array*] Projections of *ortho_to_envs* into the basis of *self*.

_matvec_without_theta_ortho [function] Backup copy of *matvec()*. Allows to monkey-patch *matvec()* with *matvec_theta_ortho()*, which is done in `_set_theta_ortho()`.

Methods

matvec(self, theta)	Apply the effective Hamiltonian to *theta*.
matvec_theta_ortho(self, theta)	Apply *self* to *theta*, and orthogonalize against *self.theta_ortho*.
to_matrix(self)	Contract *self* to a matrix.

matvec (*self*, *theta*)
Apply the effective Hamiltonian to *theta*.

This function turns *EffectiveH* to a linear operator, which can be used for *lanczos()*.

Parameters

theta [*Array*] Wave function to apply the effective Hamiltonian to.

Returns

H_theta [*Array*] Result of applying the effective Hamiltonian to *theta*, $H|\theta>$.

to_matrix (*self*)
Contract *self* to a matrix.

matvec_theta_ortho (*self*, *theta*)
Apply *self* to *theta*, and orthogonalize against *self.theta_ortho*. __ We implement this by effectively replacing `H -> P H P` with the projector `P = 1 - sum_o |o> <o|` projecting out the states from `theta_ortho`.

Parameter and return value as for *matvec()* (which this function replaces, if the class was initialized with non-empty *ortho_to_envs*.)

OneSiteH

- full name: tenpy.algorithms.mps_sweeps.OneSiteH
- parent module: *tenpy.algorithms.mps_sweeps*
- type: class

class tenpy.algorithms.mps_sweeps.**OneSiteH** (*env*, *i0*, *combine=False*, *move_right=True*, *ortho_to_envs=[]*)
Bases: *tenpy.algorithms.mps_sweeps.EffectiveH*

Class defining the one-site effective Hamiltonian for Lanczos.

The effective one-site Hamiltonian looks like this:

If *combine* is True, we define either *LHeff* as contraction of *LP* with *W* (in the case *move_right* is True) or *RHeff* as contraction of *RP* and *W*.

Parameters

env [*MPOEnvironment*] Environment for contraction `<psi|H|psi>`.

i0 [int] Index of the active site if length=1, or of the left-most active site if length>1.

combine [bool] Whether to combine legs into pipes. This combines the virtual and physical leg for the left site (when moving right) or right side (when moving left) into pipes. This reduces the overhead of calculating charge combinations in the contractions, but one *matvec()* is formally more expensive, $O(2d^3\chi^3 D)$. Is originally from the wo-site method; unclear if it works well for 1 site.

move_right [bool] Whether the the sweep is moving right or left for the next update.

Attributes

length [int] Number of (MPS) sites the effective hamiltonian covers.

combine, move_right [bool] See above.

LHeff, RHeff [*Array*] Only set `combine`, and only one of them depending on `move_right`. If *move_right* was True, *LHeff* is set with labels `'(vR*.p0)', 'wR', '(vR.p0*)'` for bra, MPO, ket; otherwise *RHeff* is set with labels `'(p0*.vL)', 'wL', '(p0, vL*)'`

LP, W0, RP [*Array*] Tensors making up the network of *self*.

Methods

combine_Heff(self)	Combine LP and RP with W to form LHeff and RHeff, depending on the direction.
combine_theta(self, theta)	Combine the legs of *theta*, such that fit to how we combined the legs of *self*.
matvec(self, theta)	Apply the effective Hamiltonian to *theta*.
matvec_theta_ortho(self, theta)	Apply *self* to *theta*, and orthogonalize against *self.theta_ortho*.
to_matrix(self)	Contract *self* to a matrix.

matvec(*self*, *theta*)
Apply the effective Hamiltonian to *theta*.

Parameters

theta [*Array*] Labels: `vL, p0, vR` if combine=False, `(vL.p0), vR` or `vL, (p0.vR)` if True (depending on the direction of movement)

Returns

theta ***Array*** Product of *theta* and the effective Hamiltonian.

combine_Heff(*self*)
Combine LP and RP with W to form LHeff and RHeff, depending on the direction.

In a move to the right, we need LHeff. In a move to the left, we need RHeff. Both contain the same W.

combine_theta(*self*, *theta*)
Combine the legs of *theta*, such that fit to how we combined the legs of *self*.

Parameters

theta [*Array*] Wave function with labels `'vL', 'p0', 'p1', 'vR'`

Returns

theta [*Array*] Wave function with labels `'vL', 'p0', 'p1', 'vR'`

to_matrix(*self*)
Contract *self* to a matrix.

matvec_theta_ortho(*self*, *theta*)
Apply *self* to *theta*, and orthogonalize against *self.theta_ortho*. __ We implement this by effectively replacing `H -> P H P` with the projector `P = 1 - sum_o |o> <o|` projecting out the states from `theta_ortho`.

Parameter and return value as for *matvec()* (which this function replaces, if the class was initialized with non-empty *ortho_to_envs*.)

Sweep

- full name: tenpy.algorithms.mps_sweeps.Sweep
- parent module: *tenpy.algorithms.mps_sweeps*
- type: class

class tenpy.algorithms.mps_sweeps.**Sweep**(*psi*, *model*, *engine_params*)
Bases: object

Prototype class for a 'sweeping' algorithm.

This is a superclass, intended to cover common procedures in all algorithms that 'sweep'. This includes DMRG, TDVP, TEBD, etc. Only DMRG is currently implemented in this way.

Parameters

psi [*MPS*] Initial guess for the ground state, which is to be optimized in-place.

model [MPOModel] The model representing the Hamiltonian for which we want to find the ground state.

engine_params [dict] Further optional parameters. These are usually algorithm-specific, and thus should be described in subclasses.

Attributes

chi_list [dict | None] A dictionary to gradually increase the *chi_max* parameter of *trunc_params*. The key defines starting from which sweep *chi_max* is set to the value, e.g. `{0: 50, 20: 100}` uses `chi_max=50` for the first 20 sweeps and `chi_max=100` afterwards. Overwrites *trunc_params['chi_list']*`. By default (None) this feature is disabled.

combine [bool] Whether to combine legs into pipes as far as possible. This reduces the overhead of calculating charge combinations in the contractions. Makes the two-site DMRG engine equivalent to the old *EngineCombine*.

E_trunc_list [list] List of truncation energies throughout a sweep.

env [*MPOEnvironment*] Environment for contraction <psi|H|psi>.

finite [bool] Whether the MPS boundary conditions are finite (True) or infinite (False)

i0 [int] Only set during sweep. Left-most of the *EffectiveH.length* sites to be updated in *update_local()*.

lanczos_params [dict] Parameters for the Lanczos algorithm.

mixer [Mixer | None] If None, no mixer is used (anymore), otherwise the mixer instance.

move_right [bool] Only set during sweep. Whether the next *i0* of the sweep will be right or left of the current one.

ortho_to_envs [list of *MPSEnvironment*] List of environments `<psi|psi_ortho>`, where *psi_ortho* is an MPS to orthogonalize against.

shelve [bool] If a simulation runs out of time (*time.time() - start_time > max_seconds*), the run will terminate with *shelve = True*.

sweeps [int] The number of sweeps already performed. (Useful for re-start).

time0 [float] Time marker for the start of the run.

trunc_err_list [list] List of truncation errors.

trunc_params [dict] Parameters for truncations.

update_LP_RP [(bool, bool)] Only set during a sweep. Whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory (inside `env`).

verbose [bool | int] Level of verbosity (i.e. how much status information to print); higher=more output.

Methods

environment_sweeps(self, N_sweeps)	Perform *N_sweeps* sweeps without optimization to update the environment.
get_sweep_schedule(self)	Define the schedule of the sweep.
init_env(self[, model])	(Re-)initialize the environment.
mixer_activate(self)	Set *self.mixer* to the class specified by *engine_params['mixer']*.
mixer_cleanup(self)	Cleanup the effects of a mixer.
post_update_local(self, **kwargs)	Algorithm-specific actions to be taken after local update.
prepare_update(self)	Prepare everything algorithm-specific to perform a local update.
reset_stats(self)	Reset the statistics.
sweep(self[, optimize, meas_E_trunc])	One 'sweep' of a sweeper algorithm.
update_local(self, theta, **kwargs)	Perform algorithm-specific local update.

init_env(*self*, *model=None*)
(Re-)initialize the environment.

This function is useful to (re-)start a Sweep with a slightly different model or different (engine) parameters. Note that we assume that we still have the same *psi*. Calls *reset_stats()*.

Parameters

model [`MPOModel`] The model representing the Hamiltonian for which we want to find the ground state. If `None`, keep the model used before.

Raises

ValueError If the engine is re-initialized with a new model, which legs are incompatible with those of hte old model.

reset_stats(*self*)
Reset the statistics. Useful if you want to start a new Sweep run.

This method is expected to be overwritten by subclass, and should then define self.update_stats and self.sweep_stats dicts consistent with the statistics generated by the algorithm particular to that subclass.

environment_sweeps(*self*, *N_sweeps*)
Perform *N_sweeps* sweeps without optimization to update the environment.

Parameters

N_sweeps [int] Number of sweeps to run without optimization

sweep(*self*, *optimize=True*, *meas_E_trunc=False*)
One 'sweep' of a sweeper algorithm.

Iteratate over the bond which is optimized, to the right and then back to the left to the starting point. If optimize=False, don't actually diagonalize the effective hamiltonian, but only update the environment.

Parameters

optimize [bool, optional] Whether we actually optimize to find the ground state of the effective Hamiltonian. (If False, just update the environments).

meas_E_trunc [bool, optional] Whether to measure truncation energies.

Returns

max_trunc_err [float] Maximal truncation error introduced.

max_E_trunc [`None` | float] `None` if meas_E_trunc is False, else the maximal change of the energy due to the truncation.

get_sweep_schedule(*self*)
Define the schedule of the sweep.

One 'sweep' is a full sequence from the leftmost site to the right and back. Only those *LP* and *RP* that can be used later should be updated.

Returns

schedule [iterable of (int, bool, (bool, bool))] Schedule for the sweep. Each entry is `(i0, move_right, (update_LP, update_RP))`, where *i0* is the leftmost of the `self.EffectiveH.length` sites to be updated in *update_local()*, *move_right* indicates whether the next *i0* in the schedule is rigth (*True*) of the current one, and *update_LP*, *update_RP* indicate whether it is necessary to update the *LP* and *RP*. The latter are chosen such that the environment is growing for infinite systems, but we only keep the minimal number of environment tensors in memory.

mixer_cleanup(*self*)
Cleanup the effects of a mixer.

A *sweep()* with an enabled `Mixer` leaves the MPS *psi* with 2D arrays in *S*. To recover the originial form, this function simply performs one sweep with disabled mixer.

mixer_activate(*self*)
Set *self.mixer* to the class specified by *engine_params['mixer']*.

It is expected that different algorithms have differen ways of implementing mixers (with different defaults). Thus, this is algorithm-specific.

prepare_update(*self*)
Prepare everything algorithm-specific to perform a local update.

update_local(*self*, *theta*, ***kwargs*)
Perform algorithm-specific local update.

post_update_local (*self*, ***kwargs*)
Algorithm-specific actions to be taken after local update.

An example would be to collect statistics.

TwoSiteH

- full name: tenpy.algorithms.mps_sweeps.TwoSiteH
- parent module: *tenpy.algorithms.mps_sweeps*
- type: class

class tenpy.algorithms.mps_sweeps.**TwoSiteH**(*env*, *i0*, *combine=False*, *move_right=True*, *ortho_to_envs=[]*)
Bases: *tenpy.algorithms.mps_sweeps.EffectiveH*

Class defining the two-site effective Hamiltonian for Lanczos.

The effective two-site Hamiltonian looks like this:

```
|         .---       ---.
|         |    |   |    |
|        LP----W0--W1---RP
|         |    |   |    |
|         .---       ---.
```

If *combine* is True, we define *LHeff* and *RHeff*, which are the contractions of *LP* with *W0*, and *RP* with *W1*, respectively.

Parameters

env [*MPOEnvironment*] Environment for contraction `<psi|H|psi>`.

i0 [int] Index of the active site if length=1, or of the left-most active site if length>1.

combine [bool] Whether to combine legs into pipes. This combines the virtual and physical leg for the left site (when moving right) or right side (when moving left) into pipes. This reduces the overhead of calculating charge combinations in the contractions, but one *matvec()* is formally more expensive, $O(2d^3\chi^3 D)$.

move_right [bool] Whether the the sweep is moving right or left for the next update.

Attributes

combine [bool] Whether to combine legs into pipes. This combines the virtual and physical leg for the left site and right site into pipes. This reduces the overhead of calculating charge combinations in the contractions, but one *matvec()* is formally more expensive, $O(2d^3\chi^3 D)$.

length [int] Number of (MPS) sites the effective hamiltonian covers.

LHeff [*Array*] Left part of the effective Hamiltonian. Labels `'(vR*.p0)'`, `'wR'`, `'(vR.p0*)'` for bra, MPO, ket.

RHeff [*Array*] Right part of the effective Hamiltonian. Labels `'(p1*.vL)'`, `'wL'`, `'(p1.vL*)'` for ket, MPO, bra.

LP, W0, W1, RP [*Array*] Tensors making up the network of *self*.

Methods

combine_Heff(self)	Combine LP and RP with W to form LHeff and RHeff.
combine_theta(self, theta)	Combine the legs of *theta*, such that fit to how we combined the legs of *self*.
matvec(self, theta)	Apply the effective Hamiltonian to *theta*.
matvec_theta_ortho(self, theta)	Apply *self* to *theta*, and orthogonalize against *self.theta_ortho*.
to_matrix(self)	Contract *self* to a matrix.

matvec(*self*, *theta*)
Apply the effective Hamiltonian to *theta*.

Parameters

theta [*Array*] Labels: `vL, p0, p1, vR` if combine=False, `(vL.p0), (p1.vR)` if True

Returns

theta ***Array*** Product of *theta* and the effective Hamiltonian.

combine_Heff(*self*)
Combine LP and RP with W to form LHeff and RHeff.

Combine LP with W0 and RP with W1 to get the effective parts of the Hamiltonian with piped legs.

combine_theta(*self*, *theta*)
Combine the legs of *theta*, such that fit to how we combined the legs of *self*.

Parameters

theta [*Array*] Wave function with labels `'vL', 'p0', 'p1', 'vR'`

Returns

theta [*Array*] Wave function with labels `'vL', 'p0', 'p1', 'vR'`

to_matrix(*self*)
Contract *self* to a matrix.

matvec_theta_ortho(*self*, *theta*)
Apply *self* to *theta*, and orthogonalize against *self.theta_ortho*. __ We implement this by effectively replacing `H -> P H P` with the projector `P = 1 - sum_o |o> <o|` projecting out the states from `theta_ortho`.

Parameter and return value as for *matvec()* (which this function replaces, if the class was initialized with non-empty *ortho_to_envs*.)

Module description

'Sweep' algorithm and effective Hamiltonians.

Many MPS-based algorithms use a 'sweep' structure, wherein local updates are performed on the MPS tensors sequentially, first from left to right, then from right to left. This procedure is common to DMRG, TDVP, sequential time evolution, etc.

Another common feature of these algorithms is the use of an effective local Hamiltonian to perform the local updates. The most prominent example of this is probably DMRG, where the local MPS object is optimized with respect to the rest of the MPS-MPO-MPS network, the latter forming the effective Hamiltonian.

The *Sweep* class attempts to generalize as many aspects of 'sweeping' algorithms as possible. *EffectiveH* and its subclasses implement the effective Hamiltonians mentioned above. Currently, effective Hamiltonians for 1-site and 2-site optimization are implemented.

Todo: Rebuild TDVP engine as subclasses of sweep Do testing

tebd

- full name: tenpy.algorithms.tebd
- parent module: *tenpy.algorithms*
- type: module

Classes

Engine(psi, model, TEBD_params)	Time Evolving Block Decimation (TEBD) 'engine'.
RandomUnitaryEvolution(psi, TEBD_params)	Evolution of an MPS with random two-site unitaries in a TEBD-like fashion.

Engine

- full name: tenpy.algorithms.tebd.Engine
- parent module: *tenpy.algorithms.tebd*
- type: class

class tenpy.algorithms.tebd.**Engine**(*psi*, *model*, *TEBD_params*)
Bases: object

Time Evolving Block Decimation (TEBD) 'engine'.

Parameters

psi [MPS] Initial state to be time evolved. Modified in place.

model [*NearestNeighborModel*] The model representing the Hamiltonian for which we want to find the ground state.

TEBD_params [dict] Further optional parameters as described in the tables in *run()* and *run_GS()* for more details. Use verbose=1 to print the used parameters during runtime.

Attributes

verbose [int] Level of verbosity (i.e. how much status information to print); higher=more output.

evolved_time [float | complex] Indicating how long *psi* has been evolved, `psi = exp(-i * evolved_time * H) psi(t=0)`.

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during the sequence of update steps.

psi [*MPS*] The MPS, time evolved in-place.

model [*NearestNeighborModel*] The model defining the Hamiltonian.

TEBD_params: dict Optional parameters, see *run()* and *run_GS()* for more details.

_U [list of list of *Array*] Exponentiated *H_bond* (bond Hamiltonians), i.e. roughly `exp(-i H_bond dt_i)`. First list for different *dt_i* as necessary for the chosen *order*, second list for the *L* different bonds.

_U_param [dict] A dictionary containing the information of the latest created *_U*. We don't recalculate *_U* if those parameters didn't change.

_trunc_err_bonds [list of *TruncationError*] The *local* truncation error introduced at each bond, ignoring the errors at other bonds. The *i*-th entry is left of site *i*.

_update_index [None | (int, int)] The indices `i_dt,i_bond` of `U_bond = self._U[i_dt][i_bond]` during update_step.

Methods

calc_U(self, order, delta_t[, type_evo, ...])	Calculate `self.U_bond` from `self.bond_eig_{vals,vecs}`.
run(self)	(Real-)time evolution with TEBD (time evolving block decimation).
run_GS(self)	TEBD algorithm in imaginary time to find the ground state.
suzuki_trotter_decomposition(order, N_steps)	Returns list of necessary steps for the suzuki trotter decomposition.
suzuki_trotter_time_steps(order)	Return time steps of U for the Suzuki Trotter decomposition of desired order.
update(self, N_steps)	Evolve by `N_steps * U_param['dt']`.
update_bond(self, i, U_bond)	Updates the B matrices on a given bond.
update_bond_imag(self, i, U_bond)	Update a bond with a (possibly non-unitary) *U_bond*.
update_imag(self, N_steps)	Perform an update suitable for imaginary time evolution.
update_step(self, U_idx_dt, odd)	Updates either even *or* odd bonds in unit cell.

property trunc_err_bonds
truncation error introduced on each non-trivial bond.

run(*self*)
(Real-)time evolution with TEBD (time evolving block decimation).

The following (optional) parameters are read out from the `TEBD_params`.

key	type	description
dt	float	Time step.
order	int	**Order of the algorithm.** The total error scales as O(t, dt^order).
N_steps	int	Number of time steps *dt* to evolve. (The Trotter decompositions of order > 1 are slightly more efficient if more than one step is performed at once.)
trunc_params	dict	Truncation parameters as described in `truncate()`.

run_GS (*self*)
TEBD algorithm in imaginary time to find the ground state.

Note: It is almost always more efficient (and hence advisable) to use DMRG. This algorithms can nonetheless be used quite well as a benchmark and for comparison.

The following (optional) parameters are read out from the `TEBD_params`. Use `verbose=1` to print the used parameters during runtime.

key	type	description
delta_tau_list	list	A list of floats: the timesteps to be used. Choosing a large timestep *delta_tau* introduces large (Trotter) errors, but a too small time step requires a lot of steps to reach `exp(-tau H) --> |psi0><psi0|`. Therefore, we start with fairly large time steps for a quick time evolution until convergence, and the gradually decrease the time step.
order	int	Order of the Suzuki-Trotter decomposition.
N_steps	int	Number of steps before measurement can be performed
trunc_params	dict	Truncation parameters as described in `truncate()`

static suzuki_trotter_time_steps (*order*)
Return time steps of U for the Suzuki Trotter decomposition of desired order.

See `suzuki_trotter_decomposition()` for details.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

time_steps [list of float] We need `U = exp(-i H_{even/odd} delta_t * dt)` for the *dt* returned in this list.

static suzuki_trotter_decomposition (*order*, *N_steps*)
Returns list of necessary steps for the suzuki trotter decomposition.

We split the Hamiltonian as $H = H_{even} + H_{odd} = H[0] + H[1]$. The Suzuki-Trotter decomposition is an approximation $\exp(tH) \approx prod_{(j,k)\in ST} \exp(d[j]tH[k]) + O(t^{order+1})$.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

ST_decomposition [list of (int, int)] Indices `j, k` of the time-steps `d = suzuki_trotter_time_step(order)` and the decomposition of *H*. They are chosen such that a subsequent application of `exp(d[j] t H[k])` to a given state `|psi>` yields `(exp(N_steps t H[k]) + O(N_steps t^{order+1}))|psi>`.

calc_U (*self*, *order*, *delta_t*, *type_evo='real'*, *E_offset=None*)
Calculate `self.U_bond` from `self.bond_eig_{vals,vecs}`.

This function calculates

- `U_bond = exp(-i dt (H_bond-E_offset_bond))` for `type_evo='real'`, or
- `U_bond = exp(- dt H_bond)` for `type_evo='imag'`.

For first order (in *delta_t*), we need just one `dt=delta_t`. Higher order requires smaller *dt* steps, as given by *suzuki_trotter_time_steps()*.

Parameters

order [int] Trotter order calculated U_bond. See update for more information.

delta_t [float] Size of the time-step used in calculating U_bond

type_evo [`'imag' | 'real'`] Determines whether we perform real or imaginary time-evolution.

E_offset [None | list of float] Possible offset added to *H_bond* for real-time evolution.

update (*self*, *N_steps*)
Evolve by `N_steps * U_param['dt']`.

Parameters

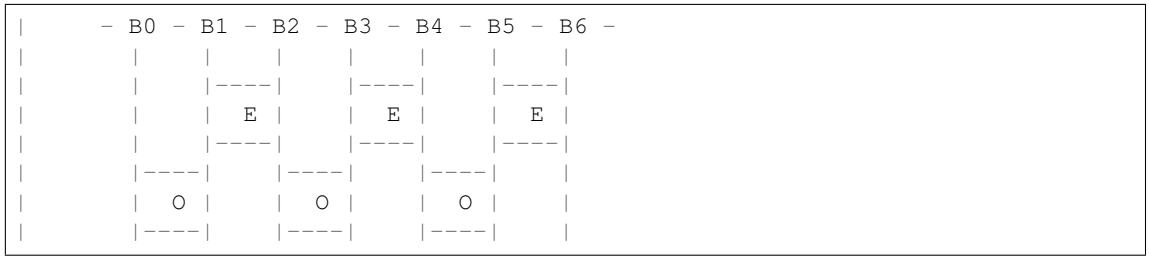
N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_step (*self*, *U_idx_dt*, *odd*)
Updates either even *or* odd bonds in unit cell.

Depending on the choice of p, this function updates all even (`E`, odd=False,0) **or** odd (`O`) (odd=True,1) bonds:



Note that finite boundary conditions are taken care of by having `Us[0] = None`.

Parameters

U_idx_dt [int] Time step index in `self._U`, evolve with `Us[i] = self.U[U_idx_dt][i]` at bond `(i-1,i)`.

odd [bool/int] Indication of whether to update even (`odd=False, 0`) or even (`odd=True, 1`) sites

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_bond(*self*, *i*, *U_bond*)

Updates the B matrices on a given bond.

Function that updates the B matrices, the bond matrix s between and the bond dimension chi for bond i. The correponding tensor networks look like this:

```
|            --S--B1--B2--            --B1--B2--
|                  |   |                |   |
|     theta:     U_bond        C:     U_bond
|                  |   |                |   |
```

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'`.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

update_imag(*self*, *N_steps*)

Perform an update suitable for imaginary time evolution.

Instead of the even/odd brick structure used for ordinary TEBD, we 'sweep' from left to right and right to left, similar as DMRG. Thanks to that, we are actually able to preserve the canonical form.

Parameters

N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_bond_imag(*self*, *i*, *U_bond*)

Update a bond with a (possibly non-unitary) *U_bond*.

Similar as *update_bond()*; but after the SVD just keep the *A, S, B* canonical form. In that way, one can sweep left or right without using old singular values, thus preserving the canonical form during imaginary time evolution.

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'`.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

RandomUnitaryEvolution

- full name: tenpy.algorithms.tebd.RandomUnitaryEvolution
- parent module: *tenpy.algorithms.tebd*
- type: class

class tenpy.algorithms.tebd.**RandomUnitaryEvolution**(*psi*, *TEBD_params*)
Bases: *tenpy.algorithms.tebd.Engine*

Evolution of an MPS with random two-site unitaries in a TEBD-like fashion.

Instead of using a model Hamiltonian, this TEBD engine evolves with random two-site unitaries. These unitaries are drawn according to the Haar measure on unitaries obeying the conservation laws dictated by the conserved charges. If no charge is preserved, this distribution is called circular unitary ensemble (CUE), see *CUE()*.

On one hand, such an evolution is of interest in recent research (see eg. arXiv:1710.09827). On the other hand, it also comes in handy to "randomize" an initial state, e.g. for DMRG. Note that the entanglement grows very quickly, choose the truncation paramters accordingly!

Parameters

psi [MPS] Initial state to be time evolved. Modified in place.

TEBD_params [dict] Use verbose=1 to print the used parameters during runtime. See *run()* and *run_GS()* for more details.

Examples

One can initialize a "random" state with total Sz = L//2 as follows:

```
>>> L = 8
>>> spin_half = SpinHalfSite(conserve='Sz')
>>> psi = MPS.from_product_state([spin_half]*L, [0, 1]*(L//2), bc='finite')  #␣
↪Neel state
>>> print(psi.chi)
[1, 1, 1, 1, 1, 1, 1]
>>> TEBD_params = dict(N_steps=2, trunc_params={'chi_max':10})
>>> eng = RandomUnitaryEvolution(psi, TEBD_params)
>>> eng.run()
>>> print(psi.chi)
[2, 4, 8, 10, 8, 4, 2]
>>> psi.canonical_form()  # necessary if you need to truncate (strongly) during␣
↪the evolution
```

The "random" unitaries preserve the specified charges, e.g. here we have Sz-conservation. If you start in a sector of all up spins, the random unitaries can only apply a phase:

```
>>> psi2 = MPS.from_product_state([spin_half]*L, [0]*L, bc='finite')  # all spins␣
↪up
>>> print(psi2.chi)
[1, 1, 1, 1, 1, 1, 1]
>>> eng2 = RandomUnitaryEvolution(psi2, TEBD_params)
>>> eng2.run()  # random unitaries respect Sz conservation -> we stay in all-up␣
↪sector
>>> print(psi2.chi)  # still a product state, not really random!!!
[1, 1, 1, 1, 1, 1, 1]
```

Attributes

trunc_err_bonds truncation error introduced on each non-trivial bond.

Methods

calc_U(self)	Draw new random two-site unitaries replacing the usual *U* of TEBD.
run(self)	Time evolution with TEBD (time evolving block decimation) and random two-site unitaries.
run_GS(self)	TEBD algorithm in imaginary time to find the ground state.
suzuki_trotter_decomposition(order, N_steps)	Returns list of necessary steps for the suzuki trotter decomposition.
suzuki_trotter_time_steps(order)	Return time steps of U for the Suzuki Trotter decomposition of desired order.
update(self, N_steps)	Apply `N_steps` random two-site unitaries to each bond (in even-odd pattern).
update_bond(self, i, U_bond)	Updates the B matrices on a given bond.
update_bond_imag(self, i, U_bond)	Update a bond with a (possibly non-unitary) *U_bond*.
update_imag(self, N_steps)	Perform an update suitable for imaginary time evolution.
update_step(self, U_idx_dt, odd)	Updates either even *or* odd bonds in unit cell.

run (*self*)
Time evolution with TEBD (time evolving block decimation) and random two-site unitaries.

The following (optional) parameters are read out from the `TEBD_params`.

key	type	description
N_steps	int	Number of two-site unitaries to be applied on each bond.
trunc_params	dict	Truncation parameters as described in *truncate()*

calc_U (*self*)
Draw new random two-site unitaries replacing the usual *U* of TEBD.

update (*self*, *N_steps*)
Apply `N_steps` random two-site unitaries to each bond (in even-odd pattern).

Parameters

N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

run_GS (*self*)
TEBD algorithm in imaginary time to find the ground state.

Note: It is almost always more efficient (and hence advisable) to use DMRG. This algorithms can nonetheless be used quite well as a benchmark and for comparison.

The following (optional) parameters are read out from the `TEBD_params`. Use `verbose=1` to print the used parameters during runtime.

key	type	description
delta_tau_list	list	A list of floats: the timesteps to be used. Choosing a large timestep *delta_tau* introduces large (Trotter) errors, but a too small time step requires a lot of steps to reach `exp(-tau H) --> |psi0><psi0|`. Therefore, we start with fairly large time steps for a quick time evolution until convergence, and the gradually decrease the time step.
order	int	Order of the Suzuki-Trotter decomposition.
N_steps	int	Number of steps before measurement can be performed
trunc_params	dict	Truncation parameters as described in *truncate()*

static suzuki_trotter_decomposition(*order*, *N_steps*)
Returns list of necessary steps for the suzuki trotter decomposition.

We split the Hamiltonian as $H = H_{even} + H_{odd} = H[0] + H[1]$. The Suzuki-Trotter decomposition is an approximation $\exp(tH) \approx prod_{(j,k)\in ST} \exp(d[j]tH[k]) + O(t^{order+1})$.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

ST_decomposition [list of (int, int)] Indices `j, k` of the time-steps `d = suzuki_trotter_time_step(order)` and the decomposition of *H*. They are chosen such that a subsequent application of `exp(d[j] t H[k])` to a given state `|psi>` yields `(exp(N_steps t H[k]) + O(N_steps t^{order+1}))|psi>`.

static suzuki_trotter_time_steps(*order*)
Return time steps of U for the Suzuki Trotter decomposition of desired order.

See *suzuki_trotter_decomposition()* for details.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

time_steps [list of float] We need `U = exp(-i H_{even/odd} delta_t * dt)` for the *dt* returned in this list.

property trunc_err_bonds
truncation error introduced on each non-trivial bond.

update_bond(*self*, *i*, *U_bond*)
Updates the B matrices on a given bond.

Function that updates the B matrices, the bond matrix s between and the bond dimension chi for bond i. The correponding tensor networks look like this:

```
|            --S--B1--B2--            --B1--B2--
|                  |   |                |   |
|     theta:       U_bond        C:     U_bond
|                  |   |                |   |
```

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'`.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

update_bond_imag(*self*, *i*, *U_bond*)

Update a bond with a (possibly non-unitary) *U_bond*.

Similar as *update_bond()*; but after the SVD just keep the *A, S, B* canonical form. In that way, one can sweep left or right without using old singular values, thus preserving the canonical form during imaginary time evolution.

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'`.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

update_imag(*self*, *N_steps*)

Perform an update suitable for imaginary time evolution.

Instead of the even/odd brick structure used for ordinary TEBD, we 'sweep' from left to right and right to left, similar as DMRG. Thanks to that, we are actually able to preserve the canonical form.

Parameters

N_steps [int] The number of steps for which the whole lattice should be updated.

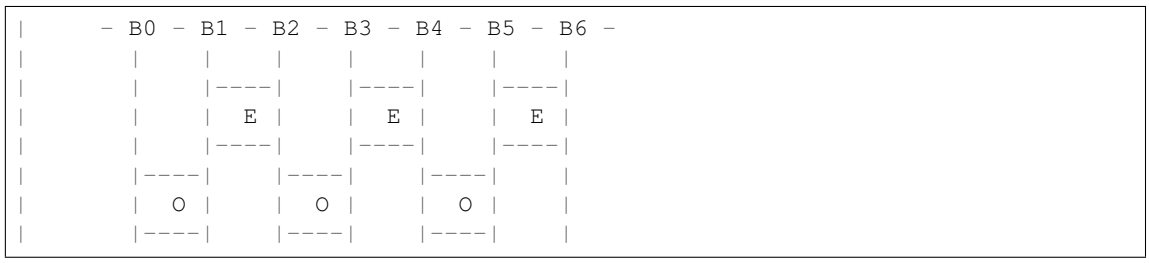
Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_step(*self*, *U_idx_dt*, *odd*)

Updates either even *or* odd bonds in unit cell.

Depending on the choice of p, this function updates all even (`E`, odd=False,0) **or** odd (`O`) (odd=True,1) bonds:



Note that finite boundary conditions are taken care of by having `Us[0] = None`.

Parameters

U_idx_dt [int] Time step index in `self._U`, evolve with `Us[i] = self.U[U_idx_dt][i]` at bond `(i-1,i)`.

odd [bool/int] Indication of whether to update even (`odd=False, 0`) or even (`odd=True, 1`) sites

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

Module description

Time evolving block decimation (TEBD).

The TEBD algorithm (proposed in [Vidal2004]) uses a trotter decomposition of the Hamiltonian to perform a time evoltion of an MPS. It works only for nearest-neighbor hamiltonians (in tenpy given by a *NearestNeighborModel*), which can be written as $H = H^{even} + H^{odd}$, such that H^{even} contains the the terms on even bonds (and similar H^{odd} the terms on odd bonds). In the simplest case, we apply first $U = \exp(-i * dt * H^{even})$, then $U = \exp(-i * dt * H^{odd})$ for each time step dt. This is correct up to errors of $O(dt^2)$, but to evolve until a time T, we need T/dt steps, so in total it is only correct up to error of $O(T * dt)$. Similarly, there are higher order schemata (in dt) (for more details see *Engine.update()*).

Remember, that bond *i* is between sites *(i-1, i)*, so for a finite MPS it looks like:

```
|      - B0 - B1 - B2 - B3 - B4 - B5 - B6 -
|        |    |    |    |    |    |    |
|        |----|    |----|    |----|    |
|        | U1 |    | U3 |    | U5 |    |
|        |----|    |----|    |----|    |
|        |    |----|    |----|    |----|
|        |    | U2 |    | U4 |    | U6 |
|        |    |----|    |----|    |----|
|                  .
|                  .
|                  .
```

After each application of a *Ui*, the MPS needs to be truncated - otherwise the bond dimension *chi* would grow indefinitely. A bound for the error introduced by the truncation is returned.

If one chooses imaginary dt, the exponential projects (for sufficiently long 'time' evolution) onto the ground state of the Hamiltonian.

Note: The application of DMRG is typically much more efficient than imaginary TEBD! Yet, imaginary TEBD might be usefull for cross-checks and testing.

---

tdvp

- full name: tenpy.algorithms.tdvp
- parent module: *tenpy.algorithms*
- type: module

Classes

Engine(psi, model, TDVP_params[, environment])	Time dependant variational principle 'Engine'.
H0_mixed(Lp, Rp)	Class defining the zero site Hamiltonian for Lanczos.
H1_mixed(Lp, Rp, W)	Class defining the one site Hamiltonian for Lanczos.
H2_mixed(Lp, Rp, W0, W1)	Class defining the two sites Hamiltonian for Lanczos.

Engine

- full name: tenpy.algorithms.tdvp.Engine
- parent module: *tenpy.algorithms.tdvp*
- type: class

class tenpy.algorithms.tdvp.**Engine**(*psi*, *model*, *TDVP_params*, *environment=None*)
Bases: object

Time dependant variational principle 'Engine'.

You can call *run_one_site()* for single-site TDVP, or *run_two_sites()* for two-site TDVP.

Parameters

psi [*MPS*] Initial state to be time evolved. Modified in place.

model [*MPOModel*] The model representing the Hamiltonian for which we want to find the ground state.

TDVP_params [dict] Further optional parameters as described in the following table. Use `verbose>0` to print the used parameters during runtime.

key	type	description
start_time	float	Initial value for `evolved_time`
dt	float	Time step of the Trotter error
trunc_params	dict	Truncation parameters as described in *truncate()*

environment [:class:'~tenpy.networks.mpo.MPOEnvironment` | None] Initial environment. If `None` (default), it will be calculated at the beginning.

Attributes

verbose [int] Level of verbosity (i.e. how much status information to print); higher=more output.

evolved_time [float | complex] Indicating how long *psi* has been evolved, `psi = exp(-i * evolved_time * H) psi(t=0)`.

psi [*MPS*] The MPS, time evolved in-place.

TDVP_params: dict Optional parameters, see `run()` and `run_GS()` for more details.

environment [*MPOEnvironment*] The environment, storing the *LP* and *RP* to avoid recalculations.

Methods

run_one_site(self[, N_steps])	Run the TDVP algorithm with the one site algorithm.
run_two_sites(self[, N_steps])	Run the TDVP algorithm with two sites update.
set_anonymous_svd(self, U, new_label)	Relabel the svd.
sweep_left_right(self)	Performs the sweep left->right of the second order TDVP scheme with one site update.
sweep_left_right_two(self)	Performs the sweep left->right of the second order TDVP scheme with two sites update.
sweep_right_left(self)	Performs the sweep right->left of the second order TDVP scheme with one site update.
sweep_right_left_two(self)	Performs the sweep left->right of the second order TDVP scheme with two sites update.
theta_svd_left_right(self, theta)	Performs the SVD from left to right.
theta_svd_right_left(self, theta)	Performs the SVD from right to left.
update_s_h0(self, s, H, dt)	Update with the zero site Hamiltonian (update of the singular value)
update_theta_h1(self, Lp, Rp, theta, W, dt)	Update with the one site Hamiltonian.
update_theta_h2(self, Lp, Rp, theta, W0, W1, dt)	Update with the two sites Hamiltonian.

run_one_site(*self*, *N_steps=None*)
Run the TDVP algorithm with the one site algorithm.

> **Warning:** Be aware that the bond dimension will not increase!

Parameters

N_steps [integer. Number of steps]

run_two_sites(*self*, *N_steps=None*)
Run the TDVP algorithm with two sites update.

The bond dimension will increase. Truncation happens at every step of the sweep, according to the parameters set in trunc_params.

Parameters

N_steps [integer. Number of steps]

sweep_left_right(*self*)
Performs the sweep left->right of the second order TDVP scheme with one site update.

Evolve from 0.5*dt.

sweep_left_right_two(*self*)
Performs the sweep left->right of the second order TDVP scheme with two sites update.

Evolve from 0.5*dt

sweep_right_left(*self*)
Performs the sweep right->left of the second order TDVP scheme with one site update.

Evolve from 0.5*dt

sweep_right_left_two (*self*)
Performs the sweep left->right of the second order TDVP scheme with two sites update.

Evolve from 0.5*dt

update_theta_h1 (*self*, *Lp*, *Rp*, *theta*, *W*, *dt*)
Update with the one site Hamiltonian.

Parameters

Lp [*Array*] tensor representing the left environment

Rp [*Array*] tensor representing the right environment

theta [*Array*] the theta tensor which needs to be updated

W [*Array*] MPO which is applied to the 'p' leg of theta

update_theta_h2 (*self*, *Lp*, *Rp*, *theta*, *W0*, *W1*, *dt*)
Update with the two sites Hamiltonian.

Parameters

Lp [*tenpy.linalg.np_conserved.Array*] tensor representing the left environment

Rp [*tenpy.linalg.np_conserved.Array*] tensor representing the right environment

theta [*tenpy.linalg.np_conserved.Array*] the theta tensor which needs to be updated

W [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p0' leg of theta

W1 [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p1' leg of theta

theta_svd_left_right (*self*, *theta*)
Performs the SVD from left to right.

Parameters

theta: :class:`tenpy.linalg.np_conserved.Array` the theta tensor on which the SVD is applied

set_anonymous_svd (*self*, *U*, *new_label*)
Relabel the svd.

Parameters

U [*tenpy.linalg.np_conserved.Array*] the tensor which lacks a leg_label

theta_svd_right_left (*self*, *theta*)
Performs the SVD from right to left.

Parameters

theta [*tenpy.linalg.np_conserved.Array*,] The theta tensor on which the SVD is applied

update_s_h0 (*self*, *s*, *H*, *dt*)
Update with the zero site Hamiltonian (update of the singular value)

Parameters

s [*tenpy.linalg.np_conserved.Array*] representing the singular value matrix which is updated

H [H0_mixed] zero site Hamiltonian that we need to apply on the singular value matrix

dt [complex number] time step of the evolution

H0_mixed

- full name: tenpy.algorithms.tdvp.H0_mixed
- parent module: *tenpy.algorithms.tdvp*
- type: class

class tenpy.algorithms.tdvp.**H0_mixed**(*Lp*, *Rp*)
Bases: object

Class defining the zero site Hamiltonian for Lanczos.

Parameters

Lp [*tenpy.linalg.np_conserved.Array*] left part of the environment

Rp [*tenpy.linalg.np_conserved.Array*] right part of the environment

Attributes

Lp [*tenpy.linalg.np_conserved.Array*] left part of the environment

Rp [*tenpy.linalg.np_conserved.Array*] right part of the environment

Methods

matvec	

H1_mixed

- full name: tenpy.algorithms.tdvp.H1_mixed
- parent module: *tenpy.algorithms.tdvp*
- type: class

class tenpy.algorithms.tdvp.**H1_mixed**(*Lp*, *Rp*, *W*)
Bases: object

Class defining the one site Hamiltonian for Lanczos.

Parameters

Lp [*tenpy.linalg.np_conserved.Array*] left part of the environment

Rp [*tenpy.linalg.np_conserved.Array*] right part of the environment

M [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p' leg of theta

Attributes

Lp [*tenpy.linalg.np_conserved.Array*] left part of the environment

Rp [*tenpy.linalg.np_conserved.Array*] right part of the environment

W [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p0' leg of theta

Methods

matvec	

H2_mixed

- full name: tenpy.algorithms.tdvp.H2_mixed
- parent module: *tenpy.algorithms.tdvp*
- type: class

class tenpy.algorithms.tdvp.**H2_mixed**(*Lp*, *Rp*, *W0*, *W1*)
Bases: object

Class defining the two sites Hamiltonian for Lanczos.

Parameters

Lp [*tenpy.linalg.np_conserved.Array*] left part of the environment

Rp [*tenpy.linalg.np_conserved.Array*] right part of the environment

W [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p0' leg of theta

Attributes

Lp [*tenpy.linalg.np_conserved.Array*] left part of the environment

Rp [*tenpy.linalg.np_conserved.Array*] right part of the environment

W0 [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p0' leg of theta

W1 [*tenpy.linalg.np_conserved.Array*] MPO which is applied to the 'p1' leg of theta

Methods

matvec	

Module description

Time Dependant Variational Principle (TDVP) with MPS (finite version only).

The TDVP MPS algorithm was first proposed by [Haegeman2011]. However the stability of the algorithm was later improved in [Haegeman2016], that we are following in this implementation. The general idea of the algorithm is to project the quantum time evolution in the manyfold of MPS with a given bond dimension. Compared to e.g. TEBD, the algorithm has several advantages: e.g. it conserves the unitarity of the time evolution and the energy (for the single-site version), and it is suitable for time evolution of Hamiltonian with arbitrary long range in the form of MPOs. We have implemented the one-site formulation which **does not** allow for growth of the bond dimension, and the two-site algorithm which does allow the bond dimension to grow - but requires truncation as in the TEBD case.

Todo: This is still a beta version, use with care. The interface might still change.

Todo: long-term: Much of the code is similar as in DMRG. To avoid too much duplicated code, we should have a general way to sweep through an MPS and updated one or two sites, used in both cases.

purification_tebd

- full name: tenpy.algorithms.purification_tebd
- parent module: *tenpy.algorithms*
- type: module

Classes

BackwardDisentangler(parent)	Disentangle with backward time evolution.
CompositeDisentangler(disentanglers)	Concatenate multiple disentanglers.
DiagonalizeDisentangler(parent)	Disentangle by diagonalizing the two-site density matrix in the auxiliar space.
Disentangler(parent)	Prototype for a disentangler.
GradientDescentDisentangler(parent)	Gradient-descent optimization, similar to *RenyiDisentangler*.
LastDisentangler(parent)	Use the last total 'U' used in `disentangle()` for the same _update_index as guess.
MinDisentangler(disentanglers, parent)	Chose the disentangler giving the smallest entropy.
NoiseDisentangler(parent)	Apply a little bit of random noise.
NormDisentangler(parent)	Find optimal U for which the truncation of U|theta> has maximal overlap with U|theta>.
PurificationTEBD(psi, model, TEBD_params)	Time evolving block decimation (TEBD) for purification MPS.
PurificationTEBD2(psi, model, TEBD_params)	Similar as PurificationTEBD, but perform sweeps instead of brickwall.
RenyiDisentangler(parent)	Iterative find U which minimized the second Renyi entropy.

BackwardDisentangler

- full name: tenpy.algorithms.purification_tebd.BackwardDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**BackwardDisentangler**(*parent*)

Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Disentangle with backward time evolution.

See [Karrasch2013] for details; only useful during real-time evolution.

For the infinite temperature state, `theta = delta_{p0, q0}*delta_{p1, q1}`. Thus, an application of *U_bond* to `p0, p1` can be reverted completely by applying `U_bond^{dagger}` to `q0, q1`, resulting in the same state. This works also for finite temperatures, since *exp(-beta H)* and *exp(-i H t)* commute. Once we apply an operator to measure correlation function, the disentangling breaks down, yet for a local operator only in it's light-cone.

Arguments and return values are the same as for *Disentangler*.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.

CompositeDisentangler

- full name: tenpy.algorithms.purification_tebd.CompositeDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**CompositeDisentangler**(*disentanglers*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Concatenate multiple disentanglers.

Applies multiple disentanglers, one after another (in iteration order).

Parameters

disentanglers [list of *Disentangler*] The disentanglers to be used.

Attributes

disentanglers [list of *Disentangler*] The disentanglers to be used.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.

DiagonalizeDisentangler

- full name: tenpy.algorithms.purification_tebd.DiagonalizeDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**DiagonalizeDisentangler**(*parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Disentangle by diagonalizing the two-site density matrix in the auxiliar space.

See arXiv:1704.01974. Problem: Sorting by eigenvalues breaks the charge conservation! Instead we just sort within the charge blocks. For non-trivial charges, this might increase the entropy!

Arguments and return values are the same as for *Disentangler*.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.

Disentangler

- full name: tenpy.algorithms.purification_tebd.Disentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**Disentangler**(*parent*)
Bases: object

Prototype for a disentangler. Trivial, does nothing.

In purification, we write $\rho_P = Tr_Q |\psi_{P,Q}><\psi_{P,Q}|$. Thus, we can actually apply any unitary to the auxiliar Q space of $|\psi>$ without changing the physical expectation values.

Note: We have to apply the *same* unitary to the 'bra' and 'ket' used for expectation values / correlation functions!

However, the unitary can strongly influence the entanglement structure of $|\psi>$. Therefore, the *PurificationTEBD* includes a hook in *PurificationTEBD.update_bond()* (and similar methods) to find and apply a disentangling unitary to the auxiliar indices of a two-site wave function by calling (`__call__` method) a *Disentangler*.

This class is a 'trivial' disentangler which does *nothing* to the two-site wave function; derived classes use different strategies to find various disentanglers.

Parameters

parent [*Engine*] The parent class calling the disentangler.

Attributes

parent [*Engine*] The parent class calling the disentangler.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.

GradientDescentDisentangler

- full name: tenpy.algorithms.purification_tebd.GradientDescentDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**GradientDescentDisentangler**(*parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Gradient-descent optimization, similar to *RenyiDisentangler*.

Reads of the following *TEBD_params*:

key	type	description
disent_eps	float	Break, if the change in the Renyi entropy `S(n=2)` per iteration is smaller than this value.
disent_max_iter	float	Maximum number of iterations to perform.
disent_n	float	Renyi index of the entropy to be used. `n=1` for von-Neumann entropy.

Arguments and return values are the same as for *Disentangler*.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.
iter(self, theta)	Given *theta*, find a unitary U towards minimizing the n-th Renyi entropy.

iter(*self*, *theta*)

Given *theta*, find a unitary U towards minimizing the n-th Renyi entropy.

This function calulates the gradiant $dS = \partial S(Utheta, n)/\partial U$. and then `U(t) = exp(-t*dS)`, where we choose the *t* from stepsizes which minimizes the entropy of `U(t) theta`.

When `R[i]` is the derivative $\partial S(Y, n)\partial Y_i$ of the (n-th Renyi) entropy, `dS` is given by:

```
|      .----X--R--Z----.
|      |    |     |    |
|      |    q0    q1   |
|      |               |
|      |    q0*   q1*  |
|      |    |     |    |
|      .----X*-Y--Z*---.
```

Parameters

theta [*Array*] Two-site wave function to be disentangled

Returns

S [float] n-th Renyi entopy of new_theta

theta [*Array*] The *disentangled* wave function `new_U theta`.

new_U [*Array*] Unitary with legs `'q0', 'q1', 'q0*', 'q1*'`, which was used to disentangle *theta*.

LastDisentangler

- full name: tenpy.algorithms.purification_tebd.LastDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**LastDisentangler**(*parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Use the last total 'U' used in `disentangle()` for the same _update_index as guess.

Useful as a starting point in a *CompositeDisentangler* to reduce the number of iterations for a following disentangler.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.

MinDisentangler

- full name: tenpy.algorithms.purification_tebd.MinDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**MinDisentangler**(*disentanglers*, *parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Chose the disentangler giving the smallest entropy.

Apply each of the disentanglers to the given *theta*, use the result with smallest entropy. Reads the TEBD_param `'disent_min_n'` which selects the *entropy()* to be used for comparison.

Parameters

disentanglers [list of *Disentangler*] The disentanglers to be used.

parent [*Engine*] The parent class calling the disentangler.

Attributes

n [float] Selects the entropy to be used for comparison.

disentanglers [list of *Disentangler*] The disentanglers to be used.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.

NoiseDisentangler

- full name: tenpy.algorithms.purification_tebd.NoiseDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**NoiseDisentangler**(*parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Apply a little bit of random noise. Useful as pre-step to *RenyiDisentangler*.

Arguments and return values are the same as for *Disentangler*.

Methods

__call__(self, theta)	Find and apply a unitary to disentangle *theta*.

NormDisentangler

- full name: tenpy.algorithms.purification_tebd.NormDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**NormDisentangler**(*parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Find optimal U for which the truncation of U|theta> has maximal overlap with U|theta>.

Reads of the following *TEBD_params* as break criteria for the iteration:

key	type	description
disent_eps	float	Break, if the change in the Renyi entropy `S(n=2)` per iteration is smaller than this value.
disent_max_iter	float	Maximum number of iterations to perform.
disent_trunc_par	dict	Truncation parameters; defaults to *trunc_params*.
disent_norm_chi	iterable	To find the optimal U it can help to increase *chi_max* of *disent_trunc_par* slowly, the default is `range(1, disent_trunc_par['chi_max']+1)`. However, that's **very** slow for large *chi_max*, so we allow to change it. (In fact, it makes the disentangler *scale* worse than the rest of TEBD.)

Arguments and return values are the same as for `disentangle()`.

Methods

`__call__`(self, theta)	Find and apply a unitary to disentangle *theta*.
iter(self, theta, U, trunc_params)	Given *theta* and *U*, find *U2* maximizing `<theta|U2 truncate(U |theta>)`.

iter(*self*, *theta*, *U*, *trunc_params*)

Given *theta* and *U*, find *U2* maximizing `<theta|U2 truncate(U |theta>)`.

Finds unitary *U2* which maximizes Tr(U

Parameters

theta [*Array*] Two-site wave function to be disentangled.

U [*Array*] The previous guess for *U*; with legs `'q0', 'q1', 'q0*', 'q1*'`.

trunc_params [dict] The truncation parameters (similar as *self.trunc_params*) used to truncate *U|theta>*.

Returns

trunc_err [TruncationError] Norm error discarded during the truncation of `U|theta>`.

new_U [*Array*] Unitary with legs `'q0', 'q1', 'q0*', 'q1*'`. Chosen such that `new_U|theta>` has maximal overlap with the truncated `U|theta>`.

PurificationTEBD

- full name: tenpy.algorithms.purification_tebd.PurificationTEBD
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**PurificationTEBD**(*psi*, *model*, *TEBD_params*)

Bases: *tenpy.algorithms.tebd.Engine*

Time evolving block decimation (TEBD) for purification MPS.

Parameters

psi [`PurificationMPS`] Initial state to be time evolved. Modified in place.

model [`NearestNeighborModel`] The model representing the Hamiltonian for which we want to find the ground state.

TEBD_params [dict] Further optional parameters as described in the following table. Use `verbose=1` to print the used parameters during runtime. See *run()* and *run_GS()* for more details.

Attributes

disent_iterations For each bond the total number of iterations performed in any *Disentangler*.

used_disentangler [*Disentangler*] The disentangler to be used on the auxiliar indices. Chosen by *get_disentangler()*, called with the TEBD parameter `'disentangle'`. Defaults to the trivial disentangler for `TEBD_params['disentangle']=None`.

_disent_iterations [1D ndarray] Number of iterations performed on all bonds, including trivial bonds; lenght *L*.

_guess_U_disent [list of list of npc.Array] Same index strucuture as *self._U*: for each two-site U of the physical time evolution the disentangler from the last application. Initialized to identities.

Methods

calc_U(self, order, delta_t[, type_evo, ...])	see `calc_U()`
disentangle(self, theta)	Disentangle *theta* before splitting with svd.
disentangle_global(self[, pair])	Try global disentangling by determining the maximally entangled pairs of sites.
disentangle_global_nsite(self[, n])	Perform a sweep through the system and disentangle with *disentangle_n_site()*.
disentangle_n_site(self, i, n, theta)	Generalization of *disentangle()* to *n* sites.
run(self)	(Real-)time evolution with TEBD (time evolving block decimation).
run_GS(self)	TEBD algorithm in imaginary time to find the ground state.
run_imaginary(self, beta)	Run imaginary time evolution to cool down to the given *beta*.
suzuki_trotter_decomposition(order, N_steps)	Returns list of necessary steps for the suzuki trotter decomposition.
suzuki_trotter_time_steps(order)	Return time steps of U for the Suzuki Trotter decomposition of desired order.
update(self, N_steps)	Evolve by `N_steps * U_param['dt']`.
update_bond(self, i, U_bond)	Updates the B matrices on a given bond.
update_bond_imag(self, i, U_bond)	Update a bond with a (possibly non-unitary) *U_bond*.
update_imag(self, N_steps)	Perform an update suitable for imaginary time evolution.
update_step(self, U_idx_dt, odd)	Updates either even *or* odd bonds in unit cell.

run_imaginary(*self*, *beta*)
Run imaginary time evolution to cool down to the given *beta*.

Note that we don't change the *norm* attribute of the MPS, i.e. normalization is preserved.

Parameters

beta [float] The inverse temperature *beta* = 1/T, by which we should cool down. We evolve to the closest multiple of `TEBD_params['dt']`, see also `evolved_time`.

property disent_iterations
For each bond the total number of iterations performed in any *Disentangler*.

calc_U(*self*, *order*, *delta_t*, *type_evo='real'*, *E_offset=None*)
see `calc_U()`

update_bond(*self*, *i*, *U_bond*)
Updates the B matrices on a given bond.

Function that updates the B matrices, the bond matrix s between and the bond dimension chi for bond i. This would look something like:

```
|            |              |
|     ... - B1  -  s  -  B2 - ...
|            |              |
|            |--------------|
|            |      U       |
|            |--------------|
|            |              |
```

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'` for *U_bond*.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

update_bond_imag(*self*, *i*, *U_bond*)

Update a bond with a (possibly non-unitary) *U_bond*.

Similar as *update_bond()*; but after the SVD just keep the *A, S, B* canonical form. In that way, one can sweep left or right without using old singular values, thus preserving the canonical form during imaginary time evolution.

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'`.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

disentangle(*self*, *theta*)

Disentangle *theta* before splitting with svd.

For the purification we write $\rho_P = Tr_Q |\psi_{P,Q}><\psi_{P,Q}|$. Thus, we can actually apply any unitary to the auxiliar Q space of $|\psi>$ without changing the result.

Note: We have to apply the *same* unitary to the 'bra' and 'ket' used for expectation values / correlation functions!

The behaviour of this function is set by `used_disentangler`, which in turn is obtained from `get_disentangler(TEBD_params['disentangle'])`, see *get_disentangler()* for details on the syntax.

Parameters

theta [*Array*] Wave function to disentangle, with legs `'vL', 'vR', 'p0', 'p1', 'q0', 'q1'`.

Returns

theta_disentangled [*Array*] Disentangled *theta*; `npc.tensordot(U, theta, axes=[['q0*', 'q1*'], ['q0', 'q1']])`.

U [`Array`] The unitary used to disentangle *theta*, with labels `'q0', 'q1', 'q0*', 'q1*'`. If no unitary was found/applied, it might also be `None`.

disentangle_global(*self*, *pair=None*)
Try global disentangling by determining the maximally entangled pairs of sites.

Caclulate the mutual information (in the auxiliar space) between two sites and determine where it is maximal. Disentangle these two sites with *disentangle()*

disentangle_global_nsite(*self*, *n=2*)
Perform a sweep through the system and disentangle with *disentangle_n_site()*.

Parameters

n: int maximal number of sites to disentangle at once.

disentangle_n_site(*self*, *i*, *n*, *theta*)
Generalization of *disentangle()* to *n* sites.

Simply group left and right *n*/2 physical legs, adjust labels, and apply *disentangle()* to disentangle the central bond. Recursively proceed to disentangle left and right parts afterwards. Scales (for even *n*) as $O(\chi^3 d^n d^{n/2})$.

run(*self*)
(Real-)time evolution with TEBD (time evolving block decimation).

The following (optional) parameters are read out from the `TEBD_params`.

key	type	description
dt	float	Time step.
order	int	**Order of the algorithm.** The total error scales as O(t, dt^order).
N_steps	int	Number of time steps *dt* to evolve. (The Trotter decompositions of order > 1 are slightly more efficient if more than one step is performed at once.)
trunc_params	dict	Truncation parameters as described in *truncate()*.

run_GS(*self*)
TEBD algorithm in imaginary time to find the ground state.

Note: It is almost always more efficient (and hence advisable) to use DMRG. This algorithms can nonetheless be used quite well as a benchmark and for comparison.

The following (optional) parameters are read out from the `TEBD_params`. Use `verbose=1` to print the used parameters during runtime.

key	type	description
delta_taus	list	A list of floats: the timesteps to be used. Choosing a large timestep *delta_tau* introduces large (Trotter) errors, but a too small time step requires a lot of steps to reach `exp(-tau H) --> |psi0><psi0|`. Therefore, we start with fairly large time steps for a quick time evolution until convergence, and the gradually decrease the time step.
order	int	Order of the Suzuki-Trotter decomposition.
N_steps	int	Number of steps before measurement can be performed
trunc_params	dict	Truncation parameters as described in *truncate()*

static suzuki_trotter_decomposition(*order*, *N_steps*)
Returns list of necessary steps for the suzuki trotter decomposition.

We split the Hamiltonian as $H = H_{even} + H_{odd} = H[0] + H[1]$. The Suzuki-Trotter decomposition is an approximation $\exp(tH) \approx prod_{(j,k)\in ST} \exp(d[j]tH[k]) + O(t^{order+1})$.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

ST_decomposition [list of (int, int)] Indices `j, k` of the time-steps `d = suzuki_trotter_time_step(order)` and the decomposition of *H*. They are chosen such that a subsequent application of `exp(d[j] t H[k])` to a given state `|psi>` yields `(exp(N_steps t H[k]) + O(N_steps t^{order+1}))|psi>`.

static suzuki_trotter_time_steps(*order*)
Return time steps of U for the Suzuki Trotter decomposition of desired order.

See *suzuki_trotter_decomposition()* for details.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

time_steps [list of float] We need `U = exp(-i H_{even/odd} delta_t * dt)` for the *dt* returned in this list.

property trunc_err_bonds
truncation error introduced on each non-trivial bond.

update(*self*, *N_steps*)
Evolve by `N_steps * U_param['dt']`.

Parameters

N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_imag(*self*, *N_steps*)
Perform an update suitable for imaginary time evolution.

Instead of the even/odd brick structure used for ordinary TEBD, we 'sweep' from left to right and right to left, similar as DMRG. Thanks to that, we are actually able to preserve the canonical form.

Parameters

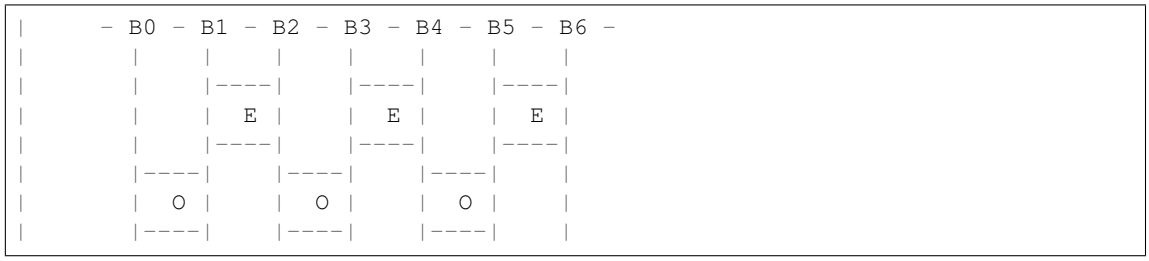
N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_step(*self*, *U_idx_dt*, *odd*)
Updates either even *or* odd bonds in unit cell.

Depending on the choice of p, this function updates all even (`E`, odd=False,0) **or** odd (`O`) (odd=True,1) bonds:



Note that finite boundary conditions are taken care of by having `Us[0] = None`.

Parameters

U_idx_dt [int] Time step index in `self._U`, evolve with `Us[i] = self.U[U_idx_dt][i]` at bond `(i-1,i)`.

odd [bool/int] Indication of whether to update even (`odd=False,0`) or even (`odd=True,1`) sites

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

PurificationTEBD2

- full name: tenpy.algorithms.purification_tebd.PurificationTEBD2
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**PurificationTEBD2**(*psi*, *model*, *TEBD_params*)
Bases: *tenpy.algorithms.purification_tebd.PurificationTEBD*

Similar as PurificationTEBD, but perform sweeps instead of brickwall.

Instead of the A-B pattern of even/odd bonds used in TEBD, perform sweeps similar as in DMRG for real-time evolution (similar as *update_imag()* does for imaginary time evolution).

Attributes

disent_iterations For each bond the total number of iterations performed in any *Disentangler*.

trunc_err_bonds truncation error introduced on each non-trivial bond.

Methods

calc_U(self, order, delta_t[, type_evo, ...])	see `calc_U()`
disentangle(self, theta)	Disentangle *theta* before splitting with svd.
disentangle_global(self[, pair])	Try global disentangling by determining the maximally entangled pairs of sites.
disentangle_global_nsite(self[, n])	Perform a sweep through the system and disentangle with *disentangle_n_site()*.
disentangle_n_site(self, i, n, theta)	Generalization of *disentangle()* to *n* sites.
run(self)	(Real-)time evolution with TEBD (time evolving block decimation).
run_GS(self)	TEBD algorithm in imaginary time to find the ground state.
run_imaginary(self, beta)	Run imaginary time evolution to cool down to the given *beta*.
suzuki_trotter_decomposition(order, N_steps)	Returns list of necessary steps for the suzuki trotter decomposition.
suzuki_trotter_time_steps(order)	Return time steps of U for the Suzuki Trotter decomposition of desired order.
update(self, N_steps)	Evolve by `N_steps * U_param['dt']`.
update_bond(self, i, U_bond)	Updates the B matrices on a given bond.
update_bond_imag(self, i, U_bond)	Update a bond with a (possibly non-unitary) *U_bond*.
update_imag(self, N_steps)	Perform an update suitable for imaginary time evolution.
update_step(self, U_idx_dt, odd)	Updates bonds in unit cell.

update (*self*, *N_steps*)
Evolve by `N_steps * U_param['dt']`.

Parameters

N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

update_step (*self*, *U_idx_dt*, *odd*)
Updates bonds in unit cell.

Depending on the choice of *odd*, perform a sweep to the left or right, updating once per site with a time step given by U_idx_dt.

Parameters

U_idx_dt [int] Time step index in `self._U`, evolve with `Us[i] = self.U[U_idx_dt][i]` at bond `(i-1,i)`.

odd [bool/int] Indication of whether to update even (`odd=False,0`) or even (`odd=True,1`) sites

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

calc_U(*self*, *order*, *delta_t*, *type_evo='real'*, *E_offset=None*)
see `calc_U()`

property disent_iterations
For each bond the total number of iterations performed in any *Disentangler*.

disentangle(*self*, *theta*)
Disentangle *theta* before splitting with svd.

For the purification we write $\rho_P = Tr_Q|\psi_{P,Q}><\psi_{P,Q}|$. Thus, we can actually apply any unitary to the auxiliar Q space of $|\psi>$ without changing the result.

Note: We have to apply the *same* unitary to the 'bra' and 'ket' used for expectation values / correlation functions!

The behaviour of this function is set by `used_disentangler`, which in turn is obtained from `get_disentangler(TEBD_params['disentangle'])`, see *get_disentangler()* for details on the syntax.

Parameters

theta [*Array*] Wave function to disentangle, with legs `'vL', 'vR', 'p0', 'p1', 'q0', 'q1'`.

Returns

theta_disentangled [*Array*] Disentangled *theta*; `npc.tensordot(U, theta, axes=[['q0*', 'q1*'], ['q0', 'q1']])`.

U [`Array`] The unitary used to disentangle *theta*, with labels `'q0', 'q1', 'q0*', 'q1*'`. If no unitary was found/applied, it might also be `None`.

disentangle_global(*self*, *pair=None*)
Try global disentangling by determining the maximally entangled pairs of sites.

Caclulate the mutual information (in the auxiliar space) between two sites and determine where it is maximal. Disentangle these two sites with *disentangle()*

disentangle_global_nsite(*self*, *n=2*)
Perform a sweep through the system and disentangle with *disentangle_n_site()*.

Parameters

n: int maximal number of sites to disentangle at once.

disentangle_n_site(*self*, *i*, *n*, *theta*)
Generalization of *disentangle()* to *n* sites.

Simply group left and right *n*/2 physical legs, adjust labels, and apply *disentangle()* to disentangle the central bond. Recursively proceed to disentangle left and right parts afterwards. Scales (for even *n*) as $O(\chi^3 d^n d^{n/2})$.

run(*self*)
(Real-)time evolution with TEBD (time evolving block decimation).

The following (optional) parameters are read out from the `TEBD_params`.

key	type	description
dt	float	Time step.
order	int	**Order of the algorithm.** The total error scales as O(t, dt^order).
N_steps	int	Number of time steps *dt* to evolve. (The Trotter decompositions of order > 1 are slightly more efficient if more than one step is performed at once.)
trunc_params	dict	Truncation parameters as described in `truncate()`.

run_GS (*self*)
TEBD algorithm in imaginary time to find the ground state.

Note: It is almost always more efficient (and hence advisable) to use DMRG. This algorithms can nonetheless be used quite well as a benchmark and for comparison.

The following (optional) parameters are read out from the `TEBD_params`. Use `verbose=1` to print the used parameters during runtime.

key	type	description
delta_tau_list	list	A list of floats: the timesteps to be used. Choosing a large timestep *delta_tau* introduces large (Trotter) errors, but a too small time step requires a lot of steps to reach `exp(-tau H) --> |psi0><psi0|`. Therefore, we start with fairly large time steps for a quick time evolution until convergence, and the gradually decrease the time step.
order	int	Order of the Suzuki-Trotter decomposition.
N_steps	int	Number of steps before measurement can be performed
trunc_params	dict	Truncation parameters as described in `truncate()`

run_imaginary (*self*, *beta*)
Run imaginary time evolution to cool down to the given *beta*.

Note that we don't change the *norm* attribute of the MPS, i.e. normalization is preserved.

Parameters

beta [float] The inverse temperature *beta* = 1/T, by which we should cool down. We evolve to the closest multiple of `TEBD_params['dt']`, see also `evolved_time`.

static suzuki_trotter_decomposition (*order*, *N_steps*)
Returns list of necessary steps for the suzuki trotter decomposition.

We split the Hamiltonian as $H = H_{even} + H_{odd} = H[0] + H[1]$. The Suzuki-Trotter decomposition is an approximation $\exp(tH) \approx prod_{(j,k)\in ST} \exp(d[j]tH[k]) + O(t^{order+1})$.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

ST_decomposition [list of (int, int)] Indices `j, k` of the time-steps `d = suzuki_trotter_time_step(order)` and the decomposition of *H*. They are chosen such that a subsequent application of `exp(d[j] t H[k])` to a given state `|psi>` yields `(exp(N_steps t H[k]) + O(N_steps t^{order+1}))|psi>`.

static suzuki_trotter_time_steps(*order*)

Return time steps of U for the Suzuki Trotter decomposition of desired order.

See *suzuki_trotter_decomposition()* for details.

Parameters

order [int] The desired order of the Suzuki-Trotter decomposition.

Returns

time_steps [list of float] We need `U = exp(-i H_{even/odd} delta_t * dt)` for the *dt* returned in this list.

property trunc_err_bonds

truncation error introduced on each non-trivial bond.

update_bond(*self*, *i*, *U_bond*)

Updates the B matrices on a given bond.

Function that updates the B matrices, the bond matrix s between and the bond dimension chi for bond i. This would look something like:

```
|            |              |
|     ... - B1  -  s  -  B2 - ...
|            |              |
|            |--------------|
|            |      U       |
|            |--------------|
|            |              |
```

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'` for *U_bond*.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

update_bond_imag(*self*, *i*, *U_bond*)

Update a bond with a (possibly non-unitary) *U_bond*.

Similar as *update_bond()*; but after the SVD just keep the *A, S, B* canonical form. In that way, one can sweep left or right without using old singular values, thus preserving the canonical form during imaginary time evolution.

Parameters

i [int] Bond index; we update the matrices at sites `i-1, i`.

U_bond [*Array*] The bond operator which we apply to the wave function. We expect labels `'p0', 'p1', 'p0*', 'p1*'`.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced by the truncation during this update step.

update_imag(*self*, *N_steps*)
Perform an update suitable for imaginary time evolution.

Instead of the even/odd brick structure used for ordinary TEBD, we 'sweep' from left to right and right to left, similar as DMRG. Thanks to that, we are actually able to preserve the canonical form.

Parameters

N_steps [int] The number of steps for which the whole lattice should be updated.

Returns

trunc_err [*TruncationError*] The error of the represented state which is introduced due to the truncation during this sequence of update steps.

RenyiDisentangler

- full name: tenpy.algorithms.purification_tebd.RenyiDisentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: class

class tenpy.algorithms.purification_tebd.**RenyiDisentangler**(*parent*)
Bases: *tenpy.algorithms.purification_tebd.Disentangler*

Iterative find U which minimized the second Renyi entropy.

See [Hauschild2018]

Reads of the following *TEBD_params* as break criteria for the iteration:

key	type	description
disent_eps	float	Break, if the change in the Renyi entropy `S(n=2)` per iteration is smaller than this value.
disent_max_iter	float	Maximum number of iterations to perform.

Arguments and return values are the same as for `disentangle()`.

Methods

`__call__`(self, theta)	Find optimal U which minimizes the second Renyi entropy.
iter(self, theta, U)	Given *theta* and U, find another U which reduces the 2nd Renyi entropy.

iter(*self*, *theta*, *U*)
Given *theta* and U, find another U which reduces the 2nd Renyi entropy.

Temporarily view the different U as independt and mimizied one of them - this corresponds to a linearization of the cost function. Defining *Utheta* as the application of U to *theata*, and combining the p legs of *theta* with `'vL', 'vR'`, this function contracts:

```
|      .----theta----.
|      |    |   |    |
|      |    q0  q1   |
|      |             |
|      |        q1*  |
|      |        |    |
|      |  .-Utheta*-.
|      |  | |
|      |  .-Utheta--.
|      |        |    |
|      |    q0* |    |
|      |    |   |    |
|      .----Utheta*-.
```

The trace yields the second Renyi entropy *S2*. Further, we calculate the unitary U with maximum overlap with this network.

Parameters

theta [*Array*] Two-site wave function to be disentangled.

U [*Array*] The previous guess for U; with legs `'q0', 'q1', 'q0*', 'q1*'`.

Returns

S2 [float] Renyi entopy (n=2), $S2 = \frac{1}{1-2} \log tr(\rho_L^2)$ of *U theta*.

new_U [*Array*] Unitary with legs `'q0', 'q1', 'q0*', 'q1*'`, which should disentangle *theta*.

Functions

get_disentangler(method, parent)	Parse the parameter *method* and construct a *Disentangler* instance.

get_disentangler

- full name: tenpy.algorithms.purification_tebd.get_disentangler
- parent module: *tenpy.algorithms.purification_tebd*
- type: function

tenpy.algorithms.purification_tebd.**get_disentangler**(*method*, *parent*)
Parse the parameter *method* and construct a *Disentangler* instance.

Parameters

method [str | None] The method to be used, of the form 'method1-method2-min(method3,method4-method5)'. The usage should be clear from the examples, the precise rule follows: We parse the full *method* string as a *composite*, and define `composite := min_atom ['-' min_atom ...]`, `min_atom := { 'min(' composite [',' composite ...] ')' } | atom`, and `atom := {any key of `disentanglers_atom_parse_dict`}`.

parent [*Engine*] The parent class calling the disentangler.

Returns

disentangler [*Disentangler*] Disentangler instance, which can be called to disentangle a 2-site *theta* with the specified *method*.

Examples

```
>>> get_disentangler(None, p)
Disentangler(p)
>>> get_disentangler('last-renyi', p)
Disentangler([LastDisentangler(p), RenyiDisentangler(p)], p)
>>> get_disentangler('min(None,noise-renyi,min(backwards,last)-graddesc)')
MinDisentangler([Disentangler,
                 CompositeDisentangler([NoiseDisentangler(p),␣
↪RenyiDisentangler(p)], p),
                 CompositeDisentangler([MinDisentangler([BackwardDisentangler(p),
                                                         LastDisentangler(p)]),
                                        GradientDescentDisentangler(p)], p), p)
```

Module description

Time evolving block decimation (TEBD) for MPS of purification.

See introduction in *purification_mps*. Time evolution for finite-temperature ensembles. This can be used to obtain correlation functions in time.

network_contractor

- full name: tenpy.algorithms.network_contractor
- parent module: *tenpy.algorithms*
- type: module

Functions

contract(tensor_list[, tensor_names, ...])	Contract a network of tensors.
ncon(tensor_list, leg_links, sequence)	Implementation of `ncon.m` for TeNPy Arrays.

contract

- full name: tenpy.algorithms.network_contractor.contract
- parent module: *tenpy.algorithms.network_contractor*
- type: function

`tenpy.algorithms.network_contractor.`**`contract`**(*tensor_list*, *tensor_names=None*, *leg_contractions=None*, *open_legs=None*, *sequence=None*)

Contract a network of tensors.

Based on the MatLab function `ncon.m` as described in arXiv:1402.0939.

Parameters

tensor_list [list of *Array*] The tensors to be contracted.

leg_contractions [list of `[n1, l1, n2, l2]`] A list of contraction instructions. An entry of leg_contractions has the form `[n1, l1, n2, l2]`, where `n1, n2` are entries of *tensor_names* and each identify an *Array* in *tensor_list*. `l1, l2` are leg labels of the corresponding *Array*. The instruction implies to contract leg `l1` of tensor `n1` with leg `l2` of tensor `n2`.

open_legs [list of `[n1, l1, l]`] A list of instructions for "open" (uncontracted) legs. `[n1, l1, l]` implies that leg `l1` of tensor `n1` is not contracted and is labelled `l` in the result.

tensor_names [list of str] A list of names for each tensor, to be used in *leg_contractions* and *open_legs*. The default value is list(range(len(tensor_list))), so that the tensor "names" are `0, 1, 2, ...`.

sequence [list of int] The order in which the leg_contractions are to be performed. An entry of network_contractor.outer_product indicates performing an outer product. This corresponds to the zero-in-sequence convention of arXiv:1304.6112

Returns

result [`Array` | complex] The number or tensor resulting from the contraction.

ncon

- full name: tenpy.algorithms.network_contractor.ncon
- parent module: *tenpy.algorithms.network_contractor*
- type: function

`tenpy.algorithms.network_contractor.`**`ncon`**(*tensor_list*, *leg_links*, *sequence*)

Implementation of `ncon.m` for TeNPy Arrays.

This function is a python implementation of `ncon.m` (arXiv:1304.6112) for tenpy *Array*. *contract()* is a wrapper that translates from a more python/tenpy input style

Parameters

tensor_list [list of :class:'Array'] Tensors to be contracted.

leg_links [list of list of int] Each entry of leg_links describes the connectivity of the corresponding tensor in *tensor_list*. Each entry is a list that has an entry for each leg of the corresponding tensor. Values `0,1,2,...` are labels of contracted legs and should appear exactly twice in *leg_links*. Values `-1,-2,-3,...` are labels of uncontracted legs and indicate the final ordering (`-1` is first axis).

sequence [list of int] The order in which the contractions are to be performed. An entry of network_contractor.outer_product indicates performing an outer product. This corresponds to the zero-in-sequence convention of arXiv:1304.6112

Returns

result [`Array` | complex] The number or tensor resulting from the contraction.

Module description

Network Contractor.

A tool to contract a network of multiple tensors.

This is an implementation of 'NCON: A tensor network contractor for MATLAB' by Robert N. C. Pfeifer, Glen Evenbly, Sukhwinder Singh, Guifre Vidal, see arXiv:1402.0939

Todo:

- **implement or wrap netcon.m, a function to find optimal contractionn sequences** (arXiv:1304.6112)
- improve helpfulness of Warnings
- _do_trace: trace over all pairs of legs at once. need the corresponding npc function first.

---

exact_diag

- full name: tenpy.algorithms.exact_diag
- parent module: *tenpy.algorithms*
- type: module

Classes

ExactDiag(model[, charge_sector, sparse, ...])	(Full) exact diagonalization of the Hamiltonian.

ExactDiag

- full name: tenpy.algorithms.exact_diag.ExactDiag
- parent module: *tenpy.algorithms.exact_diag*
- type: class

class tenpy.algorithms.exact_diag.**ExactDiag**(*model*, *charge_sector=None*, *sparse=False*, *max_size=2000000.0*)

Bases: object

(Full) exact diagonalization of the Hamiltonian.

Parameters

model [MPOmodel | CouplingModel] The model which is to be diagonalized.

charge_sector [None | charges] If not None, project onto the given charge sector.

sparse [bool] If True, don't sort/bunch the LegPipe used to combine the physical legs. This results in array *blocks* with just one entry, requires much more charge data, and is not what *np_conserved* was designed for, so it's not recommended.

max_size [int] The *build_H_** functions will do nothing (but emit a warning) if the total size of the Hamiltonian would be larger than this.

Attributes

model [MPOmodel | CouplingModel] The model which is to be diagonalized.

chinfo [*ChargeInfo*] The nature of the charge (which is the same for all sites).

charge_sector [None | charges] If not None, we project onto the given charge sector.

max_size [int] The build_H_* functions will do nothing (but emit a warning) if the total size of the Hamiltonian would be larger than this.

full_H [*Array* | None] The full Hamiltonian to be diagonalized with legs '(p0.p1....)', '(p0*,p1*...)' (in that order). None if the build_H_* functions haven't been called yet, or if *max_size* would have been exceeded.

E [ndarray | None] 1D array of eigenvalues.

V [*Array* | None] Eigenvectors. First leg 'ps' are physical legs, the second leg 'ps*' corresponds to the eigenvalues.

_sites [list of *Site*] The sites in the given order.

_labels_p [list or str] The labels use for the physical legs; just ['p0', 'p1', ..., 'p{L-1}'].

_labels_pconj [list or str] Just each of *_labels_p* with an *.

_pipe [*LegPipe*] The pipe from the single physical legs to the full combined leg.

_pipe_conj [*LegPipe*] Just _pipe.conj().

_mask [1D bool ndarray | None] Bool mask, which of the indices of the pipe are in the desired *charge_sector*.

Methods

build_full_H_from_bonds(self)	Calculate self.full_H from self.mpo.
build_full_H_from_mpo(self)	Calculate self.full_H from self.mpo.
exp_H(self, dt)	Return U(dt) := exp(-i H dt).
from_H_mpo(H_MPO, *args, **kwargs)	Wrapper taking directly an MPO instead of a Model.
full_diagonalization(self, *args, **kwargs)	Full diagonalization to obtain all eigenvalues and eigenvectors.
full_to_mps(self, psi[, canonical_form])	Convert a full state (with a single leg) to an MPS.
groundstate(self[, charge_sector])	Pick the ground state energy and ground state from self.V.
matvec(self, psi)	Allow to use *self* as LinearOperator for lanczos.
mps_to_full(self, mps)	Contract an MPS along the virtual bonds and combine its legs.
sparse_diag(self, k, *args, **kwargs)	Call *speigs()*.

classmethod from_H_mpo(*H_MPO*, **args*, ***kwargs*)
Wrapper taking directly an MPO instead of a Model.

Parameters

H_MPO [*MPO*] The MPO representing the Hamiltonian.

***args, **kwargs :** Other arguments as for the __init__ of the class.

build_full_H_from_mpo(*self*)
Calculate self.full_H from self.mpo.

build_full_H_from_bonds(*self*)
Calculate self.full_H from self.mpo.

full_diagonalization(*self*, **args*, ***kwargs*)
Full diagonalization to obtain all eigenvalues and eigenvectors.

Arguments are given to *eigh*.

groundstate(*self*, *charge_sector=None*)
Pick the ground state energy and ground state from `self.V`.

Parameters

charge_sector [None | 1D ndarray] By default (`None`), consider all charge sectors. Alternatively, give the *qtotal* which the returned state should have.

Returns

E0 [float] Ground state energy (possibly in the given sector).

psi0 [*Array*] Ground state (possibly in the given sector).

exp_H(*self*, *dt*)
Return `U(dt) := exp(-i H dt)`.

mps_to_full(*self*, *mps*)
Contract an MPS along the virtual bonds and combine its legs.

Parameters

mps [*MPS*] The MPS to be contracted.

Returns

psi [*Array*] The MPO contracted along the virtual bonds.

full_to_mps(*self*, *psi*, *canonical_form='B'*)
Convert a full state (with a single leg) to an MPS.

Parameters

psi [*Array*] The state (with a single leg) which should be splitted into an MPS.

canonical_from [*Array*] The form in which the MPS will be afterwards.

Returns

mps [*MPS*] An normalized MPS representation in canonical form.

matvec(*self*, *psi*)
Allow to use *self* as LinearOperator for lanczos.

Just applies *full_H* to (the first axis of) the given *psi*.

sparse_diag(*self*, *k*, **args*, ***kwargs*)
Call *speigs()*.

Module description

Full diagonalization (ED) of the Hamiltonian.

The full diagonalization of a small system is a simple approach to test other algorithms. In case you need the full spectrum, a full diagonalization is often the only way. This module provides functionality to quickly diagonalize the Hamiltonian of a given model. This might be used to obtain the spectrum, the ground state or highly excited states.

Note: Good use of symmetries is crucial to increase the treatable system size. While we can simply use the defined *LegCharge* of a model, we don't make use of any other symmetries like translation symmetry, SU(2) symmetry or inversion symmetries. In other words, this code does not aim to provide state-of-the-art exact diagonalization, but just the ability to diagonalize the defined models for small system sizes without addional extra work.

7.2.2 linalg

- full name: tenpy.linalg
- parent module: *tenpy*
- type: module

Module description

Linear-algebra tools for tensor networks.

Most notably is the module *np_conserved*, which contains everything needed to make use of charge conservervation in the context of tensor networks.

Relevant contents of *charges* are imported to *np_conserved*, so you propably won't need to import *charges* directly.

Submodules

np_conserved	A module to handle charge conservation in tensor networks.
charges	Basic definitions of a charge.
svd_robust	(More) robust version of singular value decomposition.
random_matrix	Provide some random matrix ensembles for numpy.
sparse	Providing support for sparse algorithms (using matrix-vector products only).
lanczos	Lanczos algorithm for np_conserved arrays.

np_conserved

- full name: tenpy.linalg.np_conserved
- parent module: *tenpy.linalg*
- type: module

Classes

Array(legcharges[, dtype, qtotal])	A multidimensional array (=tensor) for using charge conservation.

Array

- full name: tenpy.linalg.np_conserved.Array
- parent module: *tenpy.linalg.np_conserved*
- type: class

class tenpy.linalg.np_conserved.**Array**(*legcharges*, *dtype=<class 'numpy.float64'>*, *qtotal=None*)

Bases: object

A multidimensional array (=tensor) for using charge conservation.

An *Array* represents a multi-dimensional tensor, together with the charge structure of its legs (for abelian charges). Further information can be found in *Introduction to np_conserved*.

The default __init__() (i.e. Array(...)) does not insert any data, and thus yields an Array 'full' of zeros, equivalent to *zeros()*. Further, new arrays can be created with one of *from_ndarray_trivial()*, *from_ndarray()*, or *from_func()*, and of course by copying/tensordot/svd etc.

In-place methods are indicated by a name starting with i. (But *is_completely_blocked* is not inplace...)

Parameters

legcharges [list of *LegCharge*] The leg charges for each of the legs. The ChargeInfo is read out from it.

dtype [type or string] The data type of the array entries. Defaults to np.float64.

qtotal [1D array of QTYPE] The total charge of the array. Defaults to 0.

Attributes

size The number of dtype-objects stored.

stored_blocks The number of (non-zero) blocks stored in _data.

rank [int] The rank or "number of dimensions", equivalent to len(shape).

shape [tuple(int)] The number of indices for each of the legs.

dtype [np.dtype] The data type of the entries.

chinfo [*ChargeInfo*] The nature of the charge.

qtotal [1D array] The total charge of the tensor.

legs [list of *LegCharge*] The leg charges for each of the legs.

labels [dict (string -> int)] Labels for the different legs.

_data [list of arrays] The actual entries of the tensor.

_qdata [2D array (len(_data), rank), dtype np.intp] For each of the _data entries the qindices of the different legs.

_qdata_sorted [Bool] Whether self._qdata is lexsorted. Defaults to *True*, but *must* be set to *False* by algorithms changing _qdata.

Methods

add_charge(self, add_legs[, chinfo, qtotal])	Add charges.
add_leg(self, leg, i[, axis, label])	Add a leg to *self*, setting the current array as slice for a given index.
add_trivial_leg(self[, axis, label, qconj])	Add a trivial leg (with just one entry) to *self*.
as_completely_blocked(self)	Gives a version of self which is completely blocked by charges.
astype(self, dtype[, copy])	Return copy with new dtype, upcasting all blocks in `_data`.
binary_blockwise(self, func, other, *args, ...)	Roughly `return func(self, other)`, block-wise.
change_charge(self, charge, new_qmod[, ...])	Change the *qmod* of one charge in *chinfo*.
combine_legs(self, combine_legs[, new_axes, ...])	Reshape: combine multiple legs into multiple pipes.
complex_conj(self)	Return copy which is complex conjugated *without* conjugating the charge data.
conj(self[, complex_conj, inplace])	Conjugate: complex conjugate data, conjugate charge data.
copy(self[, deep])	Return a (deep or shallow) copy of self.
drop_charge(self[, charge, chinfo])	Drop (one of) the charges.
extend(self, axis, extra)	Increase the dimension of a given axis, filling the values with zeros.
from_func(func, legcharges[, dtype, qtotal, ...])	Create an Array from a numpy func.
from_func_square(func, leg[, dtype, ...])	Create an Array from a (numpy) function.
from_ndarray(data_flat, legcharges[, dtype, ...])	convert a flat (numpy) ndarray to an Array.
from_ndarray_trivial(data_flat[, dtype])	convert a flat numpy ndarray to an Array with trivial charge conservation.
gauge_total_charge(self, axis[, newqtotal, ...])	Changes the total charge by adjusting the charge on a certain leg.
get_block(self, qindices[, insert])	Return the ndarray in `_data` representing the block corresponding to *qindices*.
get_leg(self, label)	Return `self.legs[self.get_leg_index(label)]`.
get_leg_index(self, label)	translate a leg-index or leg-label to a leg-index.
get_leg_indices(self, labels)	Translate a list of leg-indices or leg-labels to leg indices.
get_leg_labels(self)	Return list of the leg labels, with *None* for anonymous legs.
has_label(self, label)	Check whether a given label exists.

Continued on next page

Table 47 – continued from previous page

iadd_prefactor_other(self, prefactor, other)	`self += prefactor * other` for scalar *prefactor* and *Array* *other*.
ibinary_blockwise(self, func, other, *args, ...)	Roughly `self = func(self, other)`, block-wise; in place.
iconj(self[, complex_conj])	Wraper around `self.conj()` with `inplace=True`.
idrop_labels(self[, old_labels])	Remove leg labels from self; in place.
iproject(self, mask, axes)	Applying masks to one or multiple axes; in place.
ipurge_zeros(self[, cutoff, norm_order])	Removes `self._data` blocks with *norm* less than cutoff; in place.
ireplace_label(self, old_label, new_label)	Replace the leg label *old_label* with *new_label*; in place.
ireplace_labels(self, old_labels, new_labels)	Replace leg label `old_labels[i]` with `new_labels[i]`; in place.
is_completely_blocked(self)	Return bool whether all legs are blocked by charge.
iscale_axis(self, s[, axis])	Scale with varying values along an axis; in place.
iscale_prefactor(self, prefactor)	`self *= prefactor` for scalar *prefactor*.
iset_leg_labels(self, labels)	Set labels for the different axes/legs; in place.
isort_qdata(self)	(Lexiographically) sort `self._qdata`; in place.
iswapaxes(self, axis1, axis2)	Similar as `np.swapaxes`; in place.
itranspose(self[, axes])	Transpose axes like *np.transpose*; in place.
iunary_blockwise(self, func, *args, **kwargs)	Roughly `self = f(self)`, block-wise; in place.
make_pipe(self, axes, **kwargs)	Generates a *LegPipe* for specified axes.
matvec(self, other)	This function is used by the Lanczos algorithm needed for DMRG.
norm(self[, ord, convert_to_float])	Norm of flattened data.
permute(self, perm, axis)	Apply a permutation in the indices of an axis.
replace_label(self, old_label, new_label)	Return a shallow copy with the leg label *old_label* replaced by *new_label*.
replace_labels(self, old_labels, new_labels)	Return a shallow copy with `old_labels[i]` replaced by `new_labels[i]`.
scale_axis(self, s[, axis])	Same as *iscale_axis()*, but return a (deep) copy.
sort_legcharge(self[, sort, bunch])	Return a copy with one or all legs sorted by charges.
sparse_stats(self)	Returns a string detailing the sparse statistics.
split_legs(self[, axes, cutoff])	Reshape: opposite of combine_legs: split (some) legs which are LegPipes.
squeeze(self[, axes])	Like `np.squeeze`.
take_slice(self, indices, axes)	Return a copy of self fixing *indices* along one or multiple *axes*.
test_sanity(self)	Sanity check.
to_ndarray(self)	Convert self to a dense numpy ndarray.
transpose(self[, axes])	Like *itranspose()*, but on a deep copy.
unary_blockwise(self, func, *args, **kwargs)	Roughly `return func(self)`, block-wise.
zeros_like(self)	Return a copy of self with only zeros as entries, containing no *_data*.

copy(*self*, *deep=True*)
Return a (deep or shallow) copy of self.

Both deep and shallow copies will share `chinfo` and the *LegCharges* in `legs`.

In contrast to a deep copy, the shallow copy will also share the tensor entries, namely the *same* instances of `_qdata` and `_data` and `labels` (and other 'immutable' properties like the shape or dtype).

Note: Shallow copies are *not* recommended unless you know the consequences! See the following examples illustrating some of the pitfalls.

Examples

Be (very!) careful when making non-deep copies: In the following example, the original *a* is changed if and only if the corresponding block existed in *a* before. >>> b = a.copy(deep=False) # shallow copy >>> b[1, 2] = 4.

Other *inplace* operations might have no effect at all (although we don't guarantee that):

```
>>> a *= 2  # has no effect on `b`
>>> b.iconj()  # nor does this change `a`
```

classmethod from_ndarray_trivial(*data_flat*, *dtype=None*)
convert a flat numpy ndarray to an Array with trivial charge conservation.

Parameters

data_flat [array_like] The data to be converted to a Array.

dtype [`np.dtype`] The data type of the array entries. Defaults to dtype of *data_flat*.

Returns

res [*Array*] An Array with data of data_flat.

classmethod from_ndarray(*data_flat*, *legcharges*, *dtype=None*, *qtotal=None*, *cutoff=None*)
convert a flat (numpy) ndarray to an Array.

Parameters

data_flat [array_like] The flat ndarray which should be converted to a npc *Array*. The shape has to be compatible with legcharges.

legcharges [list of `LegCharge`] The leg charges for each of the legs. The `ChargeInfo` is read out from it.

dtype [`np.dtype`] The data type of the array entries. Defaults to dtype of *data_flat*.

qtotal [None | charges] The total charge of the new array.

cutoff [float] Blocks with `np.max(np.abs(block)) > cutoff` are considered as zero. Defaults to `QCUTOFF`.

Returns

res [*Array*] An Array with data of *data_flat*.

See also:

detect_qtotal used to detect `qtotal` if not given.

classmethod from_func(*func*, *legcharges*, *dtype=None*, *qtotal=None*, *func_args=()*, *func_kwargs={}*, *shape_kw=None*)

Create an Array from a numpy func.

This function creates an array and fills the blocks *compatible* with the charges using *func*, where *func* is a function returning a *array_like* when given a shape, e.g. one of `np.ones` or `np.random.standard_normal`.

Parameters

func [callable] A function-like object which is called to generate the data blocks. We expect that *func* returns a flat array of the given *shape* convertible to *dtype*. If no *shape_kw* is given, it is called like `func(shape, *fargs, **fkwargs)`, otherwise as `func(*fargs, `shape_kw`=shape, **fkwargs)`. *shape* is a tuple of int.

legcharges [list of `LegCharge`] The leg charges for each of the legs. The `ChargeInfo` is read out from it.

dtype [None | type | string] The data type of the output entries. Defaults to np.float64. Defaults to *None*: obtain it from the return value of the function. Note that this argument is not given to func, but rather a type conversion is performed afterwards. You might want to set a *dtype* in *func_kwargs* as well.

qtotal [None | charges] The total charge of the new array. Defaults to charge 0.

func_args [iterable] Additional arguments given to *func*.

func_kwargs [dict] Additional keyword arguments given to *func*.

shape_kw [None | str] If given, the keyword with which shape is given to *func*.

Returns

res [*Array*] An Array with blocks filled using *func*.

classmethod from_func_square(*func*, *leg*, *dtype=None*, *func_args=()*, *func_kwargs={}*, *shape_kw=None*)

Create an Array from a (numpy) function.

This function creates an array and fills the blocks *compatible* with the charges using *func*, where *func* is a function returning a *array_like* when given a shape, e.g. one of `np.ones` or `np.random.standard_normal` or the functions defined in *random_matrix*.

Parameters

func [callable] A function-like object which is called to generate the data blocks. We expect that *func* returns a flat array of the given *shape* convertible to *dtype*. If no *shape_kw* is given, it is called like `func(shape, *fargs, **fkwargs)`, otherwise as `func(*fargs, `shape_kw`=shape, **fkwargs)`. *shape* is a tuple of int.

leg [`LegCharge`] The leg charges for the first leg; the second leg is set to `leg.conj()`. The `ChargeInfo` is read out from it.

dtype [None | type | string] The data type of the output entries. Defaults to *None*: obtain it from the return value of the function. Note that this argument is not given to func, but rather a type conversion is performed afterwards. You might want to set a *dtype* in *func_kwargs* as well.

func_args [iterable] Additional arguments given to *func*.

func_kwargs [dict] Additional keyword arguments given to *func*.

shape_kw [None | str] If given, the keyword with which shape is given to *func*.

Returns

res [*Array*] An Array with blocks filled using *func*.

zeros_like(*self*)
Return a copy of self with only zeros as entries, containing no *_data*.

test_sanity(*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

property size
The number of dtype-objects stored.

property stored_blocks
The number of (non-zero) blocks stored in `_data`.

get_leg_index(*self*, *label*)
translate a leg-index or leg-label to a leg-index.

Parameters

label [int | string] The leg-index directly or a label (string) set before.

Returns

leg_index [int] The index of the label.

See also:

get_leg_indices calls get_leg_index for a list of labels.

iset_leg_labels set the labels of different legs.

get_leg_indices(*self*, *labels*)
Translate a list of leg-indices or leg-labels to leg indices.

Parameters

labels [iterable of string/int] The leg-labels (or directly indices) to be translated in leg-indices.

Returns

leg_indices [list of int] The translated labels.

See also:

get_leg_index used to translate each of the single entries.

iset_leg_labels set the labels of different legs.

iset_leg_labels(*self*, *labels*)
Set labels for the different axes/legs; in place.

Introduction to leg labeling can be found in *Introduction to np_conserved*.

Parameters

labels [iterable (strings | None), len=self.rank] One label for each of the legs. An entry can be None for an anonymous leg.

See also:

get_leg translate the labels to indices.

get_leg_labels (*self*)
Return list of the leg labels, with *None* for anonymous legs.

has_label (*self*, *label*)
Check whether a given label exists.

get_leg (*self*, *label*)
Return `self.legs[self.get_leg_index(label)]`.

Convenient function returning the leg corresponding to a leg label/index.

ireplace_label (*self*, *old_label*, *new_label*)
Replace the leg label *old_label* with *new_label*; in place.

replace_label (*self*, *old_label*, *new_label*)
Return a shallow copy with the leg label *old_label* replaced by *new_label*.

ireplace_labels (*self*, *old_labels*, *new_labels*)
Replace leg label `old_labels[i]` with `new_labels[i]`; in place.

replace_labels (*self*, *old_labels*, *new_labels*)
Return a shallow copy with `old_labels[i]` replaced by `new_labels[i]`.

idrop_labels (*self*, *old_labels=None*)
Remove leg labels from self; in place.

Parameters

old_labels [list of str|int] The leg labels/indices for which the label should be removed. By default (None), remove all labels.

sparse_stats (*self*)
Returns a string detailing the sparse statistics.

to_ndarray (*self*)
Convert self to a dense numpy ndarray.

get_block (*self*, *qindices*, *insert=False*)
Return the ndarray in `_data` representing the block corresponding to *qindices*.

Parameters

qindices [1D array of np.intp] The qindices, for which we need to look in _qdata.

insert [bool] If True, insert a new (zero) block, if *qindices* is not existent in `self._data`. Otherwise just return `None`.

Returns

block: ndarray | None The block in `_data` corresponding to qindices. If *insert`=False and there is not block with qindices, return ``None`*.

Raises

IndexError If *qindices* are incompatible with charge and *raise_incomp_q*.

take_slice (*self*, *indices*, *axes*)
Return a copy of self fixing *indices* along one or multiple *axes*.

For a rank-4 Array `A.take_slice([i, j], [1,2])` is equivalent to `A[:, i, j, :]`.

Parameters

indices [(iterable of) int] The (flat) index for each of the legs specified by *axes*.

axes [(iterable of) str/int] Leg labels or indices to specify the legs for which the indices are given.

Returns

sliced_self [*Array*] A copy of self, equivalent to taking slices with indices inserted in axes.

See also:

add_leg opposite action of inserting a new leg.

add_trivial_leg(*self*, *axis=0*, *label=None*, *qconj=1*)
Add a trivial leg (with just one entry) to *self*.

Parameters

axis [int] The new leg is inserted before index *axis*.

label [str | None] If not None, use it as label for the new leg.

qconj [+1 | -1] The direction of the new leg.

Returns

extended [*Array*] A (possibly) *shallow* copy of self with an additional leg of ind_len 1 and charge 0.

add_leg(*self*, *leg*, *i*, *axis=0*, *label=None*)
Add a leg to *self*, setting the current array as slice for a given index.

Parameters

leg [LegCharge] The charge data of the leg to be added.

i [int] Index within the leg for which the data of *self* should be set.

axis [axis] The new leg is inserted before this current axis.

label [str | None] If not None, use it as label for the new leg.

Returns

extended [*Array*] A copy of self with the new *leg* at axis *axis* , such that `extended.take_slice(i, axis)` returns a copy of *self*.

See also:

take_slice opposite action reducing the number of legs.

extend(*self*, *axis*, *extra*)
Increase the dimension of a given axis, filling the values with zeros.

Parameters

axis [int | str] The axis (or axis-label) to be extended.

extra [LegCharge | int] By what to extend, i.e. the charges to be appended to the leg of *axis*. An int stands for extending the length of the array by a single new block of that size with zero charges.

Returns

extended [*Array*] A copy of self with the specified axis increased.

gauge_total_charge(*self*, *axis*, *newqtotal=None*, *new_qconj=None*)
Changes the total charge by adjusting the charge on a certain leg.

The total charge is given by finding a nonzero entry [i1, i2, ...] and calculating:

```
qtotal = self.chinfo.make_valid(
    np.sum([l.get_charge(l.get_qindex(qi)[0])
            for i, l in zip([i1,i2,...], self.legs)], axis=0))
```

Thus, the total charge can be changed by redefining (= shifting) the LegCharge of a single given leg. This is exaclty what this function does.

Parameters

axis [int or string] The new leg (index or label), for which the charge is changed.

newqtotal [charge values, defaults to 0] The new total charge.

new_qconj: {+1, -1, None} Whether the new LegCharge points inward (+1) or outward (-1) afterwards. By default (None) use the previous `self.legs[leg].qconj`.

Returns

copy [*Array*] A shallow copy of self with `copy.qtotal == newqtotal` and new `copy.legs[leg]`. The new leg will be a :class`LegCharge`, even if the old leg was a `LegPipe`.

add_charge(*self*, *add_legs*, *chinfo=None*, *qtotal=None*)
Add charges.

Parameters

add_legs [iterable of `LegCharge`] One *LegCharge* for each axis of *self*, to be added to the one in `legs`.

chargeinfo [`ChargeInfo`] The ChargeInfo for all charges; create new if `None`.

qtotal [None | charges] The total charge with respect to *add_legs*. If `None`, derive it from non-zero entries of `self`.

Returns

charges_added [*Array*] A copy of *self*, where the LegCharges *add_legs* where added to *self.legs*. Note that the LegCharges are neither bunched or sorted; you might want to use *sort_legcharge()*.

drop_charge(*self*, *charge=None*, *chinfo=None*)
Drop (one of) the charges.

Parameters

charge [int | str] Number or *name* of the charge (within *chinfo*) which is to be dropped. `None` means dropping all charges.

chinfo [`ChargeInfo`] The `ChargeInfo` with *charge* dropped; create a new one if `None`.

Returns

dropped [*Array*] A copy of *self*, where the specified *charge* has been removed. Note that the LegCharges are neither bunched or sorted; you might want to use *sort_legcharge()*.

change_charge(*self*, *charge*, *new_qmod*, *new_name=''*, *chinfo=None*)
Change the *qmod* of one charge in *chinfo*.

Parameters

charge [int | str] Number or *name* of the charge (within *chinfo*) which is to be changed. `None` means dropping all charges.

new_qmod [int] The new *qmod* to be set.

new_name [str] The new name of the charge.

chinfo [`ChargeInfo`] The `ChargeInfo` with *qmod* of *charge* changed; create a new one if `None`.

Returns

changed [*Array*] A copy of *self*, where the *qmod* of the specified *charge* has been changed. Note that the LegCharges are neither bunched or sorted; you might want to use *sort_legcharge()*.

is_completely_blocked(*self*)
Return bool whether all legs are blocked by charge.

sort_legcharge(*self*, *sort=True*, *bunch=True*)
Return a copy with one or all legs sorted by charges.

Sort/bunch one or multiple of the LegCharges. Legs which are sorted *and* bunched are guaranteed to be blocked by charge.

Parameters

sort [True | False | list of {True, False, perm}] A single bool holds for all legs, default=True. Else, *sort* should contain one entry for each leg, with a bool for sort/don't sort, or a 1D array perm for a given permuation to apply to a leg.

bunch [True | False | list of {True, False}] A single bool holds for all legs, default=True. Whether or not to bunch at each leg, i.e. combine contiguous blocks with equal charges.

Returns

perm [tuple of 1D arrays] The permutation applied to each of the legs, such that `cp.to_ndarray() = self.to_ndarray()[np.ix_(*perm)]`.

result [Array] A shallow copy of self, with legs sorted/bunched.

isort_qdata(*self*)
(Lexiographically) sort `self._qdata`; in place.

Lexsort `self._qdata` and `self._data` and set `self._qdata_sorted = True`.

make_pipe(*self*, *axes*, ***kwargs*)
Generates a *LegPipe* for specified axes.

Parameters

axes [iterable of str|int] The leg labels for the axes which should be combined. Order matters!

****kwargs :** Additional keyword arguments given to *LegPipe*.

Returns

pipe [*LegPipe*] A pipe of the legs specified by axes.

combine_legs(*self*, *combine_legs*, *new_axes=None*, *pipes=None*, *qconj=None*)
Reshape: combine multiple legs into multiple pipes. If necessary, transpose before.

Parameters

combine_legs [(iterable of) iterable of {str|int}] Bundles of leg indices or labels, which should be combined into a new output pipes. If multiple pipes should be created, use a list fore each new pipe.

new_axes [None | (iterable of) int] The leg-indices, at which the combined legs should appear in the resulting array. Default: for each pipe the position of its first pipe in the original array, (taking into account that some axes are 'removed' by combining). Thus no transposition is perfomed if *combine_legs* contains only contiguous ranges.

pipes [None | (iterable of) {`LegPipes` | None}] Optional: provide one or multiple of the resulting LegPipes to avoid overhead of computing new leg pipes for the same legs multiple times. The LegPipes are conjugated, if that is necessary for compatibility with the legs.

qconj [(iterable of) {+1, -1}] Specify whether new created pipes point inward or outward. Defaults to +1. Ignored for given *pipes*, which are not newly calculated.

Returns

reshaped [*Array*] A copy of self, whith some legs combined into pipes as specified by the arguments.

See also:

split_legs inverse reshaping splitting LegPipes.

Notes

Labels are inherited from self. New pipe labels are generated as `'(' + '.'.join(*leglabels) + ')'`. For these new labels, previously unlabeled legs are replaced by `'?#'`, where `#` is the leg-index in the original tensor *self*.

Examples

```
>>> oldarray.iset_leg_labels(['a', 'b', 'c', 'd', 'e'])
>>> c1 = oldarray.combine_legs([1, 2], qconj=-1)  # only single output pipe
>>> c1.get_leg_labels()
['a', '(b.c)', 'd', 'e']
```

Indices of *combine_legs* refer to the original array. If transposing is necessary, it is performed automatically:

```
>>> c2 = oldarray.combine_legs([[0, 3], [4, 1]], qconj=[+1, -1]) # two output␣
↪pipes
>>> c2.get_leg_labels()
['(a.d)', 'c', '(e.b)']
>>> c3 = oldarray.combine_legs([['a', 'd'], ['e', 'b']], new_axes=[2, 1],
>>>                            pipes=[c2.legs[0], c2.legs[2]])
>>> c3.get_leg_labels()
['c', '(e.b)', '(a.d)']
```

split_legs(*self*, *axes=None*, *cutoff=0.0*)

Reshape: opposite of combine_legs: split (some) legs which are LegPipes.

Reverts *combine_legs()* (except a possibly performed *transpose*). The splited legs are replacing the LegPipes at their position, see the examples below. Labels are split reverting what was done in *combine_legs()*. '?#' labels are replaced with `None`.

Parameters

axes [(iterable of) int|str] Leg labels or indices determining the axes to split. The corresponding entries in self.legs must be `LegPipe` instances. Defaults to all legs, which are `LegPipe` instances.

cutoff [float] Splitted data blocks with `np.max(np.abs(block)) > cutoff` are considered as zero. Defaults to 0.

Returns

reshaped [*Array*] A copy of self where the specified legs are splitted.

See also:

combine_legs this is reversed by split_legs.

Examples

Given a rank-5 Array *old_array*, you can combine it and split it again:

```
>>> old_array.iset_leg_labels(['a', 'b', 'c', 'd', 'e'])
>>> comb_array = old_array.combine_legs([[0, 3], [2, 4]] )
>>> comb_array.get_leg_labels()
['(a.d)', 'b', '(c.e)']
>>> split_array = comb_array.split_legs([0, 2])
>>> split_array.get_leg_labels()
['a', 'd', 'b', 'c', 'e']
```

as_completely_blocked(*self*)

Gives a version of self which is completely blocked by charges.

Functions like *svd()* or *eigh()* require a complete blocking by charges. This can be achieved by encapsulating each leg which is not completely blocked into a `LegPipe` (containing only that single leg). The LegPipe will then contain all necessary information to revert the blocking.

Returns

encapsulated_axes [list of int] The leg indices which have been encapsulated into Pipes.

blocked_self [*Array*] Self (if `len(encapsulated_axes) = 0`) or a copy of self, which is completely blocked.

squeeze(*self*, *axes=None*)

Like `np.squeeze`.

If a squeezed leg has non-zero charge, this charge is added to `qtotal`.

Parameters

axes [None | (iterable of) {int|str}] Labels or indices of the legs which should be 'squeezed', i.e. the legs removed. The corresponding legs must be trivial, i.e., have *ind_len* 1.

Returns

squeezed [:class:Array | scalar] A scalar of `self.dtype`, if all axes were squeezed. Else a copy of `self` with reduced `rank` as specified by *axes*.

astype(*self*, *dtype*, *copy=True*)

Return copy with new dtype, upcasting all blocks in `_data`.

Parameters

dtype [convertible to a np.dtype] The new data type. If None, deduce the new dtype as common type of `self._data`.

copy [bool] Whether to make a copy of the blocks even if the type didn't change.

Returns

copy [*Array*] Deep copy of self with new dtype.

ipurge_zeros(*self*, *cutoff=2.220446049250313e-15*, *norm_order=None*)
Removes `self._data` blocks with *norm* less than cutoff; in place.

Parameters

cutoff [float] Blocks with norm <= *cutoff* are removed. defaults to `QCUTOFF`.

norm_order : A valid *ord* argument for *np.linalg.norm*. Default `None` gives the Frobenius norm/2-norm for matrices/everything else. Note that this differs from other methods, e.g. *from_ndarray()*, which use the maximum norm.

iproject(*self*, *mask*, *axes*)
Applying masks to one or multiple axes; in place.

This function is similar as *np.compress* with boolean arrays For each specified axis, a boolean 1D array *mask* can be given, which chooses the indices to keep.

> **Warning:** Although it is possible to use an 1D int array as a mask, the order is ignored! If you need to permute an axis, use *permute()* or *sort_legcharge()*.

Parameters

mask [(list of) 1D array(boollint)] For each axis specified by *axes* a mask, which indices of the axes should be kept. If *mask* is a bool array, keep the indices where *mask* is True. If *mask* is an int array, keep the indices listed in the mask, *ignoring* the order or multiplicity.

axes [(list of) int | string] The *i`th entry in this list specifies the axis for the `i`th entry of `mask*, either as an int, or with a leg label. If axes is just a single int/string, specify just a single mask.

Returns

map_qind [list of 1D arrays] The mapping of qindices for each of the specified axes.

block_masks: list of lists of 1D bool arrays `block_masks[a][qind]` is a boolen mask which indices to keep in block `qindex` of `axes[a]`.

permute(*self*, *perm*, *axis*)
Apply a permutation in the indices of an axis.

Similar as np.take with a 1D array. Roughly equivalent to `res[:, ...] = self[perm, ...]` for the corresponding *axis*. Note: This function is quite slow, and usually not needed!

Parameters

perm [array_like 1D int] The permutation which should be applied to the leg given by *axis*.

axis [str | int] A leg label or index specifying on which leg to take the permutation.

Returns

res [*Array*] A copy of self with leg *axis* permuted, such that `res[i, ...] = self[perm[i], ...]` for `i` along *axis*.

See also:

sort_legcharge can also be used to perform a general permutation. Preferable, since it is faster for permutations which don't mix charge blocks.

itranspose(*self*, *axes=None*)
Transpose axes like *np.transpose*; in place.

Parameters

axes: iterable (int|string), len ``rank`` | None The new order of the axes. By default (None), reverse axes.

transpose(*self*, *axes=None*)
Like *itranspose()*, but on a deep copy.

iswapaxes(*self*, *axis1*, *axis2*)
Similar as `np.swapaxes`; in place.

iscale_axis(*self*, *s*, *axis=-1*)
Scale with varying values along an axis; in place.

Rescale to `new_self[i1, ..., i_axis, ...] = s[i_axis] * self[i1, ..., i_axis, ...]`.

Parameters

s [1D array, len=self.shape[axis]] The vector with which the axis should be scaled.

axis [str|int] The leg label or index for the axis which should be scaled.

See also:

iproject can be used to discard indices for which s is zero.

scale_axis(*self*, *s*, *axis=-1*)
Same as *iscale_axis()*, but return a (deep) copy.

iunary_blockwise(*self*, *func*, **args*, ***kwargs*)
Roughly `self = f(self)`, block-wise; in place.

Applies an unary function *func* to the non-zero blocks in `self._data`.

Note: Assumes implicitly that `func(np.zeros(...), *args, **kwargs)` gives 0, since we don't let *func* act on zero blocks!

Parameters

func [function] A function acting on flat arrays, returning flat arrays. It is called like `new_block = func(block, *args, **kwargs)`.

***args :** Additional arguments given to function *after* the block.

****kwargs :** Keyword arguments given to the function.

Examples

```
>>> a.iunaray_blockwise(np.real)  # get real part
>>> a.iunaray_blockwise(np.conj)  # same data as a.iconj(), but doesn't␣
↪charge conjugate.
```

unary_blockwise(*self*, *func*, **args*, ***kwargs*)
Roughly `return func(self)`, block-wise. Copies.

Same as *iunary_blockwise()*, but makes a **shallow** copy first.

iconj(*self*, *complex_conj=True*)
Wraper around `self.conj()` with `inplace=True`.

conj(*self*, *complex_conj=True*, *inplace=False*)
Conjugate: complex conjugate data, conjugate charge data.

Conjugate all legs, set negative qtotal.

Labeling: takes 'a' -> 'a*', 'a*'-> 'a' and '(a,(b*,c))' -> '(a*, (b, c*))'

Parameters

complex_conj [bool] Whether the data should be complex conjugated.

inplace [bool] Whether to apply changes to *self*, or to return a *deep* copy.

complex_conj(*self*)
Return copy which is complex conjugated *without* conjugating the charge data.

norm(*self*, *ord=None*, *convert_to_float=True*)
Norm of flattened data.

See *norm()* for details.

ibinary_blockwise(*self*, *func*, *other*, **args*, ***kwargs*)
Roughly `self = func(self, other)`, block-wise; in place.

Applies a binary function 'block-wise' to the non-zero blocks of `self._data` and `other._data`, storing result in place. Assumes that *other* is an *Array* as well, with the same shape and compatible legs. If leg labels of *other* and *self* are same up to permutations, *other* gets transposed accordingly before the action.

Note: Assumes implicitly that `func(np.zeros(...), np.zeros(...), *args, **kwargs)` gives 0, since we don't let *func* act on zero blocks!

Parameters

func [function] Binary function, called as `new_block = func(block_self, block_other, *args, **kwargs)` for blocks (=Numpy arrays) of equal shape.

other [*Array*] Other Array from which to the blocks.

***args, **kwargs:** Extra (keyword) arguments given to *func*.

Examples

```
>>> a.ibinary_blockwise(np.add, b)  # equivalent to ``a += b``, if ``b`` is
↪an `Array`.
>>> a.ibinary_blockwise(np.max, b)  # overwrites ``a`` to ``a = max(a, b)``
```

binary_blockwise(*self*, *func*, *other*, **args*, ***kwargs*)
Roughly `return func(self, other)`, block-wise. Copies.

Same as *ibinary_blockwise()*, but makes a **shallow** copy first.

matvec(*self*, *other*)
This function is used by the Lanczos algorithm needed for DMRG.

It is supposed to calculate the matrix - vector - product for a rank-2 matrix `self` and a rank-1 vector *other*.

iadd_prefactor_other(*self*, *prefactor*, *other*)
`self += prefactor * other` for scalar *prefactor* and *Array* *other*.

Note that we allow the type of *self* to change if necessary. Moreover, if *self* and *other* have the same labels in different order, other gets **transposed** before the action.

iscale_prefactor(*self*, *prefactor*)
`self *= prefactor` for scalar *prefactor*.

Note that we allow the type of *self* to change if necessary.

Functions

concatenate(arrays[, axis, copy])	Stack arrays along a given axis, similar as np.concatenate.
detect_grid_outer_legcharge(grid, grid_legs)	Derive a LegCharge for a grid used for *grid_outer()*.
detect_legcharge(flat_array, chargeinfo, ...)	Calculate a missing *LegCharge* by looking for nonzero entries of a flat array.
detect_qtotal(flat_array, legcharges[, cutoff])	Returns the total charge (w.r.t *legs*) of first non-zero sector found in *flat_array*.
diag(s, leg[, dtype])	Returns a square, diagonal matrix of entries *s*.
eig(a[, sort])	Calculate eigenvalues and eigenvectors for a non-hermitian matrix.
eigh(a[, UPLO, sort])	Calculate eigenvalues and eigenvectors for a hermitian matrix.
eigvals(a[, sort])	Calculate eigenvalues for a hermitian matrix.
eigvalsh(a[, UPLO, sort])	Calculate eigenvalues for a hermitian matrix.
expm(a)	Use scipy.linalg.expm to calculate the matrix exponential of a square matrix.
eye_like(a[, axis])	Return an identity matrix contractible with the leg *axis* of the *Array* *a*.
grid_concat(grid, axes[, copy])	Given an np.array of npc.Arrays, performs a multi-dimensional concatentation along 'axes'.
grid_outer(grid, grid_legs[, qtotal])	Given an np.array of npc.Arrays, return the corresponding higher-dimensional Array.
inner(a, b[, axes, do_conj])	Contract all legs in *a* and *b*, return scalar.
norm(a[, ord, convert_to_float])	Norm of flattened data.

Continued on next page

Table 48 – continued from previous page

outer(a, b)	Forms the outer tensor product, equivalent to `tensordot(a, b, axes=0)`.
pinv(a[, cutoff])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
qr(a[, mode, inner_labels, cutoff])	Q-R decomposition of a matrix.
speigs(a, charge_sector, k, *args, **kwargs)	Sparse eigenvalue decomposition `w, v` of square *a* in a given charge sector.
svd(a[, full_matrices, compute_uv, cutoff, ...])	Singualar value decomposition of an Array *a*.
tensordot(a, b[, axes])	Similar as `np.tensordot` but for *Array*.
to_iterable_arrays(array_list)	Similar as *to_iterable()*, but also enclose npc Arrays in a list.
trace(a[, leg1, leg2])	Trace of *a*, summing over leg1 and leg2.
zeros(legcharges[, dtype, qtotal])	Create a npc array full of zeros (with no _data).

concatenate

- full name: tenpy.linalg.np_conserved.concatenate
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**concatenate**(*arrays*, *axis=0*, *copy=True*)

Stack arrays along a given axis, similar as np.concatenate.

Stacks the qind of the array, without sorting/blocking. Labels are inherited from the first array only.

Parameters

arrays [iterable of *Array*] The arrays to be stacked. They must have the same shape and charge data except on the specified axis.

axis [int | str] Leg index or label of the first array. Defines the axis along which the arrays are stacked.

copy [bool] Whether to copy the data blocks.

Returns

stacked [*Array*] Concatenation of the given *arrays* along the specified axis.

See also:

Array.sort_legcharge can be used to block by charges along the axis.

detect_grid_outer_legcharge

- full name: tenpy.linalg.np_conserved.detect_grid_outer_legcharge
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**detect_grid_outer_legcharge**(*grid*, *grid_legs*, *qtotal=None*, *qconj=1*, *bunch=False*)

Derive a LegCharge for a grid used for *grid_outer()*.

Note: The resulting LegCharge is *not* bunched.

Parameters

grid [array_like of {*Array* | None}] The grid as it will be given to *grid_outer()*.

grid_legs [list of {LegCharge | None}] One LegCharge for each dimension of the grid, except for one entry which is None. This missing entry is to be calculated.

qtotal [charge] The desired total charge of the array. Defaults to 0.

Returns

new_grid_legs [list of LegCharge] A copy of the given *grid_legs* with the None replaced by a compatible LegCharge. The new LegCharge is neither bunched nor sorted!

See also:

detect_legcharge similar functionality for a flat numpy array instead of a grid.

detect_legcharge

- full name: tenpy.linalg.np_conserved.detect_legcharge
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**detect_legcharge**(*flat_array*, *chargeinfo*, *legcharges*, *qtotal=None*, *qconj=1*, *cutoff=None*)

Calculate a missing *LegCharge* by looking for nonzero entries of a flat array.

Parameters

flat_array [ndarray] A flat array, in which we look for non-zero entries.

chargeinfo [*ChargeInfo*] The nature of the charge.

legcharges [list of LegCharge] One LegCharge for each dimension of flat_array, except for one entry which is None. This missing entry is to be calculated.

qconj [{+1, -1}] *qconj* for the new calculated LegCharge.

qtotal [charges] Desired total charge of the array. Defaults to zeros.

cutoff [float] Blocks with `np.max(np.abs(block)) > cutoff` are considered as zero. Defaults to QCUTOFF.

Returns

new_legcharges [list of LegCharge] A copy of the given *legcharges* with the None replaced by a compatible LegCharge. The new legcharge is 'bunched', but not sorted!

See also:

detect_grid_outer_legcharge similar functionality if the flat array is given by a 'grid'.

detect_qtotal detects the total charge, if all legs are known.

detect_qtotal

- full name: tenpy.linalg.np_conserved.detect_qtotal
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**detect_qtotal**(*flat_array*, *legcharges*, *cutoff=None*)
Returns the total charge (w.r.t *legs*) of first non-zero sector found in *flat_array*.

Parameters

flat_array [array] The flat numpy array from which you want to detect the charges.

legcharges [list of `LegCharge`] For each leg the LegCharge.

cutoff [float] Blocks with `np.max(np.abs(block)) > cutoff` are considered as zero. Defaults to `QCUTOFF`.

Returns

qtotal [charge] The total charge fo the first non-zero (i.e. > cutoff) charge block.

See also:

detect_legcharge detects the charges of one missing LegCharge if *qtotal* is known.

detect_grid_outer_legcharge similar functionality if the flat array is given by a 'grid'.

diag

- full name: tenpy.linalg.np_conserved.diag
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**diag**(*s*, *leg*, *dtype=None*)
Returns a square, diagonal matrix of entries *s*.

The resulting matrix has legs `(leg, leg.conj())` and charge 0.

Parameters

s [scalar | 1D array] The entries to put on the diagonal. If scalar, all diagonal entries are the same.

leg [`LegCharge`] The first leg of the resulting matrix.

dtype [None | type] The data type to be used for the result. By default, use dtype of *s*.

Returns

diagonal [*Array*] A square matrix with diagonal entries *s*.

See also:

Array.scale_axis similar as `tensordot(diag(s), ...)`, but faster.

eig

- full name: tenpy.linalg.np_conserved.eig
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**eig**(*a*, *sort=None*)
Calculate eigenvalues and eigenvectors for a non-hermitian matrix.

`W, V = eig(a)` yields $aV = Vdiag(w)$.

Parameters

a [*Array*] The hermitian square matrix to be diagonalized.

sort [{'m>', 'm<', '>', '<', None}] How the eigenvalues should are sorted *within* each charge block. Defaults to None, which is same as '<'. See `argsort()` for details.

Returns

W [1D ndarray] The eigenvalues, sorted within the same charge blocks according to *sort*.

V [*Array*] Unitary matrix; `V[:, i]` is normalized eigenvector with eigenvalue `W[i]`. The first label is inherited from *A*, the second label is `'eig'`.

Notes

Requires the legs to be contractible. If *a* is not blocked by charge, a blocked copy is made via a permutation `P`, :math:` a' = P a P = V' W' (V')^{dagger}`. The eigenvectors V are then obtained by the reverse permutation, $V = P^{-1}V'$ such that *A = V W V^{dagger}*.

eigh

- full name: tenpy.linalg.np_conserved.eigh
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**eigh**(*a*, *UPLO='L'*, *sort=None*)
Calculate eigenvalues and eigenvectors for a hermitian matrix.

`W, V = eigh(a)` yields $a = Vdiag(w)V^\dagger$. **Assumes** that a is hermitian, `a.conj().transpose() == a`.

Parameters

a [*Array*] The hermitian square matrix to be diagonalized.

UPLO [{'L', 'U'}] Whether to take the lower ('L', default) or upper ('U') triangular part of *a*.

sort [{'m>', 'm<', '>', '<', None}] How the eigenvalues should are sorted *within* each charge block. Defaults to None, which is same as '<'. See `argsort()` for details.

Returns

W [1D ndarray] The eigenvalues, sorted within the same charge blocks according to *sort*.

V [*Array*] Unitary matrix; `V[:, i]` is normalized eigenvector with eigenvalue `W[i]`. The first label is inherited from *A*, the second label is `'eig'`.

Notes

Requires the legs to be contractible. If a is not blocked by charge, a blocked copy is made via a permutation `P`, :math:` a' = P a P = V' W' (V')^{dagger}`. The eigenvectors V are then obtained by the reverse permutation, $V = P^{-1}V'$ such that A = $V W V^{dagger}$.

eigvals

- full name: tenpy.linalg.np_conserved.eigvals
- parent module: *tenpy.linalg.np_conserved*
- type: function

`tenpy.linalg.np_conserved.`**`eigvals`**(*a*, *sort=None*)
Calculate eigenvalues for a hermitian matrix.

Parameters

a [*Array*] The hermitian square matrix to be diagonalized.

sort [{'m>', 'm<', '>', '<', `None`}] How the eigenvalues should are sorted *within* each charge block. Defaults to `None`, which is same as '<'. See `argsort()` for details.

Returns

W [1D ndarray] The eigenvalues, sorted within the same charge blocks according to *sort*.

Notes

The eigenvalues are sorted within blocks of the completely blocked legs.

eigvalsh

- full name: tenpy.linalg.np_conserved.eigvalsh
- parent module: *tenpy.linalg.np_conserved*
- type: function

`tenpy.linalg.np_conserved.`**`eigvalsh`**(*a*, *UPLO='L'*, *sort=None*)
Calculate eigenvalues for a hermitian matrix.

Assumes that a is hermitian, `a.conj().transpose() == a`.

Parameters

a [*Array*] The hermitian square matrix to be diagonalized.

UPLO [{'L', 'U'}] Whether to take the lower ('L', default) or upper ('U') triangular part of *a*.

sort [{'m>', 'm<', '>', '<', `None`}] How the eigenvalues should are sorted *within* each charge block. Defaults to `None`, which is same as '<'. See `argsort()` for details.

Returns

W [1D ndarray] The eigenvalues, sorted within the same charge blocks according to *sort*.

Notes

The eigenvalues are sorted within blocks of the completely blocked legs.

expm

- full name: tenpy.linalg.np_conserved.expm
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**expm**(*a*)
Use scipy.linalg.expm to calculate the matrix exponential of a square matrix.

Parameters

a [*Array*] A square matrix to be exponentiated.

Returns

exp_a [*Array*] The matrix exponential `expm(a)`, calculated using scipy.linalg.expm. Same legs/labels as *a*.

eye_like

- full name: tenpy.linalg.np_conserved.eye_like
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**eye_like**(*a*, *axis=0*)
Return an identity matrix contractible with the leg *axis* of the *Array* *a*.

grid_concat

- full name: tenpy.linalg.np_conserved.grid_concat
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**grid_concat**(*grid*, *axes*, *copy=True*)
Given an np.array of npc.Arrays, performs a multi-dimensional concatentation along 'axes'.

Similar to `numpy.block()`, but only for uniform blocking.

Stacks the qind of the array, *without* sorting/blocking.

Parameters

grid [array_like of *Array*] The grid of arrays.

axes [list of int] The axes along which to concatenate the arrays, same len as the dimension of the grid. Concatenate arrays of the *i`th axis of the grid along the axis ``axes[i]`*

copy [bool] Whether the _data blocks are copied.

See also:

Array.sort_legcharge can be used to block by charges.

Examples

Assume we have rank 2 Arrays `A, B, C, D` of shapes `(1, 2), (1, 4), (3, 2), (3, 4)` sharing the legs of equal sizes. Then the following grid will result in a `(1+3, 2+4)` shaped array:

```
>>> g = grid_concat([[A, B], [C, D]], axes=[0, 1])
>>> g.shape
(4, 6)
```

If `A, B, C, D` were rank 4 arrays, with the first and last leg as before, and sharing *common* legs `1` and `2` of dimensions 1, 2, then you would get a rank-4 array:

```
>>> g = grid_concat([[A, B], [C, D]], axes=[0, 3])
>>> g.shape
(4, 1, 2, 6)
```

grid_outer

- full name: tenpy.linalg.np_conserved.grid_outer
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**grid_outer**(*grid*, *grid_legs*, *qtotal=None*)

Given an np.array of npc.Arrays, return the corresponding higher-dimensional Array.

Parameters

grid [array_like of {*Array* | None}] The grid gives the first part of the axes of the resulting array. Entries have to have all the same shape and charge-data, giving the remaining axes. `None` entries in the grid are interpreted as zeros.

grid_legs [list of `LegCharge`] One LegCharge for each dimension of the grid along the grid.

qtotal [charge] The total charge of the Array. By default (`None`), derive it out from a non-trivial entry of the grid.

Returns

res [*Array*] An Array with shape `grid.shape + nontrivial_grid_entry.shape`. Constructed such that `res[idx] == grid[idx]` for any index `idx` of the *grid* the *grid* entry is not trivial (`None`).

See also:

detect_grid_outer_legcharge can calculate one missing `LegCharge` of the grid.

Examples

A typical use-case for this function is the generation of an MPO. Say you have npc.Arrays `Splus, Sminus, Sz`, each with legs `[phys.conj(), phys]`. Further, you have to define appropriate LegCharges *l_left* and *l_right*. Then one 'matrix' of the MPO for a nearest neighbour Heisenberg Hamiltonian could look like:

```
>>> Id = np.eye_like(Sz)
>>> W_mpo = grid_outer([[Id, Splus, Sminus, Sz, None],
...                     [None, None, None, None, J*Sminus],
...                     [None, None, None, None, J*Splus],
...                     [None, None, None, None, J*Sz],
...                     [None, None, None, None, Id]],
...                    leg_charges=[l_left, l_right])
>>> W_mpo.shape
(5, 5, 2, 2)
```

inner

- full name: tenpy.linalg.np_conserved.inner
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**inner**(*a*, *b*, *axes=None*, *do_conj=False*)

Contract all legs in *a* and *b*, return scalar.

Parameters

a, b [class:*Array*] The arrays for which to calculate the product. Must have same rank, and compatible LegCharges.

axes [`(axes_a, axes_b) | 'range', 'labels'`] *axes_a* and *axes_b* specifiy the legs of *a* and *b*, respectively, which should be contracted. Legs can be specified with leg labels or indices. We contract leg `axes_a[i]` of *a* with leg `axes_b[i]` of *b*. The default `axes='range'` is equivalent to `(range(rank), range(rank))`. `axes='labels'` is equivalent to either `(a.get_leg_labels(), a.get_leg_labels())` for `do_conj=True`, or to `(a.get_leg_labels(), conj_labels(a.get_leg_labels()))` for `do_conj=False`. In other words, `axes='labels'` requires *a* and *b* to have the same/conjugated labels up to a possible transposition, which is then reverted.

do_conj [bool] If `False` (Default), ignore it. if `True`, conjugate *a* before, i.e., return `inner(a.conj(), b, axes)`

Returns

inner_product [dtype] A scalar (of common dtype of *a* and *b*) giving the full contraction of *a* and *b*.

norm

- full name: tenpy.linalg.np_conserved.norm
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**norm**(*a*, *ord=None*, *convert_to_float=True*)
Norm of flattened data.

Equivalent to `np.linalg.norm(a.to_ndarray().flatten(), ord)`.

In contrast to numpy, we don't distinguish between matrices and vectors, but simply calculate the norm for the **flat** (block) data. The usual *ord*-norm is defined as $(\sum_i |a_i|^{ord})^{1/ord}$.

ord	norm
None/'fro'	Frobenius norm (same as 2-norm)
np.inf	`max(abs(x))`
-np.inf	`min(abs(x))`
0	`sum(a != 0) == np.count_nonzero(x)`
other	ususal *ord*-norm

Parameters

a [*Array* | np.ndarray] The array of which the norm should be calculated.

ord : The order of the norm. See table above.

convert_to_float : Convert integer to float before calculating the norm, avoiding int overflow.

Returns

norm [float] The norm over the *flat* data of the array.

outer

- full name: tenpy.linalg.np_conserved.outer
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**outer**(*a*, *b*)
Forms the outer tensor product, equivalent to `tensordot(a, b, axes=0)`.

Labels are inherited from *a* and *b*. In case of a collision (same label in both *a* and *b*), they are both dropped.

Parameters

a, b [*Array*] The arrays for which to form the product.

Returns

c [*Array*] Array of rank `a.rank + b.rank` such that (for `Ra = a.rank; Rb = b.rank`):

```
c[i_1, ..., i_Ra, j_1, ... j_R] = a[i_1, ..., i_Ra] * b[j_1, ..., j_rank_b]
```

pinv

- full name: tenpy.linalg.np_conserved.pinv
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**pinv**(*a*, *cutoff=1e-15*)
Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Equivalent to the following procedure: Perform a SVD, `U, S, VH = svd(a, cutoff=cutoff)` with a *cutoff* > 0, calculate `P = U * diag(1/S) * VH` (with `*` denoting tensordot) and return `P.conj.transpose()`.

Parameters

a [(M, N) *Array*] Matrix to be pseudo-inverted.

cuttof [float] Cutoff for small singular values, as given to *svd()*. (Note: different convetion than numpy.)

Returns

B [(N, M) *Array*] The pseudo-inverse of *a*.

qr

- full name: tenpy.linalg.np_conserved.qr
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**qr**(*a, mode='reduced', inner_labels=[None, None], cutoff=None*)
Q-R decomposition of a matrix.

Decomposition such that `A == npc.tensordot(q, r, axes=1)` up to numerical rounding errors.

Parameters

a [*Array*] A square matrix to be exponentiated, shape `(M,N)`.

mode [‘reduced’, ‘complete’] ‘reduced’: return *q* and *r* with shapes (M,K) and (K,N), where K=min(M,N) ‘complete’: return *q* with shape (M,M).

inner_labels: [{str|None}, {str|None}] The first label is used for `Q.legs[1]`, the second for `R.legs[0]`.

cutoff [`None` or float] If not `None`, discard linearly dependent vectors to given precision, which might reduce *K* of the ‘reduced’ mode even further.

Returns

q [*Array*] If *mode* is ‘complete’, a unitary matrix. For *mode* ‘reduced’ such thatOtherwise such that $q^*_{j,i} q_{j,k} = \delta_{i,k}$

r [*Array*] Upper triangular matrix if both legs of A are sorted by charges; Otherwise a simple transposition (performed when sorting by charges) brings it to upper triangular form.

speigs

- full name: tenpy.linalg.np_conserved.speigs
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**speigs**(*a*, *charge_sector*, *k*, **args*, ***kwargs*)
Sparse eigenvalue decomposition `w, v` of square *a* in a given charge sector.

Finds *k* right eigenvectors (chosen by `kwargs['which']`) in a given charge sector, `tensordot(A, V[i], axes=1) = W[i] * V[i]`.

Parameters

a [*Array*] A square array with contractible legs and vanishing total charge.

charge_sector [charges] *ndim* charges to select the block.

k [int] How many eigenvalues/vectors should be calculated. If the block of *charge_sector* is smaller than *k*, *k* may be reduced accordingly.

***args, **kwargs :** Additional arguments given to *scipy.sparse.linalg.eigs*.

Returns

W [ndarray] *k* (or less) eigenvalues

V [list of *Array*] *k* (or less) right eigenvectors of *A* with total charge *charge_sector*. Note that when interpreted as a matrix, this is the transpose of what `np.eigs` normally gives.

svd

- full name: tenpy.linalg.np_conserved.svd
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**svd**(*a*, *full_matrices=False*, *compute_uv=True*, *cutoff=None*, *qtotal_LR=[None, None]*, *inner_labels=[None, None]*, *inner_qconj=1*)
Singular value decomposition of an Array *a*.

Factorizes `U, S, VH = svd(a)`, such that `a = U*diag(S)*VH` (where `*` stands for a *tensordot()* and *diag* creates an correctly shaped Array with *S* on the diagonal). For a non-zero *cutoff* this holds only approximately.

There is a gauge freedom regarding the charges, see also *Array.gauge_total_charge()*. We ensure contractibility by setting `U.legs[1] = VH.legs[0].conj()`. Further, we gauge the LegCharge such that *U* and *V* have the desired *qtotal_LR*.

Parameters

a [*Array*, shape `(M, N)`] The matrix to be decomposed.

full_matrices [bool] If `False` (default), *U* and *V* have shapes `(M, K)` and `(K, N)`, where `K=len(S)`. If `True`, *U* and *V* are full square unitary matrices with shapes `(M, M)` and `(N, N)`. Note that the arrays are not directly contractible in that case; `diag(S)` would need to be a rectangluar `(M, N)` matrix.

compute_uv [bool] Whether to compute and return *U* and *V*.

cutoff [None | float] Keep only singular values which are (strictly) greater than *cutoff*. (Then the factorization holds only approximately). If None (default), ignored.

qtotal_LR [[{charges|None}, {charges|None}]] The desired *qtotal* for *U* and *VH*, respectively. [None, None] (Default) is equivalent to [None, a.qtotal]. A single *None* entry is replaced the unique charge satisfying the requirement U.qtotal + VH.qtotal = a.qtotal (modulo qmod).

inner_labels_LR: [{str|None}, {str|None}] The first label corresponds to U.legs[1], the second to VH.legs[0].

inner_qconj [{+1, -1}] Direction of the charges for the new leg. Default +1. The new LegCharge is constructed such that VH.legs[0].qconj = qconj.

Returns

U [*Array*] Matrix with left singular vectors as columns. Shape (M, M) or (M, K) depending on *full_matrices*.

S [1D ndarray] The singluar values of the array. If no *cutoff* is given, it has lenght min(M, N).

VH [*Array*] Matrix with right singular vectors as rows. Shape (N, N) or (K, N) depending on *full_matrices*.

tensordot

- full name: tenpy.linalg.np_conserved.tensordot
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**tensordot**(*a*, *b*, *axes=2*)

Similar as np.tensordot but for *Array*.

Builds the tensor product of *a* and *b* and sums over the specified axes. Does not require complete blocking of the charges.

Labels are inherited from *a* and *b*. In case of a collision (= the same label would be inherited from *a* and *b* after the contraction), both labels are dropped.

Detailed implementation notes are available in the doc-string of _tensordot_worker().

Parameters

a, b [*Array*] The first and second npc Array for which axes are to be contracted.

axes [(axes_a, axes_b) | int] A single integer is equivalent to (range(-axes, 0), range(axes)). Alternatively, *axes_a* and *axes_b* specifiy the legs of *a* and *b*, respectively, which should be contracted. Legs can be specified with leg labels or indices. Contract leg axes_a[i] of *a* with leg axes_b[i] of *b*.

Returns

a_dot_b [*Array*] The tensorproduct of *a* and *b*, summed over the specified axes. Returns a scalar in case of a full contraction.

to_iterable_arrays

- full name: tenpy.linalg.np_conserved.to_iterable_arrays
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**to_iterable_arrays**(*array_list*)
Similar as *to_iterable()*, but also enclose npc Arrays in a list.

trace

- full name: tenpy.linalg.np_conserved.trace
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**trace**(*a*, *leg1=0*, *leg2=1*)
Trace of *a*, summing over leg1 and leg2.

Requires that the contracted legs are contractible (i.e. have opposite charges). Labels are inherited from *a*.

Parameters

leg1, leg2: str|int The leg label or index for the two legs which should be contracted (i.e. summed over).

Returns

traced [*Array* | `a.dtype`] A scalar if `a.rank == 2`, else an *Array* of rank `a.rank - 2`. Equivalent to `sum([a.take_slice([i, i], [leg1, leg2]) for i in range(a.shape[leg1])])`.

zeros

- full name: tenpy.linalg.np_conserved.zeros
- parent module: *tenpy.linalg.np_conserved*
- type: function

tenpy.linalg.np_conserved.**zeros**(*legcharges*, *dtype=<class 'numpy.float64'>*, *qtotal=None*)
Create a npc array full of zeros (with no _data).

This is just a wrapper around `Array(...)`, detailed documentation can be found in the class doc-string of *Array*.

Module description

A module to handle charge conservation in tensor networks.

A detailed introduction to this module (including notations) can be found in *Introduction to np_conserved*.

This module *np_conserved* implements a class `Array` designed to make use of charge conservation in tensor networks. The idea is that the *Array* class is used in a fashion very similar to the `numpy.ndarray`, e.g you can call the functions `tensordot()` or `svd()` (of this module) on them. The structure of the algorithms (as DMRG) is thus the same as with basic numpy ndarrays.

Internally, an `Array` saves charge meta data to keep track of blocks which are nonzero. All possible operations (e.g. tensordot, svd, ...) on such arrays preserve the total charge structure. In addition, these operations make use of the charges to figure out which of the blocks it has to use/combine - this is the basis for the speed-up.

Overview

Classes

`Array`(legcharges[, dtype, qtotal])	A multidimensional array (=tensor) for using charge conservation.
`ChargeInfo`([mod, names])	Meta-data about the charge of a tensor.
`LegCharge`(chargeinfo, slices, charges[, qconj])	Save the charge data associated to a leg of a tensor.
`LegPipe`(legs[, qconj, sort, bunch])	A *LegPipe* combines multiple legs of a tensor to one.

Array creation

`Array.from_ndarray_trivial`(data_flat[, dtype])	convert a flat numpy ndarray to an Array with trivial charge conservation.
`Array.from_ndarray`(data_flat, legcharges[, ...])	convert a flat (numpy) ndarray to an Array.
`Array.from_func`(func, legcharges[, dtype, ...])	Create an Array from a numpy func.
`Array.from_func_square`(func, leg[, dtype, ...])	Create an Array from a (numpy) function.
`zeros`(legcharges[, dtype, qtotal])	Create a npc array full of zeros (with no _data).
`eye_like`(a[, axis])	Return an identity matrix contractible with the leg *axis* of the `Array` *a*.
`diag`(s, leg[, dtype])	Returns a square, diagonal matrix of entries *s*.

Concatenation

`concatenate`(arrays[, axis, copy])	Stack arrays along a given axis, similar as np.concatenate.
`grid_concat`(grid, axes[, copy])	Given an np.array of npc.Arrays, performs a multi-dimensional concatentation along 'axes'.
`grid_outer`(grid, grid_legs[, qtotal])	Given an np.array of npc.Arrays, return the corresponding higher-dimensional Array.

Detecting charges of flat arrays

detect_qtotal(flat_array, legcharges[, cutoff])	Returns the total charge (w.r.t *legs*) of first non-zero sector found in *flat_array*.
detect_legcharge(flat_array, chargeinfo, ...)	Calculate a missing *LegCharge* by looking for nonzero entries of a flat array.
detect_grid_outer_legcharge(grid, grid_legs)	Derive a LegCharge for a grid used for *grid_outer()*.

Contraction of some legs

tensordot(a, b[, axes])	Similar as `np.tensordot` but for *Array*.
outer(a, b)	Forms the outer tensor product, equivalent to `tensordot(a, b, axes=0)`.
inner(a, b[, axes, do_conj])	Contract all legs in *a* and *b*, return scalar.
trace(a[, leg1, leg2])	Trace of *a*, summing over leg1 and leg2.

Linear algebra

svd(a[, full_matrices, compute_uv, cutoff, ...])	Singualar value decomposition of an Array *a*.
pinv(a[, cutoff])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
norm(a[, ord, convert_to_float])	Norm of flattened data.
qr(a[, mode, inner_labels, cutoff])	Q-R decomposition of a matrix.
expm(a)	Use scipy.linalg.expm to calculate the matrix exponential of a square matrix.

Eigen systems

eigh(a[, UPLO, sort])	Calculate eigenvalues and eigenvectors for a hermitian matrix.
eig(a[, sort])	Calculate eigenvalues and eigenvectors for a non-hermitian matrix.
eigvalsh(a[, UPLO, sort])	Calculate eigenvalues for a hermitian matrix.
eigvals(a[, sort])	Calculate eigenvalues for a hermitian matrix.
speigs(a, charge_sector, k, *args, **kwargs)	Sparse eigenvalue decomposition `w, v` of square *a* in a given charge sector.

charges

- full name: tenpy.linalg.charges
- parent module: *tenpy.linalg*
- type: module

Classes

ChargeInfo([mod, names])	Meta-data about the charge of a tensor.
LegCharge(chargeinfo, slices, charges[, qconj])	Save the charge data associated to a leg of a tensor.
LegPipe(legs[, qconj, sort, bunch])	A *LegPipe* combines multiple legs of a tensor to one.

ChargeInfo

- full name: tenpy.linalg.charges.ChargeInfo
- parent module: *tenpy.linalg.charges*
- type: class

class tenpy.linalg.charges.**ChargeInfo**(*mod=[]*, *names=None*)

Bases: object

Meta-data about the charge of a tensor.

Saves info about the nature of the charge of a tensor. Provides *make_valid()* for taking modulo *m*.

(This class is implemented in *tenpy.linalg.charges* but also imported in *tenpy.linalg.np_conserved* for convenience.)

Parameters

mod [iterable of QTYPE] The len gives the number of charges, *qnumber*. For each charge one entry *m*: the charge is conserved modulo *m*. Defaults to trivial, i.e., no charge.

names [list of str] Descriptive names for the charges. Defaults to `['']*qnumber`.

Notes

Instances of this class can (should) be shared between different *LegCharge* and *Array*'s.

Attributes

qnumber [int] The number of charges.

mod [ndarray[QTYPE,ndim=1]] Modulo how much each of the charges is taken.

names [list of strings] A descriptive name for each of the charges. May have '' entries.

_mask_mod1 [1D array bool] mask `(mod == 1)`, to speed up *make_valid* in pure python.

_mod_masked [1D array QTYPE] Equivalent to `self.mod[self._maks_mod1]`

_qnumber, _mod : Storage of *qnumber* and *mod*.

Methods

add(chinfos)	Create a *ChargeInfo* combining multiple charges.
change(chinfo, charge, new_qmod[, new_name])	Change the *qmod* of a given charge.
check_valid(self, charges)	Check, if *charges* has all entries as expected from self.mod.
drop(chinfo[, charge])	Remove a charge from a *ChargeInfo*.
make_valid(self[, charges])	Take charges modulo self.mod.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

classmethod add(*chinfos*)
Create a *ChargeInfo* combining multiple charges.

Parameters

chinfos [iterable of *ChargeInfo*] ChargeInfo instances to be combined into a single one (in the given order).

Returns

chinfo [*ChargeInfo*] ChargeInfo combining all the given charges.

classmethod drop(*chinfo*, *charge=None*)
Remove a charge from a *ChargeInfo*.

Parameters

chinfo [*ChargeInfo*] The ChargeInfo from where to drop/remove a charge.

charge [int | str] Number or *name* of the charge (within *chinfo*) which is to be dropped. `None` means dropping all charges.

Returns

chinfo [*ChargeInfo*] ChargeInfo where the specified charge is dropped.

classmethod change(*chinfo*, *charge*, *new_qmod*, *new_name=''*)
Change the *qmod* of a given charge.

Parameters

chinfo [*ChargeInfo*] The ChargeInfo for which *qmod* of *charge* should be changed.

new_qmod [int] The new *qmod* to be set.

new_name [str] The new name of the charge.

Returns

chinfo [*ChargeInfo*] ChargeInfo where *qmod* of the specified charge was changed.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

property qnumber
The number of charges.

property mod
Modulo how much each of the charges is taken.

1 for a U(1) charge, i.e., mod 1 -> mod infinity.

make_valid(*self*, *charges=None*)
Take charges modulo self.mod.

Parameters

charges [array_like or None] 1D or 2D array of charges, last dimension *self.qnumber* None defaults to trivial charges `np.zeros(qnumber, dtype=QTYPE)`.

Returns

charges : A copy of *charges* taken modulo *mod*, but with `x % 1 := x`

check_valid(*self*, *charges*)
Check, if *charges* has all entries as expected from self.mod.

Parameters

charges [2D ndarray QTYPE_t] Charge values to be checked.

Returns

res [bool] True, if all 0 <= charges <= self.mod (wherever self.mod != 1)

LegCharge

- full name: tenpy.linalg.charges.LegCharge
- parent module: *tenpy.linalg.charges*
- type: class

class tenpy.linalg.charges.**LegCharge**(*chargeinfo*, *slices*, *charges*, *qconj=1*)
Bases: object

Save the charge data associated to a leg of a tensor.

This class is more or less a wrapper around a 2D numpy array *charges* and a 1D array *slices*. See *Introduction to np_conserved* for more details.

(This class is implemented in *tenpy.linalg.charges* but also imported in *tenpy.linalg.np_conserved* for convenience.)

Parameters

chargeinfo [*ChargeInfo*] The nature of the charge.

slices: 1D array_like, len(block_number+1) A block with 'qindex' `qi` correspondes to the leg indices in `slice(slices[qi], slices[qi+1])`.

charges [2D array_like, shape(block_number, chargeinfo.qnumber)] `charges[qi]` gives the charges for a block with 'qindex' `qi`.

qconj [{+1, -1}] A flag telling whether the charge points inwards (+1, default) or outwards (-1).

Notes

Instances of this class can be shared between different *npc.Array*. Thus, functions changing `self.slices` or `self.charges` *must* always make copies. Further they *must* set *sorted* and *bunched* to `False` (if they might not preserve them).

Attributes

ind_len: int The number of indices for this leg.

block_number: The number of blocks, i.e., a 'qindex' for this leg is in `range(block_number)`.

chinfo [*ChargeInfo* instance] The nature of the charge. Can be shared between LegCharges.

slices [ndarray[np.intp_t,ndim=1] (block_number+1)] A block with 'qindex' `qi` corresponds to the leg indices in `slice(self.slices[qi], self.slices[qi+1])`. See *get_slice()*.

charges [ndarray[QTYPE_t,ndim=1] (block_number, chinfo.qnumber)] `charges[qi]` gives the charges for a block with 'qindex' `qi`. Note: the sign might be changed by *qconj*. See also *get_charge()*.

qconj [{-1, 1}] A flag telling whether the charge points inwards (+1) or outwards (-1). Whenever charges are added, they should be multiplied with their *qconj* value.

sorted [bool] Whether the charges are guaranteed to be sorted.

bunched [bool] Whether the charges are guaranteed to be bunched.

Methods

bunch(self)	Return a copy with bunched self.charges: form blocks for contiguous equal charges.
charge_sectors(self)	Return unique rows of self.charges.
conj(self)	Return a (shallow) copy with opposite `self.qconj`.
copy(self)	Return a (shallow) copy of self.
extend(self, extra)	Return a new *LegCharge*, which extends self with futher charges.
flip_charges_qconj(self)	Return a copy with both negative *qconj* and *charges*.
from_add_charge(legs[, chargeinfo])	Add the (independent) charges of two or more legs to get larger *qnumber*.
from_change_charge(leg, charge, new_qmod[, ...])	Remove a charge from a LegCharge.
from_drop_charge(leg[, charge, chargeinfo])	Remove a charge from a LegCharge.
from_qdict(chargeinfo, qdict[, qconj])	Create a LegCharge from qdict form.
from_qflat(chargeinfo, qflat[, qconj])	Create a LegCharge from qflat form.
from_qind(chargeinfo, slices, charges[, qconj])	Just a wrapper around self.__init__(), see class docstring for parameters.
from_trivial(ind_len[, chargeinfo, qconj])	Create trivial (qnumber=0) LegCharge for given len of indices *ind_len*.
get_charge(self, qindex)	Return charge `self.charges[qindex] * self.qconj` for a given *qindex*.
get_qindex(self, flat_index)	Find qindex containing a flat index.

Continued on next page

Table 58 – continued from previous page

get_qindex_of_charges(self, charges)	Return the slice selecting the block for given charge values.
get_slice(self, qindex)	Return slice selecting the block for a given *qindex*.
is_blocked(self)	Returns whether self is blocked, i.e.
is_bunched(self)	Checks whether *bunch()* would change something.
is_sorted(self)	Returns whether *self.charges* is sorted lexiographically.
perm_flat_from_perm_qind(self, perm_qind)	Convert a permutation of qind (acting on self) into a flat permutation.
perm_qind_from_perm_flat(self, perm_flat)	Convert flat permutation into qind permutation.
project(self, mask)	Return copy keeping only the indices specified by *mask*.
sort(self[, bunch])	Return a copy of *self* sorted by charges (but maybe not bunched).
test_contractible(self, other)	Raises a ValueError if charges are incompatible for contraction with other.
test_equal(self, other)	Test if charges are *equal* including *qconj*.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
to_qdict(self)	Return charges in *qdict* form.
to_qflat(self)	Return charges in *qflat* form.

copy(*self*)
Return a (shallow) copy of self.

classmethod from_trivial(*ind_len*, *chargeinfo=None*, *qconj=1*)
Create trivial (qnumber=0) LegCharge for given len of indices *ind_len*.

classmethod from_qflat(*chargeinfo*, *qflat*, *qconj=1*)
Create a LegCharge from qflat form.

Does *neither* bunch *nor* sort. We recommend to sort (and bunch) afterwards, if you expect that tensors using the LegCharge have entries at all positions compatible with the charges.

Parameters

chargeinfo [*ChargeInfo*] The nature of the charge.

qflat [array_like (ind_len, *qnumber*)] *qnumber* charges for each index of the leg on entry.

qconj [{-1, 1}] A flag telling whether the charge points inwards (+1) or outwards (-1).

See also:

sort sorts by charges

bunch bunches contiguous blocks of the same charge.

classmethod from_qind(*chargeinfo*, *slices*, *charges*, *qconj=1*)
Just a wrapper around self.__init__(), see class doc-string for parameters.

See also:

sort sorts by charges

bunch bunches contiguous blocks of the same charge.

classmethod from_qdict(*chargeinfo*, *qdict*, *qconj=1*)
Create a LegCharge from qdict form.

Parameters

chargeinfo [*ChargeInfo*] The nature of the charge.

qdict [dict] A dictionary mapping a tuple of charges to slices.

classmethod from_add_charge(*legs*, *chargeinfo=None*)
Add the (independent) charges of two or more legs to get larger *qnumber*.

Parameters

legs [iterable of *LegCharge*] The legs for which the charges are to be combined/added.

chargeinfo [*ChargeInfo*] The ChargeInfo for all charges; create new if `None`.

Returns

combined [*LegCharge*] A LegCharge with the charges of both legs. Is neither sorted nor bunched!

classmethod from_drop_charge(*leg*, *charge=None*, *chargeinfo=None*)
Remove a charge from a LegCharge.

Parameters

leg [*LegCharge*] The leg from which to drop/remove a charge.

charge [int | str] Number or *name* of the charge (within *chinfo*) which is to be dropped. `None` means dropping all charges.

chargeinfo [*ChargeInfo*] The ChargeInfo with *charge* dropped; create new if `None`.

Returns

dropped [*LegCharge*] A LegCharge with the specified charge dropped. Is neither sorted nor bunched!

classmethod from_change_charge(*leg*, *charge*, *new_qmod*, *new_name=''*, *chargeinfo=None*)
Remove a charge from a LegCharge.

Parameters

leg [*LegCharge*] The leg from which to drop/remove a charge.

charge [int | str] Number or *name* of the charge (within *chinfo*) for which *mod* is to be changed.

new_qmod [int] The new *mod* to be set for *charge* in the *ChargeInfo*.

new_name [str] The new name for *charge*.

chargeinfo [*ChargeInfo*] The ChargeInfo with *charge* changed; create new if `None`.

Returns

leg [*LegCharge*] A LegCharge with the specified charge changed. Is neither sorted nor bunched!

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

conj(*self*)
Return a (shallow) copy with opposite `self.qconj`.

Returns

conjugated [*LegCharge*] Shallow copy of *self* with flipped `qconj`. *test_contractible()* of *self* with *conjugated* will not raise an error.

flip_charges_qconj(*self*)
Return a copy with both negative *qconj* and *charges*.

Returns

conj_charges [*LegCharge*] (Shallow) copy of self with negative *qconj* and *charges*, thus representing the very same charges. *test_equal()* of *self* with *conj_charges* will not raise an error.

to_qflat(*self*)
Return charges in *qflat* form.

to_qdict(*self*)
Return charges in *qdict* form.

Raises ValueError, if not blocked.

is_blocked(*self*)
Returns whether self is blocked, i.e. qindex map 1:1 to charge values.

is_sorted(*self*)
Returns whether *self.charges* is sorted lexiographically.

is_bunched(*self*)
Checks whether *bunch()* would change something.

test_contractible(*self*, *other*)
Raises a ValueError if charges are incompatible for contraction with other.

Parameters

other [*LegCharge*] The LegCharge of the other leg condsidered for contraction.

Raises

ValueError If the charges are incompatible for direct contraction.

See also:

test_equal `self.test_contractible(other)` just performs `self.test_equal(other.conj())`.

Notes

This function checks that two legs are *ready* for contraction. This is the case, if all of the following conditions are met:

- the `ChargeInfo` is equal
- the *slices* are equal
- the *charges* are the same up to *opposite* signs `qconj`:

```
self.charges * self.qconj = - other.charges * other.qconj
```

In general, there could also be a change of the total charge, see *Introduction to np_conserved* This special case is not considered here - instead use `gauge_total_charge()`, if a change of the charge is desired.

If you are sure that the legs should be contractable, check whether the charges are actually valid or whether `self` and `other` are blocked or should be sorted.

test_equal(*self*, *other*)
Test if charges are *equal* including *qconj*.

Check that all of the following conditions are met:

- the `ChargeInfo` is equal
- the *slices* are equal
- the *charges* are the same up to the signs `qconj`:

```
self.charges * self.qconj = other.charges * other.qconj
```

See also:

test_contractible `self.test_equal(other)` is equivalent to `self.test_contractible(other.conj())`.

get_slice(*self*, *qindex*)
Return slice selecting the block for a given *qindex*.

get_qindex(*self*, *flat_index*)
Find qindex containing a flat index.

Given a flat index, to find the corresponding entry in an Array, we need to determine the block it is saved in. For example, if `slices = [[0, 3], [3, 7], [7, 12]]`, the flat index `5` corresponds to the second entry, `qindex = 1` (since 5 is in [3:7]), and the index within the block would be `2 = 5 - 3`.

Parameters

flat_index [int] A flat index of the leg. Negative index counts from behind.

Returns

qindex [int] The qindex, i.e. the index of the block containing *flat_index*.

index_within_block [int] The index of *flat_index* within the block given by *qindex*.

get_qindex_of_charges(*self*, *charges*)
Return the slice selecting the block for given charge values.

Inverse function of *get_charge()*.

Parameters

charges [1D array_like] Charge values for which the slice of the block is to be determined.

Returns

slice(i, j) [slice] Slice of the charge values for

Raises

ValueError [if the answer is not unique (because *self* is not blocked).]

get_charge(*self*, *qindex*)
Return charge `self.charges[qindex] * self.qconj` for a given *qindex*.

sort(*self*, *bunch=True*)
Return a copy of *self* sorted by charges (but maybe not bunched).

If bunch=True, the returned copy is completely blocked by charge.

Parameters

bunch [bool] Whether *self.bunch* is called after sorting. If True, the leg is guaranteed to be fully blocked by charge.

Returns

perm_qind [array (self.block_len,)] The permutation of the qindices (before bunching) used for the sorting. To obtain the flat permuation such that `sorted_array[..., :] = unsorted_array[..., perm_flat]`, use `perm_flat = unsorted_leg.perm_flat_from_perm_qind(perm_qind)`

sorted_copy [*LegCharge*] A shallow copy of self, with new qind sorted (and thus blocked if bunch) by charges.

See also:

bunch enlarge blocks for contiguous qind of the same charges.

numpy.take can apply *perm_flat* to a given axis

tenpy.tools.misc.inverse_permutation returns inverse of a permutation

bunch(*self*)

Return a copy with bunched self.charges: form blocks for contiguous equal charges.

Returns

idx [1D array] `idx[:-1]` are the indices of the old qind which are kept, `idx[-1] = old_block_number`.

cp [*LegCharge*] A new LegCharge with the same charges at given indices of the leg, but (possibly) shorter `self.charges` and `self.slices`.

See also:

sort sorts by charges, thus enforcing complete blocking in combination with bunch.

project(*self*, *mask*)

Return copy keeping only the indices specified by *mask*.

Parameters

mask [1D array(bool)] Whether to keep of the indices.

Returns

map_qind [1D array] Map of qindices, such that `qind_new = map_qind[qind_old]`, and `map_qind[qind_old] = -1` for qindices projected out.

block_masks [1D array] The bool mask for each of the *remaining* blocks.

projected_copy [*LegCharge*] Copy of self with the qind projected by *mask*.

extend(*self*, *extra*)

Return a new *LegCharge*, which extends self with futher charges.

This is needed to formally increase the dimension of an Array.

Parameters

extra [*LegCharge* | int] By what to extend, i.e. the charges to be appended to *self*. An int stands for extending the length of the array by a single new block of that size and zero charges.

Returns

extended_leg [*LegCharge*] Copy of *self* extended by the charge blocks of the *extra* leg.

charge_sectors(*self*)
Return unique rows of self.charges.

Returns

charges [array[QTYPE, ndim=2]] Rows are the rows of self.charges lexsorted and without duplicates.

perm_flat_from_perm_qind(*self*, *perm_qind*)
Convert a permutation of qind (acting on self) into a flat permutation.

perm_qind_from_perm_flat(*self*, *perm_flat*)
Convert flat permutation into qind permutation.

Parameters

perm_flat [1D array] A permutation acting on self, which doesn't mix the blocks of qind.

Returns

perm_qind [1D array] The permutation of self.qind described by perm_flat.

Raises

ValueError If perm_flat mixes blocks of different qindex.

LegPipe

- full name: tenpy.linalg.charges.LegPipe
- parent module: *tenpy.linalg.charges*
- type: class

class tenpy.linalg.charges.**LegPipe**(*legs*, *qconj=1*, *sort=True*, *bunch=True*)
Bases: *tenpy.linalg.charges.LegCharge*

A *LegPipe* combines multiple legs of a tensor to one.

Often, it is necessary to "combine" multiple legs into one: for example to perfom a SVD, the tensor needs to be viewed as a matrix.

This class does exactly this job: it combines multiple LegCharges ('incoming legs') into one 'pipe' (*the* 'outgoing leg'). The pipe itself is a *LegCharge*, with indices running from 0 to the product of the individual legs' *ind_len*, corresponding to all possible combinations of input leg indices.

(This class is implemented in *tenpy.linalg.charges* but also imported in *tenpy.linalg.np_conserved* for convenience.)

Parameters

legs [list of *LegCharge*] The legs which are to be combined.

qconj [{+1, -1}] A flag telling whether the charge of the *resulting* pipe points inwards (+1, default) or outwards (-1).

sort [bool] Whether the outgoing pipe should be sorted. Default `True`; recommended. Note: calling *sort()* after initialization converts to a LegCharge.

bunch [bool] Whether the outgoing pipe should be bunched. Default `True`; recommended. Note: calling *bunch()* after initialization converts to a LegCharge.

Notes

For np.reshape, taking, for example, $i, j, ... \to k$ amounted to $k = s_1 * i + s_2 * j + ...$ for appropriate strides s_1, s_2.

In the charged case, however, we want to block k by charge, so we must implicitly permute as well. This reordering is encoded in *q_map*.

Each qindex combination of the *nlegs* input legs $(i_1, ..., i_{nlegs})$, will end up getting placed in some slice $a_j : a_{j+1}$ of the outgoing pipe. Within this slice, the data is simply reshaped in usual row-major fashion ('C'-order), i.e., with strides $s_1 > s_2 > ....$

It will be a subslice of a new total block labeled by qindex I_s. Because many charge combinations fuse to the same total charge, in general there will be many tuples $(i_1, ..., i_{nlegs})$ belonging to the same I_s. The rows of *q_map* are precisely the collections of `[b_j, b_{j+1}, I_s, i_1, . . . , i_{nlegs}]`. Here, $b_j : b_{j+1}$ denotes the slice of this qindex combination *within* the total block *I_s*, i.e., `b_j = a_j - self.slices[I_s]`.

The rows of *q_map* are lex-sorted first by `I_s`, then the `i`. Each `I_s` will have multiple rows, and the order in which they are stored in *q_map* is the order the data is stored in the actual tensor, i.e., it might look like

```
[ ...,
 [ b_j,     b_{j+1},  I_s,     i_1,     ..., i_{nlegs}   ],
 [ b_{j+1}, b_{j+2},  I_s,     i'_1,    ..., i'_{nlegs}  ],
 [ 0,       b_{j+3},  I_s + 1, i''_1,   ..., i''_{nlegs} ],
 [ b_{j+3}, b_{j+4},  I_s + 1, i'''_1, ..., i'''_{nlegs}],
 ...]
```

The charge fusion rule is:

```
self.charges[Qi]*self.qconj == sum([l.charges[qi_l]*l.qconj for l in self.legs]) ␣
↪mod qmod
```

Here the qindex `Qi` of the pipe corresponds to qindices `qi_l` on the individual legs.

Attributes

nlegs [int] The number of legs.

legs [tuple of *LegCharge*] The original legs, which were combined in the pipe.

subshape [tuple of int] *ind_len* for each of the incoming legs.

subqshape [tuple of int] *block_number* for each of the incoming legs.

q_map: array[np.intp, ndim=2] Shape (*block_number*, 3 + *nlegs*). Rows: `[ b_j, b_{j+1}, I_s, i_1, ..., i_{nlegs}]`, See Notes below for details.

q_map_slices [array[np.intp, ndim=1]] Defined such that the row indices of in `range(q_map_slices[I_s], q_map_slices[I_s+1])` have `q_map[:, 2] == I_s`.

_perm [1D array] A permutation such that `q_map[_perm, 3:]` is sorted by *i_l*.

_strides [1D array] Strides for mapping incoming qindices *i_l* to the index of `q_map[_perm, :]`.

Methods

bunch(self, *args, **kwargs)	Convert to LegCharge and call *LegCharge.bunch()*.
charge_sectors(self)	Return unique rows of self.charges.
conj(self)	Return a shallow copy with opposite `self.qconj`.
copy(self)	Return a (shallow) copy of self.
extend(self, extra)	Return a new *LegCharge*, which extends self with futher charges.
flip_charges_qconj(self)	Return a copy with both negative *qconj* and *charges*.
from_add_charge(legs[, chargeinfo])	Add the (independent) charges of two or more legs to get larger *qnumber*.
from_change_charge(leg, charge, new_qmod[, ...])	Remove a charge from a LegCharge.
from_drop_charge(leg[, charge, chargeinfo])	Remove a charge from a LegCharge.
from_qdict(chargeinfo, qdict[, qconj])	Create a LegCharge from qdict form.
from_qflat(chargeinfo, qflat[, qconj])	Create a LegCharge from qflat form.
from_qind(chargeinfo, slices, charges[, qconj])	Just a wrapper around self.__init__(), see class docstring for parameters.
from_trivial(ind_len[, chargeinfo, qconj])	Create trivial (qnumber=0) LegCharge for given len of indices *ind_len*.
get_charge(self, qindex)	Return charge `self.charges[qindex] * self.qconj` for a given *qindex*.
get_qindex(self, flat_index)	Find qindex containing a flat index.
get_qindex_of_charges(self, charges)	Return the slice selecting the block for given charge values.
get_slice(self, qindex)	Return slice selecting the block for a given *qindex*.
is_blocked(self)	Returns whether self is blocked, i.e.
is_bunched(self)	Checks whether *bunch()* would change something.
is_sorted(self)	Returns whether *self.charges* is sorted lexiographically.
map_incoming_flat(self, incoming_indices)	Map (flat) incoming indices to an index in the outgoing pipe.
outer_conj(self)	Like *conj()*, but don't change `qconj` for incoming legs.
perm_flat_from_perm_qind(self, perm_qind)	Convert a permutation of qind (acting on self) into a flat permutation.
perm_qind_from_perm_flat(self, perm_flat)	Convert flat permutation into qind permutation.
project(self, *args, **kwargs)	Convert self to LegCharge and call *LegCharge.project()*.
sort(self, *args, **kwargs)	Convert to LegCharge and call *LegCharge.sort()*.
test_contractible(self, other)	Raises a ValueError if charges are incompatible for contraction with other.
test_equal(self, other)	Test if charges are *equal* including *qconj*.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
to_LegCharge(self)	Convert self to a LegCharge, discarding the information how to split the legs.
to_qdict(self)	Return charges in *qdict* form.

Continued on next page

Table 59 – continued from previous page

to_qflat(self)	Return charges in *qflat* form.

copy (*self*)
Return a (shallow) copy of self.

test_sanity (*self*)
Sanity check, raises ValueErrors, if something is wrong.

to_LegCharge (*self*)
Convert self to a LegCharge, discarding the information how to split the legs.

Usually not needed, but called by functions, which are not implemented for a LegPipe.

conj (*self*)
Return a shallow copy with opposite `self.qconj`.

Returns

conjugated [*LegCharge*] Shallow copy of *self* with flipped `qconj`. Whenever we contract two legs, they need to be conjugated to each other. The incoming legs of the pipe are also conjugated.

outer_conj (*self*)
Like *conj()*, but don't change `qconj` for incoming legs.

sort (*self*, **args*, ***kwargs*)
Convert to LegCharge and call *LegCharge.sort()*.

bunch (*self*, **args*, ***kwargs*)
Convert to LegCharge and call *LegCharge.bunch()*.

project (*self*, **args*, ***kwargs*)
Convert self to LegCharge and call *LegCharge.project()*.

In general, this could be implemented for a LegPipe, but would make *split_legs()* more complicated, thus we keep it simple. If you really want to project and split afterwards, use the following work-around, which is for example used in `exact_diagonalization`:

1) Create the full pipe and save it separetely.
2) Convert the Pipe to a Leg & project the array with it.
3) [... do calculations ...]
4) To split the 'projected pipe' of A, create and empty array B with the legs of A, but replace the projected leg by the full pipe. Set A as a slice of B. Finally split the pipe.

map_incoming_flat (*self*, *incoming_indices*)
Map (flat) incoming indices to an index in the outgoing pipe.

Parameters

incoming_indices [iterable of int] One (flat) index on each of the incoming legs.

Returns

outgoing_index [int] The index in the outgoing leg.

charge_sectors (*self*)
Return unique rows of self.charges.

Returns

charges [array[QTYPE, ndim=2]] Rows are the rows of self.charges lexsorted and without duplicates.

extend(*self*, *extra*)
Return a new *LegCharge*, which extends self with futher charges.

This is needed to formally increase the dimension of an Array.

Parameters

extra [*LegCharge* | int] By what to extend, i.e. the charges to be appended to *self*. An int stands for extending the length of the array by a single new block of that size and zero charges.

Returns

extended_leg [*LegCharge*] Copy of *self* extended by the charge blocks of the *extra* leg.

flip_charges_qconj(*self*)
Return a copy with both negative *qconj* and *charges*.

Returns

conj_charges [*LegCharge*] (Shallow) copy of self with negative *qconj* and *charges*, thus representing the very same charges. *test_equal()* of *self* with *conj_charges* will not raise an error.

classmethod from_add_charge(*legs*, *chargeinfo=None*)
Add the (independent) charges of two or more legs to get larger *qnumber*.

Parameters

legs [iterable of *LegCharge*] The legs for which the charges are to be combined/added.

chargeinfo [*ChargeInfo*] The ChargeInfo for all charges; create new if `None`.

Returns

combined [*LegCharge*] A LegCharge with the charges of both legs. Is neither sorted nor bunched!

classmethod from_change_charge(*leg*, *charge*, *new_qmod*, *new_name=''*, *chargeinfo=None*)
Remove a charge from a LegCharge.

Parameters

leg [*LegCharge*] The leg from which to drop/remove a charge.

charge [int | str] Number or *name* of the charge (within *chinfo*) for which *mod* is to be changed.

new_qmod [int] The new *mod* to be set for *charge* in the *ChargeInfo*.

new_name [str] The new name for *charge*.

chargeinfo [*ChargeInfo*] The ChargeInfo with *charge* changed; create new if `None`.

Returns

leg [*LegCharge*] A LegCharge with the specified charge changed. Is neither sorted nor bunched!

classmethod from_drop_charge(*leg*, *charge=None*, *chargeinfo=None*)
Remove a charge from a LegCharge.

Parameters

leg [*LegCharge*] The leg from which to drop/remove a charge.

charge [int | str] Number or *name* of the charge (within *chinfo*) which is to be dropped. `None` means dropping all charges.

chargeinfo [*ChargeInfo*] The ChargeInfo with *charge* dropped; create new if `None`.

Returns

dropped [*LegCharge*] A LegCharge with the specified charge dropped. Is neither sorted nor bunched!

classmethod from_qdict(*chargeinfo*, *qdict*, *qconj=1*)
Create a LegCharge from qdict form.

Parameters

chargeinfo [*ChargeInfo*] The nature of the charge.

qdict [dict] A dictionary mapping a tuple of charges to slices.

classmethod from_qflat(*chargeinfo*, *qflat*, *qconj=1*)
Create a LegCharge from qflat form.

Does *neither* bunch *nor* sort. We recommend to sort (and bunch) afterwards, if you expect that tensors using the LegCharge have entries at all positions compatible with the charges.

Parameters

chargeinfo [*ChargeInfo*] The nature of the charge.

qflat [array_like (ind_len, *qnumber*)] *qnumber* charges for each index of the leg on entry.

qconj [{-1, 1}] A flag telling whether the charge points inwards (+1) or outwards (-1).

See also:

sort sorts by charges

bunch bunches contiguous blocks of the same charge.

classmethod from_qind(*chargeinfo*, *slices*, *charges*, *qconj=1*)
Just a wrapper around self.__init__(), see class doc-string for parameters.

See also:

sort sorts by charges

bunch bunches contiguous blocks of the same charge.

classmethod from_trivial(*ind_len*, *chargeinfo=None*, *qconj=1*)
Create trivial (qnumber=0) LegCharge for given len of indices *ind_len*.

get_charge(*self*, *qindex*)
Return charge `self.charges[qindex] * self.qconj` for a given *qindex*.

get_qindex(*self*, *flat_index*)
Find qindex containing a flat index.

Given a flat index, to find the corresponding entry in an Array, we need to determine the block it is saved in. For example, if `slices = [[0, 3], [3, 7], [7, 12]]`, the flat index `5` corresponds to the second entry, `qindex = 1` (since 5 is in [3:7]), and the index within the block would be `2 = 5 - 3`.

Parameters

flat_index [int] A flat index of the leg. Negative index counts from behind.

Returns

qindex [int] The qindex, i.e. the index of the block containing *flat_index*.

index_within_block [int] The index of *flat_index* within the block given by *qindex*.

get_qindex_of_charges(*self*, *charges*)
Return the slice selecting the block for given charge values.

Inverse function of *get_charge()*.

Parameters

charges [1D array_like] Charge values for which the slice of the block is to be determined.

Returns

slice(i, j) [slice] Slice of the charge values for

Raises

ValueError [if the answer is not unique (because *self* is not blocked).]

get_slice(*self*, *qindex*)
Return slice selecting the block for a given *qindex*.

is_blocked(*self*)
Returns whether self is blocked, i.e. qindex map 1:1 to charge values.

is_bunched(*self*)
Checks whether *bunch()* would change something.

is_sorted(*self*)
Returns whether *self.charges* is sorted lexiographically.

perm_flat_from_perm_qind(*self*, *perm_qind*)
Convert a permutation of qind (acting on self) into a flat permutation.

perm_qind_from_perm_flat(*self*, *perm_flat*)
Convert flat permutation into qind permutation.

Parameters

perm_flat [1D array] A permutation acting on self, which doesn't mix the blocks of qind.

Returns

perm_qind [1D array] The permutation of self.qind described by perm_flat.

Raises

ValueError If perm_flat mixes blocks of different qindex.

test_contractible(*self*, *other*)
Raises a ValueError if charges are incompatible for contraction with other.

Parameters

other [*LegCharge*] The LegCharge of the other leg condsidered for contraction.

Raises

ValueError If the charges are incompatible for direct contraction.

See also:

test_equal `self.test_contractible(other)` just performs `self.test_equal(other.conj())`.

Notes

This function checks that two legs are *ready* for contraction. This is the case, if all of the following conditions are met:

- the `ChargeInfo` is equal
- the *slices* are equal
- the *charges* are the same up to *opposite* signs `qconj`:

```
self.charges * self.qconj = - other.charges * other.qconj
```

In general, there could also be a change of the total charge, see *Introduction to np_conserved* This special case is not considered here - instead use `gauge_total_charge()`, if a change of the charge is desired.

If you are sure that the legs should be contractable, check whether the charges are actually valid or whether `self` and `other` are blocked or should be sorted.

test_equal (*self*, *other*)
Test if charges are *equal* including *qconj*.

Check that all of the following conditions are met:

- the `ChargeInfo` is equal
- the *slices* are equal
- the *charges* are the same up to the signs `qconj`:

```
self.charges * self.qconj = other.charges * other.qconj
```

See also:

test_contractible `self.test_equal(other)` is equivalent to `self.test_contractible(other.conj())`.

to_qdict (*self*)
Return charges in *qdict* form.

Raises ValueError, if not blocked.

to_qflat (*self*)
Return charges in *qflat* form.

Module description

Basic definitions of a charge.

This module contains implementations for handling the quantum numbers ("charges") of the *Array*.

In particular, the classes *ChargeInfo*, *LegCharge* and *LegPipe* are implemented here.

Note: The contents of this module are imported in *np_conserved*, so you usually don't need to import this module in your application.

A detailed introduction to *np_conserved* can be found in *Introduction to np_conserved*.

In this module, some functions have the python decorator `@use_cython`. Functions with this decorator are replaced by the ones written in Cython, implemented in the file `tenpy/linalg/_npc_helper.pyx`. For further details, see the definition of *use_cython()*.

svd_robust

- full name: tenpy.linalg.svd_robust
- parent module: *tenpy.linalg*
- type: module

Functions

svd(a[, full_matrices, compute_uv, ...])	Wrapper around `scipy.linalg.svd()` with *gesvd* backup plan.
svd_gesvd(a[, full_matrices, compute_uv, ...])	svd with LAPACK's '#gesvd' (with # = d/z for float/complex).

svd

- full name: tenpy.linalg.svd_robust.svd
- parent module: *tenpy.linalg.svd_robust*
- type: function

`tenpy.linalg.svd_robust.`**`svd`**(*a*, *full_matrices=True*, *compute_uv=True*, *overwrite_a=False*, *check_finite=True*, *lapack_driver='gesdd'*, *warn=True*)

Wrapper around `scipy.linalg.svd()` with *gesvd* backup plan.

Tries to avoid raising an LinAlgError by using using the lapack_driver *gesvd*, if *gesdd* failed.

Parameters

overwrite_a [bool] Ignored (i.e. set to `False`) if `lapack_driver='gesdd'`. Otherwise described in `scipy.linalg.svd()`.

lapack_driver [{'gesdd', 'gesvd'}, optional] Whether to use the more efficient divide-and-conquer approach (`'gesdd'`) or general rectangular approach (`'gesvd'`) to compute the SVD. MATLAB and Octave use the `'gesvd'` approach. Default is `'gesdd'`. If `'gesdd'` fails, `'gesvd'` is used as backup.

warn [bool] whether to create a warning when the SVD failed.

Other parameters as described in doc-string of :func:`scipy.linalg.svd`

Returns

U, S, Vh [ndarray] As described in doc-string of `scipy.linalg.svd()`

svd_gesvd

- full name: tenpy.linalg.svd_robust.svd_gesvd
- parent module: *tenpy.linalg.svd_robust*
- type: function

`tenpy.linalg.svd_robust.`**`svd_gesvd`**(*a*, *full_matrices=True*, *compute_uv=True*, *check_finite=True*)

svd with LAPACK's '#gesvd' (with # = d/z for float/complex).

Similar as `numpy.linalg.svd()`, but use LAPACK 'gesvd' driver. Works only with 2D arrays. Outer part is based on the code of *numpy.linalg.svd*.

Parameters

a, full_matrices, compute_uv : See `numpy.linalg.svd()` for details.

check_finite : check whether input arrays contain 'NaN' or 'inf'.

Returns

U, S, Vh [ndarray] See `numpy.linalg.svd()` for details.

Module description

(More) robust version of singular value decomposition.

We often need to perform an SVD. In general, an SVD is a matrix factorization that is always well defined and should also work for ill-conditioned matrices. But sadly, both `numpy.linalg.svd()` and `scipy.linalg.svd()` fail from time to time, raising `LinalgError("SVD did not converge")`. The reason is that both of them call the LAPACK function *#gesdd* (where # depends on the data type), which takes an iterative approach that can fail. However, it is usually much faster than the alternative (and robust) *#gesvd*.

Our workaround is as follows: we provide a function *svd()* with call signature as scipy's svd. This function is basically just a wrapper around scipy's svd, i.e., we keep calling the faster *dgesdd*. But if that fails, we can still use *dgesvd* as a backup.

Sadly, *dgesvd* and *zgesvd* were not included into scipy until version '0.18.0' (nor in numpy), which is as the time of this writing the latest stable scipy version. For scipy version newer than '0.18.0', we make use of the new keyword 'lapack_driver' for svd, otherwise we (try to) load *dgesvd* and *zgesvd* from shared LAPACK libraries.

The tribute for the dgesvd wrapper code goes to 'jgarcke', originally posted at http://projects.scipy.org/numpy/ticket/990, which is now hosted at https://github.com/numpy/numpy/issues/1588 He explains a bit more in detail what fails.

The include of *dgesvd* to scipy was done in https://github.com/scipy/scipy/pull/5994.

Examples

The idea is that you just import the *svd* from this module and use it as replacement for `np.linalg.svd` or `scipy.linalg.svd`:

```
>>> from svd_robust import svd
>>> U, S, VT = svd([[1., 1.], [0., [1.]])
```

random_matrix

- full name: tenpy.linalg.random_matrix
- parent module: *tenpy.linalg*
- type: module

Functions

COE(size)	Circular orthogonal ensemble (COE).
CRE(size)	Circular real ensemble (CRE).
CUE(size)	Circular unitary ensemble (CUE).
GOE(size)	Gaussian orthogonal ensemble (GOE).
GUE(size)	Gaussian unitary ensemble (GUE).
O_close_1(size[, a])	return an random orthogonal matrix 'close' to the Identity.
U_close_1(size[, a])	return an random orthogonal matrix 'close' to the identity.
box(size[, W])	return random number uniform in (-W, W].
standard_normal_complex(size)	return `(R + 1.j*I)` for independent *R* and *I* from np.random.standard_normal.

COE

- full name: tenpy.linalg.random_matrix.COE
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**COE**(*size*)
Circular orthogonal ensemble (COE).

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

Returns

U [ndarray] Unitary, symmetric (complex) matrix drawn from the COE (=Haar measure on this space).

CRE

- full name: tenpy.linalg.random_matrix.CRE
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**CRE**(*size*)
Circular real ensemble (CRE).

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

Returns

U [ndarray] Orthogonal matrix drawn from the CRE (=Haar measure on O(n)).

CUE

- full name: tenpy.linalg.random_matrix.CUE
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**CUE**(*size*)
Circular unitary ensemble (CUE).

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

Returns

U [ndarray] Unitary matrix drawn from the CUE (=Haar measure on U(n)).

GOE

- full name: tenpy.linalg.random_matrix.GOE
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**GOE**(*size*)
Gaussian orthogonal ensemble (GOE).

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

Returns

H [ndarray] Real, symmetric numpy matrix drawn from the GOE, i.e. $p(H) = 1/Zexp(-n/4tr(H^2))$

GUE

- full name: tenpy.linalg.random_matrix.GUE
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**GUE**(*size*)
Gaussian unitary ensemble (GUE).

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

Returns

H [ndarray] Hermitian (complex) numpy matrix drawn from the GUE, i.e. $p(H) = 1/Zexp(-n/4tr(H^2))$.

O_close_1

- full name: tenpy.linalg.random_matrix.O_close_1
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**O_close_1**(*size*, *a=0.01*)
return an random orthogonal matrix 'close' to the Identity.

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

a [float] Parameter determining how close the result is on O; $\lim_{a\to 0} < |O-E| >_a = 0$ (where E is the identity).

Returns

O [ndarray] Orthogonal matrix close to the identiy (for small *a*).

U_close_1

- full name: tenpy.linalg.random_matrix.U_close_1
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**U_close_1**(*size*, *a=0.01*)
return an random orthogonal matrix 'close' to the identity.

Parameters

size [tuple] `(n, n)`, where *n* is the dimension of the output matrix.

a [float] Parameter determining how close the result is to the identity. $\lim_{a\to 0} < |O-E| >_a = 0$ (where E is the identity).

Returns

U [ndarray] Unitary matrix close to the identiy (for small *a*). Eigenvalues are chosen i.i.d. as `exp(1.j*a*x)` with *x* uniform in [-1, 1].

box

- full name: tenpy.linalg.random_matrix.box
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**box**(*size*, *W=1.0*)
return random number uniform in (-W, W].

standard_normal_complex

- full name: tenpy.linalg.random_matrix.standard_normal_complex
- parent module: *tenpy.linalg.random_matrix*
- type: function

tenpy.linalg.random_matrix.**standard_normal_complex**(*size*)
return `(R + 1.j*I)` for independent *R* and *I* from np.random.standard_normal.

Module description

Provide some random matrix ensembles for numpy.

The implemented ensembles are:

ensemble	matrix class drawn from	measure	invariant under	beta
GOE	real, symmetric	`~ exp(-n/4 tr(H^2))`	orthogonal O	1
GUE	hermitian	`~ exp(-n/2 tr(H^2))`	unitary U	2
CRE	O(n)	Haar	orthogonal O	/
COE	U in U(n) with U = U^T	Haar	orthogonal O	1
CUE	U(n)	Haar	unitary U	2
O_close_1	O(n)	?	/	/
U_close_1	U(n)	?	/	/

All functions in this module take a tuple `(n, n)` as first argument, such that we can use the function *from_func()* to generate a block diagonal *Array* with the block from the corresponding ensemble, for example:

```
npc.Array.from_func_square(GOE, [leg, leg.conj()])
```

sparse

- full name: tenpy.linalg.sparse
- parent module: *tenpy.linalg*
- type: module

Classes

FlatHermitianOperator(npc_matvec, leg, dtype)	Hermitian variant of *FlatLinearOperator*.
FlatLinearOperator(npc_matvec, leg, dtype[, ...])	Square Linear operator acting on numpy arrays based on a *matvec* acting on npc Arrays.
NpcLinearOperator	Prototype for a Linear Operator acting on *Array*.

FlatHermitianOperator

- full name: tenpy.linalg.sparse.FlatHermitianOperator
- parent module: *tenpy.linalg.sparse*
- type: class

class tenpy.linalg.sparse.**FlatHermitianOperator**(*npc_matvec*, *leg*, *dtype*, *charge_sector=0*, *vec_label=None*)

Bases: *tenpy.linalg.sparse.FlatLinearOperator*

Hermitian variant of *FlatLinearOperator*.

Note that we don't check *matvec()* to return a hermitian result, we only define an adjoint to be *self*.

Attributes

H Hermitian adjoint.

T Transpose this linear operator.

charge_sector Charge sector of the vector which is acted on.

Methods

__call__(self, x)	Call self as a function.
adjoint(self)	Hermitian adjoint.
dot(self, x)	Matrix-matrix or matrix-vector multiplication.
flat_to_npc(self, vec)	Convert flat vector of selected charge sector into npc Array.
from_NpcArray(mat[, charge_sector])	Create a *FlatLinearOperator* from a square *Array*.
from_guess_with_pipe(npc_matvec, v0_guess[, ...])	Create a *FlatLinearOperator*` from a *matvec* function acting on multiple legs.
matmat(self, X)	Matrix-matrix multiplication.
matvec(self, x)	Matrix-vector multiplication.
npc_to_flat(self, npc_vec)	Convert npc Array with qtotal = self.charge_sector into ndarray.
rmatmat(self, X)	Adjoint matrix-matrix multiplication.
rmatvec(self, x)	Adjoint matrix-vector multiplication.
transpose(self)	Transpose this linear operator.

property H

Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns

A_H [LinearOperator] Hermitian adjoint of self.

property T

Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint(*self*)
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns

A_H [LinearOperator] Hermitian adjoint of self.

property charge_sector
Charge sector of the vector which is acted on.

dot(*self*, *x*)
Matrix-matrix or matrix-vector multiplication.

Parameters

x [array_like] 1-d or 2-d array, representing a vector or matrix.

Returns

Ax [array] 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

flat_to_npc(*self*, *vec*)
Convert flat vector of selected charge sector into npc Array.

Parameters

vec [1D ndarray] Numpy vector to be converted. Should have the entries according to self.charge_sector.

Returns

npc_vec [*Array*] Same as *vec*, but converted into a flat array.

classmethod from_NpcArray(*mat*, *charge_sector=0*)
Create a *FlatLinearOperator* from a square *Array*.

Parameters

mat [*Array*] A square matrix, with contractable legs.

charge_sector [None | charges | 0] Selects the charge sector of the vector onto which the Linear operator acts. None stands for *all* sectors, 0 stands for the zero-charge sector. Defaults to 0, i.e., *assumes* the dominant eigenvector is in charge sector 0.

classmethod from_guess_with_pipe(*npc_matvec*, *v0_guess*, *labels_split=None*, *dtype=None*)
Create a *FlatLinearOperator*` from a *matvec* function acting on multiple legs.

This function creates a wrapper *matvec* function to allow acting on a "vector" with multiple legs. The wrapper combines the legs into a *LegPipe* before calling the actual *matvec* function, and splits them again in the end.

Parameters

npc_matvec [function] Function to calculate the action of the linear operator on an npc vector with the given split labels *labels_split*. Has to return an npc vector with the same legs.

v0_guess [*Array*] Initial guess/starting vector which can be applied to *npc_matvec*.

labels_split [None | list of str] Labels of v0_guess in the order in which they are to be combined into a *LegPipe*. `None` defaults to `v0_guess.get_leg_labels()`.

dtype [np.dtype | None] The data type of the arrays. `None` defaults to dtype of *v0_guess* (!).

Returns

lin_op [cls] Instance of the class to be used as linear operator

guess_flat [np.ndarray] Numpy vector representing the guess *v0_guess*.

matmat (*self*, *X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters

X [{matrix, ndarray}] An array with shape (N,K).

Returns

Y [{matrix, ndarray}] A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec (*self*, *x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters

x [{matrix, ndarray}] An array with shape (N,) or (N,1).

Returns

y [{matrix, ndarray}] A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

npc_to_flat (*self*, *npc_vec*)
Convert npc Array with qtotal = self.charge_sector into ndarray.

Parameters

npc_vec [*Array*] Npc Array to be converted. Should only have entries in *self.charge_sector*.

Returns

vec [1D ndarray] Same as *npc_vec*, but converted into a flat Numpy array.

rmatmat (*self*, *X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters

X [{matrix, ndarray}] A matrix or 2D array.

Returns

Y [{matrix, ndarray}] A matrix or 2D array depending on the type of the input.

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec (*self*, *x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters

x [{matrix, ndarray}] An array with shape (M,) or (M,1).

Returns

y [{matrix, ndarray}] A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

transpose (*self*)
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

FlatLinearOperator

- full name: tenpy.linalg.sparse.FlatLinearOperator
- parent module: *tenpy.linalg.sparse*
- type: class

class tenpy.linalg.sparse.**FlatLinearOperator**(*npc_matvec*, *leg*, *dtype*, *charge_sector=0*, *vec_label=None*)
Bases: `scipy.sparse.linalg.interface.LinearOperator`

Square Linear operator acting on numpy arrays based on a *matvec* acting on npc Arrays.

Note that this class represents a square linear operator. In terms of charges, this means it has legs `[self.leg.conj(), self.leg]` and trivial (zero) `qtotal`.

Parameters

npc_matvec [function] Function to calculate the action of the linear operator on an npc vector (with the specified *leg*). Has to return an npc vector with the same leg.

leg [*LegCharge*] Leg of the vector on which *npc_matvec* can act on.

dtype [np.dtype] The data type of the arrays.

charge_sector [None | charges | 0] Selects the charge sector of the vector onto which the Linear operator acts. None stands for *all* sectors, 0 stands for the zero-charge sector. Defaults to 0, i.e., *assumes* the dominant eigenvector is in charge sector 0.

vec_label [None | str] Label to be set to the npc vector before acting on it with *npc_matvec*. Ignored if *None*.

Attributes

charge_sector Charge sector of the vector which is acted on.

possible_charge_sectors [ndarray[QTYPE, ndim=2]] Each row corresponds to one possible choice for *charge_sector*.

shape [(int, int)] The dimensions for the selected charge sector.

dtype [np.dtype] The data type of the arrays.

leg [*LegCharge*] Leg of the vector on which *npc_matvec* can act on.

vec_label [None | str] Label to be set to the npc vector before acting on it with *npc_matvec*. Ignored if *None*.

npc_matvec [function] Function to calculate the action of the linear operator on an npc vector (with one *leg*).

matvec_count [int] The number of times *npc_matvec* was called.

_mask [ndarray[ndim=1, bool]] The indices of *leg* corresponding to the *charge_sector* to be diagonalized.

_npc_matvec_multileg [function | None] Only set if initalized with *from_guess_with_pipe()*. The *npc_matvec* function to be wrapped around. Takes the npc Array in multidimensional form and returns it that way.

_labels_split [list of str] Only set if initalized with *from_guess_with_pipe()*. Labels of the guess before combining them into a pipe (stored as *leg*).

Methods

__call__(self, x)	Call self as a function.
adjoint(self)	Hermitian adjoint.
dot(self, x)	Matrix-matrix or matrix-vector multiplication.
flat_to_npc(self, vec)	Convert flat vector of selected charge sector into npc Array.
from_NpcArray(mat[, charge_sector])	Create a *FlatLinearOperator* from a square *Array*.
from_guess_with_pipe(npc_matvec, v0_guess[, ...])	Create a *FlatLinearOperator*` from a *matvec* function acting on multiple legs.
matmat(self, X)	Matrix-matrix multiplication.
matvec(self, x)	Matrix-vector multiplication.

Continued on next page

Table 64 – continued from previous page

npc_to_flat(self, npc_vec)	Convert npc Array with qtotal = self.charge_sector into ndarray.
rmatmat(self, X)	Adjoint matrix-matrix multiplication.
rmatvec(self, x)	Adjoint matrix-vector multiplication.
transpose(self)	Transpose this linear operator.

classmethod from_NpcArray(*mat*, *charge_sector=0*)
Create a *FlatLinearOperator* from a square *Array*.

Parameters

mat [*Array*] A square matrix, with contractable legs.

charge_sector [None | charges | 0] Selects the charge sector of the vector onto which the Linear operator acts. None stands for *all* sectors, 0 stands for the zero-charge sector. Defaults to 0, i.e., *assumes* the dominant eigenvector is in charge sector 0.

classmethod from_guess_with_pipe(*npc_matvec*, *v0_guess*, *labels_split=None*, *dtype=None*)
Create a *FlatLinearOperator*` from a *matvec* function acting on multiple legs.

This function creates a wrapper *matvec* function to allow acting on a "vector" with multiple legs. The wrapper combines the legs into a *LegPipe* before calling the actual *matvec* function, and splits them again in the end.

Parameters

npc_matvec [function] Function to calculate the action of the linear operator on an npc vector with the given split labels *labels_split*. Has to return an npc vector with the same legs.

v0_guess [*Array*] Initial guess/starting vector which can be applied to *npc_matvec*.

labels_split [None | list of str] Labels of v0_guess in the order in which they are to be combined into a *LegPipe*. None defaults to v0_guess.get_leg_labels().

dtype [np.dtype | None] The data type of the arrays. None defaults to dtype of *v0_guess* (!).

Returns

lin_op [cls] Instance of the class to be used as linear operator

guess_flat [np.ndarray] Numpy vector representing the guess *v0_guess*.

property charge_sector
Charge sector of the vector which is acted on.

flat_to_npc(*self*, *vec*)
Convert flat vector of selected charge sector into npc Array.

Parameters

vec [1D ndarray] Numpy vector to be converted. Should have the entries according to self.charge_sector.

Returns

npc_vec [*Array*] Same as *vec*, but converted into a flat array.

npc_to_flat(*self*, *npc_vec*)
Convert npc Array with qtotal = self.charge_sector into ndarray.

Parameters

npc_vec [*Array*] Npc Array to be converted. Should only have entries in *self.charge_sector*.

Returns

vec [1D ndarray] Same as *npc_vec*, but converted into a flat Numpy array.

property H
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns

A_H [LinearOperator] Hermitian adjoint of self.

property T
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

adjoint(*self*)
Hermitian adjoint.

Returns the Hermitian adjoint of self, aka the Hermitian conjugate or Hermitian transpose. For a complex matrix, the Hermitian adjoint is equal to the conjugate transpose.

Can be abbreviated self.H instead of self.adjoint().

Returns

A_H [LinearOperator] Hermitian adjoint of self.

dot(*self*, *x*)
Matrix-matrix or matrix-vector multiplication.

Parameters

x [array_like] 1-d or 2-d array, representing a vector or matrix.

Returns

Ax [array] 1-d or 2-d array (depending on the shape of x) that represents the result of applying this linear operator on x.

matmat(*self*, *X*)
Matrix-matrix multiplication.

Performs the operation y=A*X where A is an MxN linear operator and X dense N*K matrix or ndarray.

Parameters

X [{matrix, ndarray}] An array with shape (N,K).

Returns

Y [{matrix, ndarray}] A matrix or ndarray with shape (M,K) depending on the type of the X argument.

Notes

This matmat wraps any user-specified matmat routine or overridden _matmat method to ensure that y has the correct type.

matvec (*self*, *x*)
Matrix-vector multiplication.

Performs the operation y=A*x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters

x [{matrix, ndarray}] An array with shape (N,) or (N,1).

Returns

y [{matrix, ndarray}] A matrix or ndarray with shape (M,) or (M,1) depending on the type and shape of the x argument.

Notes

This matvec wraps the user-specified matvec routine or overridden _matvec method to ensure that y has the correct shape and type.

rmatmat (*self*, *X*)
Adjoint matrix-matrix multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array, or 2-d array. The default implementation defers to the adjoint.

Parameters

X [{matrix, ndarray}] A matrix or 2D array.

Returns

Y [{matrix, ndarray}] A matrix or 2D array depending on the type of the input.

Notes

This rmatmat wraps the user-specified rmatmat routine.

rmatvec (*self*, *x*)
Adjoint matrix-vector multiplication.

Performs the operation y = A^H * x where A is an MxN linear operator and x is a column vector or 1-d array.

Parameters

x [{matrix, ndarray}] An array with shape (M,) or (M,1).

Returns

y [{matrix, ndarray}] A matrix or ndarray with shape (N,) or (N,1) depending on the type and shape of the x argument.

Notes

This rmatvec wraps the user-specified rmatvec routine or overridden _rmatvec method to ensure that y has the correct shape and type.

transpose(*self*)
Transpose this linear operator.

Returns a LinearOperator that represents the transpose of this one. Can be abbreviated self.T instead of self.transpose().

NpcLinearOperator

- full name: tenpy.linalg.sparse.NpcLinearOperator
- parent module: *tenpy.linalg.sparse*
- type: class

class tenpy.linalg.sparse.**NpcLinearOperator**
Bases: object

Prototype for a Linear Operator acting on *Array*.

Note that an *Array* implements a matvec function. Thus you can use any (square) npc Array as an NpcLinearOperator.

Attributes

dtype [np.type] The data type of its action.

acts_on [list of str] Labels of the state on which the operator can act. NB: Class attribute.

Methods

matvec(self, vec)	Calculate the action of the operator on a vector *vec*.

matvec(*self*, *vec*)
Calculate the action of the operator on a vector *vec*.

Note that we don't require *vec* to be one-dimensional. However, for square operators we require that the result of *matvec* has the same legs (in the same order) as *vec* such that they can be added. Note that this excludes a non-trivial *qtotal* for square operators.

Module description

Providing support for sparse algorithms (using matrix-vector products only).

Some linear algebra algorithms, e.g. Lanczos, do not require the full representations of a linear operator, but only the action on a vector, i.e., a matrix-vector product *matvec*. Here we define the strucuture of such a general operator, *NpcLinearOperator*, as it is used in our own implementations of these algorithms (e.g., *lanczos*). Moreover, the *FlatLinearOperator* allows to use all the scipy sparse methods by providing functionality to convert flat numpy arrays to and from np_conserved arrays.

lanczos

- full name: tenpy.linalg.lanczos
- parent module: *tenpy.linalg*
- type: module

Classes

LanczosEvolution(H, psi0, params)	Calculate $exp(deltaH)\|psi0>$ using Lanczos.
LanczosGroundState(H, psi0, params[, ...])	Lanczos algorithm working on npc arrays.

LanczosEvolution

- full name: tenpy.linalg.lanczos.LanczosEvolution
- parent module: *tenpy.linalg.lanczos*
- type: class

class tenpy.linalg.lanczos.**LanczosEvolution**(*H*, *psi0*, *params*)

Bases: *tenpy.linalg.lanczos.LanczosGroundState*

Calculate $exp(deltaH)|psi0>$ using Lanczos.

It turns out that the Lanczos algorithm is also good for calculating the matrix exponential applied to the starting vector. Instead of diagonalizing the tri-diagonal T and taking the ground state, we now calculate `exp(delta T) e_0 in the Krylov ONB, where ``e_0 = (1, 0, 0, ...)` corresponds to `psi0` in the original basis.

Parameters

H, psi0, params : Hamiltonian, starting vector and parameters as defined in *LanczosGroundState*. The parameters *E_tol* and *min_gap* are ignored, the parameters *P_tol* defines when convergence is reached, see `_converged()` for details.

Attributes

delta [float/complex] Prefactor of H in the exponential.

_result_norm [float] Norm of the resulting vector.

Methods

run(self, delta)	Calculate `expm(delta H).dot(psi0)` using Lanczos.

run(*self*, *delta*)

Calculate `expm(delta H).dot(psi0)` using Lanczos.

Parameters

delta [float/complex] Time step by which we should evolve psi0: prefactor of H in the exponential. Note that the complex *i* is *not* included!

Returns

psi_f [*Array*] Best approximation for `expm(delta H).dot(psi0)`

N [int] Krylov space dimension used.

LanczosGroundState

- full name: tenpy.linalg.lanczos.LanczosGroundState
- parent module: *tenpy.linalg.lanczos*
- type: class

class tenpy.linalg.lanczos.**LanczosGroundState**(*H*, *psi0*, *params*, *orthogonal_to=[]*)
Bases: object

Lanczos algorithm working on npc arrays.

The Lanczos algorithm can finds extremal eigenvalues (in terms of magnitude) along with the corresponding eigenvectors. It assumes that the linear operator *H* is hermitian. Given a start vector *psi0*, it generates an orthonormal basis of the Krylov space, in which *H* is a small tridiagonal matrix, and solves the eigenvalue problem there. Finally, it transform the resulting ground state back into the original space.

Parameters

H [*NpcLinearOperator*-like] A hermitian linear operator. Must implement the method *matvec* acting on a *Array*; nothing else required. The result has to have the same legs as the argument.

psi0 [*Array*] The starting vector defining the Krylov basis. For finding the ground state, this should be the best guess available. Note that it must not be a 1D "vector", we are fine with viewing higher-rank tensors as vectors.

params [dict] Further optional parameters as described in the following table. Add a parameter `verbose >=1` to print the used parameters during runtime. The algorithm stops if *both* criteria for *e_tol* and *p_tol* are met or if the maximum number of steps was reached.

key	type	description
N_min	int	Minimum number of steps to perform.
N_max	int	Maximum number of steps to perform.
E_tol	float	Stop if energy difference per step < *E_tol*
P_tol	float	Tolerance for the error estimate from the Ritz Residual, stop if `(RitzRes/gap)**2 < P_tol`
min_gap	float	Lower cutoff for the gap estimate used in the P_tol criterion.
N_cache	int	The maximum number of *psi* to keep in memory during the first iteration. By default, we keep all states (up to N_max). Set this to a number >= 2 if you are short on memory. The penalty is that one needs another Lanczos iteration to determine the ground state in the end, i.e., runtime is large.
reortho	bool	For poorly conditioned matrices, one can quickly loose orthogonality of the generated Krylov basis. If *reortho* is True, we re-orthogonalize against all the vectors kept in cache to avoid that problem.
cutoff	float	Cutoff to abort if *beta* (= norm of next vector in Krylov basis before normalizing) is too small. This is necessary if the rank of A is smaller than N_max - then we get a complete basis of the Krylov space, and *beta* will be zero.

orthogonal_to [list of *Array*] Vectors (same tensor structure as psi) against which Lanczos will orthogonalize, ensuring that the result is perpendicular to them. (Assumes that the

smallest eigenvalue is smaller than 0, which should *always* be the case if you want to find ground states with Lanczos!)

Notes

I have computed the Ritz residual *RitzRes* according to http://web.eecs.utk.edu/~dongarra/etemplates/node103.html#estimate_residual. Given the gap, the Ritz residual gives a bound on the error in the wavefunction, `err < (RitzRes/gap)**2`. The gap is estimated from the full Lanczos spectrum.

Attributes

H [*NpcLinearOperator*-like] The hermitian linear operator.

psi0 [*Array*] The starting vector.

orthogonal_to [list of *Array*] Vectors to orthogonalize against.

N_min, N_max, E_tol, P_tol, N_cache, reortho: Parameters as described above.

Es [ndarray, shape(N_max, N_max)] `Es[n, :]` contains the energies of `_T[:n+1, :n+1]` in step *n*.

_T [ndarray, shape (N_max + 1, N_max +1)] The tridiagonal matrix representing *H* in the orthonormalized Krylov basis.

_cutoff [float] See parameter *cutoff*.

_cache [list of psi0-like vectors] The ONB of the Krylov space generated during the iteration. FIFO (first in first out) cache of at most N_cache vectors.

_result_krylov [ndarray] Result in the ONB of the Krylov space: ground state of *_T*.

Methods

run(self)	Find the ground state of H.

run (*self*)
Find the ground state of H.

Returns

E0 [float] Ground state energy (estimate).

psi0 [*Array*] Ground state vector (estimate).

N [int] Used dimension of the Krylov space, i.e., how many iterations where performed.

Functions

gram_schmidt(vecs[, rcond, verbose])	In place Gram-Schmidt Orthogonalization and normalization for npc Arrays.
lanczos(H, psi[, lanczos_params, orthogonal_to])	Simple wrapper calling `LanczosGroundState(H, psi, params, orthogonal_to).run()`
lanczos_arpack(H, psi[, lanczos_params, ...])	Use `scipy.sparse.linalg.eigsh()` to find the ground state of *H*.
plot_stats(ax, Es)	Plot the convergence of the energies.

gram_schmidt

- full name: tenpy.linalg.lanczos.gram_schmidt
- parent module: *tenpy.linalg.lanczos*
- type: function

tenpy.linalg.lanczos.**gram_schmidt**(*vecs*, *rcond=1e-14*, *verbose=0*)
In place Gram-Schmidt Orthogonalization and normalization for npc Arrays.

Parameters

vecs [list of *Array*] The vectors which should be orthogonalized. All with the same *order* of the legs. Entries are modified *in place*. if a norm < rcond, the entry is set to *None*.

rcond [float] Vectors of `norm < rcond` (after projecting out previous vectors) are discarded.

verbose [int] Print additional output if verbose >= 1.

Returns

vecs [list of Array] The ortho-normalized vectors (without any `None`).

ov [2D Array] For `j >= i`, `ov[j, i] = npc.inner(vecs[j], vecs[i], 'range', do_conj=True)` (where vecs[j] was orthogonalized to all `vecs[k], k < i`).

lanczos

- full name: tenpy.linalg.lanczos.lanczos
- parent module: *tenpy.linalg.lanczos*
- type: function

tenpy.linalg.lanczos.**lanczos**(*H*, *psi*, *lanczos_params={}*, *orthogonal_to=[]*)
Simple wrapper calling `LanczosGroundState(H, psi, params, orthogonal_to).run()`

Parameters

H, psi, lanczos_params, orthogonal_to : See *LanczosGroundState*.

Returns

E0, psi0, N : See *LanczosGroundState.run()*.

lanczos_arpack

- full name: tenpy.linalg.lanczos.lanczos_arpack
- parent module: *tenpy.linalg.lanczos*
- type: function

tenpy.linalg.lanczos.**lanczos_arpack**(*H*, *psi*, *lanczos_params={}*, *orthogonal_to=[]*)
Use `scipy.sparse.linalg.eigsh()` to find the ground state of *H*.

This function has the same call/return structure as *lanczos()*, but uses the ARPACK package through the functions *speigsh()* instead of the custom lanczos implementation in *LanczosGroundState*. This function is mostly intended for debugging, since it requires to convert the vector from np_conserved *Array* into a flat numpy array and back during *each matvec*-operation!

Parameters

H, psi, lanczos_params, orthogonal_to : See *LanczosGroundState*. *H* and *psi* should have/use labels.

Returns

E0 [float] Ground state energy.

psi0 [*Array*] Ground state vector.

plot_stats

- full name: tenpy.linalg.lanczos.plot_stats
- parent module: *tenpy.linalg.lanczos*
- type: function

`tenpy.linalg.lanczos.`**`plot_stats`**(*ax*, *Es*)
Plot the convergence of the energies.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

Es [list of ndarray.] The energies `Lanczos.Es`.

Module description

Lanczos algorithm for np_conserved arrays.

7.2.3 models

- full name: tenpy.models
- parent module: *tenpy*
- type: module

Module description

Definition of the various models.

For an introduction to models see *Introduction to models*.

The module *tenpy.models.model* contains base classes for models. The module *tenpy.models.lattice* contains base classes and implementations of lattices. All other modules in this folder contain model classes derived from these base classes.

Submodules

lattice	Classes to define the lattice structure of a model.
model	This module contains some base classes for models.

lattice

- full name: tenpy.models.lattice
- parent module: *tenpy.models*
- type: module

Classes

Chain(L, site, **kwargs)	A chain of L equal sites.
Honeycomb(Lx, Ly, sites, **kwargs)	A honeycomb lattice.
IrregularLattice(mps_sites, based_on[, order])	A variant of a regular lattice, where we might have extra sites or sites missing.
Kagome(Lx, Ly, sites, **kwargs)	A Kagome lattice.
Ladder(L, sites, **kwargs)	A ladder coupling two chains.
Lattice(Ls, unit_cell[, order, bc, bc_MPS, ...])	A general, regular lattice.
SimpleLattice(Ls, site, **kwargs)	A lattice with a unit cell consiting of just a single site.
Square(Lx, Ly, site, **kwargs)	A square lattice.
Triangular(Lx, Ly, site, **kwargs)	A triangular lattice.
TrivialLattice(mps_sites, **kwargs)	Trivial lattice consisting of a single (possibly large) unit cell in 1D.

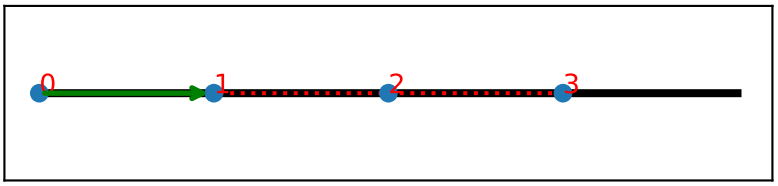
Chain

- full name: tenpy.models.lattice.Chain
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**Chain**(*L*, *site*, ***kwargs*)

Bases: *tenpy.models.lattice.SimpleLattice*

A chain of L equal sites.



Parameters

L [int] The lenght of the chain.

site [*Site*] The local lattice site. The *unit_cell* of the *Lattice* is just `[site]`.

****kwargs :** Additional keyword arguments given to the *Lattice*. *pairs* are initialize with `[next_]next_]nearest_neighbors`. *positions* can be specified as a single vector.

Attributes

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

ordering(*self*, *order*)

Provide possible orderings of the *N* lattice sites.

The following orders are defined in this method compared to `Lattice.ordering()`:

order	Resulting order
`'default'`	`0, 1, 2, 3, 4, ... , L-1`
`'folded'`	`0, L-1, 1, L-2, ... , L//2`. This order might be usefull if you want to consider a ring with periodic boundary conditions with a finite MPS: It avoids the ultra-long range of the coupling from site 0 to L present in the default order.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)
Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values (*self*, *A*, *axes=0*, *u=None*)
same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.

mps_idx_fix_u (*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u (*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites (*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape (*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors (*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If `None`, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of `unit_cell_positions`.

****kwargs :** Keyword arguments for the used `ax.plot`.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (`None`), use `self.pairs['nearest_neighbors']`. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site `(i0, i1, ..., u1)` to the site `(i0+dx[0], i1+dx[1], ..., u2)`, taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (`None`) use *order*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites (*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position (*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings (*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length `dim`. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings (*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary

conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

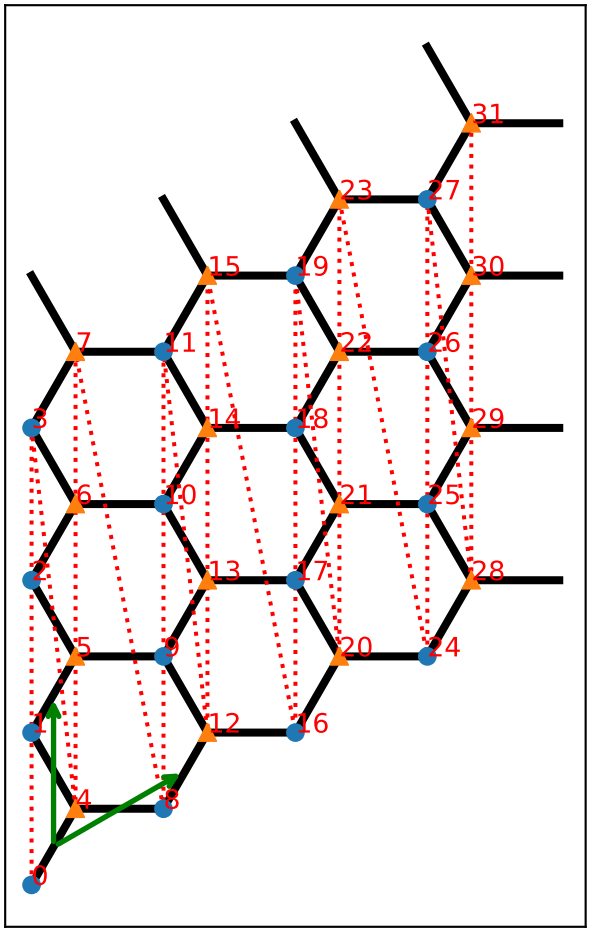
Raises ValueErrors, if something is wrong.

Honeycomb

- full name: tenpy.models.lattice.Honeycomb
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**Honeycomb**(*Lx*, *Ly*, *sites*, ***kwargs*)
Bases: *tenpy.models.lattice.Lattice*

A honeycomb lattice.



Parameters

Lx, Ly [int] The length in each direction.

sites [(list of) *Site*] The two local lattice sites making the *unit_cell* of the *Lattice*. If only a single *Site* is given, it is used for both sites.

****kwargs :** Additional keyword arguments given to the *Lattice*. *basis*, *pos* and *[[next_]next_]nearest_neighbors* are set accordingly. For the Honeycomb lattice `'fourth_nearest_neighbors', 'fifth_nearest_neighbors'` are set in `pairs`.

Attributes

fifth_nearest_neighbors

fourth_nearest_neighbors

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

ordering (*self*, *order*)
Provide possible orderings of the *N* lattice sites.

The following orders are defined in this method compared to *Lattice.ordering()*:

order	equivalent *priority*	equivalent `snake_winding`
`'default'`	(0, 2, 1)	(False, False, False)
`'snake'`	(0, 2, 1)	(False, True, False)

count_neighbors (*self*, *u=0*, *key='nearest_neighbors'*)
Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)

Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

lat2mps_idx(*self*, *lat_idx*)

Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)

Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)

Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [None | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If None, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of unit_cell_positions.

****kwargs :** Keyword arguments for the used ax.plot.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (None), use self.pairs['nearest_neighbors']. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site (i0, i1, ..., u1) to the site (i0+dx[0], i1+dx[1], ..., u2), taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to ax.plot().

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not None, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for ax.text().

****kwargs :** Further keyword arguments given to ax.plot().

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of ax.plot() to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker markers[u % len(markers)].

****kwargs :** Further keyword arguments given to ax.plot().

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, (... , dim+1)] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length `dim`. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

IrregularLattice

- full name: tenpy.models.lattice.IrregularLattice
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**IrregularLattice**(*mps_sites*, *based_on*, *order=None*)
Bases: *tenpy.models.lattice.Lattice*

A variant of a regular lattice, where we might have extra sites or sites missing.

Todo:

- this doesn't fully work yet...

Attributes

dim The dimension of the lattice.

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.

Continued on next page

Table 74 – continued from previous page

plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

from_add_sites	
from_mps_sites	

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)
Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

property dim
The dimension of the lattice.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)
Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [`None` | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors (*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors (*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering (*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, . . . , dim-1, dim)	(False, . . . , False, False)
`'default'`		
`'snake'`	(0, 1, . . . , dim-1, dim)	(True, . . . , True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, . . . , 1, 0, dim)	(False, . . . , False, False)
`'snakeFstyle'`	(dim-1, . . . , 1, 0, dim)	(False, . . . , False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function, `'standard'` for *get_order()* and `'grouped'` for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis (*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If None, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of unit_cell_positions.

****kwargs :** Keyword arguments for the used ax.plot.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (None), use self.pairs['nearest_neighbors']. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site (i0, i1, ..., u1) to the site (i0+dx[0], i1+dx[1], ..., u2), taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to ax.plot().

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not None, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for ax.text().

****kwargs :** Further keyword arguments given to ax.plot().

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of ax.plot() to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker markers[u % len(markers)].

****kwargs :** Further keyword arguments given to ax.plot().

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, (... , dim+1)] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length *dim*. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

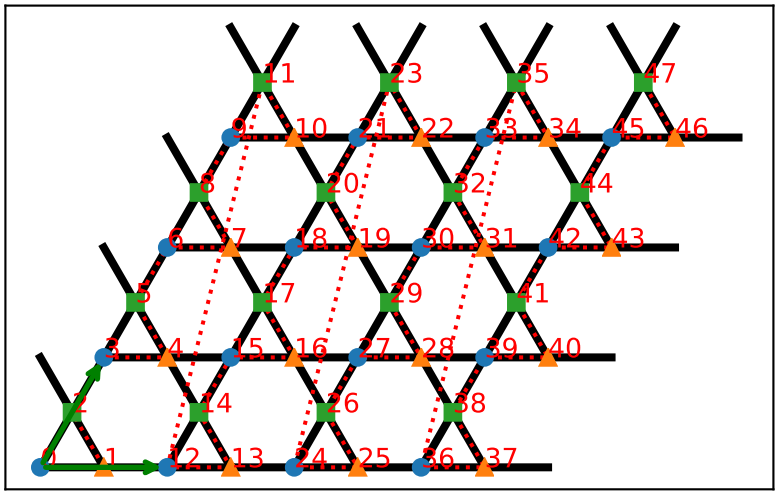
Raises ValueErrors, if something is wrong.

Kagome

- full name: tenpy.models.lattice.Kagome
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**Kagome**(*Lx*, *Ly*, *sites*, ***kwargs*)
Bases: *tenpy.models.lattice.Lattice*

A Kagome lattice.



Parameters

Lx, Ly [int] The length in each direction.

sites [(list of) *Site*] The two local lattice sites making the *unit_cell* of the *Lattice*. If only a single *Site* is given, it is used for both sites.

****kwargs :** Additional keyword arguments given to the *Lattice*. *basis*, *pos* and *[[next_]next_]nearest_neighbors* are set accordingly.

Attributes

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

count_neighbors (*self*, *u=0*, *key='nearest_neighbors'*)

Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape (*self*, *dx*)

Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)
Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [`None` | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

---

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function,

'standard' for *get_order()* and 'grouped' for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of ax.annotate.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If None, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of unit_cell_positions.

****kwargs :** Keyword arguments for the used ax.plot.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (None), use self.pairs['nearest_neighbors']. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site (i0, i1, ..., u1) to the site (i0+dx[0], i1+dx[1], ..., u2), taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to ax.plot().

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites (*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position (*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings (*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length `dim`. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings (*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site (*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity (*self*)
Sanity check.

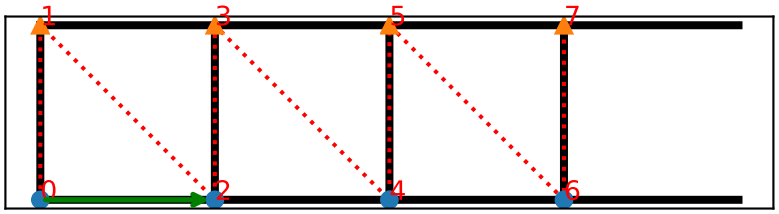
Raises ValueErrors, if something is wrong.

Ladder

- full name: tenpy.models.lattice.Ladder
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**Ladder** (*L*, *sites*, ***kwargs*)
Bases: *tenpy.models.lattice.Lattice*

A ladder coupling two chains.



Parameters

L [int] The length of each chain, we have 2*L sites in total.

sites [(list of) *Site*] The two local lattice sites making the *unit_cell* of the *Lattice*. If only a single *Site* is given, it is used for both chains.

****kwargs :** Additional keyword arguments given to the *Lattice*. *basis*, *pos* and *[[next_]next_]nearest_neighbors* are set accordingly.

Attributes

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)

Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)

Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)
Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [`None` | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

---

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function,

'standard' for *get_order()* and 'grouped' for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of ax.annotate.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If None, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of unit_cell_positions.

****kwargs :** Keyword arguments for the used ax.plot.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (None), use self.pairs['nearest_neighbors']. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site (i0, i1, ..., u1) to the site (i0+dx[0], i1+dx[1], ..., u2), taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to ax.plot().

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length `dim`. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

Lattice

- full name: tenpy.models.lattice.Lattice
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**Lattice**(*Ls*, *unit_cell*, *order='default'*, *bc='open'*, *bc_MPS='finite'*, *basis=None*, *positions=None*, *nearest_neighbors=None*, *next_nearest_neighbors=None*, *next_next_nearest_neighbors=None*, *pairs=None*)

Bases: `object`

A general, regular lattice.

The lattice consists of a **unit cell** which is repeated in *dim* different directions. A site of the lattice is thus identified by **lattice indices** `(x_0, ..., x_{dim-1}, u)`, where `0 <= x_l < Ls[l]` pick the position of the unit cell in the lattice and `0 <= u < len(unit_cell)` picks the site within the unit cell. The site is located in 'space' at `sum_l x_l*basis[l] + unit_cell_positions[u]` (see *position()*). (Note that the position in space is only used for plotting, not for defining the couplings.)

In addition to the pure geometry, this class also defines an *order* of all sites. This order maps the lattice to a finite 1D chain and defines the geometry of MPSs and MPOs. The **MPS index** *i* corresponds thus to the lattice sites given by `(x_0, ..., x_{dim-1}, u) = tuple(self.order[i])`. Infinite boundary conditions of the MPS repeat in the first spatial direction of the lattice, i.e., if the site at (x_0, x_1, ..., x_{dim-1},u)`` has MPS index *i*, the site at at `(x_0 + a*Ls[0], x_1 ..., x_{dim-1}, u)` corresponds to MPS index `i + N_sites`. Use *mps2lat_idx()* and *lat2mps_idx()* for conversion of indices. The function *mps2lat_values()* performs the necessary reshaping and re-ordering from arrays indexed in MPS form to arrays indexed in lattice form.

Parameters

Ls [list of int] the length in each direction

unit_cell [list of *Site*] The sites making up a unit cell of the lattice. If you want to specify it only after initialization, use `None` entries in the list.

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] A string or tuple specifying the order, given to *ordering()*.

bc [(iterable of) {'open' | 'periodic' | int}] Boundary conditions in each direction of the lattice. A single string holds for all directions. An integer *shift* means that we have periodic boundary conditions along this direction, but shift/tilt by `-shift*lattice.basis[0]` (~cylinder axis for `bc_MPS='infinite'`) when going around the boundary along this direction.

bc_MPS ['finite' | 'segment' | 'infinite'] Boundary conditions for an MPS/MPO living on the ordered lattice. If the system is `'infinite'`, the infinite direction is always along the first basis vector (justifying the definition of *N_rings* and *N_sites_per_ring*).

basis [iterable of 1D arrays] For each direction one translation vectors shifting the unit cell. Defaults to the standard ONB `np.eye(dim)`.

positions [iterable of 1D arrays] For each site of the unit cell the position within the unit cell. Defaults to `np.zeros((len(unit_cell), dim))`.

nearest_neighbors [`None` | list of `(u1, u2, dx)`] Deprecated. Specify as `pairs['nearest_neighbors']` instead.

next_nearest_neighbors [`None` | list of `(u1, u2, dx)`] Deprecated. Specify as `pairs['next_nearest_neighbors']` instead.

next_next_nearest_neighbors [`None` | list of `(u1, u2, dx)`] Deprecated. Specify as `pairs['next_next_nearest_neighbors']` instead.

pairs [dict] Of the form `{'nearest_neighbors': [(u1, u2, dx), ...], ...}`. Typical keys are `'nearest_neighbors'`, `'next_nearest_neighbors'`. For each of them, it specifies a list of tuples `(u1, u2, dx)` which can be used as parameters for *add_coupling()* to generate couplings over each pair of e.g. `'nearest_neighbors'`. Note that this adds couplings for each pair *only in one direction*!

Attributes

dim [int] The dimension of the lattice.

order [ndarray (N_sites, dim+1)] Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

N_cells [int] the number of unit cells in the lattice, `np.prod(self.Ls)`.

N_sites [int] the number of sites in the lattice, `np.prod(self.shape)`.

N_sites_per_ring [int] Defined as `N_sites / Ls[0]`, for an infinite system the number of cites per "ring".

N_rings [int] Alias for `Ls[0]`, for an infinite system the number of "rings" in the unit cell.

Ls [tuple of int] the length in each direction.

shape [tuple of int] the 'shape' of the lattice, same as `Ls + (len(unit_cell), )`

unit_cell [list of *Site*] the lattice sites making up a unit cell of the lattice.

bc [bool ndarray] Boundary conditions of the couplings in each direction of the lattice, translated into a bool array with the global *bc_choices*.

bc_shift [None | ndarray(int)] The shift in x-direction when going around periodic boundaries in other directions.

bc_MPS ['finite' | 'segment' | 'infinite'] Boundary conditions for an MPS/MPO living on the ordered lattice. If the system is `'infinite'`, the infinite direction is always along the first basis vector (justifying the definition of *N_rings* and *N_sites_per_ring*).

basis [ndarray (dim, Dim)] translation vectors shifting the unit cell. The row *i* gives the vector shifting in direction *i*.

unit_cell_positions [ndarray, shape (len(unit_cell), Dim)] for each site in the unit cell a vector giving its position within the unit cell.

pairs [dict] See above.

_order [ndarray (N_sites, dim+1)] The place where *order* is stored.

_strides [ndarray (dim,)] necessary for *mps2lat_idx()*

_perm [ndarray (N,)] permutation needed to make *order* lexsorted.

_mps2lat_vals_idx [ndarray *shape*] index array for reshape/reordering in `mps2lat_vals()`

_mps_fix_u [tuple of ndarray (N_cells,) np.intp] for each site of the unit cell an index array selecting the mps indices of that site.

_mps2lat_vals_idx_fix_u [tuple of ndarray of shape *Ls*] similar as *_mps2lat_vals_idx*, but for a fixed *u* picking a site from the unit cell.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.

Continued on next page

Table 77 – continued from previous page

site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

test_sanity (*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

property dim
The dimension of the lattice.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering (*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, . . . , dim-1, dim)	(False, . . . , False, False)
`'default'`		
`'snake'`	(0, 1, . . . , dim-1, dim)	(True, . . . , True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, . . . , 1, 0, dim)	(False, . . . , False, False)
`'snakeFstyle'`	(dim-1, . . . , 1, 0, dim)	(False, . . . , False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function, `'standard'` for *get_order()* and `'grouped'` for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

position (*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

site(*self*, *i*)
return `Site` instance corresponding to an MPS index *i*

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as `mps_idx_fix_u()`, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)

Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [`None` | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

count_neighbors (*self*, *u=0*, *key='nearest_neighbors'*)
Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

number_nearest_neighbors (*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors (*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

possible_couplings (*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length *dim*. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings (*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (`None`) use *order*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (`None`), use `self.pairs['nearest_neighbors']`. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site `(i0, i1, ..., u1)` to the site `(i0+dx[0], i1+dx[1], ..., u2)`, taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If `None`, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of `unit_cell_positions`.

****kwargs :** Keyword arguments for the used `ax.plot`.

SimpleLattice

- full name: tenpy.models.lattice.SimpleLattice
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**SimpleLattice**(*Ls*, *site*, ***kwargs*)
Bases: *tenpy.models.lattice.Lattice*

A lattice with a unit cell consiting of just a single site.

In many cases, the unit cell consists just of a single site, such that the the last entry of *u* of an 'lattice index' can only be `0`. From the point of internal algorithms, we handle this class like a *Lattice* – in that way we don't need to distinguish special cases in the algorithms.

Yet, from the point of a tenpy user, for example if you measure an expectation value on each site in a *SimpleLattice*, you expect to get an ndarray of dimensions `self.Ls`, not `self.shape`. To avoid that problem, *SimpleLattice* overwrites just the meaning of `u=None` in *mps2lat_values()* to be the same as `u=0`.

Parameters

Ls [list of int] the length in each direction

site [*Site*] the lattice site. The *unit_cell* of the *Lattice* is just `[site]`.

****kwargs :** Additional keyword arguments given to the *Lattice*. If *order* is specified in the form `('standard', snake_windingi, priority)`, the *snake_winding* and *priority* should only be specified for the spatial directions. Similarly, *positions* can be specified as a single vector.

Attributes

dim The dimension of the lattice.

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.

Continued on next page

Table 78 – continued from previous page

position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)
same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)
Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

property dim
The dimension of the lattice.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function, `'standard'` for *get_order()* and `'grouped'` for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If `None`, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of `unit_cell_positions`.

****kwargs :** Keyword arguments for the used `ax.plot`.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (`None`), use `self.pairs['nearest_neighbors']`. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site `(i0, i1, ..., u1)` to the site `(i0+dx[0], i1+dx[1], ..., u2)`, taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *`ordering()`*; by default (`None`) use *`order`*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length *dim*. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

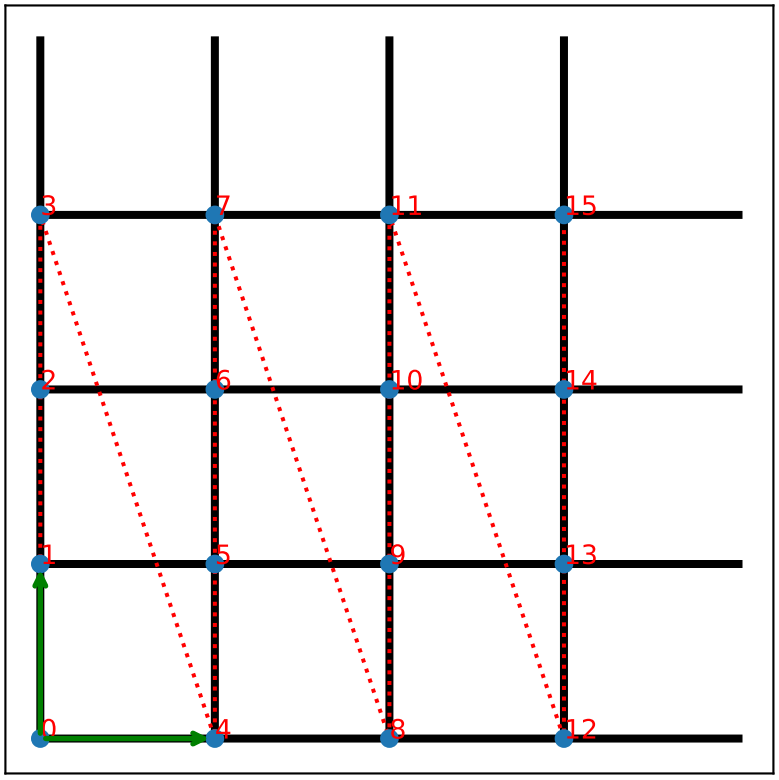
Raises ValueErrors, if something is wrong.

Square

- full name: tenpy.models.lattice.Square
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**Square**(*Lx*, *Ly*, *site*, ***kwargs*)
Bases: *tenpy.models.lattice.SimpleLattice*

A square lattice.



Parameters

Lx, Ly [int] The length in each direction.

site [*Site*] The local lattice site. The *unit_cell* of the *Lattice* is just [site].

****kwargs :** Additional keyword arguments given to the *Lattice*. *[[next_]next_]nearest_neighbors* are set accordingly. If *order* is specified in the form ('standard', snake_windingi, priority), the *snake_winding* and *priority* should only be specified for the spatial directions. Similarly, *positions* can be specified as a single vector.

Attributes

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)

Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)
same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function, `'standard'` for *get_order()* and `'grouped'` for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If `None`, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of `unit_cell_positions`.

****kwargs :** Keyword arguments for the used `ax.plot`.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (`None`), use `self.pairs['nearest_neighbors']`. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site `(i0, i1, ..., u1)` to the site `(i0+dx[0], i1+dx[1], ..., u2)`, taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not None, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for ax.text().

****kwargs :** Further keyword arguments given to ax.plot().

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of ax.plot() to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker markers[u % len(markers)].

****kwargs :** Further keyword arguments given to ax.plot().

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, (... , dim+1)] Lattice indices.

Returns

pos [ndarray, (..., dim)] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (bc[a] == False) the index x_a is taken modulo Ls[a] and runs through range(Ls[a]). For open boundary conditions, x_a is limited to 0 <= x_a < Ls[a] and 0 <= x_a+dx[a] < lat.Ls[a].

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length dim. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len dim. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in `add_multi_coupling()`.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in `add_multi_coupling()`.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return `Site` instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

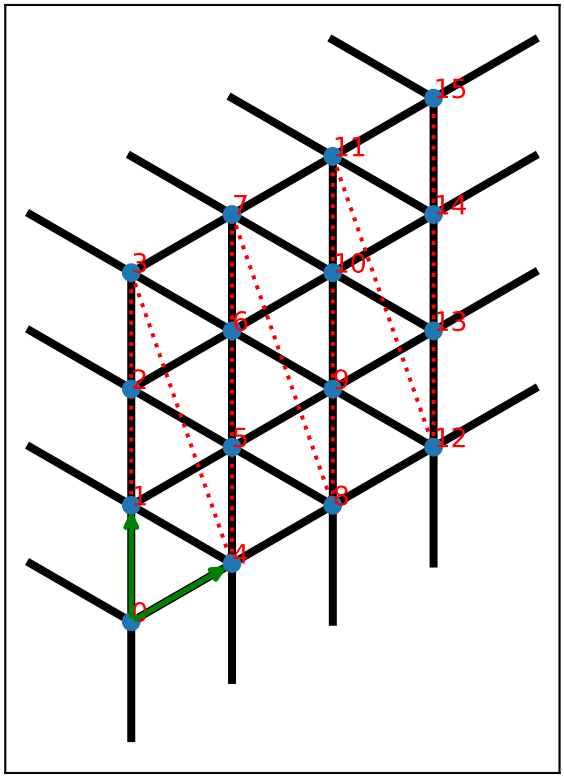
Raises ValueErrors, if something is wrong.

Triangular

- full name: tenpy.models.lattice.Triangular
- parent module: `tenpy.models.lattice`
- type: class

class tenpy.models.lattice.**Triangular**(*Lx*, *Ly*, *site*, ***kwargs*)
Bases: `tenpy.models.lattice.SimpleLattice`

A triangular lattice.



Parameters

Lx, Ly [int] The length in each direction.

site [*Site*] The local lattice site. The *unit_cell* of the *Lattice* is just `[site]`.

****kwargs :** Additional keyword arguments given to the *Lattice*. *[[next_]next_]nearest_neighbors* are set accordingly. If *order* is specified in the form `('standard', snake_windingi, priority)`, the *snake_winding* and *priority* should only be specified for the spatial directions. Similarly, *positions* can be specified as a single vector.

Attributes

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)

Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)

Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

lat2mps_idx(*self*, *lat_idx*)
Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)
Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)
same as *Lattice.mps2lat_values()*, but ignore `u`, setting it to `0`.

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function,

'standard' for *get_order()* and 'grouped' for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of ax.annotate.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If None, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of unit_cell_positions.

****kwargs :** Keyword arguments for the used ax.plot.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (None), use self.pairs['nearest_neighbors']. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site (i0, i1, ..., u1) to the site (i0+dx[0], i1+dx[1], ..., u2), taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to ax.plot().

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length `dim`. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len `dim`. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

TrivialLattice

- full name: tenpy.models.lattice.TrivialLattice
- parent module: *tenpy.models.lattice*
- type: class

class tenpy.models.lattice.**TrivialLattice**(*mps_sites*, ***kwargs*)
Bases: *tenpy.models.lattice.Lattice*

Trivial lattice consisting of a single (possibly large) unit cell in 1D.

This is usefull if you need a valid *Lattice* given just the *mps_sites()*.

Parameters

mps_sites [list of *Site*] The sites making up a unit cell of the lattice.

****kwargs :** Further keyword arguments given to *Lattice*.

Attributes

dim The dimension of the lattice.

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)

Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)

Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

property dim

The dimension of the lattice.

lat2mps_idx(*self*, *lat_idx*)

Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)

Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)

Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [`None` | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

---

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function,

'standard' for *get_order()* and 'grouped' for *get_order_grouped()*, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of ax.annotate.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If None, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of unit_cell_positions.

****kwargs :** Keyword arguments for the used ax.plot.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (None), use self.pairs['nearest_neighbors']. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site (i0, i1, ..., u1) to the site (i0+dx[0], i1+dx[1], ..., u2), taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to ax.plot().

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [matplotlib.axes.Axes] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *ordering()*; by default (None) use *order*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *add_coupling()*

dx [array] Length *dim*. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

Functions

get_lattice(lattice_name)	Given the name of a *Lattice* class, create an instance of it with gi.
get_order(shape, snake_winding[, priority])	Built the *Lattice.order* in (Snake-) C-Style for a given lattice shape.
get_order_grouped(shape, groups)	Variant of *get_order()*, grouping some sites of the unit cell.

get_lattice

- full name: tenpy.models.lattice.get_lattice
- parent module: *tenpy.models.lattice*
- type: function

tenpy.models.lattice.**get_lattice**(*lattice_name*)
Given the name of a *Lattice* class, create an instance of it with gi.

Parameters

lattice_name [str] Name of a *Lattice* class defined in the module *lattice*, for example `"Chain", "Square", "Honeycomb", ...`.

***args, **kwargs** Arguments and keyword-arguments for the initialization of the specified lattice class.

Returns

LatticeClass [(subclass of) *Lattice*] An instance of the lattice class specified by *lattice_name*.

get_order

- full name: tenpy.models.lattice.get_order
- parent module: *tenpy.models.lattice*
- type: function

tenpy.models.lattice.**get_order**(*shape*, *snake_winding*, *priority=None*)
Built the *Lattice.order* in (Snake-) C-Style for a given lattice shape.

In this function, the word 'direction' referst to a physical direction of the lattice or the index *u* of the unit cell as an "artificial direction".

Parameters

shape [tuple of int] The shape of the lattice, i.e., the length in each direction.

snake_winding [tuple of bool] For each direction one bool, whether we should wind as a "snake" (True) in that direction (i.e., going forth and back) or simply repeat ascending (False)

priority [`None` | tuple of float] If `None` (default), use C-Style ordering. Otherwise, this defines the priority along which direction to wind first; the direction with the highest priority increases fastest. For example, "C-Style" order is enforced by `priority=(0, 1, 2, ...)`, and Fortrans F-style order is enforced by `priority=(dim, dim-1, ..., 1, 0)`

group [`None` | tuple of tuple] If `None` (default), ignore it. Otherwise, it specifies that we group the fastests changing dimension

Returns

order [ndarray (np.prod(shape), len(shape))] An order of the sites for *Lattice.order* in the specified *ordering*.

See also:

Lattice.ordering method in *Lattice* to obtain the order from parameters.

Lattice.plot_order visualizes the resulting order in a *Lattice*.

get_order_grouped a variant grouping sites of the unit cell.

get_order_grouped

- full name: tenpy.models.lattice.get_order_grouped
- parent module: *tenpy.models.lattice*
- type: function

tenpy.models.lattice.**get_order_grouped**(*shape*, *groups*)
Variant of *get_order()*, grouping some sites of the unit cell.

In this function, the word 'direction' referst to a physical direction of the lattice or the index *u* of the unit cell as an "artificial direction". This function is usefull for lattices with a unit cell of more than 2 sites (e.g. Kagome). The argument *group* is a To explain the order, assume we have a 3-site unit cell in a 2D lattice with shape (Lx, Ly, Lu). Calling this function with groups=((1,), (2, 0)) returns an order of the following form:

```
# columns: [x, y, u]
[0, 0, 1]  # first for u = 1 along y
[0, 1, 1]
    :
[0, Ly-1, 1]
[0, 0, 2]  # then for u = 2 and 0
[0, 0, 0]
[0, 1, 2]
[0, 1, 0]
    :
[0, Ly-1, 2]
[0, Ly-1, 0]
# and then repeat the above for increasing `x`.
```

Parameters

shape [tuple of int] The shape of the lattice, i.e., the length in each direction.

groups [tuple of tuple of int] A partition and reordering of range(shape[-1]) into smaller groups. The ordering goes first within a group, then along the last spatial dimensions, then changing between different groups and finally in Cstyle order along the remaining spatial dimensions.

Returns

order [ndarray (np.prod(shape), len(shape))] An order of the sites for *Lattice.order* in the specified *ordering*.

See also:

Lattice.ordering() method in *Lattice* to obtain the order from parameters.

Lattice.plot_order() visualizes the resulting order in a *Lattice*.

Module description

Classes to define the lattice structure of a model.

The base class *Lattice* defines the general structure of a lattice, you can subclass this to define you own lattice. The *SimpleLattice* is a slight simplification for lattices with a single-site unit cell. Further, we have some predefined lattices, namely *Chain*, *Ladder* in 1D and *Square*, *Triangular*, *Honeycomb*, and *Kagome* in 2D.

See also the *Introduction to models*.

model

- full name: tenpy.models.model
- parent module: *tenpy.models*
- type: module

Classes

CouplingMPOModel(model_params)	Combination of the *CouplingModel* and *MPOModel*.
CouplingModel(lattice[, bc_coupling])	Base class for a general model of a Hamiltonian consisting of two-site couplings.
MPOModel(lattice, H_MPO)	Base class for a model with an MPO representation of the Hamiltonian.
Model(lattice)	Base class for all models.
MultiCouplingModel(lattice[, bc_coupling])	Generalizes *CouplingModel* to allow couplings involving more than two sites.
NearestNeighborModel(lattice, H_bond)	Base class for a model of nearest neigbor interactions w.r.t.

CouplingMPOModel

- full name: tenpy.models.model.CouplingMPOModel
- parent module: *tenpy.models.model*
- type: class

class tenpy.models.model.**CouplingMPOModel**(*model_params*)

Bases: *tenpy.models.model.CouplingModel*, *tenpy.models.model.MPOModel*

Combination of the *CouplingModel* and *MPOModel*.

This class provides the interface for most of the model classes in *tenpy*. Examples based on this class are given in *xxz_chain* and *tf_ising*.

The `__init__` of this function performs the standard initialization explained in *Introduction to models*, by calling the methods *init_lattice()* (step 1-4) to initialize a lattice (which in turn calls *init_sites()*) and *init_terms()*. The latter should be overwritten by subclasses to add the desired terms.

As shown in *tf_ising*, you can get a 1D version suitable for TEBD from a general-lattice model by subclassing it once more, only redefining the `__init__` as follows:

```
def __init__(self, model_params):
    CouplingMPOModel.__init__(self, model_params)
```

Parameters

model_params [dict] A dictionary with all the model parameters. These parameters are given to the different `init_...()` methods, and should be read out using *get_parameter()*. This may happen in any of the `init_...()` methods. The parameter `'verbose'` is read out in the *__init__* of this function and specifies how much status information should be printed during initialization. The parameter `'sort_mpo_legs'` specifies whether the virtual legs of the MPO should be sorted by charges (see *sort_legcharges()*).

Attributes

name [str] The name of the model, e.g. `"XXZChain" or ``"SpinModel"`.

verbose [int] Level of verbosity (i.e. how much status information to print); higher=more output.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of *NearestNeighborModel*. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of *NearestNeighborModel*. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)

Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

CouplingModel

- full name: tenpy.models.model.CouplingModel
- parent module: *tenpy.models.model*
- type: class

class tenpy.models.model.**CouplingModel**(*lattice*, *bc_coupling=None*)
Bases: *tenpy.models.model.Model*

Base class for a general model of a Hamiltonian consisting of two-site couplings.

In this class, the terms of the Hamiltonian are specified explicitly as *OnsiteTerms* or *CouplingTerms*.

Deprecated since version 0.4.0: *bc_coupling* will be removed in 1.0.0. To specify the full geometry in the lattice, use the *bc* parameter of the `Lattice`.

Parameters

lattice [`Lattice`] The lattice defining the geometry and the local Hilbert space(s).

bc_coupling [(iterable of) {`'open'` | `'periodic'` | `int`}] Boundary conditions of the couplings in each direction of the lattice. Defines how the couplings are added in *add_coupling()*. A single string holds for all directions. An integer *shift* means that we have periodic boundary conditions along this direction, but shift/tilt by `-shift*lattice.basis[0]` (~cylinder axis for `bc_MPS='infinite'`) when going around the boundary along this direction.

Attributes

onsite_terms [{'category': *OnsiteTerms*}] The *OnsiteTerms* ordered by category.

coupling_terms [{'category': *CouplingTerms*}] The *CouplingTerms* ordered by category. In a *MultiCouplingModel*, values may also be *MultiCouplingTerms*.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of *NearestNeighborModel*. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

MPOModel

- full name: tenpy.models.model.MPOModel
- parent module: *tenpy.models.model*
- type: class

class tenpy.models.model.**MPOModel**(*lattice*, *H_MPO*)

Bases: *tenpy.models.model.Model*

Base class for a model with an MPO representation of the Hamiltonian.

In this class, the Hamiltonian gets represented by an *MPO*. Thus, instances of this class are suitable for MPO-based algorithms like DMRG *dmrg* and MPO time evolution.

Todo: implement MPO for time evolution...

Parameters

H_MPO [*MPO*] The Hamiltonian rewritten as an MPO.

Attributes

H_MPO [*tenpy.networks.mpo.MPO*] MPO representation of the Hamiltonian.

Methods

calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.

test_sanity

group_sites (*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

calc_H_bond_from_MPO (*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of *NearestNeighborModel*. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

Model

- full name: tenpy.models.model.Model
- parent module: *tenpy.models.model*
- type: class

class tenpy.models.model.**Model**(*lattice*)
Bases: object

Base class for all models.

The common base to all models is the underlying Hilbert space and geometry, specified by a Lattice.

Parameters

lattice [Lattice] The lattice defining the geometry and the local Hilbert space(s).

Attributes

lat [Lattice] The lattice defining the geometry and the local Hilbert space(s).

Methods

group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

MultiCouplingModel

- full name: tenpy.models.model.MultiCouplingModel
- parent module: *tenpy.models.model*
- type: class

class tenpy.models.model.**MultiCouplingModel**(*lattice*, *bc_coupling=None*)
Bases: *tenpy.models.model.CouplingModel*

Generalizes *CouplingModel* to allow couplings involving more than two sites.

The corresponding couplings can be added with *add_multi_coupling()* and *add_multi_coupling_term()* and are saved in coupling_terms, which can now contain instances of *MultiCouplingTerms*.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_multi_coupling(self, strength, u0, op0, ...)	Add multi-site coupling terms to the Hamiltonian, summing over lattice sites.
add_multi_coupling_term(self, strength, ...)	Add a general M-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

add_multi_coupling(*self*, *strength*, *u0*, *op0*, *other_ops*, *op_string=None*, *category=None*)

Add multi-site coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $sum_{x_0,...,x_{dim-1}} strength[loc(\vec{x})] * OP0 * OP1 * ... * OPM$, where `OP_0 := lat.unit_cell[u0].get_op(op0)` acts on the site `(x_0, ..., x_{dim-1}, u0)`, and `OP_m := lat.unit_cell[other_u[m]].get_op(other_op[m])`, m=1...M, acts on the site `(x_0+other_dx[m][0], ..., x_{dim-1}+other_dx[m][dim-1], other_u[m])`. For periodic boundary conditions along direction *a* (`lat.bc[a] == False`) the index `x_a` is taken modulo `lat.Ls[a]` and runs through `range(lat.Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+other_dx[m,a] < lat.Ls[a]`. The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing all the involved sites $\vec{x}, \vec{x} + other_d\vec{x}[m, :]$.

The necessary terms are just added to `coupling_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially and is tiled to the required shape.

u0 [int] Picks the site `lat.unit_cell[u0]` for OP0.

op0 [str] Valid operator name of an onsite operator in `lat.unit_cell[u0]` for OP0.

other_ops [list of `(u, op_m, dx)`] One tuple for each of the other operators `OP1, OP2, ... OPM` involved. *u* picks the site `lat.unit_cell[u]`, *op_name* is a valid operator acting on that site, and *dx* gives the translation vector between `OP0` and the specified operator.

op_string [str | None] Name of an operator to be used inbetween the operators, excluding the sites on which the operators act. This operator should be defined on all sites in the unit cell.

Special case: If `None`, auto-determine whether a Jordan-Wigner string is needed (using *op_needs_JW()*), for each of the segments inbetween the operators and also on the sites of the left operators. Note that in this case the ordering of the operators *is* important and handled in the usual convention that `OPM` acts first and `OP0` last on a physical state.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op0}_i {other_ops[0]}_j {other_ops[1]}_k ..."`.

add_multi_coupling_term(*self*, *strength*, *ijkl*, *ops_ijkl*, *op_string*, *category=None*)

Add a general M-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_multi_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

ijkl [list of int] The MPS indices of the sites on which the operators acts. With *i, j, k, ...* = *ijkl*, we require that they are ordered ascending, `i < j < k < ...` and that `0 <= i < N_sites`. Inidces >= N_sites indicate couplings between different unit cells of an infinite MPS.

ops_ijkl [list of str] Names of the involved operators on sites *i, j, k, ...*.

op_string [list of str] Names of the operator to be inserted between the operators, e.g., op_string[0] is inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op0}_i {other_ops[0]}_j {other_ops[1]}_k ..."`.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of *NearestNeighborModel*. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

NearestNeighborModel

- full name: tenpy.models.model.NearestNeighborModel
- parent module: *tenpy.models.model*
- type: class

class tenpy.models.model.**NearestNeighborModel**(*lattice*, *H_bond*)

Bases: *tenpy.models.model.Model*

Base class for a model of nearest neigbor interactions w.r.t. the MPS index.

In this class, the Hamiltonian $H = \sum_i H_{i,i+1}$ is represented by "bond terms" $H_{i,i+1}$ acting only on two neighboring sites *i* and *i+1*, where *i* is an integer. Instances of this class are suitable for *tebd*.

Note that the "nearest-neighbor" in the name referst to the MPS index, not the lattice. In short, this works only for 1-dimensional (1D) nearest-neighbor models: A 2D lattice is internally mapped to a 1D MPS "snake", and even a nearest-neighbor coupling in 2D becomes long-range in the MPS chain.

Parameters

lattice [tenpy.model.lattice.Lattice] The lattice defining the geometry and the local Hilbert space(s).

H_bond [list of {*Array* | None}] The Hamiltonian rewritten as `sum_i H_bond[i]` for MPS indices `i`. `H_bond[i]` acts on sites `(i-1, i)`; we require `len(H_bond) == lat.N_sites`. Legs of each `H_bond[i]` are `['p0', 'p0*', 'p1', 'p1*']`.

Attributes

H_bond [list of {*Array* | None}] The Hamiltonian rewritten as `sum_i H_bond[i]` for MPS indices `i`. `H_bond[i]` acts on sites `(i-1, i)`, `None` represents 0. Legs of each `H_bond[i]` are `['p0', 'p0*', 'p1', 'p1*']`.

Methods

bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

test_sanity

classmethod from_MPOModel(*mpo_model*)

Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with *MPOModel.group_sites()*.

Parameters

mpo_model [*MPOModel*] A model instance implementing the MPO. Does not need to be a *NearestNeighborModel*, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

Module description

This module contains some base classes for models.

A 'model' is supposed to represent a Hamiltonian in a generalized way. The *Lattice* specifies the geometry and underlying Hilbert space, and is thus common to all models. It is needed to intialize the common base class *Model* of all models.

Different algorithms require different representations of the Hamiltonian. For example for DMRG, the Hamiltonian needs to be given as an MPO, while TEBD needs the Hamiltonian to be represented by 'nearest neighbor' bond terms. This module contains the base classes defining these possible representations, namley the *MPOModel* and *NearestNeighborModel*.

A particular model like the `XXZChain` should then yet another class derived from these classes. In it's __init__, it needs to explicitly call the `MPOModel.__init__(self, lattice, H_MPO)`, providing an MPO representation of H, and also the `NearestNeighborModel.__init__(self, lattice, H_bond)`, providing a representation of H by bond terms *H_bond*.

The *CouplingModel* is the attempt to generalize the representation of *H* by explicitly specifying the couplings in a general way, and providing functionality for converting them into *H_MPO* and *H_bond*. This allows to quickly generate new model classes for a very broad class of Hamiltonians.

For simplicity, the *CouplingModel* is limited to interactions involving only two sites. Yet, we also provide the *MultiCouplingModel* to generate Models for Hamiltonians involving couplings between multiple sites.

The *CouplingMPOModel* aims at structuring the initialization for most models and is used as base class in (most of) the predefined models in TeNPy.

See also the introduction in *Introduction to models*.

Specific models

tf_ising	Prototypical example of a quantum model: the transverse field Ising model.
xxz_chain	Prototypical example of a 1D quantum model: the spin-1/2 XXZ chain.
spins	Nearest-neighbour spin-S models.
spins_nnn	Next-Nearest-neighbour spin-S models.
fermions_spinless	Spinless fermions with hopping and interaction.
hubbard	Bosonic and fermionic Hubbard models.
hofstadter	Cold atomic (Harper-)Hofstadter model on a strip or cylinder.
haldane	Bosonic and fermionic Haldane models.
toric_code	Kitaev's exactly solvable toric code model.

tf_ising

- full name: tenpy.models.tf_ising
- parent module: *tenpy.models*
- type: module

Classes

TFIChain(model_params)	The *TFIModel* on a Chain, suitable for TEBD.
TFIModel(model_params)	Transverse field Ising model on a general lattice.

TFIChain

- full name: tenpy.models.tf_ising.TFIChain
- parent module: *tenpy.models.tf_ising*
- type: class

class tenpy.models.tf_ising.**TFIChain**(*model_params*)

Bases: *tenpy.models.tf_ising.TFIModel*, *tenpy.models.model.NearestNeighborModel*

The *TFIModel* on a Chain, suitable for TEBD.

See the *TFIModel* for the documentation of parameters.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,...,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1`

`:= lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)

Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [`MPOModel`] A model instance implementing the MPO. Does not need to be a `NearestNeighborModel`, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve*

model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

TFIModel

- full name: tenpy.models.tf_ising.TFIModel
- parent module: *tenpy.models.tf_ising*
- type: class

class tenpy.models.tf_ising.**TFIModel**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Transverse field Ising model on a general lattice.

The Hamiltonian reads:

$$H = -\sum_{\langle i,j\rangle, i<j} \mathtt{J}\sigma_i^x\sigma_j^x - \sum_i \mathtt{g}\sigma_i^z$$

Here, $\langle i, j\rangle, i < j$ denotes nearest neighbor pairs, each pair appearing exactly once. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

conserve [None | 'parity'] What should be conserved. See `SpinHalfSite`.

J, g [float | array] Couplings as defined for the Hamiltonian above.

lattice [str | *Lattice*] Instance of a lattice class for the underlaying geometry. Alternatively a string being the name of one of the Lattices defined in *lattice*, e.g. `"Chain", "Square", "HoneyComb", ...`.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)
Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)

Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

Module description

Prototypical example of a quantum model: the transverse field Ising model.

Like the *XXZChain*, the transverse field ising chain *TFIChain* is contained in the more general *SpinChain*; the idea is more to serve as a pedagogical example for a 'model'.

We choose the field along z to allow to conserve the parity, if desired.

xxz_chain

- full name: tenpy.models.xxz_chain
- parent module: *tenpy.models*
- type: module

Classes

XXZChain(model_params)	Spin-1/2 XXZ chain with Sz conservation.
XXZChain2(model_params)	Another implementation of the Spin-1/2 XXZ chain with Sz conservation.

XXZChain

- full name: tenpy.models.xxz_chain.XXZChain
- parent module: *tenpy.models.xxz_chain*
- type: class

class tenpy.models.xxz_chain.**XXZChain**(*model_params*)

Bases: *tenpy.models.model.CouplingModel*, *tenpy.models.model.NearestNeighborModel*, *tenpy.models.model.MPOModel*

Spin-1/2 XXZ chain with Sz conservation.

The Hamiltonian reads:

$$H = \sum_i \mathtt{Jxx}/2(S_i^+ S_{i+1}^- + S_i^- S_{i+1}^+) + \mathtt{Jz} S_i^z S_{i+1}^z - \sum_i \mathtt{hz} S_i^z$$

All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

L [int] Length of the chain.

Jxx, Jz, hz [float | array] Couplings as defined for the Hamiltonian above.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions. Coupling boundary conditions are chosen appropriately.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)

Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)
Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [`MPOModel`] A model instance implementing the MPO. Does not need to be a `NearestNeighborModel`, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

XXZChain2

- full name: tenpy.models.xxz_chain.XXZChain2
- parent module: *tenpy.models.xxz_chain*
- type: class

class tenpy.models.xxz_chain.**XXZChain2**(*model_params*)

Bases: *tenpy.models.model.CouplingMPOModel*, *tenpy.models.model.NearestNeighborModel*

Another implementation of the Spin-1/2 XXZ chain with Sz conservation.

This implementation takes the same parameters as the *XXZChain*, but is implemented based on the *CouplingMPOModel*.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first

on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S^z_i S^z_j$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)

Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in `add_coupling()`.

Parameters

strength [scalar | array] The strength to be used in `add_coupling()`, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in `add_coupling()`.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in `add_coupling()` with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the `FermionSite`. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)

Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [`MPOModel`] A model instance implementing the MPO. Does not need to be a `NearestNeighborModel`, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

Module description

Prototypical example of a 1D quantum model: the spin-1/2 XXZ chain.

The XXZ chain is contained in the more general *SpinChain*; the idea of this module is more to serve as a pedagogical example for a model.

spins

- full name: tenpy.models.spins
- parent module: *tenpy.models*
- type: module

Classes

SpinChain(model_params)	The *SpinModel* on a Chain, suitable for TEBD.
SpinModel(model_params)	Spin-S sites coupled by nearest neighbour interactions.

SpinChain

- full name: tenpy.models.spins.SpinChain
- parent module: *tenpy.models.spins*
- type: class

class tenpy.models.spins.**SpinChain**(*model_params*)

Bases: *tenpy.models.spins.SpinModel*, *tenpy.models.model.NearestNeighborModel*

The *SpinModel* on a Chain, suitable for TEBD.

See the *SpinModel* for the documentation of parameters.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

Continued on next page

Table 98 – continued from previous page

trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)

Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [`MPOModel`] A model instance implementing the MPO. Does not need to be a `NearestNeighborModel`, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_sites(*self*, *model_params*)
Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

SpinModel

- full name: tenpy.models.spins.SpinModel
- parent module: *tenpy.models.spins*
- type: class

class tenpy.models.spins.**SpinModel**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Spin-S sites coupled by nearest neighbour interactions.

The Hamiltonian reads:

$$H = \sum_{\langle i,j\rangle, i<j} (\mathtt{Jx} S_i^x S_j^x + \mathtt{Jy} S_i^y S_j^y + \mathtt{Jz} S_i^z S_j^z + \mathtt{muJ} i/2 (S_i^- S_j^+ - S_i^+ S_j^-)) \\ - \sum_i (\mathtt{hx} S_i^x + \mathtt{hy} S_i^y + \mathtt{hz} S_i^z) \\ + \sum_i (\mathtt{D} (S_i^z)^2 + \mathtt{E} ((S_i^x)^2 - (S_i^y)^2))$$

Here, $\langle i, j\rangle, i < j$ denotes nearest neighbor pairs. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

S [{0.5, 1, 1.5, 2, ...}] The 2S+1 local states range from m = -S, -S+1, ... +S.

conserve ['best' | 'Sz' | 'parity' | None] What should be conserved. See `SpinSite`. For `'best'`, we check the parameters what can be preserved.

Jx, Jy, Jz, hx, hy, hz, muJ, D, E: float | array Couplings as defined for the Hamiltonian above.

lattice [str | *Lattice*] Instance of a lattice class for the underlaying geometry. Alternatively a string being the name of one of the Lattices defined in *lattice*, e.g. `"Chain", "Square", "HoneyComb", ...`.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)

Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

Module description

Nearest-neighbour spin-S models.

Uniform lattice of spin-S sites, coupled by nearest-neighbour interactions.

spins_nnn

- full name: tenpy.models.spins_nnn
- parent module: *tenpy.models*
- type: module

Classes

SpinChainNNN(model_params)	Spin-S sites coupled by (next-)nearest neighbour interactions on a *GroupedSite*.
SpinChainNNN2(model_params)	Spin-S sites coupled by next-nearest neighbour interactions.

SpinChainNNN

- full name: tenpy.models.spins_nnn.SpinChainNNN
- parent module: *tenpy.models.spins_nnn*
- type: class

class tenpy.models.spins_nnn.**SpinChainNNN**(*model_params*)

Bases: *tenpy.models.model.CouplingMPOModel*, *tenpy.models.model.NearestNeighborModel*

Spin-S sites coupled by (next-)nearest neighbour interactions on a *GroupedSite*.

The Hamiltonian reads:

$$H = \sum_{\langle i,j\rangle, i<j} \mathtt{Jx} S_i^x S_j^x + \mathtt{Jy} S_i^y S_j^y + \mathtt{Jz} S_i^z S_j^z + \sum_{\langle\langle i,j\rangle\rangle, i<j} \mathtt{Jxp} S_i^x S_j^x + \mathtt{Jyp} S_i^y S_j^y + \mathtt{Jzp} S_i^z S_j^z - \sum_i \mathtt{hx} S_i^x + \mathtt{hy} S_i^y + \mathtt{hz} S_i^z$$

Here, $\langle i,j\rangle, i<j$ denotes nearest neighbors and $\langle\langle i,j\rangle\rangle, i<j$ denotes next nearest neighbors. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

L [int] Length of the chain in terms of *GroupedSite*, i.e. we have `2*L` spin sites.

S [{0.5, 1, 1.5, 2, ...}] The 2S+1 local states range from m = -S, -S+1, ... +S.

conserve ['best' | 'Sz' | 'parity' | None] What should be conserved. See `SpinSite`.

Jx, Jy, Jz, Jxp, Jyp, Jzp, hx, hy, hz [float | array] Couplings as defined for the Hamiltonian above.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions. Coupling boundary conditions are chosen appropriately.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.

Continued on next page

Table 101 – continued from previous page

all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ...,`

`x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)

Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [`MPOModel`] A model instance implementing the MPO. Does not need to be a `NearestNeighborModel`, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

SpinChainNNN2

- full name: tenpy.models.spins_nnn.SpinChainNNN2
- parent module: *tenpy.models.spins_nnn*
- type: class

class tenpy.models.spins_nnn.**SpinChainNNN2**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Spin-S sites coupled by next-nearest neighbour interactions.

The Hamiltonian reads:

$$H = \sum_{\langle i,j\rangle, i<j} \mathtt{Jx} S_i^x S_j^x + \mathtt{Jy} S_i^y S_j^y + \mathtt{Jz} S_i^z S_j^z + \sum_{\langle\langle i,j\rangle\rangle, i<j} \mathtt{Jxp} S_i^x S_j^x + \mathtt{Jyp} S_i^y S_j^y + \mathtt{Jzp} S_i^z S_j^z - \sum_i \mathtt{hx} S_i^x + \mathtt{hy} S_i^y + \mathtt{hz} S_i^z$$

Here, $\langle i,j\rangle, i<j$ denotes nearest neighbors and $\langle\langle i,j\rangle\rangle, i<j$ denotes next nearest neighbors. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

S [{0.5, 1, 1.5, 2, ...}] The 2S+1 local states range from m = -S, -S+1, ... +S.

conserve ['best' | 'Sz' | 'parity' | None] What should be conserved. See `SpinSite`. For `'best'`, we check the parameters what can be preserved.

Jx, Jy, Jz, Jxp, Jyp, Jzp, hx, hy, hz [float | array] Couplings as defined for the Hamiltonian above.

lattice [str | *Lattice*] Instance of a lattice class for the underlaying geometry. Alternatively a string being the name of one of the Lattices defined in *lattice*, e.g. `"Chain"`, `"Square"`, `"HoneyComb"`, ...

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)
Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ...,`

`x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)

Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

Module description

Next-Nearest-neighbour spin-S models.

Uniform lattice of spin-S sites, coupled by next-nearest-neighbour interactions. We have two variants implementing the same hamiltonian. The *SpinChainNNN* uses the *GroupedSite* to keep it a *NearestNeighborModel* suitable for TEBD, while the *SpinChainNNN2* just involves longer-range couplings in the MPO. The latter is preferable for pure DMRG calculations and avoids having to add each of the short range couplings twice for the grouped sites.

Note that you can also get a *NearestNeighborModel* for TEBD from the latter by using *group_sites()* and `from_MPOModel()`. An example for such a case is given in the file `examples/c_tebd.py`.

fermions_spinless

- full name: tenpy.models.fermions_spinless
- parent module: *tenpy.models*
- type: module

Classes

FermionChain(model_params)	The *FermionModel* on a Chain, suitable for TEBD.
FermionModel(model_params)	Spinless fermions with particle number conservation.

FermionChain

- full name: tenpy.models.fermions_spinless.FermionChain
- parent module: *tenpy.models.fermions_spinless*
- type: class

class tenpy.models.fermions_spinless.**FermionChain**(*model_params*)

Bases: *tenpy.models.fermions_spinless.FermionModel*, *tenpy.models.model.NearestNeighborModel*

The *FermionModel* on a Chain, suitable for TEBD.

See the *FermionModel* for the documentation of parameters.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.

Continued on next page

Table 104 – continued from previous page

calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should

be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S^z_i S^z_j$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)
Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [MPOModel] A model instance implementing the MPO. Does not need to be a NearestNeighborModel, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_sites(*self*, *model_params*)
Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

FermionModel

- full name: tenpy.models.fermions_spinless.FermionModel
- parent module: *tenpy.models.fermions_spinless*
- type: class

class tenpy.models.fermions_spinless.**FermionModel**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Spinless fermions with particle number conservation.

The Hamiltonian reads:

$$H = \sum_{\langle i,j \rangle, i<j} -\mathtt{J}(c_i^\dagger c_j + c_j^\dagger c_i) + \mathtt{V} n_i n_j - \sum_i \mathtt{mu} n_i$$

Here, $\langle i, j \rangle, i < j$ denotes nearest neighbor pairs. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Warning: Using the Jordan-Wigner string (`JW`) is crucial to get correct results! See *Fermions and the Jordan-Wigner transformation* for details.

Parameters

conserve ['best' | 'N' | 'parity' | None] What should be conserved. See `FermionSite`. For `'best'`, we check the parameters what can be preserved.

J, V, mu [float | array] Hopping, interaction and chemical potential as defined for the Hamiltonian above.

lattice [str | *Lattice*] Instance of a lattice class for the underlaying geometry. Alternatively a string being the name of one of the Lattices defined in *lattice*, e.g. `"Chain"`, `"Square"`, `"HoneyComb"`, ...

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)
Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ...,`

`x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)

Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

Module description

Spinless fermions with hopping and interaction.

Todo: -add further terms (e.g. c^dagger c^dagger + h.c.) to the Hamiltonian.

hubbard

- full name: tenpy.models.hubbard
- parent module: *tenpy.models*
- type: module

Classes

BoseHubbardChain(model_params)	The *BoseHubbardModel* on a Chain, suitable for TEBD.
BoseHubbardModel(model_params)	Spinless Bose-Hubbard model.
FermiHubbardChain(model_params)	The *FermiHubbardModel* on a Chain, suitable for TEBD.
FermiHubbardModel(model_params)	Spin-1/2 Fermi-Hubbard model.

BoseHubbardChain

- full name: tenpy.models.hubbard.BoseHubbardChain
- parent module: *tenpy.models.hubbard*
- type: class

class tenpy.models.hubbard.**BoseHubbardChain**(*model_params*)

Bases: *tenpy.models.hubbard.BoseHubbardModel*, *tenpy.models.model.NearestNeighborModel*

The *BoseHubbardModel* on a Chain, suitable for TEBD.

See the *BoseHubbardModel* for the documentation of parameters.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.

Continued on next page

Table 107 – continued from previous page

calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should

be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S^z_i S^z_j$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)
Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [MPOModel] A model instance implementing the MPO. Does not need to be a NearestNeighborModel, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel(*self*)

Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

BoseHubbardModel

- full name: tenpy.models.hubbard.BoseHubbardModel
- parent module: *tenpy.models.hubbard*
- type: class

class tenpy.models.hubbard.**BoseHubbardModel**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Spinless Bose-Hubbard model.

The Hamiltonian is:

$$H = -t \sum_{\langle i,j \rangle, i<j} (b_i^\dagger b_j + b_j^\dagger b_i) + V \sum_{\langle i,j \rangle, i<j} n_i n_j + \frac{U}{2} \sum_i n_i(n_i - 1) - \mu \sum_i n_i$$

Here, $\langle i, j \rangle, i < j$ denotes nearest neighbor pairs. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

n_max [int] Maximum number of bosons per site.

filling [float] Average filling.

conserve: {'best' | 'N' | 'parity' | None} What should be conserved. See `BosonSite`.

t, U, V, mu [float | array] Couplings as defined in the Hamiltonian above. Note the signs!

lattice [str | *Lattice*] Instance of a lattice class for the underlaying geometry. Alternatively a string being the name of one of the Lattices defined in *lattice*, e.g. `"Chain", "Square", "HoneyComb", ...`.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.

Continued on next page

Table 108 – continued from previous page

calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in `add_coupling()`.

Parameters

strength [scalar | array] The strength to be used in `add_coupling()`, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in `add_coupling()`.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in `add_coupling()` with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the `FermionSite`. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* in place to group sites.

Group each *n* sites together using the `GroupedSite`. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of `GroupedSite`] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

FermiHubbardChain

- full name: tenpy.models.hubbard.FermiHubbardChain
- parent module: *tenpy.models.hubbard*
- type: class

class tenpy.models.hubbard.**FermiHubbardChain**(*model_params*)
Bases: *tenpy.models.hubbard.FermiHubbardModel*, *tenpy.models.model.NearestNeighborModel*

The *FermiHubbardModel* on a Chain, suitable for TEBD.

See the *FermiHubbardModel* for the documentation of parameters.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
bond_energies(self, psi)	Calculate bond energies <psi|H_bond|psi>.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_MPO_from_bond(self[, tol_zero])	Calculate the MPO Hamiltonian from the bond Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
from_MPOModel(mpo_model)	Initialize a NearestNeighborModel from a model class defining an MPO.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
trivial_like_NNModel(self)	Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

bond_energies(*self*, *psi*)
Calculate bond energies <psi|H_bond|psi>.

Parameters

psi [*MPS*] The MPS for which the bond energies should be calculated.

Returns

E_bond [1D ndarray] List of bond energies: for finite bc, `E_Bond[i]` is the energy of bond `i, i+1`. (i.e. we omit bond 0 between sites L-1 and 0); for infinite bc `E_bond[i]` is the energy of bond `i-1, i`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_MPO_from_bond(*self*, *tol_zero=1e-15*)
Calculate the MPO Hamiltonian from the bond Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

classmethod from_MPOModel(*mpo_model*)

Initialize a NearestNeighborModel from a model class defining an MPO.

This is especially usefull in combination with `MPOModel.group_sites()`.

Parameters

mpo_model [`MPOModel`] A model instance implementing the MPO. Does not need to be a `NearestNeighborModel`, but should only have nearest-neighbor couplings.

Examples

The *SpinChainNNN2* has next-nearest-neighbor couplings and thus only implements an MPO:

```
>>> from tenpy.models.spins_nnn import SpinChainNNN2
>>> nnn_chain = SpinChainNNN2({'L': 20})
parameter 'L'=20 for SpinChainNNN2
>>> print(isinstance(nnn_chain, NearestNeighborModel))
False
>>> print("range before grouping:", nnn_chain.H_MPO.max_range)
range before grouping: 2
```

By grouping each two neighboring sites, we can bring it down to nearest neighbors.

```
>>> nnn_chain.group_sites(2)
>>> print("range after grouping:", nnn_chain.H_MPO.max_range)
range after grouping: 1
```

Yet, TEBD will not yet work, as the model doesn't define *H_bond*. However, we can initialize a NearestNeighborModel from the MPO:

```
>>> nnn_chain_for_tebd = NearestNeighborModel.from_MPOModel(nnn_chain)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_sites(*self*, *model_params*)
Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms (*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

test_sanity (*self*)
Sanity check, raises ValueErrors, if something is wrong.

trivial_like_NNModel (*self*)
Return a NearestNeighborModel with same lattice, but trivial (H=0) bonds.

FermiHubbardModel

- full name: tenpy.models.hubbard.FermiHubbardModel
- parent module: *tenpy.models.hubbard*
- type: class

class tenpy.models.hubbard.**FermiHubbardModel** (*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Spin-1/2 Fermi-Hubbard model.

The Hamiltonian reads:

$$H = -\sum_{\langle i,j\rangle, i<j,\sigma} t(c^\dagger_{\sigma,i}c_{\sigma j} + h.c.) + \sum_i U n_{\uparrow,i}n_{\downarrow,i} - \sum_i \mu(n_{\uparrow,i}+n_{\downarrow,i}) + \sum_{\langle i,j\rangle, i<j,\sigma} V(n_{\uparrow,i}+n_{\downarrow,i})(n_{\uparrow,j}+n_{\downarrow,j})$$

Here, $\langle i,j\rangle, i < j$ denotes nearest neighbor pairs. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Warning: Using the Jordan-Wigner string (JW) is crucial to get correct results! See *Fermions and the Jordan-Wigner transformation* for details.

Parameters

cons_N [{'N' | 'parity' | None}] Whether particle number is conserved, see *SpinHalfFermionSite* for details.

cons_Sz [{'Sz' | 'parity' | None}] Whether spin is conserved, see *SpinHalfFermionSite* for details.

t, U, mu [float | array] Parameters as defined for the Hamiltonian above. Note the signs!

lattice [str | *Lattice*] Instance of a lattice class for the underlaying geometry. Alternatively a string being the name of one of the Lattices defined in *lattice*, e.g. "Chain", "Square", "HoneyComb", ...

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

Module description

Bosonic and fermionic Hubbard models.

hofstadter

- full name: tenpy.models.hofstadter
- parent module: *tenpy.models*
- type: module

Classes

HofstadterBosons(model_params)	Bosons on a square lattice with magnetic flux.
HofstadterFermions(model_params)	Fermions on a square lattice with magnetic flux.

HofstadterBosons

- full name: tenpy.models.hofstadter.HofstadterBosons
- parent module: *tenpy.models.hofstadter*
- type: class

class tenpy.models.hofstadter.**HofstadterBosons**(*model_params*)

Bases: *tenpy.models.model.CouplingModel*, *tenpy.models.model.MPOModel*

Bosons on a square lattice with magnetic flux.

For now, the Hamiltonian reads:

$$\begin{aligned} H = &-\sum_{x,y} \mathtt{Jx}(e^{i\mathtt{phi}_{x,y}} a^\dagger_{x+1,y} a_{x,y} + h.c.) \\ &-\sum_{x,y} \mathtt{Jy}(e^{i\mathtt{phi}_{x,y}} a^\dagger_{x,y+1} a_{x,y} + h.c.) \\ &+\sum_{x,y} \frac{\mathtt{U}}{2} n_{x,y}(n_{x,y}-1) - \mathtt{mu} n_{x,y} \end{aligned}$$

where $e^{i\mathtt{phi}_{x,y}}$ is a complex Aharonov-Bohm hopping phase, depending on lattice coordinates and gauge choice (see *tenpy.models.hofstadter.gauge_hopping()*).

All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

Lx, Ly [int] Size of the simulation unit cell in terms of lattice sites.

mx, my [int] Size of the magnetic unit cell in terms of lattice sites.

Nmax [int] Maximum number of bosons per site.

filling [tuple] Average number of fermions per site, defined as a fraction (numerator, denominator) Changes the definition of `'dN'` in the *BosonSite*.

Jx, Jy, mu, U: float Hamiltonian parameters as defined above.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly.

bc_x ['periodic' | 'open'] Boundary conditions in x-direction

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction.

conserve [{'N' | 'parity' | None}] What quantum number to conserve.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*

phi [tuple] Magnetic flux density, defined as a fraction (numerator, denominator)

phi_ext [float] External magnetic flux 'threaded' through the cylinder.

gauge ['landau_x' | 'landau_y' | 'symmetric'] Choice of the gauge used for the magnetic field. This changes the magnetic unit cell.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_lattice	
init_sites	
init_terms	

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,...,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in `add_coupling()`.

Parameters

strength [scalar | array] The strength to be used in `add_coupling()`, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in `add_coupling()`.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in `add_coupling()` with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the `FermionSite`. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the `GroupedSite`. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of `GroupedSite`] The sites grouped together.

Returns

grouped_sites [list of `GroupedSite`] The sites grouped together.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

HofstadterFermions

- full name: tenpy.models.hofstadter.HofstadterFermions
- parent module: *tenpy.models.hofstadter*
- type: class

class tenpy.models.hofstadter.**HofstadterFermions**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Fermions on a square lattice with magnetic flux.

For now, the Hamiltonian reads:

$$H = -\sum_{x,y} \mathtt{Jx}(e^{i\mathtt{phi}_{x,y}} c^\dagger_{x,y} c_{x+1,y} + h.c.)$$
$$-\sum_{x,y} \mathtt{Jy}(e^{i\mathtt{phi}_{x,y}} c^\dagger_{x,y} c_{x,y+1} + h.c.) + \sum_{x,y} \mathtt{v}(n_{x,y} n_{x,y+1} + n_{x,y} n_{x+1,y} - \sum_{x,y} \mathtt{mu} n_{x,y},$$

where $e^{i\mathtt{phi}_{x,y}}$ is a complex Aharonov-Bohm hopping phase, depending on lattice coordinates and gauge choice (see *tenpy.models.hofstadter.gauge_hopping()*).

All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

Lx, Ly [int] Size of the simulation unit cell in terms of lattice sites.

mx, my [int] Size of the magnetic unit cell in terms of lattice sites.

filling [tuple] Average number of fermions per site, defined as a fraction (numerator, denominator) Changes the definition of `'dN'` in the *FermionSite*.

Jx, Jy, mu, v: float Hamiltonian parameters as defined above.

bc_MPS [{'finite' | 'infinite'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly.

bc_x ['periodic' | 'open'] Lattice boundary conditions in x-direction

bc_y ['ladder' | 'cylinder'] Lattice boundary conditions in y-direction.

conserve [{'N' | 'parity' | None}] What quantum number to conserve.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*.

phi [tuple] Magnetic flux density, defined as a fraction (numerator, denominator)

phi_ext [float] External magnetic flux 'threaded' through the cylinder.

gauge ['landau_x' | 'landau_y' | 'symmetric'] Choice of the gauge used for the magnetic field. This changes the magnetic unit cell. See *gauge_hopping()* for details.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in `add_coupling()`.

Parameters

strength [scalar | array] The strength to be used in `add_coupling()`, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in `add_coupling()`.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in `add_coupling()` with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the `FermionSite`. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the `GroupedSite`. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of `GroupedSite`] The sites grouped together.

Returns

grouped_sites [list of `GroupedSite`] The sites grouped together.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

Functions

gauge_hopping(model_params)	Compute hopping amplitudes for the Hofstadter models based on a gauge choice.

gauge_hopping

- full name: tenpy.models.hofstadter.gauge_hopping
- parent module: *tenpy.models.hofstadter*
- type: function

tenpy.models.hofstadter.**gauge_hopping**(*model_params*)

Compute hopping amplitudes for the Hofstadter models based on a gauge choice.

In the Hofstadter model, the magnetic field enters as an Aharonov-Bohm phase. This phase is dependent on a choice of gauge, which simultaneously defines a 'magnetic unit cell' (MUC).

The magnetic unit cell is the smallest set of lattice plaquettes that encloses an integer number of flux quanta. It can be user-defined by setting mx and my, but for common gauge choices is computed based on the flux density.

The gauge choices are:

- 'landau_x': Landau gauge along the x-axis. The magnetic unit cell will have shape :math`(mathtt{mx}, 1)`. For flux densities p/q, *mx* will default to q. Example: at a flux density $1/3$, the magnetic unit cell will have shape $(3, 1)$, so it encloses exactly 1 flux quantum.
- 'landau_y': Landau gauge along the y-axis. The magnetic unit cell will have shape :math`(1, mathtt{my})`. For flux densities :math`p/q`, *my* will default to q. Example: at a flux density $3/7$, the magnetic unit cell will have shape $(1, 7)$, so it encloses axactly 3 flux quanta.
- 'symmetric': symmetric gauge. The magnetic unit cell will have shape $(\mathtt{mx}, \mathtt{my})$, with $mx = my$. For flux densities p/q, *mx* and *my* will default to q Example: at a flux density 4/9, the magnetic unit cell will have shape (9,9).

Parameters

gauge ['landau_x' | 'landau_y' | 'symmetric'] Choice of the gauge, see table above.

mx, my [int | None] Dimensions of the magnetic unit cell in terms of lattice sites. `None` defaults to the minimal choice compatible with *gauge* and *phi_pq*.

Jx, Jy: float 'Bare' hopping amplitudes (without phase). Without any flux we have `hop_x = -Jx` and `hop_y = -Jy`.

phi_pq [tuple (int, int)] Magnetic flux as a fraction p/q, defined as (p, q)

Returns

hop_x, hop_y [float | array] Hopping amplitudes to be used as prefactors for $c^\dagger_{x,y}c_{x+1,y}$ (*hop_x*) and $c^\dagger_{x,y}c_{x,y+1}$ (*hop_x*), respectively, with the necessary phases for the gauge.

Module description

Cold atomic (Harper-)Hofstadter model on a strip or cylinder.

Todo: WARNING: These models are still under development and not yet tested for correctness. Use at your own risk! Replicate known results to confirm models work correctly. Long term: implement different lattices. Long term: implement variable hopping strengths Jx, Jy.

haldane

- full name: tenpy.models.haldane
- parent module: *tenpy.models*
- type: module

Classes

BosonicHaldaneModel(model_params)	Hardcore bosonic Haldane model.
FermionicHaldaneModel(model_params)	Spinless fermionic Haldane model.

BosonicHaldaneModel

- full name: tenpy.models.haldane.BosonicHaldaneModel
- parent module: *tenpy.models.haldane*
- type: class

class tenpy.models.haldane.**BosonicHaldaneModel**(*model_params*)

Bases: *tenpy.models.model.CouplingMPOModel*

Hardcore bosonic Haldane model.

The Hamiltonian reads:

$$H = \sum_{ij} t_{ij} b_i^\dagger b_j + \sum_i \mu (n_{A,i} - n_{B,i}) + V \sum_{\langle ij \rangle, i<j} n_{A,i} n_{B,j}$$

Here, $\langle i, j \rangle, i < j$ denotes nearest neighbor pairs and n_A, n_B are the number operators on the A and B sites. Hopping is allowed to nearest and next-nearest neighbor sites with amplitudes $t_{\langle ij \rangle} = t_1 \in \mathbb{R}$ and $t_{\langle\langle ij \rangle\rangle} = t_2 e^{\pm i\phi} \in \mathbb{C}$ respectively, where $\pm\phi$ is the phase acquired by a boson hopping between atoms in the same sublattice with a sign given by the direction of the hopping. This Hamiltonian is translated from [Grushin2015]. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

conserve ['best' | 'N' | 'parity' | None] What should be conserved. See `BosonSite`. For `'best'`, we check the parameters that can be preserved.

t1, t2, V, mu [float | array] Hopping, interaction and chemical potential as defined for the Hamiltonian above. The default value for t2 is chosen to achieve the optimal band flatness ratio.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite"`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

FermionicHaldaneModel

- full name: tenpy.models.haldane.FermionicHaldaneModel
- parent module: *tenpy.models.haldane*
- type: class

class tenpy.models.haldane.**FermionicHaldaneModel**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*

Spinless fermionic Haldane model.

The Hamiltonian reads:

$$H = \sum_{ij} t_{ij} c_i^\dagger c_j + \sum_i \mu (n_{A,i} - n_{B,i}) + V \sum_{\langle ij \rangle, i<j} n_{A,i} n_{B,j}$$

Here, $\langle i, j \rangle, i < j$ denotes nearest neighbor pairs and n_A, n_B are the number operators on the A and B sites. Hopping is allowed to nearest and next-nearest neighbor sites with amplitudes $t_{\langle ij \rangle} = t_1 \in \mathbb{R}$ and $t_{\langle\langle ij \rangle\rangle} = t_2 e^{\pm i\phi} \in \mathbb{C}$ respectively, where $\pm\phi$ is the phase acquired by an electron hopping between atoms in the same sublattice with a sign given by the direction of the hopping. This Hamiltonian is described in [Grushin2015]. All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Warning: Using the Jordan-Wigner string (`JW`) is crucial to get correct results! See *Fermions and the Jordan-Wigner transformation* for details.

Parameters

conserve ['best' | 'N' | 'parity' | None] What should be conserved. See `FermionSite`. For `'best'`, we check the parameters what can be preserved.

t1, t2, V, mu [float | array] Hopping, interaction and chemical potential as defined for the Hamiltonian above. The default value for t2 is chosen to achieve the optimal band flatness ratio.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions along the x-direction. For 'infinite' boundary conditions, repeat the unit cell in x-direction. Coupling boundary conditions in x-direction are chosen accordingly. Only used if *lattice* is a string.

order [string] Ordering of the sites in the MPS, e.g. 'default', 'snake'; see *ordering()*. Only used if *lattice* is a string.

L [int] Lenght of the lattice. Only used if *lattice* is the name of a 1D Lattice.

Lx, Ly [int] Length of the lattice in x- and y-direction. Only used if *lattice* is the name of a 2D Lattice.

bc_y ['ladder' | 'cylinder'] Boundary conditions in y-direction. Only used if *lattice* is the name of a 2D Lattice.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_terms(*self*, *model_params*)

Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)

Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (*FermionSite*), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)

Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)
Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

init_lattice(*self*, *model_params*)

Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

Module description

Bosonic and fermionic Haldane models.

toric_code

- full name: tenpy.models.toric_code
- parent module: *tenpy.models*
- type: module

Classes

DualSquare(Lx, Ly, sites, **kwargs)	The dual lattice of the square lattice (again square).
ToricCode(model_params)	Toric code model.

DualSquare

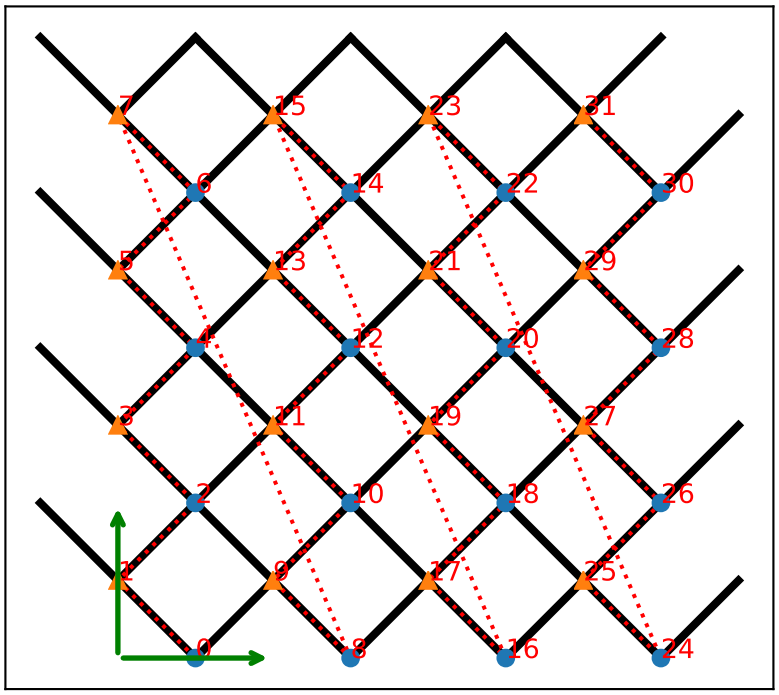
- full name: tenpy.models.toric_code.DualSquare
- parent module: *tenpy.models.toric_code*
- type: class

class tenpy.models.toric_code.**DualSquare**(*Lx*, *Ly*, *sites*, ***kwargs*)

Bases: *tenpy.models.lattice.Lattice*

The dual lattice of the square lattice (again square).

The sites in this lattice correspond to the vertical and horizontal (nearest neighbor) bonds of a common *Square* lattice with the same dimensions *Lx, Ly*.



Parameters

Lx, Ly [int] Dimensions of the original lattice. This lattice has *2*Lx*Ly* sites.

sites [*Site*] The sites for the horizontal (first entry) and vertical (second entry) bonds.

****kwargs :** Additional keyword arguments given to the `Lattice`. *basis*, *pos* and *[[next_]next_]nearest_neighbors* are set accordingly.

Attributes

dim The dimension of the lattice.

nearest_neighbors

next_nearest_neighbors

next_next_nearest_neighbors

order Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

Methods

count_neighbors(self[, u, key])	Count e.g.
coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a coupling.
lat2mps_idx(self, lat_idx)	Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.
mps2lat_idx(self, i)	Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.
mps2lat_values(self, A[, axes, u])	Reshape/reorder *A* to replace an MPS index by lattice indices.
mps_idx_fix_u(self[, u])	return an index array of MPS indices for which the site within the unit cell is *u*.
mps_lat_idx_fix_u(self[, u])	Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.
mps_sites(self)	Return a list of sites for all MPS indices.
multi_coupling_shape(self, dx)	Calculate correct shape of the *strengths* for a multi_coupling.
number_nearest_neighbors(self[, u])	Deprecated.
number_next_nearest_neighbors(self[, u])	Deprecated.
ordering(self, order)	Provide possible orderings of the *N* lattice sites.
plot_basis(self, ax, **kwargs)	Plot arrows indicating the basis vectors of the lattice.
plot_bc_identified(self, ax[, direction, shift])	Mark two sites indified by periodic boundary conditions.
plot_coupling(self, ax[, coupling])	Plot lines connecting nearest neighbors of the lattice.
plot_order(self, ax[, order, textkwargs])	Plot a line connecting sites in the specified "order" and text labels enumerating them.
plot_sites(self, ax[, markers])	Plot the sites of the lattice with markers.
position(self, lat_idx)	return 'space' position of one or multiple sites.
possible_couplings(self, u1, u2, dx)	Find possible MPS indices for two-site couplings.
possible_multi_couplings(self, u0, other_us, dx)	Generalization of *possible_couplings()* to couplings with more than 2 sites.
site(self, i)	return *Site* instance corresponding to an MPS index *i*
test_sanity(self)	Sanity check.

count_neighbors(*self*, *u=0*, *key='nearest_neighbors'*)

Count e.g. the number of nearest neighbors for a site in the bulk.

Parameters

u [int] Specifies the site in the unit cell, for which we should count the number of neighbors (or whatever *key* specifies).

key [str] Key of `pairs` to select what to count.

Returns

number [int] Number of nearest neighbors (or whatever *key* specified) for the *u*-th site in the unit cell, somewhere in the bulk of the lattice. Note that it might not be the correct value at the edges of a lattice with open boundary conditions.

coupling_shape(*self*, *dx*)

Calculate correct shape of the *strengths* for a coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

property dim

The dimension of the lattice.

lat2mps_idx(*self*, *lat_idx*)

Translate lattice indices `(x_0, ..., x_{D-1}, u)` to MPS index *i*.

Parameters

lat_idx [array_like [..., dim+1]] The last dimension corresponds to lattice indices `(x_0, ..., x_{D-1}, u)`. All lattice indices should be positive and smaller than the corresponding entry in `self.shape`. Exception: for "infinite" *bc_MPS*, an *x_0* outside indicates shifts accross the boundary.

Returns

i [array_like] MPS index/indices corresponding to *lat_idx*. Has the same shape as *lat_idx* without the last dimension.

mps2lat_idx(*self*, *i*)

Translate MPS index *i* to lattice indices `(x_0, ..., x_{dim-1}, u)`.

Parameters

i [int | array_like of int] MPS index/indices.

Returns

lat_idx [array] First dimensions like *i*, last dimension has len *dim`+1 and contains the lattice indices ``(x_0, ..., x_{dim-1}, u)`* corresponding to *i*. For *i* accross the MPS unit cell and "infinite" *bc_MPS*, we shift *x_0* accordingly.

mps2lat_values(*self*, *A*, *axes=0*, *u=None*)

Reshape/reorder *A* to replace an MPS index by lattice indices.

Parameters

A [ndarray] Some values. Must have `A.shape[axes] = self.N_sites` if *u* is `None`, or `A.shape[axes] = self.N_cells` if *u* is an int.

axes [(iterable of) int] chooses the axis which should be replaced.

u [`None` | int] Optionally choose a subset of MPS indices present in the axes of *A*, namely the indices corresponding to `self.unit_cell[u]`, as returned by *mps_idx_fix_u()*. The resulting array will not have the additional dimension(s) of *u*.

Returns

res_A [ndarray] Reshaped and reordered verions of A. Such that an MPS index *j* is replaced by `res_A[..., self.order, ...] = A[..., np.arange(self.N_sites), ...]`

Examples

Say you measure expection values of an onsite term for an MPS, which gives you an 1D array *A*, where *A[i]* is the expectation value of the site given by `self.mps2lat_idx(i)`. Then this function gives you the expectation values ordered by the lattice:

```
>>> print(lat.shape, A.shape)
(10, 3, 2) (60,)
>>> A_res = lat.mps2lat_values(A)
>>> A_res.shape
(10, 3, 2)
>>> A_res[lat.mps2lat_idx(5)] == A[5]
True
```

If you have a correlation function `C[i, j]`, it gets just slightly more complicated:

```
>>> print(lat.shape, C.shape)
(10, 3, 2) (60, 60)
>>> lat.mps2lat_values(C, axes=[0, 1]).shape
(10, 3, 2, 10, 3, 2)
```

If the unit cell consists of different physical sites, an onsite operator might be defined only on one of the sites in the unit cell. Then you can use *mps_idx_fix_u()* to get the indices of sites it is defined on, measure the operator on these sites, and use the argument *u* of this function.

```
>>> u = 0
>>> idx_subset = lat.mps_idx_fix_u(u)
>>> A_u = A[idx_subset]
>>> A_u_res = lat.mps2lat_values(A_u, u=u)
>>> A_u_res.shape
(10, 3)
>>> np.all(A_res[:, :, u] == A_u_res[:, :])
True
```

Todo: make sure this function is used for expectation values...

mps_idx_fix_u(*self*, *u=None*)
return an index array of MPS indices for which the site within the unit cell is *u*.

If you have multiple sites in your unit-cell, an onsite operator is in general not defined for all sites. This functions returns an index array of the mps indices which belong to sites given by `self.unit_cell[u]`.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices for which `self.site(i) is self.unit_cell[u]`. Ordered ascending.

mps_lat_idx_fix_u(*self*, *u=None*)
Similar as *mps_idx_fix_u()*, but return also the corresponding lattice indices.

Parameters

u [None | int] Selects a site of the unit cell. `None` (default) means all sites.

Returns

mps_idx [array] MPS indices *i* for which `self.site(i) is self.unit_cell[u]`.

lat_idx [2D array] The row *j* contains the lattice index (without *u*) corresponding to `mps_idx[j]`.

mps_sites(*self*)
Return a list of sites for all MPS indices.

Equivalent to `[self.site(i) for i in range(self.N_sites)]`.

This should be used for *sites* of 1D tensor networks (MPS, MPO,...).

multi_coupling_shape(*self*, *dx*)
Calculate correct shape of the *strengths* for a multi_coupling.

Parameters

dx [tuple of int] Translation vector in the lattice for a coupling of two operators.

Returns

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

shift_lat_indices [array] Translation vector from lower left corner of box spanned by *dx* to the origin.

number_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

number_next_nearest_neighbors(*self*, *u=0*)
Deprecated.

Use *count_neighbors()* instead.

property order
Defines an ordering of the lattice sites, thus mapping the lattice to a 1D chain.

This order defines how an MPS/MPO winds through the lattice.

ordering(*self*, *order*)
Provide possible orderings of the *N* lattice sites.

This function can be overwritten by derived lattices to define additional orderings. The following orders are defined in this method:

order	equivalent *priority*	equivalent `snake_winding`
`'Cstyle'`	(0, 1, ..., dim-1, dim)	(False, ..., False, False)
`'default'`		
`'snake'`	(0, 1, ..., dim-1, dim)	(True, ..., True, True)
`'snakeCstyle'`		
`'Fstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)
`'snakeFstyle'`	(dim-1, ..., 1, 0, dim)	(False, ..., False, False)

Parameters

order [str | `('standard', snake_winding, priority)` | `('grouped', groups)`] Specifies the desired ordering using one of the strings of the above tables. Alternatively, an ordering is specified by a tuple with first entry specifying a function, `'standard'` for `get_order()` and `'grouped'` for `get_order_grouped()`, and other arguments in the tuple as specified in the documentation of these functions.

Returns

order [array, shape (N, D+1), dtype np.intp] the order to be used for *order*.

See also:

get_order generates the *order* from equivalent *priority* and *snake_winding*.

get_order_grouped variant of *get_order*.

plot_order visualizes the resulting *order*.

plot_basis(*self*, *ax*, ***kwargs*)
Plot arrows indicating the basis vectors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

****kwargs :** Keyword arguments specifying the "arrowprops" of `ax.annotate`.

plot_bc_identified(*self*, *ax*, *direction=-1*, *shift=None*, ***kwargs*)
Mark two sites indified by periodic boundary conditions.

Works only for lattice with a 2-dimensional basis.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

direction [int] The direction of the lattice along which we should mark the idenitified sites. If `None`, mark it along all directions with periodic boundary conditions.

shift [None | np.ndarray] The origin starting from where we mark the identified sites. Defaults to the first entry of `unit_cell_positions`.

****kwargs :** Keyword arguments for the used `ax.plot`.

plot_coupling(*self*, *ax*, *coupling=None*, ***kwargs*)
Plot lines connecting nearest neighbors of the lattice.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

coupling [list of (u1, u2, dx)] By default (`None`), use `self.pairs['nearest_neighbors']`. Specifies the connections to be plotted; iteating over lattice indices *(i0, i1, ...)*, we plot a connection from the site `(i0, i1, ..., u1)` to the site `(i0+dx[0], i1+dx[1], ..., u2)`, taking into account the boundary conditions.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_order(*self*, *ax*, *order=None*, *textkwargs={}*, ***kwargs*)
Plot a line connecting sites in the specified "order" and text labels enumerating them.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

order [None | 2D array (self.N_sites, self.dim+1)] The order as returned by *`ordering()`*; by default (`None`) use *`order`*.

textkwargs: ``None`` | dict If not `None`, we add text labels enumerating the sites in the plot. The dictionary can contain keyword arguments for `ax.text()`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

plot_sites(*self, ax, markers=['o', '^', 's', 'p', 'h', 'D'], **kwargs*)
Plot the sites of the lattice with markers.

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

markers [list] List of values for the keywork *marker* of `ax.plot()` to distinguish the different sites in the unit cell, a site *u* in the unit cell is plotted with a marker `markers[u % len(markers)]`.

****kwargs :** Further keyword arguments given to `ax.plot()`.

position(*self*, *lat_idx*)
return 'space' position of one or multiple sites.

Parameters

lat_idx [ndarray, `(... , dim+1)`] Lattice indices.

Returns

pos [ndarray, `(..., dim)`] The position of the lattice sites specified by *lat_idx* in real-space.

possible_couplings(*self*, *u1*, *u2*, *dx*)
Find possible MPS indices for two-site couplings.

For periodic boundary conditions (`bc[a] == False`) the index `x_a` is taken modulo `Ls[a]` and runs through `range(Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+dx[a] < lat.Ls[a]`.

Parameters

u1, u2 [int] Indices within the unit cell; the *u1* and *u2* of *`add_coupling()`*

dx [array] Length *`dim`*. The translation in terms of basis vectors for the coupling.

Returns

mps1, mps2 [array] For each possible two-site coupling the MPS indices for the *u1* and *u2*.

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

possible_multi_couplings(*self*, *u0*, *other_us*, *dx*)
Generalization of *possible_couplings()* to couplings with more than 2 sites.

Given the arguments of *add_coupling()* determine the necessary shape of *strength*.

Parameters

u0 [int] Argument *u0* of *add_multi_coupling()*.

other_us [list of int] The *u* of the *other_ops* in *add_multi_coupling()*.

dx [array, shape (len(other_us), lat.dim+1)] The *dx* specifying relative operator positions of the *other_ops* in *add_multi_coupling()*.

Returns

mps_ijkl [2D int array] Each row contains MPS indices *i,j,k,l,...* ` for each of the operators positions. The positions are defined by *dx* (j,k,l,... relative to *i*) and boundary coundary conditions of *self* (how much the *box* for given *dx* can be shifted around without hitting a boundary - these are the different rows).

lat_indices [2D int array] Rows of *lat_indices* correspond to rows of *mps_ijkl* and contain the lattice indices of the "lower left corner" of the box containing the coupling.

coupling_shape [tuple of int] Len *dim*. The correct shape for an array specifying the coupling strength. *lat_indices* has only rows within this shape.

site(*self*, *i*)
return *Site* instance corresponding to an MPS index *i*

test_sanity(*self*)
Sanity check.

Raises ValueErrors, if something is wrong.

ToricCode

- full name: tenpy.models.toric_code.ToricCode
- parent module: *tenpy.models.toric_code*
- type: class

class tenpy.models.toric_code.**ToricCode**(*model_params*)
Bases: *tenpy.models.model.CouplingMPOModel*, *tenpy.models.model.MultiCouplingModel*

Toric code model.

The Hamiltonian reads:

$$H = -\mathtt{Jv} \sum_{vertices v} \prod_{i \in v} \sigma_i^x - \mathtt{Jp} \sum_{plaquettes p} \prod_{i \in p} \sigma_i^z$$

(Note that this are Pauli matrices, not spin-1/2 operators.) All parameters are collected in a single dictionary *model_params* and read out with *get_parameter()*.

Parameters

Lx, Ly [int] Dimension of the lattice, number of plaquettes around the cylinder.

conserve ['parity' | None] What should be conserved. See `SpinHalfSite`.

Jc, Jp: float | array Couplings as defined for the Hamiltonian above.

bc_MPS [{'finite' | 'infinte'}] MPS boundary conditions. Coupling boundary conditions are chosen appropriately.

order [str] The order of the lattice sites in the lattice, see *DualSquare*.

Methods

add_coupling(self, strength, u1, op1, u2, ...)	Add twosite coupling terms to the Hamiltonian, summing over lattice sites.
add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_multi_coupling(self, strength, u0, op0, ...)	Add multi-site coupling terms to the Hamiltonian, summing over lattice sites.
add_multi_coupling_term(self, strength, ...)	Add a general M-site coupling term on given MPS sites.
add_onsite(self, strength, u, opname[, category])	Add onsite terms to `onsite_terms`.
add_onsite_term(self, strength, i, op[, ...])	Add an onsite term on a given MPS site.
all_coupling_terms(self)	Sum of all `coupling_terms`.
all_onsite_terms(self)	Sum of all `onsite_terms`.
calc_H_MPO(self[, tol_zero])	Calculate MPO representation of the Hamiltonian.
calc_H_bond(self[, tol_zero])	calculate *H_bond* from `coupling_terms` and `onsite_terms`.
calc_H_bond_from_MPO(self[, tol_zero])	Calculate the bond Hamiltonian from the MPO Hamiltonian.
calc_H_onsite(self[, tol_zero])	Calculate *H_onsite* from *self.onsite_terms*.
coupling_strength_add_ext_flux(self, ...)	Add an external flux to the coupling strength.
group_sites(self[, n, grouped_sites])	Modify *self* in place to group sites.
init_lattice(self, model_params)	Initialize a lattice for the given model parameters.
init_sites(self, model_params)	Define the local Hilbert space and operators; needs to be implemented in subclasses.
init_terms(self, model_params)	Add the onsite and coupling terms to the model; subclasses should implement this.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

init_sites(*self*, *model_params*)

Define the local Hilbert space and operators; needs to be implemented in subclasses.

This function gets called by *init_lattice()* to get the *Site* for the lattice unit cell.

Note: Initializing the sites requires to define the conserved quantum numbers. All pre-defined sites accept `conserve=None` to disable using quantum numbers. Many models in TeNPy read out the *conserve* model parameter, which can be set to `"best"` to indicate the optimal parameters.

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

sites [(tuple of) *Site*] The local sites of the lattice, defining the local basis states and operators.

init_lattice(*self*, *model_params*)
Initialize a lattice for the given model parameters.

This function reads out the model parameter *lattice*. This can be a full *Lattice* instance, in which case it is just returned without further action. Alternatively, the *lattice* parameter can be a string giving the name of one of the predefined lattices, which then gets initialized. Depending on the dimensionality of the lattice, this requires different model parameters.

The following model parameters get read out.

key	type	description
lattice	str | Lattice	The name of a lattice pre-defined in TeNPy to be initialized. Alternatively, a (possibly self-defined) Lattice instance. In the latter case, no further parameters are read out.
bc_MPS	str	Boundary conditions for the MPS
order	str	The order of sites within the lattice for non-trivial lattices.
L	int	The length in x-direction (or lenght of the unit cell for infinite systems). Only read out for 1D lattices.
Lx, Ly	int	The length in x- and y-direction. For `"infinite"` *bc_MPS*, the system is infinite in x-direction and *Lx* is the number of "rings" in the infinite MPS unit cell, while *Ly* gives the circumference around the cylinder or width of th the rung for a ladder (depending on *bc_y*. Only read out for 2D lattices.
bc_y	str	`"cylinder" | "ladder"`. The boundary conditions in y-direction. Only read out for 2D lattices.
bc_x	str	`"open" | "periodic"`. Can be used to force "periodic" boundaries for the lattice, i.e., for the couplings in the Hamiltonian, even if the MPS is finite. Defaults to `"open"` for `bc_MPS="finite"` and `"periodic"` for `bc_MPS="infinite`. If you are not aware of the consequences, you should probably *not* use "periodic" boundary conditions. (The MPS is still "open", so this will introduce long-range couplings between the first and last sites of the MPS!)

Parameters

model_params [dict] The model parameters given to `__init__`.

Returns

lat [*Lattice*] An initialized lattice.

init_terms(*self*, *model_params*)
Add the onsite and coupling terms to the model; subclasses should implement this.

add_coupling(*self*, *strength*, *u1*, *op1*, *u2*, *op2*, *dx*, *op_string=None*, *str_on_first=True*, *raise_op2_left=False*, *category=None*)
Add twosite coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $\sum_{x_0,\ldots,x_{dim-1}} strength[loc(\vec{x})] * OP1 * OP2$, where `OP1 := lat.unit_cell[u1].get_op(op1)` acts on the site `(x_0, ..., x_{dim-1}, u1)`, and `OP2 := lat.unit_cell[u2].get_op(op2)` acts on the site `(x_0+dx[0], ..., x_{dim-1}+dx[dim-1], u2)`. Possible combinations `x_0, ..., x_{dim-1}` are determined from the boundary conditions in *possible_couplings()*.

The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing the involved sites $\vec{x}$ and $\vec{x} + \vec{dx}$.

The necessary terms are just added to `coupling_terms`; doesn't (re)build the MPO.

Deprecated since version 0.4.0: The arguments *str_on_first* and *raise_op2_left* will be removed in version 1.0.0.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially (see above). If an array of smaller size is provided, it gets tiled to the required shape.

u1 [int] Picks the site `lat.unit_cell[u1]` for OP1.

op1 [str] Valid operator name of an onsite operator in `lat.unit_cell[u1]` for OP1.

u2 [int] Picks the site `lat.unit_cell[u2]` for OP2.

op2 [str] Valid operator name of an onsite operator in `lat.unit_cell[u2]` for OP2.

dx [iterable of int] Translation vector (of the unit cell) between OP1 and OP2. For a 1D lattice, a single int is also fine.

op_string [str | None] Name of an operator to be used between the OP1 and OP2 sites. Typical use case is the phase for a Jordan-Wigner transformation. The operator should be defined on all sites in the unit cell. If `None`, auto-determine whether a Jordan-Wigner string is needed, using *op_needs_JW()*.

str_on_first [bool] Whether the provided *op_string* should also act on the first site. This option should be chosen as `True` for Jordan-Wigner strings. When handling Jordan-Wigner strings we need to extend the *op_string* to also act on the 'left', first site (in the sense of the MPS ordering of the sites given by the lattice). In this case, there is a well-defined ordering of the operators in the physical sense (i.e. which of *op1* or *op2* acts first on a given state). We follow the convention that *op2* acts first (in the physical sense), independent of the MPS ordering. Deprecated.

raise_op2_left [bool] Raise an error when *op2* appears left of *op1* (in the sense of the MPS ordering given by the lattice). Deprecated.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

Examples

When initializing a model, you can add a term $J \sum_{<i,j>} S_i^z S_j^z$ on all nearest-neighbor bonds of the lattice like this:

```
>>> J = 1.  # the strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(J, u1, 'Sz', u2, 'Sz', dx)
```

The strength can be an array, which get's tiled to the correct shape. For example, in a 1D *Chain* with an even number of sites and periodic (or infinite) boundary conditions, you can add alternating strong and weak couplings with a line like:

```
>>> self.add_coupling([1.5, 1.], 0, 'Sz', 0, 'Sz', dx)
```

To add the hermitian conjugate, e.g. for a hopping term, you should add it in the opposite direction `-dx`, complex conjugate the strength, and take the hermitian conjugate of the operators in swapped order (including a swap of *u1* <-> *u2*). For spin-less fermions (`FermionSite`), this would be

```
>>> t = 1.  # hopping strength
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(t), u2, 'Cd', u1, 'C', -dx)  # h.c.
```

With spin-full fermions (`SpinHalfFermions`), it could be:

```
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(t, u1, 'Cdu', u2, 'Cd', dx)  # Cdagger_up C_down
...     self.add_coupling(np.conj(t), u2, 'Cdd', u1, 'Cu', -dx)  # h.c.␣
↪Cdagger_down C_up
```

Note that the Jordan-Wigner strings are figured out automatically!

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*, *category=None*)
Add a two-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op1}_i {op2}_j"`.

add_multi_coupling(*self*, *strength*, *u0*, *op0*, *other_ops*, *op_string=None*, *category=None*)
Add multi-site coupling terms to the Hamiltonian, summing over lattice sites.

Represents couplings of the form $sum_{x_0,...,x_{dim-1}} strength[loc(\vec{x})] * OP0 * OP1 * ... * OPM$, where `OP_0 := lat.unit_cell[u0].get_op(op0)` acts on the site `(x_0, ..., x_{dim-1}, u0)`, and `OP_m := lat.unit_cell[other_u[m]].get_op(other_op[m])`, m=1...M, acts on the site `(x_0+other_dx[m][0], ..., x_{dim-1}+other_dx[m][dim-1], other_u[m])`. For periodic boundary conditions along direction *a* (`lat.bc[a] == False`) the index `x_a` is taken modulo `lat.Ls[a]` and runs through `range(lat.Ls[a])`. For open boundary conditions, `x_a` is limited to `0 <= x_a < Ls[a]` and `0 <= x_a+other_dx[m,a] < lat.Ls[a]`. The coupling *strength* may vary spatially, $loc(\vec{x})$ indicates the lower left corner of the hypercube containing all the involved sites $\vec{x}, \vec{x} + other_{\vec{d}}\vec{x}[m, :]$.

The necessary terms are just added to `coupling_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the coupling. May vary spatially and is tiled to the required shape.

u0 [int] Picks the site `lat.unit_cell[u0]` for OP0.

op0 [str] Valid operator name of an onsite operator in `lat.unit_cell[u0]` for OP0.

other_ops [list of `(u, op_m, dx)`] One tuple for each of the other operators `OP1, OP2, ... OPM` involved. *u* picks the site `lat.unit_cell[u]`, *op_name* is a valid operator acting on that site, and *dx* gives the translation vector between `OP0` and the specified operator.

op_string [str | None] Name of an operator to be used inbetween the operators, excluding the sites on which the operators act. This operator should be defined on all sites in the unit cell.

Special case: If `None`, auto-determine whether a Jordan-Wigner string is needed (using *op_needs_JW()*), for each of the segments inbetween the operators and also on the sites of the left operators. Note that in this case the ordering of the operators *is* important and handled in the usual convention that `OPM` acts first and `OP0` last on a physical state.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op0}_i {other_ops[0]}_j {other_ops[1]}_k ..."`.

add_multi_coupling_term(*self*, *strength*, *ijkl*, *ops_ijkl*, *op_string*, *category=None*)
Add a general M-site coupling term on given MPS sites.

Wrapper for `self.coupling_terms[category].add_multi_coupling_term(...)`.

Parameters

strength [float] The strength of the coupling term.

ijkl [list of int] The MPS indices of the sites on which the operators acts. With *i, j, k, ...* = *ijkl*, we require that they are ordered ascending, `i < j < k < ...` and that `0 <= i < N_sites`. Inidces >= N_sites indicate couplings between different unit cells of an infinite MPS.

ops_ijkl [list of str] Names of the involved operators on sites *i, j, k, ...*.

op_string [list of str] Names of the operator to be inserted between the operators, e.g., op_string[0] is inserted between *i* and *j*.

category [str] Descriptive name used as key for `coupling_terms`. Defaults to a string of the form `"{op0}_i {other_ops[0]}_j {other_ops[1]}_k ..."`.

add_onsite(*self*, *strength*, *u*, *opname*, *category=None*)
Add onsite terms to `onsite_terms`.

Adds a term $\sum_{x_0,...,x_{dim-1}} strength[x_0, ..., x_{dim-1}] * OP$, where the operator `OP=lat.unit_cell[u].get_op(opname)` acts on the site given by a lattice index `(x_0, ..., x_{dim-1}, u)`, to the represented Hamiltonian.

The necessary terms are just added to `onsite_terms`; doesn't rebuild the MPO.

Parameters

strength [scalar | array] Prefactor of the onsite term. May vary spatially. If an array of smaller size is provided, it gets tiled to the required shape.

u [int] Picks a `Site` `lat.unit_cell[u]` out of the unit cell.

opname [str] valid operator name of an onsite operator in `lat.unit_cell[u]`.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *opname*.

add_onsite_term(*self*, *strength*, *i*, *op*, *category=None*)
Add an onsite term on a given MPS site.

Wrapper for `self.onsite_terms[category].add_onsite_term(...)`.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

category [str] Descriptive name used as key for `onsite_terms`. Defaults to *op*.

all_coupling_terms(*self*)
Sum of all `coupling_terms`.

all_onsite_terms(*self*)
Sum of all `onsite_terms`.

calc_H_MPO(*self*, *tol_zero=1e-15*)
Calculate MPO representation of the Hamiltonian.

Uses `onsite_terms` and `coupling_terms` to build an MPO graph (and then an MPO).

Parameters

tol_zero [float] Prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_MPO [*MPO*] MPO representation of the Hamiltonian.

calc_H_bond(*self*, *tol_zero=1e-15*)
calculate *H_bond* from `coupling_terms` and `onsite_terms`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_bond_from_MPO(*self*, *tol_zero=1e-15*)
Calculate the bond Hamiltonian from the MPO Hamiltonian.

Parameters

tol_zero [float] Arrays with norm < *tol_zero* are considered to be zero.

Returns

H_bond [list of *Array*] Bond terms as required by the constructor of `NearestNeighborModel`. Legs are `['p0', 'p0*', 'p1', 'p1*']`

Raises

ValueError [if the Hamiltonian contains longer-range terms.]

calc_H_onsite(*self*, *tol_zero=1e-15*)
Calculate *H_onsite* from *self.onsite_terms*.

Deprecated since version 0.4.0: This function will be removed in 1.0.0. Replace calls to this function by `self.all_onsite_terms().remove_zeros(tol_zero).to_Arrays(self.lat.mps_sites())`.

Parameters

tol_zero [float] prefactors with `abs(strength) < tol_zero` are considered to be zero.

Returns

H_onsite [list of npc.Array] onsite terms of the Hamiltonian.

coupling_strength_add_ext_flux(*self*, *strength*, *dx*, *phase*)

Add an external flux to the coupling strength.

When performing DMRG on a "cylinder" geometry, it might be useful to put an "external flux" through the cylinder. This means that a particle hopping around the cylinder should pick up a phase given by the external flux [Resta1997]. This is also called "twisted boundary conditions" in literature. This function adds a complex phase to the *strength* array on some bonds, such that particles hopping in positive direction around the cylinder pick up *exp(+i phase)*.

Warning: For the sign of *phase* it is important that you consistently use the creation operator as *op1* and the annihilation operator as *op2* in *add_coupling()*.

Parameters

strength [scalar | array] The strength to be used in *add_coupling()*, when no external flux would be present.

dx [iterable of int] Translation vector (of the unit cell) between *op1* and *op2* in *add_coupling()*.

phase [iterable of float] The phase of the external flux for hopping in each direction of the lattice. E.g., if you want flux through the cylinder on which you have an infinite MPS, you should give `phase=[0, phi]` souch that particles pick up a phase *phi* when hopping around the cylinder.

Returns

strength [complex array] The strength array to be used as *strength* in *add_coupling()* with the given *dx*.

Examples

Let's say you have an infinite MPS on a cylinder, and want to add nearest-neighbor hopping of fermions with the *FermionSite*. The cylinder axis is the *x*-direction of the lattice, so to put a flux through the cylinder, you want particles hopping *around* the cylinder to pick up a phase *phi* given by the external flux.

```
>>> strength = 1. # hopping strength without external flux
>>> phi = np.pi/4 # determines the external flux strength
>>> strength_with_flux = self.coupling_strength_add_ext_flux(strength, dx, [0,
↪ phi])
>>> for u1, u2, dx in self.lat.pairs['nearest_neighbors']:
...     self.add_coupling(strength_with_flux, u1, 'Cd', u2, 'C', dx)
...     self.add_coupling(np.conj(strength_with_flux), u2, 'Cd', u1, 'C', -dx)
```

group_sites(*self*, *n=2*, *grouped_sites=None*)

Modify *self* in place to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

This has to be done after finishing initialization and can not be reverted.

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

Returns

grouped_sites [list of *GroupedSite*] The sites grouped together.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

Module description

Kitaev's exactly solvable toric code model.

As we put the model on a cylinder, the name "toric code" is a bit misleading, but it is the established name for this model...

7.2.4 networks

- full name: tenpy.networks
- parent module: *tenpy*
- type: module

Module description

Definitions of tensor networks like MPS and MPO.

Here, 'tensor network' refers just to the (parital) contraction of tensors. For example an MPS represents the contraction along the 'virtual' legs/bonds of its B.

Submodules

site	Defines a class describing the local physical Hilbert space.
mps	This module contains a base class for a Matrix Product State (MPS).
mpo	Matrix product operator (MPO).
terms	Classes to store a collection of operator names and sites they act on, together with prefactors.
purification_mps	This module contains an MPS class representing an density matrix by purification.

site

- full name: tenpy.networks.site
- parent module: *tenpy.networks*
- type: module

Classes

BosonSite([Nmax, conserve, filling])	Create a *Site* for up to *Nmax* bosons.
FermionSite([conserve, filling])	Create a *Site* for spin-less fermions.
GroupedSite(sites[, labels, charges])	Group two or more *Site* into a larger one.
Site(leg[, state_labels])	Collects necessary information about a single local site of a lattice.
SpinHalfFermionSite([cons_N, cons_Sz, filling])	Create a *Site* for spinful (spin-1/2) fermions.
SpinHalfSite([conserve])	Spin-1/2 site.
SpinSite([S, conserve])	General Spin S site.

BosonSite

- full name: tenpy.networks.site.BosonSite
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**BosonSite**(*Nmax=1*, *conserve='N'*, *filling=0.0*)

Bases: *tenpy.networks.site.Site*

Create a *Site* for up to *Nmax* bosons.

Local states are `vac, 1, 2, ... , Nc`. (Exception: for parity conservation, we sort as `vac, 2, 4, ..., 1, 3, 5, ...`.)

operator	description
`Id, JW`	Identity $\mathbb{1}$
`B`	Annihilation operator b
`Bd`	Creation operator $b^\dagger$
`N`	Number operator $n = b^\dagger b$
`NN`	n^2
`dN`	$\delta n := n - filling$
`dNdN`	$(\delta n)^2$
`P`	Parity $Id - 2(n \mod 2)$.

conserve	qmod	*excluded* onsite operators
`'N'`	[1]	–
`'parity'`	[2]	–
`None`	[]	–

Parameters

Nmax [int] Cutoff defining the maximum number of bosons per site. The default `Nmax=1` describes hard-core bosons.

conserve [str] Defines what is conserved, see table above.

filling [float] Average filling. Used to define `dN`.

Attributes

conserve [str] Defines what is conserved, see table above.

filling [float] Average filling. Used to define `dN`.

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

add_op (*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be `[leg, leg.conj()]`. We set labels `'p', 'p*'`.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If `True`, the function adds *name* to `need_JW_string`.

change_charge (*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [`LegCharge` | None] The new charges to be used. If `None`, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust `state_labels` and `perm`. If you sorted the previous leg with `perm_qind, new_leg_charge = leg.sort()`, use `leg.perm_flat_from_perm_qind(perm_qind)`. Ignored if `None`.

property dim
Dimension of the local Hilbert space.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels `'p', 'p*'`. If name already was an npc Array, it's directly returned.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from `need_JW_string`.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

FermionSite

- full name: tenpy.networks.site.FermionSite
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**FermionSite**(*conserve='N'*, *filling=0.5*)
Bases: *tenpy.networks.site.Site*

Create a *Site* for spin-less fermions.

Local states are `empty` and `full`.

Warning: Using the Jordan-Wigner string (`JW`) is crucial to get correct results, otherwise you just describe hardcore bosons! Further details in *Fermions and the Jordan-Wigner transformation*.

operator	description
`Id`	Identity $\mathbb{1}$
`JW`	Sign for the Jordan-Wigner string.
`C`	Annihilation operator c (up to 'JW'-string left of it)
`Cd`	Creation operator $c^\dagger$ (up to 'JW'-string left of it)
`N`	Number operator $n = c^\dagger c$
`dN`	$\delta n := n - filling$
`dNdN`	$(\delta n)^2$

conserve	qmod	*exluded* onsite operators
`'N'`	[1]	–
`'parity'`	[2]	–
`None`	[]	–

Parameters

conserve [str] Defines what is conserved, see table above.

filling [float] Average filling. Used to define `dN`.

Attributes

conserve [str] Defines what is conserved, see table above.

filling [float] Average filling. Used to define `dN`.

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

add_op (*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be `[leg, leg.conj()]`. We set labels `'p', 'p*'`.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If `True`, the function adds *name* to `need_JW_string`.

change_charge (*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [`LegCharge` | None] The new charges to be used. If `None`, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust `state_labels` and `perm`. If you sorted the previous leg with `perm_qind, new_leg_charge = leg.sort()`, use `leg.perm_flat_from_perm_qind(perm_qind)`. Ignored if `None`.

property dim
Dimension of the local Hilbert space.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels `'p', 'p*'`. If name already was an npc Array, it's directly returned.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from `need_JW_string`.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

GroupedSite

- full name: tenpy.networks.site.GroupedSite
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**GroupedSite**(*sites*, *labels=None*, *charges='same'*)
Bases: *tenpy.networks.site.Site*

Group two or more *Site* into a larger one.

A typical use-case is that you want a NearestNeighborModel for TEBD although you have next-nearest neighbor interactions: you just double your local Hilbertspace to consist of two original sites. Note that this is a 'hack' at the cost of other things (e.g., measurements of 'local' operators) getting more complicated/computationally expensive.

If the individual sites indicate fermionic operators (with entries in *need_JW_string*), we construct the new onsite oerators of *site1* to include the JW string of *site0*, i.e., we use the Kronecker product of `[JW, op]` instead of `[Id, op]` if necessary (but always `[op, Id]`). In that way the onsite operators of this DoubleSite automatically fulfill the expected commutation relations. See also *Fermions and the Jordan-Wigner transformation*.

Parameters

sites [list of *Site*] The individual sites being grouped together. Copied before use if `charges!='same'`.

labels : Include the Kronecker product of the each onsite operator *op* on `sites[i]` and identities on other sites with the name `opname+labels[i]`. Similarly, set state labels for `' '.join(state[i]+'_'+labels[i])`. Defaults to `[str(i) for i in range(n_sites)]`, which for example grouping two SpinSites gives operators name like `"Sz0"` and sites labels like `'up_0 down_1'`.

charges [`'same' | 'drop' | 'independent'`] How to handle charges, defaults to 'same'. `'same'` means that all *sites* have the same *ChargeInfo*, and the total charge is the sum of the charges on the individual *sites*. `'independent'` means that the *sites* have possibly different *ChargeInfo*, and the charges are conserved separately, i.e., we have *n_sites* conserved charges. For `'drop'`, we drop any charges, such that the remaining legcharges are trivial.

Attributes

n_sites [int] The number of sites grouped together, i.e. `len(sites)`.

sites [list of *Site*] The sites grouped together into self.

labels: list of str The labels using which the single-site operators are added during construction.

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
kroneckerproduct(self, ops)	Return the Kronecker product $op0 \otimes op1$ of local operators.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

kroneckerproduct(*self*, *ops*)
Return the Kronecker product $op0 \otimes op1$ of local operators.

Parameters

ops [list of *Array*] One operator (or operator name) on each of the ungrouped sites. Each operator should have labels `['p', 'p*']`.

Returns

prod [*Array*] Kronecker product $ops[0] \otimes ops[1] \otimes \cdots$, with labels `['p', 'p*']`.

add_op(*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be `[leg, leg.conj()]`. We set labels `'p', 'p*'`.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If `True`, the function adds *name* to `need_JW_string`.

change_charge(*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [LegCharge | None] The new charges to be used. If None, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust state_labels and perm. If you sorted the previous leg with perm_qind, new_leg_charge = leg.sort(), use leg.perm_flat_from_perm_qind(perm_qind). Ignored if None.

property dim
Dimension of the local Hilbert space.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels 'p', 'p*'. If name already was an npc Array, it's directly returned.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from need_JW_string.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

Site

- full name: tenpy.networks.site.Site
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**Site**(*leg*, *state_labels=None*, ***site_ops*)
Bases: object

Collects necessary information about a single local site of a lattice.

This class defines what the local basis states are: it provides the `leg` defining the charges of the physical leg for this site. Moreover, it stores (local) on-site operators, which are directly available as attribute, e.g., `self.Sz` is the Sz operator for the *SpinSite*. Alternatively, operators can be obained with *get_op()*. The operator names `Id` and `JW` are reserved for the identy and Jordan-Wigner strings.

Warning: The order of the local basis can change depending on the charge conservation! This is a *necessary* feature since we need to sort the basis by charges for efficiency. We use the `state_labels` and `perm` to keep track of these permutations.

Parameters

leg [*LegCharge*] Charges of the physical states, to be used for the physical leg of MPS.

state_labels [None | list of str] Optionally a label for each local basis states. `None` entries are ignored / not set.

****site_ops :** Additional keyword arguments of the form `name=op` given to *add_op()*. The identity operator `'Id'` is automatically included. If no `'JW'` for the Jordan-Wigner string is given, `'JW'` is set as an alias to `'Id'`.

Examples

The following generates a site for spin-1/2 with Sz conservation. Note that `Sx = (Sp + Sm)/2` violates Sz conservation and is thus not a valid on-site operator.

```
>>> chinfo = npc.ChargeInfo([1], ['Sz'])
>>> ch = npc.LegCharge.from_qflat(chinfo, [1, -1])
>>> Sp = [[0, 1.], [0, 0]]
>>> Sm = [[0, 0], [1., 0]]
>>> Sz = [[0.5, 0], [0, -0.5]]
>>> site = Site(ch, ['up', 'down'], Splus=Sp, Sminus=Sm, Sz=Sz)
>>> print(site.Splus.to_ndarray())
array([[ 0.,  1.],
       [ 0.,  0.]])
>>> print(site.get_op('Sminus').to_ndarray())
array([[ 0.,  0.],
       [ 1.,  0.]])
>>> print(site.get_op('Splus Sminus').to_ndarray())
array([[ 1.,  0.],
       [ 0.,  0.]])
```

Attributes

dim Dimension of the local Hilbert space.

onsite_ops Dictionary of on-site operators for iteration.

leg [*LegCharge*] Charges of the local basis states.

state_labels [{str: int}] (Optional) labels for the local basis states.

opnames [set] Labels of all onsite operators (i.e. `self.op` exists if `'op'` in `self.opnames`). Note that *get_op()* allows arbitrary concatenations of them.

need_JW_string [set] Labels of all onsite operators that need a Jordan-Wigner string. Used in *op_needs_JW()* to determine whether an operator anticommutes or commutes with operators on other sites.

ops [*Array*] Onsite operators are added directly as attributes to self. For example after `self.add_op('Sz', Sz)` you can use `self.Sz` for the S^z operator. All onsite operators have labels `'p'`, `'p*'`.

perm [1D array] Index permutation of the physical leg compared to *conserve=None*, i.e. `OP_conserved = OP_nonconserved[np.ix_(perm, perm)]` and `perm[state_labels_conserved["some_state"]] == state_labels_nonconserved["some_state"]`.

JW_exponent [1D array] Exponents of the `'JW'` operator, such that `self.JW.to_ndarray() = np.diag(np.exp(1.j*np.pi* JW_exponent))`

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

change_charge(*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [LegCharge | None] The new charges to be used. If None, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust state_labels and perm. If you sorted the previous leg with perm_qind, new_leg_charge = leg.sort(), use leg.perm_flat_from_perm_qind(perm_qind). Ignored if None.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

property dim
Dimension of the local Hilbert space.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

add_op(*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be [leg, leg.conj()]. We set labels 'p', 'p*'.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If True, the function adds *name* to need_JW_string.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels `'p', 'p*'`. If name already was an npc Array, it's directly returned.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from `need_JW_string`.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

SpinHalfFermionSite

- full name: tenpy.networks.site.SpinHalfFermionSite
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**SpinHalfFermionSite**(*cons_N='N'*, *cons_Sz='Sz'*, *filling=1.0*)

Bases: *tenpy.networks.site.Site*

Create a *Site* for spinful (spin-1/2) fermions.

Local states are: `empty` (vacuum), `up` (one spin-up electron), `down` (one spin-down electron), and `full` (both electrons)

Local operators can be built from creation operators.

Warning: Using the Jordan-Wigner string (`JW`) in the correct way is crucial to get correct results, otherwise you just describe hardcore bosons!

operator	description
`Id`	Identity $\mathbb{1}$
`JW`	Sign for the Jordan-Wigner string $(-1)^{n_\uparrow + n_\downarrow}$
`JWu`	Partial sign for the Jordan-Wigner string $(-1)^{n_\uparrow}$
`JWd`	Partial sign for the Jordan-Wigner string $(-1)^{n_\downarrow}$
`Cu`	Annihilation operator spin-up $c_\uparrow$ (up to 'JW'-string on sites left of it).
`Cdu`	Creation operator spin-up $c_\uparrow^\dagger$ (up to 'JW'-string on sites left of it).
`Cd`	Annihilation operator spin-down $c_\downarrow$ (up to 'JW'-string on sites left of it). Includes `JWu` such that it anti-commutes onsite with `Cu, Cdu`.
`Cdd`	Creation operator spin-down $c_\downarrow^\dagger$ (up to 'JW'-string on sites left of it). Includes `JWu` such that it anti-commutes onsite with `Cu, Cdu`.
`Nu`	Number operator $n_\uparrow = c_\uparrow^\dagger c_\uparrow$
`Nd`	Number operator $n_\downarrow = c_\downarrow^\dagger c_\downarrow$
`NuNd`	Dotted number operators $n_\uparrow n_\downarrow$
`Ntot`	Total number operator $n_t = n_\uparrow + n_\downarrow$
`dN`	Total number operator compared to the filling $\Delta n = n_t - filling$
`Sx, Sy, Sz`	Spin operators $S^{x,y,z}$, in particular $S^z = \frac{1}{2}(n_\uparrow - n_\downarrow)$
`Sp, Sm`	Spin flips $S^\pm = S^x \pm i S^y$, e.g. $S^+ = c_\uparrow^\dagger c_\downarrow$

The spin operators are defined as $S^\gamma = (c_\uparrow^\dagger, c_\downarrow^\dagger)\sigma^\gamma(c_\uparrow, c_\downarrow)^T$, where σ^γ are spin-1/2 matrices (i.e. half the pauli matrices).

cons_N	*cons_Sz*	qmod	*excluded* onsite operators
`'N'`	`'Sz'`	[1, 1]	`Sx, Sy`
`'N'`	`'parity'`	[1, 2]	–
`'N'`	`None`	[1]	–
`'parity'`	`'Sz'`	[2, 1]	`Sx, Sy`
`'parity'`	`'parity'`	[2, 2]	–
`'parity'`	`None`	[2]	–
`None`	`'Sz'`	[1]	`Sx, Sy`
`None`	`'parity'`	[2]	–
`None`	`None`	[]	–

Todo: Check if Jordan-Wigner strings for 4x4 operators are correct.

Parameters

cons_N [`'N'` | `'parity'` | `None`] Whether particle number is conserved, c.f. table above.

cons_Sz [`'Sz'` | `'parity'` | `None`] Whether spin is conserved, c.f. table above.

filling [float] Average filling. Used to define `dN`.

Attributes

cons_N [`'N'` | `'parity'` | `None`] Whether particle number is conserved, c.f. table above.

cons_Sz [`'Sz'` | `'parity'` | `None`] Whether spin is conserved, c.f. table above.

filling [float] Average filling. Used to define `dN`.

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

add_op (*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be `[leg, leg.conj()]`. We set labels `'p', 'p*'`.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If `True`, the function adds *name* to `need_JW_string`.

change_charge(*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [`LegCharge` | None] The new charges to be used. If `None`, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust `state_labels` and `perm`. If you sorted the previous leg with `perm_qind, new_leg_charge = leg.sort()`, use `leg.perm_flat_from_perm_qind(perm_qind)`. Ignored if `None`.

property dim
Dimension of the local Hilbert space.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels `'p', 'p*'`. If name already was an npc Array, it's directly returned.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from `need_JW_string`.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

SpinHalfSite

- full name: tenpy.networks.site.SpinHalfSite
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**SpinHalfSite**(*conserve='Sz'*)
Bases: *tenpy.networks.site.Site*

Spin-1/2 site.

Local states are `up` (0) and `down` (1). Local operators are the usual spin-1/2 operators, e.g. `Sz = [[0.5, 0.], [0., -0.5]]`, `Sx = 0.5*sigma_x` for the Pauli matrix *sigma_x*.

operator	description
`Id, JW`	Identity $\mathbb{1}$
`Sx, Sy, Sz`	Spin components $S^{x,y,z}$, equal to half the Pauli matrices.
`Sigmax, Sigmay, Sigmaz`	Pauli matrices $\sigma^{x,y,z}$
`Sp, Sm`	Spin flips $S^{\pm} = S^x \pm iS^y$

conserve	qmod	*excluded* onsite operators
`'Sz'`	[1]	`Sx, Sy, Sigmax, Sigmay`
`'parity'`	[2]	–
`None`	[]	–

Parameters

conserve [str] Defines what is conserved, see table above.

Attributes

conserve [str] Defines what is conserved, see table above.

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

add_op (*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be `[leg, leg.conj()]`. We set labels `'p', 'p*'`.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If `True`, the function adds *name* to `need_JW_string`.

change_charge (*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [`LegCharge` | None] The new charges to be used. If `None`, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust `state_labels` and `perm`. If you sorted the previous leg with `perm_qind, new_leg_charge = leg.sort()`, use `leg.perm_flat_from_perm_qind(perm_qind)`. Ignored if `None`.

property dim
Dimension of the local Hilbert space.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels `'p', 'p*'`. If name already was an npc Array, it's directly returned.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from `need_JW_string`.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

SpinSite

- full name: tenpy.networks.site.SpinSite
- parent module: *tenpy.networks.site*
- type: class

class tenpy.networks.site.**SpinSite**(*S=0.5*, *conserve='Sz'*)
Bases: *tenpy.networks.site.Site*

General Spin S site.

There are *2S+1* local states range from `down` (0) to `up` (2S+1), corresponding to `Sz=-S, -S+1, ..., S-1, S`. Local operators are the spin-S operators, e.g. `Sz = [[0.5, 0.], [0., -0.5]]`, `Sx = 0.5*sigma_x` for the Pauli matrix *sigma_x*.

operator	description
`Id, JW`	Identity $\mathbb{1}$
`Sx, Sy, Sz`	Spin components $S^{x,y,z}$, equal to half the Pauli matrices.
`Sp, Sm`	Spin flips $S^{\pm} = S^x \pm iS^y$

conserve	qmod	*excluded* onsite operators
`'Sz'`	[1]	`Sx, Sy`
`'parity'`	[2]	–
`None`	[]	–

Parameters

conserve [str] Defines what is conserved, see table above.

Attributes

S [{0.5, 1, 1.5, 2, ...}] The 2S+1 states range from m = -S, -S+1, ... +S.

conserve [str] Defines what is conserved, see table above.

Methods

add_op(self, name, op[, need_JW])	Add one on-site operators.
change_charge(self[, new_leg_charge, permute])	Change the charges of the site (in place).
get_op(self, name)	Return operator of given name.
multiply_op_names(self, names)	Multiply operator names together.
op_needs_JW(self, name)	Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.
remove_op(self, name)	Remove an added operator.
rename_op(self, old_name, new_name)	Rename an added operator.
state_index(self, label)	Return index of a basis state from its label.
state_indices(self, labels)	Same as *state_index()*, but for multiple labels.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.
valid_opname(self, name)	Check whether 'name' labels a valid onsite-operator.

add_op(*self*, *name*, *op*, *need_JW=False*)
Add one on-site operators.

Parameters

name [str] A valid python variable name, used to label the operator. The name under which *op* is added as attribute to self.

op [np.ndarray | *Array*] A matrix acting on the local hilbert space representing the local operator. Dense numpy arrays are automatically converted to *Array*. LegCharges have to be `[leg, leg.conj()]`. We set labels `'p', 'p*'`.

need_JW [bool] Whether the operator needs a Jordan-Wigner string. If `True`, the function adds *name* to `need_JW_string`.

change_charge(*self*, *new_leg_charge=None*, *permute=None*)
Change the charges of the site (in place).

Parameters

new_leg_charge [`LegCharge` | None] The new charges to be used. If `None`, use trivial charges.

permute [ndarray | None] The permuation applied to the physical leg, which gets used to adjust `state_labels` and `perm`. If you sorted the pre-

vious leg with `perm_qind, new_leg_charge = leg.sort()`, use `leg.perm_flat_from_perm_qind(perm_qind)`. Ignored if `None`.

property dim
Dimension of the local Hilbert space.

get_op(*self*, *name*)
Return operator of given name.

Parameters

name [str] The name of the operator to be returned. In case of multiple operator names separated by whitespace, we multiply them together to a single on-site operator (with the one on the right acting first).

Returns

op [*np_conserved*] The operator given by *name*, with labels `'p', 'p*'`. If name already was an npc Array, it's directly returned.

multiply_op_names(*self*, *names*)
Multiply operator names together.

Join the operator names in *names* such that *get_op* returns the product of the corresponding operators.

Parameters

names [list of str] List of valid operator labels.

Returns

combined_opname [str] A valid operator name Operatorname representing the product of operators in *names*.

property onsite_ops
Dictionary of on-site operators for iteration.

Single operators are accessible as attributes.

op_needs_JW(*self*, *name*)
Whether an (composite) onsite operator is fermionic and needs a Jordan-Wigner string.

Parameters

name [str] The name of the operator, as in *get_op()*.

Returns

needs_JW [bool] Whether the operator needs a Jordan-Wigner string, judging from `need_JW_string`.

remove_op(*self*, *name*)
Remove an added operator.

Parameters

name [str] The name of the operator to be removed.

rename_op(*self*, *old_name*, *new_name*)
Rename an added operator.

Parameters

old_name [str] The old name of the operator.

new_name [str] The new name of the operator.

state_index(*self*, *label*)
Return index of a basis state from its label.

Parameters

label [int | string] eather the index directly or a label (string) set before.

Returns

state_index [int] the index of the basis state associated with the label.

state_indices(*self*, *labels*)
Same as *state_index()*, but for multiple labels.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

valid_opname(*self*, *name*)
Check whether 'name' labels a valid onsite-operator.

Parameters

name [str] Label for the operator. Can be multiple operator(labels) separated by whitespace, indicating that they should be multiplied together.

Returns

valid [bool] `True` if *name* is a valid argument to *get_op()*.

Functions

group_sites(sites[, n, labels, charges])	Given a list of sites, group each *n* sites together.
multi_sites_combine_charges(sites[, ...])	Adjust the charges of the given sites (in place) such that they can be used together.

group_sites

- full name: tenpy.networks.site.group_sites
- parent module: *tenpy.networks.site*
- type: function

tenpy.networks.site.**group_sites**(*sites*, *n=2*, *labels=None*, *charges='same'*)
Given a list of sites, group each *n* sites together.

Parameters

sites [list of *Site*] The sites to be grouped together.

n [int] We group each *n* consecutive sites from *sites* together in a *GroupedSite*.

labels, charges : See `GroupedSites`.

Returns

grouped_sites [list of *GroupedSite*] The grouped sites. Has length `(len(sites)-1)//n + 1`.

multi_sites_combine_charges

- full name: tenpy.networks.site.multi_sites_combine_charges
- parent module: *tenpy.networks.site*
- type: function

tenpy.networks.site.**multi_sites_combine_charges**(*sites*, *same_charges=[]*)

Adjust the charges of the given sites (in place) such that they can be used together.

When we want to contract tensors corresponding to different *Site* instances, these sites need to share a single *ChargeInfo*. This function adjusts the charges of these sites such that they can be used together.

Parameters

sites [list of *Site*] The sites to be combined. Modified **in place**.

same_charges [[[(int, int|str), (int, int|str), ...], ...]] Defines which charges actually are the same, i.e. their quantum numbers are added up. Each charge is specified by a tuple (s, i)= (int, int|str), where *s* gives the index of the site within sites and *i* the index or name of the charge in the *ChargeInfo* of this site.

Returns

perms [list of ndarray] For each site the permutation performed on the physical leg to sort by charges.

Examples

```
>>> ferm = SpinHalfFermionSite(cons_N='N', cons_Sz='Sz')
>>> spin = SpinSite(1.0, 'Sz')
>>> ferm.leg.chinfo is spin.leg.chinfo
False
>>> print(spin.leg)
+1
0 [[-1]
1  [ 1]]
2
>>> multi_sites_combine_charges([ferm, spin], same_charges=[[(0, 'Sz'), (1, 0)]])
[array([0, 1, 2, 3]), array([0, 1])]
>>> # no permutations where needed
>>> ferm.leg.chinfo is spin.leg.chinfo
True
>> ferm.leg.chinfo.names
['N', 'Sz']
>>> print(spin.leg)
+1
0 [[ 0 -1]
1  [ 0  1]]
2
```

Module description

Defines a class describing the local physical Hilbert space.

The *Site* is the prototype, read it's docstring.

mps

- full name: tenpy.networks.mps
- parent module: *tenpy.networks*
- type: module

Classes

MPS(sites, Bs, SVs[, bc, form, norm])	A Matrix Product State, finite (MPS) or infinite (iMPS).
MPSEnvironment(bra, ket[, init_LP, init_RP, ...])	Stores partial contractions of $< bra\|Op\|ket >$ for local operators Op.
TransferMatrix(bra, ket[, shift_bra, ...])	Transfer matrix of two MPS (bra & ket).

MPS

- full name: tenpy.networks.mps.MPS
- parent module: *tenpy.networks.mps*
- type: class

class tenpy.networks.mps.**MPS**(*sites*, *Bs*, *SVs*, *bc='finite'*, *form='B'*, *norm=1.0*)
Bases: object

A Matrix Product State, finite (MPS) or infinite (iMPS).

Parameters

sites [list of *Site*] Defines the local Hilbert space for each site.

Bs [list of *Array*] The 'matrices' of the MPS. Labels are `vL, vR, p` (in any order).

SVs [list of 1D array] The singular values on *each* bond. Should always have length *L+1*. Entries out of *nontrivial_bonds* are ignored.

bc [`'finite' | 'segment' | 'infinite'`] Boundary conditions as described in the tabel of the module doc-string.

form [(list of) {`'B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)}] The form of the stored 'matrices', see table in module doc-string. A single choice holds for all of the entries.

Attributes

L Number of physical sites; for an iMPS the len of the MPS unit cell.

chi Dimensions of the (nontrivial) virtual bonds.

finite Distinguish MPS vs iMPS.

nontrivial_bonds Slice of the non-trivial bond indices, depending on `self.bc`.

sites [list of *Site*] Defines the local Hilbert space for each site.

bc [{'finite', 'segment', 'infinite'}] Boundary conditions as described in above table.

form [list of {None | tuple(float, float)}] Describes the canonical form on each site. None means non-canonical form. For form = (nuL, nuR), the stored _B[i] are s**form[0] -- Gamma -- s**form[1] (in Vidal's notation).

chinfo [ChargeInfo] The nature of the charge.

dtype [type] The data type of the _B.

norm [float] The norm of the state, i.e. sqrt(<psi|psi>). Ignored for (normalized) *expectation_value()*, but important for *overlap()*.

grouped [int] Number of sites grouped together, see *group_sites()*.

_B [list of npc.Array] The 'matrices' of the MPS. Labels are vL, vR, p (in any order). We recommend using *get_B()* and *set_B()*, which will take care of the different canonical forms.

_S [list of (None | 1D array)] The singular values on each virtual bond, length L+1. May be None if the MPS is not in canonical form. Otherwise, _S[i] is to the left of _B[i]. We recommend using *get_SL()*, *get_SR()*, *set_SL()*, *set_SR()*, which takes proper care of the boundary conditions.

_valid_forms [dict] Mapping for canonical forms to a tuple (nuL, nuR) indicating that self._Bs[i] = s[i]**nuL -- Gamma[i] -- s[i]**nuR is saved.

_valid_bc [tuple of str] Valid boundary conditions.

_transfermatrix_keep [int] How many states to keep at least when diagonalizing a *TransferMatrix*. Important if the state develops a near-degeneracy.

Methods

add(self, other, alpha, beta[, cutoff])	Return an MPS which represents alpha|self> + beta |others>.
apply_local_op(self, i, op[, unitary, ...])	Apply a local (one or multi-site) operator to *self*.
average_charge(self[, bond])	Return the average charge for the block on the left of a given bond.
canonical_form(self[, renormalize])	Bring self into canonical 'B' form, (re-)calculate singular values.
canonical_form_finite(self[, renormalize, ...])	Bring a finite (or segment) MPS into canonical form (in place).
canonical_form_infinite(self[, renormalize, ...])	Bring an infinite MPS into canonical form (in place).
charge_variance(self[, bond])	Return the charge variance on the left of a given bond.
compute_K(self, perm[, swap_op, trunc_par, ...])	Compute the momentum quantum numbers of the entanglement spectrum for 2D states.
convert_form(self[, new_form])	Tranform self into different canonical form (by scaling the legs with singular values).
copy(self)	Returns a copy of *self*.
correlation_function(self, ops1, ops2[, ...])	Correlation function <psi|op1_i op2_j|psi>/<psi|psi> of single site operators.

Continued on next page

Table 132 – continued from previous page

correlation_length(self[, target, tol_ev0, ...])	Calculate the correlation length by diagonalizing the transfer matrix.
entanglement_entropy(self[, n, bonds, ...])	Calculate the (half-chain) entanglement entropy for all nontrivial bonds.
entanglement_entropy_segment(self[, ...])	Calculate entanglement entropy for general geometry of the bipartition.
entanglement_spectrum(self[, by_charge])	return entanglement energy spectrum.
expectation_value(self, ops[, sites, axes])	Expectation value <psi|ops|psi>/<psi|psi> of (n-site) operator(s).
expectation_value_multi_sites(self, ...)	Expectation value <psi|op0_{i0}op1_{i0+1}...opN_{i0+N}|psi>/<psi|psi>.
expectation_value_term(self, term[, autoJW])	Expectation value <psi|op_{i0}op_{i1}...op_{iN}|psi>/<psi|psi>.
expectation_value_terms_sum(self, term_list)	Calculate expectation values for a bunch of terms and sum them up.
from_Bflat(sites, Bflat[, SVs, bc, dtype, ...])	Construct a matrix product state from a set of numpy arrays *Bflat* and singular vals.
from_full(sites, psi[, form, cutoff, ...])	Construct an MPS from a single tensor *psi* with one leg per physical site.
from_product_state(sites, p_state[, bc, ...])	Construct a matrix product state from a given product state.
from_singlets(site, L, pairs[, up, down, ...])	Create an MPS of entangled singlets.
gauge_total_charge(self[, qtotal, vL_leg, ...])	Gauge the legcharges of the virtual bonds such that the MPS has a total *qtotal*.
get_B(self, i[, form, copy, cutoff, label_p])	Return (view of) *B* at site *i* in canonical form.
get_SL(self, i)	Return singular values on the left of site *i*
get_SR(self, i)	Return singular values on the right of site *i*
get_grouped_mps(self, blocklen)	contract blocklen subsequent tensors into a single one and return result as a new MPS.
get_op(self, op_list, i)	Given a list of operators, select the one corresponding to site *i*.
get_rho_segment(self, segment)	Return reduced density matrix for a segment.
get_theta(self, i[, n, cutoff, formL, formR])	Calculates the *n*-site wavefunction on sites[i:i+n].
get_total_charge(self[, only_physical_legs])	Calculate and return the *qtotal* of the whole MPS (when contracted).
group_sites(self[, n, grouped_sites])	Modify *self* inplace to group sites.
group_split(self[, trunc_par])	Modify *self* inplace to split previously grouped sites.
increase_L(self[, new_L])	Modify *self* inplace to enlarge the unit cell.
mutinf_two_site(self[, max_range, n])	Calculate the two-site mutual information $I(i:j)$.
norm_test(self)	Check that self is in canonical form.
overlap(self, other[, charge_sector, ...])	Compute overlap <self|other>.
permute_sites(self, perm[, swap_op, ...])	Applies the permutation perm to the state (inplace).
probability_per_charge(self[, bond])	Return probabilites of charge value on the left of a given bond.
set_B(self, i, B[, form])	Set *B* at site *i*.
set_SL(self, i, S)	Set singular values on the left of site *i*
set_SR(self, i, S)	Set singular values on the right of site *i*
swap_sites(self, i[, swap_op, trunc_par])	Swap the two neighboring sites *i* and *i+1* (inplace).
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

classmethod from_product_state(*sites*, *p_state*, *bc='finite'*, *dtype=<class 'numpy.float64'>*, *permute=True*, *form='B'*, *chargeL=None*)
Construct a matrix product state from a given product state.

Parameters

sites [list of *Site*] The sites defining the local Hilbert space.

p_state [iterable of {int | str | 1D array}] Defines the product state to be represented. If `p_state[i]` is str, then site `i` is in state `self.sites[i].state_labels(p_state[i])`. If `p_state[i]` is int, then site `i` is in state `p_state[i]`. If `p_state[i]` is an array, then site `i` wavefunction is `p_state[i]`.

bc [{'infinite', 'finite', 'segmemt'}] MPS boundary conditions. See docstring of *MPS*.

dtype [type or string] The data type of the array entries.

permute [bool] The `Site` might permute the local basis states if charge conservation gets enabled. If *permute* is True (default), we permute the given *p_state* locally according to each site's `perm`. The *p_state* argument should then always be given as if *conserve=None* in the Site.

form [(list of) {`'B' | 'A' | 'C' | 'G' | None` | tuple(float, float)}] Defines the canonical form. See module doc-string. A single choice holds for all of the entries.

chargeL [charges] Leg charge at bond 0, which are purely conventional.

Returns

product_mps [*MPS*] An MPS representing the specified product state.

Examples

Example to get a Neel state for a `TIChain`:

```
>>> M = TFIChain({'L': 10})
>>> p_state = ["up", "down"] * (L//2)  # repeats entries L/2 times
>>> psi = MPS.from_product_state(M.lat.mps_sites(), p_state, bc=M.lat.bc_MPS)
```

The meaning of the labels `"up","down"` is defined by the `Site`, in this example a *SpinHalfSite*.

Extending the example, we can replace the spin in the center with one with arbitrary angles `theta, phi` in the bloch sphere:

```
>>> M = TFIChain({'L': 8, 'conserve': None})
>>> p_state = ["up", "down"] * (L//2)  # repeats entries L/2 times
>>> bloch_sphere_state = np.array([np.cos(theta/2), np.exp(1.j*phi)*np.
↪sin(theta/2)])
>>> p_state[L//2] = bloch_sphere_state   # replace one spin in center
>>> psi = MPS.from_product_state(M.lat.mps_sites(), p_state, bc=M.lat.bc_MPS,␣
↪dtype=np.complex)
```

Note that for the more general *SpinChain*, the order of the two entries for the `bloch_sphere_state` would be *exactly the opposite* (when we keep the the north-pole of the bloch sphere being the up-state). The reason is that the *SpinChain* uses the general *SpinSite*, where the states are orderd ascending from `'down'` to `'up'`. The *SpinHalfSite* on the other hand uses the order `'up', 'down'` where that the Pauli matrices look as usual.

Moreover, note that you can not write this bloch state (for `theta != 0, pi`) when conserving symmetries, as the two physical basis states correspond to different symmetry sectors.

classmethod from_Bflat(*sites*, *Bflat*, *SVs=None*, *bc='finite'*, *dtype=None*, *permute=True*, *form='B'*, *legL=None*)

Construct a matrix product state from a set of numpy arrays *Bflat* and singular vals.

Parameters

sites [list of *Site*] The sites defining the local Hilbert space.

Bflat [iterable of numpy ndarrays] The matrix defining the MPS on each site, with legs `'p', 'vL', 'vR'` (physical, virtual left/right).

SVs [list of 1D array | `None`] The singular values on *each* bond. Should always have length *L+1*. By default (`None`), set all singular values to the same value. Entries out of *nontrivial_bonds* are ignored.

bc [{'infinite', 'finite', 'segmemt'}] MPS boundary conditions. See docstring of *MPS*.

dtype [type or string] The data type of the array entries. Defaults to the common dtype of *Bflat*.

permute [bool] The `Site` might permute the local basis states if charge conservation gets enabled. If *permute* is True (default), we permute the given *Bflat* locally according to each site's `perm`. The *p_state* argument should then always be given as if *conserve=None* in the Site.

form [(list of) {`'B' | 'A' | 'C' | 'G' | None` | tuple(float, float)}] Defines the canonical form of *Bflat*. See module doc-string. A single choice holds for all of the entries.

leg_L [LegCharge | `None`] Leg charges at bond 0, which are purely conventional. If `None`, use trivial charges.

Returns

mps [*MPS*] An MPS with the matrices *Bflat* converted to npc arrays.

classmethod from_full(*sites*, *psi*, *form=None*, *cutoff=1e-16*, *normalize=True*, *bc='finite'*, *outer_S=None*)

Construct an MPS from a single tensor *psi* with one leg per physical site.

Performs a sequence of SVDs of psi to split off the *B* matrices and obtain the singular values, the result will be in canonical form. Obviously, this is only well-defined for *finite* or *segment* boundary conditions.

Parameters

sites [list of *Site*] The sites defining the local Hilbert space.

psi [*Array*] The full wave function to be represented as an MPS. Should have labels `'p0', 'p1', ..., 'p{L-1}'`. Additionally, it may have (or must have for 'segment' *bc*) the legs `'vL', 'vR'`, which are trivial for 'finite' *bc*.

form [`'B' | 'A' | 'C' | 'G' | None`] The canonical form of the resulting MPS, see module doc-string. `None` defaults to 'A' form on the first site and 'B' form on all following sites.

cutoff [float] Cutoff of singular values used in the SVDs.

normalize [bool] Whether the resulting MPS should have 'norm' 1.

bc ['finite' | 'segment'] Boundary conditions.

outer_S [None | (array, array)] For 'semgent' *bc* the singular values on the left and right of the considered segment, *None* for 'finite' boundary conditions.

Returns

psi_mps [*MPS*] MPS representation of *psi*, in canonical form and possibly normalized.

classmethod from_singlets(*site*, *L*, *pairs*, *up='up'*, *down='down'*, *lonely=[]*, *lonely_state='up'*, *bc='finite'*)

Create an MPS of entangled singlets.

Parameters

site [*Site*] The *site* defining the local Hilbert space, taken uniformly for all sites.

L [int] The number of sites.

pairs [list of (int, int)] Pairs of sites to be entangled; the returned MPS will have a singlet for each pair in *pairs*.

up, down [int | str] A singlet is defined as `(|up down> - |down up>)/2**0.5`, `up` and `down` give state indices or labels defined on the corresponding site.

lonely [list of int] Sites which are not included into a singlet pair.

lonely_state [int | str] The state for the lonely sites.

bc [{'infinite', 'finite', 'segmemt'}] MPS boundary conditions. See docstring of *MPS*.

Returns

singlet_mps [*MPS*] An MPS representing singlets on the specified pairs of sites.

copy(*self*)

Returns a copy of *self*.

The copy still shares the sites, chinfo, and LegCharges of the B tensors, but the values of B and S are deeply copied.

property L

Number of physical sites; for an iMPS the len of the MPS unit cell.

property dim

List of local physical dimensions.

property finite

Distinguish MPS vs iMPS.

True for an MPS (`bc='finite', 'segment'`), False for an iMPS (`bc='infinite'`).

property chi

Dimensions of the (nontrivial) virtual bonds.

property nontrivial_bonds

Slice of the non-trivial bond indices, depending on `self.bc`.

get_B(*self*, *i*, *form='B'*, *copy=False*, *cutoff=1e-16*, *label_p=None*)

Return (view of) *B* at site *i* in canonical form.

Parameters

i [int] Index choosing the site.

form `['B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)] The (canonical) form of the returned B. For `None`, return the matrix in whatever form it is. If any of the tuple entry is None, also don't scale on the corresponding axis.

copy [bool] Whether to return a copy even if *form* matches the current form.

cutoff [float] During DMRG with a mixer, *S* may be a matrix for which we need the inverse. This is calculated as the Penrose pseudo-inverse, which uses a cutoff for the singular values.

label_p [None | str] Ignored by default (`None`). Otherwise replace the physical label `'p'` with `'p'+label_p'`. (For derived classes with more than one "physical" leg, replace all the physical leg labels accordingly.)

Returns

B [*Array*] The MPS 'matrix' *B* at site *i* with leg labels `'vL', 'p', 'vR'`. May be a view of the matrix (if `copy=False`), or a copy (if the form changed or `copy=True`).

Raises

ValueError [if self is not in canoncial form and *form* is not None.]

set_B (*self*, *i*, *B*, *form='B'*)
Set *B* at site *i*.

Parameters

i [int] Index choosing the site.

B [*Array*] The 'matrix' at site *i*. No copy is made! Should have leg labels `'vL', 'p', 'vR'` (not necessarily in that order).

form [`'B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)] The (canonical) form of the *B* to set. `None` stands for non-canonical form.

get_SL (*self*, *i*)
Return singular values on the left of site *i*

get_SR (*self*, *i*)
Return singular values on the right of site *i*

set_SL (*self*, *i*, *S*)
Set singular values on the left of site *i*

set_SR (*self*, *i*, *S*)
Set singular values on the right of site *i*

get_op (*self*, *op_list*, *i*)
Given a list of operators, select the one corresponding to site *i*.

Parameters

op_list [(list of) {str | npc.array}] List of operators from which we choose. We assume that `op_list[j]` acts on site `j`. If the length is shorter than *L*, we repeat it periodically. Strings are translated using *get_op()* of site *i*.

i [int] Index of the site on which the operator acts.

Returns

op [npc.array] One of the entries in *op_list*, not copied.

get_theta (*self*, *i*, *n=2*, *cutoff=1e-16*, *formL=1.0*, *formR=1.0*)
Calculates the *n*-site wavefunction on `sites[i:i+n]`.

Parameters

i [int] Site index.

n [int] Number of sites. The result lives on `sites[i:i+n]`.

cutoff [float] During DMRG with a mixer, *S* may be a matrix for which we need the inverse. This is calculated as the Penrose pseudo-inverse, which uses a cutoff for the singular values.

formL [float] Exponent for the singular values to the left.

formR [float] Exponent for the singular values to the right.

Returns

theta [*Array*] The n-site wave function with leg labels `vL, p0, p1, .... p{n-1}, vR`. In Vidal's notation (with s=lambda, G=Gamma): `theta = s**form_L G_i s G_{i+1} s ... G_{i+n-1} s**form_R`.

convert_form(*self*, *new_form='B'*)
Tranform self into different canonical form (by scaling the legs with singular values).

Parameters

new_form [(list of) {`'B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)}] The form the stored 'matrices'. The table in module doc-string. A single choice holds for all of the entries.

Raises

ValueError [if trying to convert from a `None` form. Use *canonical_form()* instead!]

increase_L(*self*, *new_L=None*)
Modify *self* inplace to enlarge the unit cell.

For an infinite MPS, we have unit cells.

Parameters

new_L [int] New number of sites. Defaults to twice the number of current sites.

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* inplace to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

See also:

group_split Reverts the grouping.

group_split(*self*, *trunc_par=None*)
Modify *self* inplace to split previously grouped sites.

Parameters

trunc_par [dict] Parameters for truncation, see *truncate()*. Defaults to `{'chi_max': max(self.chi)}`.

Returns

trunc_err [*TruncationError*] The error introduced by the truncation for the splitting.

See also:

group_sites Should have been used before to combine sites.

get_grouped_mps(*self*, *blocklen*)
contract blocklen subsequent tensors into a single one and return result as a new MPS.

blocklen = number of subsequent sites to be combined.

Returns

new MPS object with bunched sites.

get_total_charge(*self*, *only_physical_legs=False*)
Calculate and return the *qtotal* of the whole MPS (when contracted).

Parameters

only_physical_legs [bool] For `'finite'` boundary conditions, the total charge can be gauged away by changing the LegCharge of the trivial legs on the left and right of the MPS. This option allows to project out the trivial legs to get the actual "physical" total charge.

Returns

qtotal [charges] The sum of the *qtotal* of the individual *B* tensors.

gauge_total_charge(*self*, *qtotal=None*, *vL_leg=None*, *vR_leg=None*)
Gauge the legcharges of the virtual bonds such that the MPS has a total *qtotal*.

Parameters

qtotal [(list of) charges] If a single set of charges is given, it is the desired total charge of the MPS (which *get_total_charge()* will return afterwards). By default (`None`), use 0 charges, unless vL_leg and vR_leg are specified, in which case we adjust the total charge to match these legs.

vL_leg [None | LegCharge] Desired new virtual leg on the very left. Needs to have the same block strucuture as current leg, but can have shifted charge entries.

vR_leg [None | LegCharge] Desired new virtual leg on the very right. Needs to have the same block strucuture as current leg, but can have shifted charge entries. Should be *vL_leg.conj()* for infinite MPS, if *qtotal* is not given.

entanglement_entropy(*self*, *n=1*, *bonds=None*, *for_matrix_S=False*)
Calculate the (half-chain) entanglement entropy for all nontrivial bonds.

Consider a bipartition of the sytem into $A = \{j : j <= i_b\}$ and $B = \{j : j > i_b\}$ and the reduced density matrix $\rho_A = tr_B(\rho)$. The von-Neumann entanglement entropy is defined as $S(A, n = 1) = -tr(\rho_A \log(\rho_A)) = S(B, n = 1)$. The generalization for `n != 1, n>0` are the Renyi entropies: $S(A, n) = \frac{1}{1-n} \log(tr(\rho_A^2)) = S(B, n = 1)$

This function calculates the entropy for a cut at different bonds *i*, for which the the eigenvalues of the reduced density matrix ρ_A and ρ_B is given by the squared schmidt values *S* of the bond.

Parameters

n [int/float] Selects which entropy to calculate; *n=1* (default) is the ususal von-Neumann entanglement entropy.

bonds [`None` | (iterable of) int] Selects the bonds at which the entropy should be calculated. `None` defaults to `range(0, L+1)[self.nontrivial_bonds]`.

for_matrix_S [bool] Switch calculate the entanglement entropy even if the *_S* are matrices. Since $O(\chi^3)$ is expensive compared to the ususal $O(\chi)$, we raise an error by default.

Returns

entropies [1D ndarray] Entanglement entropies for half-cuts. *entropies[j]* contains the entropy for a cut at bond `bonds[j]` (i.e. left to site `bonds[j]`).

entanglement_entropy_segment(*self, segment=[0], first_site=None, n=1*)
Calculate entanglement entropy for general geometry of the bipartition.

This function is similar as *entanglement_entropy()*, but for more general geometry of the region *A* to be a segment of a *few* sites.

This is acchieved by explicitly calculating the reduced density matrix of *A* and thus works only for small segments.

Parameters

segment [list of int] Given a first site *i*, the region `A_i` is defined to be `[i+j for j in segment]`.

first_site [`None` | (iterable of) int] Calculate the entropy for segments starting at these sites. `None` defaults to `range(L-segment[-1])` for finite or *range(L)* for infinite boundary conditions.

n [int | float] Selects which entropy to calculate; *n=1* (default) is the ususal von-Neumann entanglement entropy, otherwise the *n*-th Renyi entropy.

Returns

entropies [1D ndarray] `entropies[i]` contains the entropy for the the region `A_i` defined above.

entanglement_spectrum(*self*, *by_charge=False*)
return entanglement energy spectrum.

Parameters

by_charge [bool] Wheter we should sort the spectrum on each bond by the possible charges.

Returns

ent_spectrum [list] For each (non-trivial) bond the entanglement spectrum. If *by_charge* is `False`, return (for each bond) a sorted 1D ndarray with the convention $S_i^2 = e^{-\xi_i}$, where S_i labels a Schmidt value and ξ_i labels the entanglement 'energy' in the returned spectrum. If *by_charge* is True, return a a list of tuples `(charge, sub_spectrum)` for each possible charge on that bond.

get_rho_segment(*self*, *segment*)
Return reduced density matrix for a segment.

Note that the dimension of rho_A scales exponentially in the length of the segment.

Parameters

segment [iterable of int] Sites for which the reduced density matrix is to be calculated. Assumed to be sorted.

Returns

rho [*Array*] Reduced density matrix of the segment sites. Labels `'p0', 'p1', ..., 'pk', 'p0*', 'p1*', ..., 'pk*'` with `k=len(segment)`.

probability_per_charge(*self*, *bond=0*)
Return probabilites of charge value on the left of a given bond.

For example for particle number conservation, define $N_b = sum_{i<b} n_i$ for a given bond b. This function returns the possible values of *N_b* as rows of *charge_values*, and for each row the probabilty that this combination occurs in the given state.

Parameters

bond [int] The bond to be considered. The returned charges are summed on the left of this bond.

Returns

charge_values [2D array] Columns correspond to the different charges in *self.chinfo*. Rows are the different charge fluctuations at this bond

probabilities [1D array] For each row of *charge_values* the probablity for these values of charge fluctuations.

average_charge(*self*, *bond=0*)

Return the average charge for the block on the left of a given bond.

For example for particle number conservation, define $N_b = sum_{i<b} n_i$ for a given bond b. Then this function returns $< \psi | N_b | \psi >$.

Parameters

bond [int] The bond to be considered. The returned charges are summed over the sites left of *bond*.

Returns

average_charge [1D array] For each type of charge in `chinfo` the average value when summing the charge values over sites left of the given bond.

charge_variance(*self*, *bond=0*)

Return the charge variance on the left of a given bond.

For example for particle number conservation, define $N_b = sum_{i<b} n_i$ for a given bond b. Then this function returns $< \psi | N_b^2 | \psi > - (< \psi | N_b | \psi >)^2$.

Parameters

bond [int] The bond to be considered. The returned charges are summed over the sites left of *bond*.

Returns

average_charge [1D array] For each type of charge in `chinfo` the variance of of the charge values left of the given bond.

mutinf_two_site(*self*, *max_range=None*, *n=1*)

Calculate the two-site mutual information $I(i:j)$.

Calculates $I(i:j) = S(i) + S(j) - S(i,j)$, where $S(i)$ is the single site entropy on site i and $S(i,j)$ the two-site entropy on sites i, j.

Parameters

max_range [int] Maximal distance `|i-j|` for which the mutual information should be calculated. `None` defaults to *L-1*.

n [float] Selects the entropy to use, see *entropy()*.

Returns

coords [2D array] Coordinates for the mutinf array.

mutinf [1D array] `mutinf[k]` is the mutual information $I(i:j)$ between the sites `i, j = coords[k]`.

overlap (*self*, *other*, *charge_sector=0*, *ignore_form=False*, ***kwargs*)
Compute overlap `<self|other>`.

Parameters

other [*MPS*] An MPS with the same physical sites.

charge_sector [None | charges | `0`] Selects the charge sector in which the dominant eigenvector of the TransferMatrix is. `None` stands for *all* sectors, `0` stands for the zero-charge sector. Defaults to `0`, i.e., *assumes* the dominant eigenvector is in charge sector 0.

ignore_form [bool] If `False` (default), take into account the canonical form `form` at each site. If `True`, we ignore the canonical form (i.e., whether the MPS is in left, right, mixed or no canonical form) and just contract all the `_B` as they are. (This can give different results!)

****kwargs :** Further keyword arguments given to *TransferMatrix.eigenvectors()*; only used for infinite boundary conditions.

Returns

overlap [dtype.type] The contraction `<self|other> * self.norm * other.norm` (i.e., taking into account the `norm` of both MPS). For an infinite MPS, `<self|other>` is the overlap per unit cell, i.e., the largest eigenvalue of the TransferMatrix.

expectation_value (*self*, *ops*, *sites=None*, *axes=None*)
Expectation value `<psi|ops|psi>/<psi|psi>` of (n-site) operator(s).

Given the MPS in canonical form, it calculates n-site expectation values. For example the contraction for a two-site ($n = 2$) operator on site i would look like:

```
|           .--S--B[i]--B[i+1]--.
|           |     |     |       |
|           |     |-----|       |
|           |     | op  |       |
|           |     |-----|       |
|           |     |     |       |
|           .--S--B*[i]-B*[i+1]-.
```

Parameters

ops [(list of) { *Array* | str }] The operators, for wich the expectation value should be taken, All operators should all have the same number of legs (namely *2 n*). If less than *self.L* operators are given, we repeat them periodically. Strings (like `'Id', 'Sz'`) are translated into single-site operators defined by `sites`.

sites [None | list of int] List of site indices. Expectation values are evaluated there. If `None` (default), the entire chain is taken (clipping for finite b.c.)

axes [None | (list of str, list of str)] Two lists of each n leg labels giving the physical legs of the operator used for contraction. The first n legs are contracted with conjugated B, the second n legs with the non-conjugated B. `None` defaults to `(['p'], ['p*'])` for single site operators ($n = 1$), or `(['p0', 'p1', ... 'p{n-1}'], ['p0*', 'p1*', .... 'p{n-1}*'])` for $n > 1$.

Returns

exp_vals [1D ndarray] Expectation values, `exp_vals[i] = <psi|ops[i]|psi>`, where `ops[i]` acts on site(s) `j, j+1, ..., j+{n-1}` with `j=sites[i]`.

Examples

One site examples (n=1):

```
>>> psi.expectation_value('Sz')
[Sz0, Sz1, ..., Sz{L-1}]
>>> psi.expectation_value(['Sz', 'Sx'])
[Sz0, Sx1, Sz2, Sx3, ... ]
>>> psi.expectation_value('Sz', sites=[0, 3, 4])
[Sz0, Sz3, Sz4]
```

Two site example (n=2), assuming homogeneous sites:

```
>>> SzSx = npc.outer(psi.sites[0].Sz.replace_labels(['p', 'p*'], ['p0', 'p0*
↪']),
                     psi.sites[1].Sx.replace_labels(['p', 'p*'], ['p1', 'p1*
↪']))
>>> psi.expectation_value(SzSx)
[Sz0Sx1, Sz1Sx2, Sz2Sx3, ... ]   # with len L-1 for finite bc, or L for␣
↪infinite
```

Example measuring <psi|SzSx|psi2> on each second site, for inhomogeneous sites:

```
>>> SzSx_list = [npc.outer(psi.sites[i].Sz.replace_labels(['p', 'p*'], ['p0',
↪'p0*']),
                           psi.sites[i+1].Sx.replace_labels(['p', 'p*'], ['p1
↪', 'p1*']))
                  for i in range(0, psi.L-1, 2)]
>>> psi.expectation_value(SzSx_list, range(0, psi.L-1, 2))
[Sz0Sx1, Sz2Sx3, Sz4Sx5, ...]
```

expectation_value_term(*self*, *term*, *autoJW=True*)

Expectation value `<psi|op_{i0}op_{i1}...op_{iN}|psi>/<psi|psi>`.

Calculates the expectation value of a tensor product of single-site operators acting on different sites *i0*, *i1*, ... (not necessarily next to each other). In other words, evaluate the expectation value of a term `op0_i0 op1_i1 op2_i2 ...`.

For example the contraction of three one-site operators on sites *i0*, *i1=i0+1*, *i2=i0+3* would look like:

```
|          .--S--B[i0]---B[i0+1]--B[i0+2]--B[i0+3]--.
|          |     |        |         |         |        |
|          |    op1      op2        |        op3       |
|          |     |        |         |         |        |
|          .--S--B*[i0]--B*[i0+1]-B*[i0+2]-B*[i0+3]-.
```

Parameters

term [list of (str, int)] List of tuples `op, i` where *i* is the MPS index of the site the operator named *op* acts on. The order inside *term* determines the order in which they act (in the mathematical convention: the last operator in *term* is right-most, so it acts first on a Ket).

autoJW [bool] If True (default), automatically insert Jordan Wigner strings for Fermions as needed.

Returns

exp_val [float/complex] The expectation value of the tensorproduct of the given onsite operators, `<psi|op_i0 op_i1 ... op_iN |psi>/<psi|psi>`, where `|psi>` is the represented MPS.

See also:

correlation_function efficient way to evaluate many correlation functions.

Examples

```
>>> a = psi.expectation_value_term([('Sx', 2), ('Sz', 4)])
>>> b = psi.expectation_value_term([('Sz', 4), ('Sx', 2)])
>>> c = psi.expectation_value_multi_sites(['Sz', 'Id', 'Sz'], i0=2)
>>> assert a == b == c
```

expectation_value_multi_sites (*self*, *operators*, *i0*)
Expectation value `<psi|op0_{i0}op1_{i0+1}...opN_{i0+N}|psi>/<psi|psi>`.

Calculates the expectation value of a tensor product of single-site operators acting on different sites next to each other. In other words, evaluate the expectation value of a term `op0_i0 op1_{i0+1} op2_{i0+2} ....`

Parameters

operators [List of { *Array* | str }] List of one-site operators. This method calculates the expectation value of the n-sites operator given by their tensor product.

i0 [int] The left most index on which an operator acts, i.e., `operators[i]` acts on site `i + i0`.

Returns

exp_val [float/complex] The expectation value of the tensorproduct of the given onsite operators, `<psi|operators[0]_{i0} operators[1]_{i0+1} ... |psi>/<psi|psi>`, where `|psi>` is the represented MPS.

expectation_value_terms_sum (*self*, *term_list*, *prefactors=None*)
Calculate expectation values for a bunch of terms and sum them up.

This is equivalent to the following expression:

```
sum([self.expectation_value_term(term)*strength for term, strength in term_
↪list])
```

However, for effiency, the term_list is converted to an MPO and the expectation value of the MPO is evaluated.

Note: Due to the way MPO expectation values are evaluated for infinite systems, it works only if all terms in the *term_list* start within the MPS unit cell.

Deprecated since version 0.4.0: *prefactor* will be removed in version 1.0.0. Instead, directly give just `TermList(term_list, prefactors)` as argument.

Parameters

term_list [*TermList*] The terms and prefactors (*strength*) to be summed up.

prefactors : Instead of specifying a *TermList*, one can also specify the term_list and strength separately. This is deprecated.

Returns

terms_sum [list of (complex) float] Equivalent to the expression `sum([self.expectation_value_term(term)*strength for term, strength in term_list])`.

_mpo : Intermediate results: the generated MPO. For a finite MPS, `terms_sum = _mpo.expectation_value(self)`, for an infinite MPS `terms_sum = _mpo.expectation_value(self) * self.L`

See also:

expectation_value_term evaluates a single *term*.

tenpy.networks.mpo.MPO.expectation_value expectation value density of an MPO.

correlation_function(*self*, *ops1*, *ops2*, *sites1=None*, *sites2=None*, *opstr=None*, *str_on_first=True*, *hermitian=False*)

Correlation function `<psi|op1_i op2_j|psi>/<psi|psi>` of single site operators.

Given the MPS in canonical form, it calculates 2-site correlation functions. For examples the contraction for a two-site operator on site *i* would look like:

```
|           .--S--B[i]--B[i+1]--...--B[j]---.
|           |      |      |             |       |
|           |      |      |             op2     |
|           |      op1    |             |       |
|           |      |      |             |       |
|           .--S--B*[i]-B*[i+1]-...--B*[j]--.
```

Onsite terms are taken in the order `<psi | op1 op2 | psi>`.

If *opstr* is given and `str_on_first=True`, it calculates:

```
|            for i < j                                  for i > j
|
|           .--S--B[i]---B[i+1]--...- B[j]---.      .--S--B[j]---B[j+1]--...-␣
↪B[i]---.
|           |      |      |             |       |      |      |      |             |␣
↪      |
|           |      opstr  opstr         op2     |      |      op2    |             |␣
↪      |
|           |      |      |             |       |      |      |      |             |␣
↪      |
|           |      op1    |             |       |      |      opstr  opstr         ␣
↪op1    |
|           |      |      |             |       |      |      |      |             |␣
↪      |
|           .--S--B*[i]--B*[i+1]-...- B*[j]--.      .--S--B*[j]--B*[j+1]-...-␣
↪B*[i]--.
```

For `i==j`, no *opstr* is included. For `str_on_first=False`, the *opstr* on site `min(i, j)` is always left out.

Strings (like `'Id', 'Sz'`) in the arguments are translated into single-site operators defined by the *Site* on which they act. Each operator should have the two legs `'p', 'p*'`.

Parameters

ops1 [(list of) { *Array* | str }] First operator of the correlation function (acting after ops2). `ops1[x]` acts on site `sites1[x]`. If less than `len(sites1)` operators are given, we repeat them periodically.

ops2 [(list of) { *Array* | str }] Second operator of the correlation function (acting before ops1). `ops2[y]` acts on site `sites2[y]`. If less than `len(sites2)` operators are given, we repeat them periodically.

sites1 [None | int | list of int] List of site indices; a single *int* is translated to `range(0, sites1)`. `None` defaults to all sites `range(0, L)`. Is sorted before use, i.e. the order is ignored.

sites2 [None | int | list of int] List of site indices; a single *int* is translated to `range(0, sites2)`. `None` defaults to all sites `range(0, L)`. Is sorted before use, i.e. the order is ignored.

opstr [None | (list of) { *Array* | str }] Ignored by default (`None`). Operator(s) to be inserted between `ops1` and `ops2`. If less than *L* operators are given, we repeat them periodically. If given as a list, `opstr[r]` is inserted at site *r* (independent of *sites1* and *sites2*).

str_on_first [bool] Whether the *opstr* is included on the site `min(i, j)`. Note the order, which is chosen that way to handle fermionic Jordan-Wigner strings correctly. (In other words: choose `str_on_first=True` for fermions!)

hermitian [bool] Optimization flag: if `sites1 == sites2` and `Ops1[i]^\dagger == Ops2[i]` (which is not checked explicitly!), the resulting `C[x, y]` will be hermitian. We can use that to avoid calculations, so `hermitian=True` will run faster.

Returns

C [2D ndarray] The correlation function `C[x, y] = <psi|ops1[i] ops2[j]|psi>`, where `ops1[i]` acts on site `i=sites1[x]` and `ops2[j]` on site `j=sites2[y]`. If *opstr* is given, it gives (for `str_on_first=True`):

- For `i < j`: `C[x, y] = <psi|ops1[i] prod_{i <= r < j} opstr[r] ops2[j]|psi>`.
- For `i > j`: `C[x, y] = <psi|prod_{j <= r < i} opstr[r] ops1[i] ops2[j]|psi>`.
- For `i = j`: `C[x, y] = <psi|ops1[i] ops2[j]|psi>`.

The condition `<= r` is replaced by a strict `< r`, if `str_on_first=False`.

norm_test(*self*)

Check that self is in canonical form.

Returns

norm_error: array, shape (L, 2) For each site the norm error to the left and right. The error `norm_error[i, 0]` is defined as the norm-difference between the following networks:

```
|   --theta[i]---.       --s[i]--.
|       |        |   vs         |
|   --theta*[i]--.       --s[i]--.
```

Similarly, `norm_errror[i, 1]` is the norm-difference of:

```
|   .--theta[i]---         .--s[i+1]--
|   |    |          vs     |
|   .--theta*[i]--         .--s[i+1]--
```

canonical_form(*self*, *renormalize=True*)
Bring self into canonical 'B' form, (re-)calculate singular values.

Simply calls *canonical_form_finite()* or *canonical_form_infinite()*.

canonical_form_finite(*self*, *renormalize=True*, *cutoff=0.0*)
Bring a finite (or segment) MPS into canonical form (in place).

If any site is in `form` `None`, it does *not* use any of the singular values *S* (for 'finite' boundary conditions, or only the very left *S* for 'segment' b.c.). If all sites have a *form*, it respects the *form* to ensure that one *S* is included per bond. The final state is always in right-canonical 'B' form.

Performs one sweep left to right doing QR decompositions, and one sweep right to left doing SVDs calculating the singular values.

Parameters

renormalize: bool Whether a change in the norm should be discarded or used to update `norm`.

cutoff [float | None] Cutoff of singular values used in the SVDs.

Returns

U_L, V_R [*Array*] Only returned for `'segment'` boundary conditions. The unitaries defining the new left and right Schmidt states in terms of the old ones, with legs `'vL'`, `'vR'`.

canonical_form_infinite(*self*, *renormalize=True*, *tol_xi=1000000.0*)
Bring an infinite MPS into canonical form (in place).

If any site is in `form` `None`, it does *not* use any of the singular values *S*. If all sites have a *form*, it respects the *form* to ensure that one *S* is included per bond. The final state is always in right-canonical 'B' form.

Proceeds in three steps, namely 1) diagonalize right and left transfermatrix on a given bond to bring that bond into canonical form, and then 2) sweep right to left, and 3) left to right to bringing other bonds into canonical form.

Parameters

renormalize: bool Whether a change in the norm should be discarded or used to update `norm`.

tol_xi [float] Raise an error if the correlation length is larger than that (which indicates a degenerate "cat" state, e.g., for spontaneous symmetry breaking).

correlation_length(*self*, *target=1*, *tol_ev0=1e-08*, *charge_sector=0*)
Calculate the correlation length by diagonalizing the transfer matrix.

Assumes that *self* is in canonical form.

Works only for infinite MPS, where the transfer matrix is a useful concept. Assuming a single-site unit cell, any correlation function splits into $C(A_i, B_j) = A_i' T^{j-i-1} B_j'$ with some parts left and right and the $j-i-1$-th power of the transfer matrix in between. The largest eigenvalue is 1 (if self is properly normalized) and gives the dominant contribution of $A_i' E_1 * 1^{j-i-1} * E_1^T B_j' =< A >< B >$, and the second largest one gives a contribution $\propto \lambda_2^{j-i-1}$. Thus $\lambda_2 = \exp(-\frac{1}{\xi})$.

More general for a *L*-site unit cell we get $\lambda_2 = \exp(-\frac{L}{\xi})$, where the *xi* is given in units of 1 lattice spacing in the MPS.

Warning: For a higher-dimensional lattice (which the MPS class doesn't know about), the correct unit is the lattice spacing in x-direction, and the correct formula is $\lambda_2 = \exp(-\frac{L_x}{\xi})$, where *L_x* is the number of lattice spacings in the infinite direction within the MPS unit cell, e.g. the number of "rings" of a cylinder in the MPS unit cell. To get to these units, divide the returned *xi* by the number of sites within a "ring", for a lattice given in `N_sites_per_ring`.

Parameters

target [int] We look for the *target* + 1 largest eigenvalues.

tol_ev0 [float] Print warning if largest eigenvalue deviates from 1 by more than *tol_ev0*.

charge_sector [None | charges | `0`] Selects the charge sector in which the dominant eigenvector of the TransferMatrix is. `None` stands for *all* sectors, `0` stands for the zero-charge sector. Defaults to `0`, i.e., *assumes* the dominant eigenvector is in charge sector 0.

Returns

xi [float | 1D array] If *target`=1, return just the correlation length, otherwise an array of the `target* largest correlation lengths. It is measured in units of a single lattice spacing in the MPS language, see the warning above.

add(*self*, *other*, *alpha*, *beta*, *cutoff=1e-15*)
Return an MPS which represents `alpha|self> + beta |others>`.

Works only for 'finite', 'segment' boundary conditions. For 'segment' boundary conditions, the virtual legs on the very left/right are assumed to correspond to each other (i.e. self and other have the same state outside of the considered segment). Takes into account `norm`.

Parameters

other [*MPS*] Another MPS of the same length to be added with self.

alpha, beta [complex float] Prefactors for self and other. We calculate `alpha * |self> + beta * |other>`

cutoff [float | None] Cutoff of singular values used in the SVDs.

Returns

sum [*MPS*] An MPS representing `alpha|self> + beta |other>`. Has same total charge as *self*.

U_L, V_R [*Array*] Only returned for `'segment'` boundary conditions. The unitaries defining the new left and right Schmidt states in terms of the old ones, with legs `'vL', 'vR'`.

apply_local_op(*self*, *i*, *op*, *unitary=None*, *renormalize=False*, *cutoff=1e-13*)
Apply a local (one or multi-site) operator to *self*.

Note that this destroys the canonical form if the local operator is non-unitary. Therefore, this function calls *canonical_form()* if necessary.

Parameters

i [int] (Left-most) index of the site(s) on which the operator should act.

op [str | npc.Array] A physical operator acting on site *i*, with legs `'p', 'p*'` for a single-site operator or with legs `['p0', 'p1', ...], ['p0*', 'p1*', ...]` for an operator acting on *n*>=2 sites. Strings (like `'Id', 'Sz'`) are translated into single-site operators defined by `sites`.

unitary [None | bool] Whether *op* is unitary, i.e., whether the canonical form is preserved (`True`) or whether we should call *canonical_form()* (`False`). `None` checks whether `norm(op dagger(op) - identity)` is smaller than *cutoff*.

renormalize [bool] Whether the final state should keep track of the norm (False, default) or be renormalized to have norm 1 (True).

cutoff [float] Cutoff for singular values if *op* acts on more than one site (see *from_full()*). (And used as cutoff for a unspecified *unitary*.)

swap_sites(*self*, *i*, *swap_op='auto'*, *trunc_par={}*)
Swap the two neighboring sites *i* and *i+1* (inplace).

Exchange two neighboring sites: form theta, 'swap' the physical legs and split with an svd. While the 'swap' is just a transposition/relabeling for bosons, one needs to be careful about the sign for fermions.

Parameters

i [int] Swap the two sites at positions *i* and *i+1*.

swap_op [`None` | `'auto'` | *Array*] The operator used to swap the phyiscal legs of the two-site wave function *theta*. For `None`, just transpose/relabel the legs, for `'auto'` also take care of fermionic signs. Alternative give an npc *Array* which represents the full operator used for the swap. Should have legs `['p0', 'p1', 'p0*', 'p1*']` whith `'p0', 'p1*'` contractible.

trunc_par [dict] Parameters for truncation, see *truncate()*. *chi_max* defaults to `max(self.chi)`.

Returns

trunc_err [*TruncationError*] The error of the represented state introduced by the truncation after the swap.

permute_sites(*self*, *perm*, *swap_op='auto'*, *trunc_par={}*, *verbose=0*)
Applies the permutation perm to the state (inplace).

Parameters

perm [ndarray[ndim=1, int]] The applied permutation, such that `psi.permute_sites(perm)[i] = psi[perm[i]]` (where `[i]` indicates the *i*-th site).

swap_op [`None` | `'auto'` | *Array*] The operator used to swap the phyiscal legs of a two-site wave function *theta*, see *swap_sites()*.

trunc_par [dict] Parameters for truncation, see *truncate()*. *chi_max* defaults to `max(self.chi)`.

verbose [float] Level of verbosity, print status messages if verbose > 0.

Returns

trunc_err [*TruncationError*] The error of the represented state introduced by the truncation after the swaps.

compute_K(*self*, *perm*, *swap_op='auto'*, *trunc_par=None*, *canonicalize=1e-06*, *verbose=0*)
Compute the momentum quantum numbers of the entanglement spectrum for 2D states.

Works for an infinite MPS living on a cylinder, infinitely long in *x* direction and with periodic boundary conditions in *y* directions. If the state is invariant under 'rotations' around the cylinder axis, one can find the momentum quantum numbers of it. (The rotation is nothing more than a translation in *y*.) This function permutes some sites (on a copy of *self*) to enact the rotation, and then finds the dominant eigenvector of

the mixed transfer matrix to get the quantum numbers, along the lines of [PollmannTurner2012], see also (the appendix and Fig. 11 in the arXiv version of) [CincioVidal2013].

Parameters

perm [1D ndarray | *Lattice*] Permuation to be applied to the physical indices, see *permute_sites()*. If a lattice is given, we use it to read out the lattice structure and shift each site by one lattice-vector in y-direction (assuming periodic boundary conditions). (If you have a *CouplingModel*, give its *lat* attribute for this argument)

swap_op [None | 'auto' | *Array*] The operator used to swap the phyiscal legs of a two-site wave function *theta*, see *swap_sites()*.

trunc_par [dict] Parameters for truncation, see *truncate()*. If not set, *chi_max* defaults to max(self.chi).

canonicalize [float] Check that *self* is in canonical form; call *canonical_form()* if *norm_test()* yields np.linalg.norm(self.norm_test()) > canonicalize.

verbose [float] Level of verbosity, print status messages if verbose > 0.

Returns

U [*Array*] Unitary representation of the applied permutation on left Schmidt states.

W [ndarray] 1D array of the form S**2 exp(i K), where *S* are the Schmidt values on the left bond. You can use np.abs() and np.angle() to extract the Schmidt values *S* and momenta *K* from *W*.

q [*LegCharge*] LegCharge corresponding to *W*.

ov [complex] The eigenvalue of the mixed transfer matrix <*psi*|*T*|*psi*> per *L* sites. An absolute value different smaller than 1 indicates that the state is not invariant under the permutation or that the truncation error *trunc_err* was too large!

trunc_err [*TruncationError*] The error of the represented state introduced by the truncation after swaps when performing the truncation.

MPSEnvironment

- full name: tenpy.networks.mps.MPSEnvironment
- parent module: *tenpy.networks.mps*
- type: class

class tenpy.networks.mps.**MPSEnvironment**(*bra*, *ket*, *init_LP=None*, *init_RP=None*, *age_LP=0*, *age_RP=0*)

Bases: object

Stores partial contractions of $< bra|Op|ket >$ for local operators *Op*.

The network for a contraction $< bra|Op|ket >$ of a local operator *Op*, say exemplary at sites *i, i+1* looks like:

```
|      .-----M[0]--- ... --M[1]---M[2]--- ... ->--.
|      |      |              |       |               |
|      |      |              |-------|               |
|      LP[0]  |              |  Op   |              RP[-1]
|      |      |              |-------|               |
|      |      |              |       |               |
|      .-----N[0]*-- ... --N[1]*--N[2]*-- ... -<--.
```

Of course, we can also calculate the overlap <*bra*|*ket*> by using the special case `Op = Id`.

We use the following label convention (where arrows indicate *qconj*):

```
|     .-->- vR            vL ->-.
|     |                         |
|     LP                        RP
|     |                         |
|     .--<- vR*          vL* -<-.
```

To avoid recalculations of the whole network e.g. in the DMRG sweeps, we store the contractions up to some site index in this class. For `bc='finite','segment'`, the very left and right part `LP[0]` and `RP[-1]` are trivial and don't change, but for `bc='infinite'` they are might be updated (by inserting another unit cell to the left/right).

The MPS *bra* and *ket* have to be in canonical form. All the environments are constructed without the singular values on the open bond. In other words, we contract left-canonical *A* to the left parts *LP* and right-canonical *B* to the right parts *RP*. Thus, the special case `ket=bra` should yield identity matrices for *LP* and *RP*.

Parameters

bra [*MPS*] The MPS to project on. Should be given in usual 'ket' form; we call *conj()* on the matrices directly. Stored in place, without making copies. If necessary to match charges, we call *gauge_total_charge()*.

ket [*MPO* | None] The MPS on which the local operator acts. Stored in place, without making copies. If `None`, use *bra*.

init_LP [`None` | *Array*] Initial very left part `LP`. If `None`, build trivial one with *init_LP()*.

init_RP [`None` | *Array*] Initial very right part `RP`. If `None`, build trivial one with *init_RP()*.

age_LP [int] The number of physical sites involved into the contraction yielding *firstLP*.

age_RP [int] The number of physical sites involved into the contraction yielding *lastRP*.

Attributes

L [int] Number of physical sites involved into the Environment, i.e. the least common multiple of `bra.L` and `ket.L`.

bra, ket [*MPS*] The two MPS for the contraction.

dtype [type] The data type.

_finite [bool] Whether the boundary conditions of the MPS are finite.

_LP [list of {`None` | *Array*}] Left parts of the environment, len *L*. `LP[i]` contains the contraction strictly left of site *i* (or `None`, if we don't have it calculated).

_RP [list of {`None` | *Array*}] Right parts of the environment, len *L*. `RP[i]` contains the contraction strictly right of site *i* (or `None`, if we don't have it calculated).

_LP_age [list of int | `None`] Used for book-keeping, how large the DMRG system grew: `_LP_age[i]` stores the number of physical sites invovled into the contraction network which yields `self._LP[i]`.

_RP_age [list of int | `None`] Used for book-keeping, how large the DMRG system grew: `_RP_age[i]` stores the number of physical sites invovled into the contraction network which yields `self._RP[i]`.

Methods

del_LP(self, i)	Delete stored part strictly to the left of site *i*.
del_RP(self, i)	Delete storde part scrictly to the right of site *i*.
expectation_value(self, ops[, sites, axes])	Expectation value `<bra|ops|ket>` of (n-site) operator(s).
full_contraction(self, i0)	Calculate the overlap by a full contraction of the network.
get_LP(self, i[, store])	Calculate LP at given site from nearest available one (including *i*).
get_LP_age(self, i)	Return number of physical sites in the contractions of get_LP(i).
get_RP(self, i[, store])	Calculate RP at given site from nearest available one (including *i*).
get_RP_age(self, i)	Return number of physical sites in the contractions of get_RP(i).
init_LP(self, i)	Build initial left part `LP`.
init_RP(self, i)	Build initial right part `RP` for an MPS/MPOEnvironment.
set_LP(self, i, LP, age)	Store part to the left of site *i*.
set_RP(self, i, RP, age)	Store part to the right of site *i*.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

init_LP(*self*, *i*)
Build initial left part `LP`.

Parameters

i [int] Build `LP` left of site *i*.

Returns

init_LP [*Array*] Identity contractible with the *vL* leg of `ket.get_B(i)`, labels `'vR*'`, `'vR'`.

init_RP(*self*, *i*)
Build initial right part `RP` for an MPS/MPOEnvironment.

Parameters

i [int] Build `RP` right of site *i*.

Returns

init_RP [*Array*] Identity contractible with the *vR* leg of `ket.get_B(i)`, labels `'vL*'`, `'vL'`.

get_LP(*self*, *i*, *store=True*)
Calculate LP at given site from nearest available one (including *i*).

The returned `LP_i` corresponds to the following contraction, where the M's and the N's are in the 'A' form:

```
|      .-------M[0]--- ... --M[i-1]--->-    'vR'
|      |        |              |
|      LP[0]    |              |
|      |        |              |
|      .-------N[0]*-- ... --N[i-1]*--<-    'vR*'
```

Parameters

i [int] The returned *LP* will contain the contraction *strictly* left of site *i*.

store [bool] Wheter to store the calculated *LP* in *self* (`True`) or discard them (`False`).

Returns

LP_i [*Array*] Contraction of everything left of site *i*, with labels `'vR*', 'vR'` for *bra*, *ket*.

get_RP (*self*, *i*, *store=True*)

Calculate RP at given site from nearest available one (including *i*).

The returned `RP_i` corresponds to the following contraction, where the M's and the N's are in the 'B' form:

```
|      'vL'  ->---M[i+1]-- ... --M[L-1]----.
|                |              |          |
|                |              |         RP[-1]
|                |              |          |
|      'vL*' -<---N[i+1]*- ... --N[L-1]*---.
```

Parameters

i [int] The returned *RP* will contain the contraction *strictly* right of site *i*.

store [bool] Wheter to store the calculated *RP* in *self* (`True`) or discard them (`False`).

Returns

RP_i [*Array*] Contraction of everything left of site *i*, with labels `'vL*', 'vL'` for *bra*, *ket*.

get_LP_age (*self*, *i*)

Return number of physical sites in the contractions of get_LP(i).

Might be `None`.

get_RP_age (*self*, *i*)

Return number of physical sites in the contractions of get_RP(i).

Might be `None`.

set_LP (*self*, *i*, *LP*, *age*)

Store part to the left of site *i*.

set_RP (*self*, *i*, *RP*, *age*)

Store part to the right of site *i*.

del_LP (*self*, *i*)

Delete stored part strictly to the left of site *i*.

del_RP (*self*, *i*)

Delete storde part scrictly to the right of site *i*.

full_contraction(*self*, *i0*)
Calculate the overlap by a full contraction of the network.

The full contraction of the environments gives the overlap `<bra|ket>`, taking into account `MPS.norm` of both *bra* and *ket*. For this purpose, this function contracts `get_LP(i0+1, store=False)` and `get_RP(i0, store=False)` with appropriate singular values in between.

Parameters

i0 [int] Site index.

expectation_value(*self*, *ops*, *sites=None*, *axes=None*)
Expectation value `<bra|ops|ket>` of (n-site) operator(s).

Calculates n-site expectation values of operators sandwiched between bra and ket. For examples the contraction for a two-site operator on site *i* would look like:

```
|           .--S--B[i]--B[i+1]--.
|           |      |      |      |
|           |      |------|      |
|           LP[i]  |  op  |      RP[i+1]
|           |      |------|      |
|           |      |      |      |
|           .--S--B*[i]-B*[i+1]-.
```

Here, the *B* are taken from *ket*, the *B** from *bra*. The call structure is the same as for *MPS.expectation_value()*.

Parameters

ops [(list of) { *Array* | str }] The operators, for wich the expectation value should be taken, All operators should all have the same number of legs (namely *2 n*). If less than `len(sites)` operators are given, we repeat them periodically. Strings (like `'Id'`, `'Sz'`) are translated into single-site operators defined by `sites`.

sites [list] List of site indices. Expectation values are evaluated there. If `None` (default), the entire chain is taken (clipping for finite b.c.)

axes [None | (list of str, list of str)] Two lists of each *n* leg labels giving the physical legs of the operator used for contraction. The first *n* legs are contracted with conjugated *B*, the second *n* legs with the non-conjugated *B*. `None` defaults to `(['p'], ['p*'])` for single site (n=1), or `(['p0', 'p1', ... 'p{n-1}'], ['p0*', 'p1*', ... 'p{n-1}*'])` for $n > 1$.

Returns

exp_vals [1D ndarray] Expectation values, `exp_vals[i] = <bra|ops[i]|ket>`, where `ops[i]` acts on site(s) `j, j+1, ..., j+{n-1}` with `j=sites[i]`.

Examples

One site examples (n=1):

```
>>> env.expectation_value('Sz')
[Sz0, Sz1, ..., Sz{L-1}]
>>> env.expectation_value(['Sz', 'Sx'])
[Sz0, Sx1, Sz2, Sx3, ... ]
>>> env.expectation_value('Sz', sites=[0, 3, 4])
[Sz0, Sz3, Sz4]
```

Two site example (n=2), assuming homogeneous sites:

```
>>> SzSx = npc.outer(psi.sites[0].Sz.replace_labels(['p', 'p*'], ['p0', 'p0*
↪']),
                     psi.sites[1].Sx.replace_labels(['p', 'p*'], ['p1', 'p1*
↪']))
>>> env.expectation_value(SzSx)
[Sz0Sx1, Sz1Sx2, Sz2Sx3, ... ]   # with len L-1 for finite bc, or L for␣
↪infinite
```

Example measuring <bra|SzSx|ket> on each second site, for inhomogeneous sites:

```
>>> SzSx_list = [npc.outer(psi.sites[i].Sz.replace_labels(['p', 'p*'], ['p0',
↪'p0*']),
                           psi.sites[i+1].Sx.replace_labels(['p', 'p*'], ['p1
↪', 'p1*']))
                  for i in range(0, psi.L-1, 2)]
>>> env.expectation_value(SzSx_list, range(0, psi.L-1, 2))
[Sz0Sx1, Sz2Sx3, Sz4Sx5, ...]
```

TransferMatrix

- full name: tenpy.networks.mps.TransferMatrix
- parent module: *tenpy.networks.mps*
- type: class

class tenpy.networks.mps.**TransferMatrix**(*bra*, *ket*, *shift_bra=0*, *shift_ket=None*, *transpose=False*, *charge_sector=0*, *form='B'*)

Bases: *tenpy.linalg.sparse.NpcLinearOperator*

Transfer matrix of two MPS (bra & ket).

For an iMPS in the thermodynamic limit, we often need to find the 'dominant *RP*' (and *LP*). This mean nothing else than to take the transfer matrix of the unit cell and find the (right/left) eigenvector with the largest (magnitude) eigenvalue, since it will dominate $(TM)^n RP$ (or $LP(TM)^n$) in the limit $n \to \infty$ - whatever the initial *RP* is. This class provides exactly that functionality with *eigenvectors()*.

Given two MPS, we define the transfer matrix as:

```
|     ---M[i]---M[i+1]- ... --M[i+L]---
|        |       |              |
|     ---N[j]*--N[j+1]* ... --N[j+L]*--
```

Here the *M* denotes the matrices of the bra and *N* the ones of the ket, respectively. To view it as a *matrix*, we combine the left and right indices to pipes:

```
|   (vL.vL*) ->-TM->- (vR.vR*)    acting on   (vL.vL*) ->-RP
```

Note that we keep all M and N as copies.

Parameters

bra [MPS] The MPS which is to be (complex) conjugated.

ket [MPS] The MPS which is not (complex) conjugated.

shift_bra [int] We start the *N* of the bra at site *shift_bra* (i.e. the *j* in the above network).

shift_ket [int | None] We start the *M* of the ket at site *shift_ket* (i.e. the *i* in the above network). `None` defaults to *shift_bra*.

transpose [bool] Wheter *self.matvec* acts on *RP* (`False`) or *LP* (`True`).

charge_sector [None | charges | `0`] Selects the charge sector of the vector onto which the Linear operator acts. `None` stands for *all* sectors, `0` stands for the zero-charge sector. Defaults to `0`, i.e., *assumes* the dominant eigenvector is in charge sector 0.

form [`'B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)] In which canonical form we take the *M* and *N* matrices.

Attributes

L [int] Number of physical sites involved in the transfer matrix, i.e. the least common multiple of *bra.L* and *ket.L*.

shift_bra [int] We start the *N* of the bra at site *shift_bra*.

shift_ket [int | None] We start the *M* of the ket at site *shift_ket*. `None` defaults to *shift_bra*.

transpose [bool] Wheter *self.matvec* acts on *RP* (`True`) or *LP* (`False`).

qtotal [charges] Total charge of the transfer matrix (which is gauged away in matvec).

form [tuple(float, float) | None] In which canonical form (all of) the *M* and *N* matrices are.

flat_linop [*FlatLinearOperator*] Class lifting *matvec()* to ndarrays in order to use *speigs()*.

pipe [*LegPipe*] Pipe corresponding to `'(vL.vL*)'` for `transpose=False` or to `'(vR.vR*)'` for `transpose=True`.

label_split : `['vL', 'vL*']` if `tranpose=False` or `['vR', 'vR*']` if `transpose=True`.

_bra_N [list of npc.Array] Complex conjugated matrices of the bra, transposed for fast *matvec*.

_ket_M [list of npc.Array] The matrices of the ket, transposed for fast *matvec*.

_contract_legs [int] Number of physical legs per site + 1.

Methods

eigenvectors(self[, num_ev, max_num_ev, ...])	Find (dominant) eigenvector(s) of self using `scipy.sparse`.
initial_guess(self[, diag])	Return a diagonal matrix as initial guess for the eigenvector.
matvec(self, vec)	Given *vec* as an npc.Array, apply the transfer matrix.

matvec(*self*, *vec*)

Given *vec* as an npc.Array, apply the transfer matrix.

Parameters

vec [*Array*] Vector to act on with the transfermatrix. If not *transposed*, *vec* is the right part *RP* of an environment, with legs `'(vL.vL*)'` in a pipe or splitted. If *transposed*, the left part *LP* of an environment with legs `'(vR*.vR)'`.

Returns

mat_vec [*Array*] The tranfer matrix acted on *vec*, in the same form as given.

initial_guess(*self*, *diag=1.0*)
Return a diagonal matrix as initial guess for the eigenvector.

Parameters

diag [float | 1D ndarray] Should be `1.` for the identity or some singular values squared.

Returns

mat [*Array*] A 2D array with *diag* on the diagonal such that *matvec()* can act on it.

eigenvectors(*self*, *num_ev=1*, *max_num_ev=None*, *max_tol=1e-12*, *which='LM'*, *v0=None*, ***kwargs*)
Find (dominant) eigenvector(s) of self using `scipy.sparse`.

If no charge_sector was selected, we look in *all* charge sectors.

Parameters

num_ev [int] Number of eigenvalues/vectors to look for.

max_num_ev [int] `scipy.sparse.linalg.speigs()` somtimes raises a NoConvergenceError for small *num_ev*, which might be avoided by increasing *num_ev*. As a workaround, we try it again in the case of an error, just with larger *num_ev* up to *max_num_ev*. `None` defaults to `num_ev + 2`.

max_tol [float] After the first *NoConvergenceError* we increase the *tol* argument to that value.

which [str] Which eigenvalues to look for, see *scipy.sparse.linalg.speigs*.

****kwargs :** Further keyword arguments are given to *speigs()*.

Returns

eta [1D ndarray] The eigenvalues, sorted according to *which*.

w [list of *Array*] The eigenvectors corresponding to *eta*, as npc.Array with LegPipe.

Functions

build_initial_state(size, states, filling[, ...])	Build an "initial state" list.

build_initial_state

- full name: tenpy.networks.mps.build_initial_state
- parent module: *tenpy.networks.mps*
- type: function

tenpy.networks.mps.**build_initial_state**(*size*, *states*, *filling*, *mode='random'*, *seed=None*)
Build an "initial state" list.

Uses two iterables ('states' and 'filling') to determine how to fill the state. The two lists should have the same length as every element in 'filling' gives the filling fraction for the corresponding state in 'states'.

Example: size = 6, states = [0, 1, 2], filling = [1./3, 2./3, 0.] n_states = size * filling = [2, 4, 0] ==> Two sites will get state 0, 4 sites will get state 1, 0 sites will get state 2.

Todo: Make more general: it should be possible to specify states as strings.

Parameters

size [int] length of state

states [iterable] Containing the possible local states

filling [iterable] Fraction of the total number of sites to get a certain state. If infinite fractions (e.g. 1/3) are needed, one should supply a fraction (1./3.)

mode [str | None] State filling pattern. Only 'random' is implemented

seed [int | None] Seed for random number generators

Returns

initial_state (list) [the initial state]

Raises

ValueError If fractonal fillings are incommensurate with system size.

AssertionError If the total filling is not equal to 1, or the length of *filling* does not equal the length of *states*.

Module description

This module contains a base class for a Matrix Product State (MPS).

An MPS looks roughly like this:

```
|   -- B[0] -- B[1] -- B[2] -- ...
|       |       |       |
```

We use the following label convention for the *B* (where arrows indicate *qconj*):

```
|  vL ->- B ->- vR
|          |
|          ^
|          p
```

We store one 3-leg tensor *_B[i]* with labels `'vL', 'vR', 'p'` for each of the *L* sites `0 <= i < L`. Additionally, we store `L+1` singular value arrays *_S[ib]* on each bond `0 <= ib <= L`, independent of the boundary conditions. `_S[ib]` gives the singlur values on the bond `i-1, i`. However, be aware that e.g. *chi* returns only the dimensions of the *nontrivial_bonds* depending on the boundary conditions.

The matrices and singular values always represent a normalized state (i.e. `np.linalg.norm(psi._S[ib]) == 1` up to roundoff errors), but (for finite MPS) we keep track of the norm in `norm` (which is respected by *overlap()*, ...).

Valid MPS boundary conditions (not to confuse with *bc_coupling* of *tenpy.models.model.CouplingModel*) are the following:

bc	description
'finite'	Finite MPS, `G0 s1 G1 ... s{L-1} G{l-1}`. This is acchieved by using a trivial left and right bond `s[0] = s[-1] = np.array([1.])`.
'segment'	Generalization of 'finite', describes an MPS embedded in left and right environments. The left environment is described by `chi[0]` *orthonormal* states which are weighted by the singular values `s[0]`. Similar, `s[L]` weight some right orthonormal states. You can think of the left and right states to be generated by additional MPS, such that the overall structure is something like `... s L s L [s0 G0 s1 G1 ... s{L-1} G{L-1} s{L}] R s R s R ...` (where we save the part in the brackets `[ ... ]`).
'infinite'	infinite MPS (iMPS): we save a 'MPS unit cell' `[s0 G0 s1 G1 ... s{L-1} G{L-1}]` which is repeated periodically, identifying all indices modulo `self.L`. In particular, the last bond `L` is identified with `0`. (The MPS unit cell can differ from a lattice unit cell). bond is identified with the first one.

An MPS can be in different 'canonical forms' (see [Vidal2004], [Schollwoeck2011]). To take care of the different canonical forms, algorithms should use functions like *get_theta()*, *get_B()* and *set_B()* instead of accessing them directly, as they return the B in the desired form (which can be chosen as an argument). The values of the tuples for the form correspond to the exponent of the singular values on the left and right. To keep track of a "mixed" canonical form `A A A s B B`, we save the tuples for each site of the MPS in `MPS.form`.

form	tuple	description
`'B'`	(0, 1)	right canonical: `_B[i] = -- Gamma[i] -- s[i+1]--` The default form, which algorithms asssume.
`'C'`	(0.5, 0.5)	symmetric form: `_B[i] = -- s[i]**0.5 -- Gamma[i] -- s[i+1]**0.5--`
`'A'`	(1, 0)	left canonical: `_B[i] = -- s[i] -- Gamma[i] --`.
`'G'`	(0, 0)	Save only `_B[i] = -- Gamma[i] --`.
`'Th'`	(1, 1)	Form of a local wave function *theta* with singular value on both sides. `psi.get_B(i, 'Th')` is equivalent to ``psi.get_theta(i, n=1)`.
`None`	`None`	General non-canoncial form. Valid form for initialization, but you need to call *canonical_form()* (or similar) before using algorithms.

mpo

- full name: tenpy.networks.mpo
- parent module: *tenpy.networks*
- type: module

Classes

MPO(sites, Ws[, bc, IdL, IdR, max_range])	Matrix product operator, finite (MPO) or infinite (iMPO).
MPOEnvironment(bra, H, ket[, init_LP, ...])	Stores partial contractions of $< bra\|H\|ket >$ for an MPO H.
MPOGraph(sites[, bc, max_range])	Representation of an MPO by a graph, based on a 'finite state machine'.

MPO

- full name: tenpy.networks.mpo.MPO
- parent module: *tenpy.networks.mpo*
- type: class

class tenpy.networks.mpo.**MPO**(*sites*, *Ws*, *bc='finite'*, *IdL=None*, *IdR=None*, *max_range=None*)
Bases: object

Matrix product operator, finite (MPO) or infinite (iMPO).

Parameters

sites [list of Site] Defines the local Hilbert space for each site.

Ws [list of *Array*] The matrices of the MPO. Should have labels wL, wR, p, p*.

bc [{'finite' | 'segment' | 'infinite'}] Boundary conditions as described in *mps*. 'finite' requires Ws[0].get_leg('wL').ind_len = 1.

IdL [(iterable of) {int | None}] Indices on the bonds, which correpond to 'only identities to the left'. A single entry holds for all bonds.

IdR [(iterable of) {int | None}] Indices on the bonds, which correpond to 'only identities to the right'.

max_range [int | np.inf | None] Maximum range of hopping/interactions (in unit of sites) of the MPO. None for unknown.

Attributes

L [int] Number of physical sites; for an iMPO the len of the MPO unit cell.

chinfo [ChargeInfo] The nature of the charge.

sites [list of Site] Defines the local Hilbert space for each site.

dtype [type] The data type of the *_W*.

bc [{'finite' | 'segment' | 'infinite'}] Boundary conditions as described in *mps*. 'finite' requires Ws[0].get_leg('wL').ind_len = 1.

IdL [list of {int | None}] Indices on the bonds (length *L`+1), which correpond to 'only identities to the left'. ``None`* for bonds where it is not set. In standard form, this is *0* (except for unset bonds in finite case)

IdR [list of {int | None}] Indices on the bonds (length *L`+1), which correpond to 'only identities to the right'. ``None`* for bonds where it is not set. In standard form, this is the last index on the bond (except for unset bonds in finite case).

max_range [int | np.inf | None] Maximum range of hopping/interactions (in unit of sites) of the MPO. None for unknown.

grouped [int] Number of sites grouped together, see *group_sites()*.

_W [list of *Array*] The matrices of the MPO. Labels are 'wL', 'wR', 'p', 'p*'.

_valid_bc [tuple of str] Valid boundary conditions. The same as for an MPS.

Methods

dagger(self)	Return hermition conjugate copy of self.
expectation_value(self, psi[, tol, max_range])	Calculate <psi|self|psi>/<psi|psi>.
from_grids(sites, grids[, bc, IdL, IdR, ...])	Initialize an MPO from *grids*.
get_IdL(self, i)	Return index of *IdL* at bond to the *left* of site *i*.
get_IdR(self, i)	Return index of *IdR* at bond to the *right* of site *i*.
get_W(self, i[, copy])	Return *W* at site *i*.
get_full_hamiltonian(self[, maxsize])	extract the full Hamiltonian as a d**L x d**L matrix.
get_grouped_mpo(self, blocklen)	Contract *blocklen* subsequent tensors into a single one and return result as a new MPO object.
group_sites(self[, n, grouped_sites])	Modify *self* inplace to group sites.
is_equal(self, other[, eps, max_range])	Check if *self* and *other* represent the same MPO to precision *eps*.
is_hermitian(self[, eps, max_range])	Check if *self* is a hermitian MPO.
set_W(self, i, W)	Set *W* at site *i*.
sort_legcharges(self)	Sort virtual legs by charges.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

classmethod from_grids(*sites*, *grids*, *bc='finite'*, *IdL=None*, *IdR=None*, *Ws_qtotal=None*, *leg0=None*, *max_range=None*)

Initialize an MPO from *grids*.

Parameters

sites [list of `Site`] Defines the local Hilbert space for each site.

grids [list of list of list of entries] For each site (outer-most list) a matrix-grid (corresponding to `wL, wR`) with entries being or representing (see *grid_insert_ops()*) onsite-operators.

bc [{'finite' | 'segment' | 'infinite'}] Boundary conditions as described in *mps*.

IdL [(iterable of) {int | None}] Indices on the bonds, which correpond to 'only identities to the left'. A single entry holds for all bonds.

IdR [(iterable of) {int | None}] Indices on the bonds, which correpond to 'only identities to the right'.

Ws_qtotal [(list of) total charge] The *qtotal* to be used for each grid. Defaults to zero charges.

leg0 [`LegCharge`] LegCharge for 'wL' of the left-most *W*. By default, construct it.

max_range [int | np.inf | None] Maximum range of hopping/interactions (in unit of sites) of the MPO. `None` for unknown.

See also:

grid_insert_ops used to plug in *entries* of the grid.

tenpy.linalg.np_conserved.grid_outer used for final conversion.

test_sanity(*self*)

Sanity check, raises ValueErrors, if something is wrong.

property L
Number of physical sites; for an iMPO the len of the MPO unit cell.

property dim
List of local physical dimensions.

property finite
Distinguish MPO vs iMPO.

True for an MPO (`bc='finite', 'segment'`), False for an iMPO (`bc='infinite'`).

property chi
Dimensions of the virtual bonds.

get_W(*self*, *i*, *copy=False*)
Return *W* at site *i*.

set_W(*self*, *i*, *W*)
Set *W* at site *i*.

get_IdL(*self*, *i*)
Return index of *IdL* at bond to the *left* of site *i*.

May be `None`.

get_IdR(*self*, *i*)
Return index of *IdR* at bond to the *right* of site *i*.

May be `None`.

group_sites(*self*, *n=2*, *grouped_sites=None*)
Modify *self* inplace to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

sort_legcharges(*self*)
Sort virtual legs by charges. In place.

The MPO seen as matrix of the `wL, wR` legs is usually very sparse. This sparsity is captured by the LegCharges for these bonds not being sorted and bunched. This requires a tensordot to do more block-multiplications with smaller blocks. This is in general faster for large blocks, but might lead to a larger overhead for small blocks. Therefore, this function allows to sort the virtual legs by charges.

expectation_value(*self*, *psi*, *tol=1e-10*, *max_range=100*)
Calculate `<psi|self|psi>/<psi|psi>`.

For a finite MPS, simply contract the network `<psi|self|psi>`. For an infinite MPS, it assumes that *self* is the a of terms, with `IdL` and `IdR` defined on each site. Under this assumption, it calculates the expectation value of terms with the left-most non-trivial operator inside the MPO unit cell and returns the average value per site.

Parameters

psi [*MPS*] State for which the expectation value should be taken.

tol [float] Ignored for finite *psi*. For infinite MPO containing exponentially decaying long-range terms, stop evaluating further terms if the terms in *LP* have norm < *tol*.

max_range [int] Ignored for finite *psi*. Contract at most `self.L * max_range` sites, even if *tol* is not reached. In that case, issue a warning.

Returns

exp_val [float/complex] The expectation value of *self* with respect to the state *psi*. For an infinite MPS: the density per site.

dagger (*self*)
Return hermition conjugate copy of self.

is_hermitian (*self*, *eps=1e-10*, *max_range=None*)
Check if *self* is a hermitian MPO.

Shorthand for `self.is_equal(self.dagger(), eps, max_range)`.

is_equal (*self*, *other*, *eps=1e-10*, *max_range=None*)
Check if *self* and *other* represent the same MPO to precision *eps*.

To compare them efficiently we view *self* and *other* as MPS and compare the overlaps `abs(<self|self> + <other|other> - 2 Re(<self|other>)) < eps*(<self|self>+<other|other>)`

Parameters

other [*MPO*] The MPO to compare to.

eps [float] Precision threshold what counts as zero.

max_range [None | int] Ignored for finite MPS; for finite MPS we consider only the terms contained in the sites with indices `range(self.L + max_range)`. None defaults to `max_range` (or *L* in case this is infinite or None).

Returns

equal [bool] Whether *self* equals *other* to the desired precision.

get_grouped_mpo (*self*, *blocklen*)
Contract *blocklen* subsequent tensors into a single one and return result as a new MPO object.

get_full_hamiltonian (*self*, *maxsize=1000000.0*)
extract the full Hamiltonian as a d**L x d**L matrix.

MPOEnvironment

- full name: tenpy.networks.mpo.MPOEnvironment
- parent module: *tenpy.networks.mpo*
- type: class

class tenpy.networks.mpo.**MPOEnvironment** (*bra*, *H*, *ket*, *init_LP=None*, *init_RP=None*, *age_LP=0*, *age_RP=0*)

Bases: *tenpy.networks.mps.MPSEnvironment*

Stores partial contractions of $< bra|H|ket >$ for an MPO H.

The network for a contraction $< bra|H|ket >$ of an MPO H bewteen two MPS looks like:

```
|      .------>-M[0]-->-M[1]-->-M[2]-->- ...  ->--.
|      |         |        |        |                |
|      |         ^        ^        ^                |
|      |         |        |        |                |
```

(continues on next page)

(continued from previous page)

```
|      LP[0] ->-W[0]-->-W[1]-->-W[2]-->- ...  ->- RP[-1]
|       |         |        |        |                |
|       |         ^        ^        ^                |
|       |         |        |        |                |
|       .-------<-N[0]*-<-N[1]*-<-N[2]*-<- ...  -<--.
```

We use the following label convention (where arrows indicate *qconj*):

```
|     .-->- vR            vL ->-.
|     |                         |
|     LP->- wR            wL ->-RP
|     |                         |
|     .--<- vR*          vL* -<-.
```

To avoid recalculations of the whole network e.g. in the DMRG sweeps, we store the contractions up to some site index in this class. For `bc='finite','segment'`, the very left and right part `LP[0]` and `RP[-1]` are trivial and don't change in the DMRG algorithm, but for iDMRG (`bc='infinite'`) they are also updated (by inserting another unit cell to the left/right).

The MPS *bra* and *ket* have to be in canonical form. All the environments are constructed without the singular values on the open bond. In other words, we contract left-canonical *A* to the left parts *LP* and right-canonical *B* to the right parts *RP*.

Parameters

bra [*MPS*] The MPS to project on. Should be given in usual 'ket' form; we call *conj()* on the matrices directly.

H [*MPO*] The MPO sandwiched between *bra* and *ket*. Should have 'IdL' and 'IdR' set on the first and last bond.

ket [MPS] The MPS on which *H* acts. May be identical with *bra*.

init_LP [None | *Array*] Initial very left part LP. If None, build trivial one with :meth`init_LP`.

init_RP [None | *Array*] Initial very right part RP. If None, build trivial one with *init_RP()*.

age_LP [int] The number of physical sites involved into the contraction yielding *firstLP*.

age_RP [int] The number of physical sites involved into the contraction yielding *lastRP*.

Attributes

H [*MPO*] The MPO sandwiched between *bra* and *ket*.

Methods

del_LP(self, i)	Delete stored part strictly to the left of site *i*.
del_RP(self, i)	Delete storde part scrictly to the right of site *i*.
expectation_value(self, ops[, sites, axes])	Expectation value <bra|ops|ket> of (n-site) operator(s).
full_contraction(self, i0)	Calculate the energy by a full contraction of the network.
get_LP(self, i[, store])	Calculate LP at given site from nearest available one (including *i*).

Continued on next page

Table 138 – continued from previous page

get_LP_age(self, i)	Return number of physical sites in the contractions of get_LP(i).
get_RP(self, i[, store])	Calculate RP at given site from nearest available one (including *i*).
get_RP_age(self, i)	Return number of physical sites in the contractions of get_RP(i).
init_LP(self, i)	Build initial left part `LP`.
init_RP(self, i)	Build initial right part `RP` for an MPS/MPOEnvironment.
set_LP(self, i, LP, age)	Store part to the left of site *i*.
set_RP(self, i, RP, age)	Store part to the right of site *i*.
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

init_LP(*self*, *i*)
Build initial left part `LP`.

Parameters

i [int] Build `LP` left of site *i*.

Returns

init_LP [*Array*] Identity contractible with the *vL* leg of `.ket.get_B(i)`, multiplied with a unit vector nonzero in `H.IdL[i]`, with labels `'vR*', 'wR', 'vR'`.

init_RP(*self*, *i*)
Build initial right part `RP` for an MPS/MPOEnvironment.

Parameters

i [int] Build `RP` right of site *i*.

Returns

init_RP [*Array*] Identity contractible with the *vR* leg of `self.get_B(i)`, multiplied with a unit vector nonzero in `H.IdR[i]`, with labels `'vL*', 'wL', 'vL'`.

get_LP(*self*, *i*, *store=True*)
Calculate LP at given site from nearest available one (including *i*).

The returned `LP_i` corresponds to the following contraction, where the M's and the N's are in the 'A' form:

```
|      .-------M[0]--- ... --M[i-1]--->-   'vR'
|      |        |              |
|      LP[0]---W[0]--- ... --W[i-1]--->-   'wR'
|      |        |              |
|      .-------N[0]*-- ... --N[i-1]*--<-   'vR*'
```

Parameters

i [int] The returned *LP* will contain the contraction *strictly* left of site *i*.

store [bool] Wheter to store the calculated *LP* in *self* (`True`) or discard them (`False`).

Returns

LP_i [*Array*] Contraction of everything left of site *i*, with labels `'vR*', 'wR', 'vR'` for *bra*, *H*, *ket*.

get_RP (*self*, *i*, *store=True*)
Calculate RP at given site from nearest available one (including *i*).

The returned `RP_i` corresponds to the following contraction, where the M's and the N's are in the 'B' form:

```
|      'vL'  ->---M[i+1]-- ... --M[L-1]----.
|                  |               |          |
|      'wL'  ->---W[i+1]-- ... --W[L-1]----RP[-1]
|                  |               |          |
|      'vL*' -<---N[i+1]*- ... --N[L-1]*---.
```

Parameters

i [int] The returned *RP* will contain the contraction *strictly* rigth of site *i*.

store [bool] Wheter to store the calculated *RP* in *self* (`True`) or discard them (`False`).

Returns

RP_i [*Array*] Contraction of everything right of site *i*, with labels `'vL*', 'wL', 'vL'` for *bra*, *H*, *ket*.

full_contraction (*self*, *i0*)
Calculate the energy by a full contraction of the network.

The full contraction of the environments gives the value `<bra|H|ket> / (norm(|bra>)*norm(|ket>))`, i.e. if *bra* is *ket* and normalized, the total energy. For this purpose, this function contracts `get_LP(i0+1, store=False)` and `get_RP(i0, store=False)`.

Parameters

i0 [int] Site index.

del_LP (*self*, *i*)
Delete stored part strictly to the left of site *i*.

del_RP (*self*, *i*)
Delete storde part scrictly to the right of site *i*.

expectation_value (*self*, *ops*, *sites=None*, *axes=None*)
Expectation value `<bra|ops|ket>` of (n-site) operator(s).

Calculates n-site expectation values of operators sandwiched between bra and ket. For examples the contraction for a two-site operator on site *i* would look like:

```
|          .--S--B[i]--B[i+1]--.
|          |      |       |         |
|          |      |-------|         |
|          LP[i]  |  op   |        RP[i+1]
|          |      |-------|         |
|          |      |       |         |
|          .--S--B*[i]-B*[i+1]-.
```

Here, the *B* are taken from *ket*, the *B** from *bra*. The call structure is the same as for `MPS.expectation_value()`.

Parameters

ops [(list of) { *Array* | str }] The operators, for wich the expectation value should be taken, All operators should all have the same number of legs (namely *2 n*). If less than `len(sites)` operators are given, we repeat them periodically. Strings (like `'Id'`, `'Sz'`) are translated into single-site operators defined by `sites`.

sites [list] List of site indices. Expectation values are evaluated there. If `None` (default), the entire chain is taken (clipping for finite b.c.)

axes [None | (list of str, list of str)] Two lists of each *n* leg labels giving the physical legs of the operator used for contraction. The first *n* legs are contracted with conjugated *B*, the second *n* legs with the non-conjugated *B*. `None` defaults to `(['p'], ['p*'])` for single site (n=1), or `(['p0', 'p1', ... 'p{n-1}'], ['p0*', 'p1*', ... 'p{n-1}*'])` for $n > 1$.

Returns

exp_vals [1D ndarray] Expectation values, `exp_vals[i] = <bra|ops[i]|ket>`, where `ops[i]` acts on site(s) `j, j+1, ..., j+{n-1}` with `j=sites[i]`.

Examples

One site examples (n=1):

```
>>> env.expectation_value('Sz')
[Sz0, Sz1, ..., Sz{L-1}]
>>> env.expectation_value(['Sz', 'Sx'])
[Sz0, Sx1, Sz2, Sx3, ... ]
>>> env.expectation_value('Sz', sites=[0, 3, 4])
[Sz0, Sz3, Sz4]
```

Two site example (n=2), assuming homogeneous sites:

```
>>> SzSx = npc.outer(psi.sites[0].Sz.replace_labels(['p', 'p*'], ['p0', 'p0*
↪']),
                     psi.sites[1].Sx.replace_labels(['p', 'p*'], ['p1', 'p1*
↪']))
>>> env.expectation_value(SzSx)
[Sz0Sx1, Sz1Sx2, Sz2Sx3, ... ]   # with len L-1 for finite bc, or L for␣
↪infinite
```

Example measuring <bra|SzSx|ket> on each second site, for inhomogeneous sites:

```
>>> SzSx_list = [npc.outer(psi.sites[i].Sz.replace_labels(['p', 'p*'], ['p0',
↪'p0*']),
                           psi.sites[i+1].Sx.replace_labels(['p', 'p*'], ['p1
↪', 'p1*']))
                  for i in range(0, psi.L-1, 2)]
>>> env.expectation_value(SzSx_list, range(0, psi.L-1, 2))
[Sz0Sx1, Sz2Sx3, Sz4Sx5, ...]
```

get_LP_age (*self*, *i*)
Return number of physical sites in the contractions of get_LP(i).

Might be `None`.

get_RP_age (*self*, *i*)
Return number of physical sites in the contractions of get_RP(i).

Might be `None`.

set_LP (*self*, *i*, *LP*, *age*)
Store part to the left of site *i*.

set_RP (*self*, *i*, *RP*, *age*)
Store part to the right of site *i*.

MPOGraph

- full name: tenpy.networks.mpo.MPOGraph
- parent module: *tenpy.networks.mpo*
- type: class

class tenpy.networks.mpo.**MPOGraph** (*sites*, *bc='finite'*, *max_range=None*)
Bases: object

Representation of an MPO by a graph, based on a 'finite state machine'.

This representation is used for building H_MPO from the interactions. The idea is to view the MPO as a kind of 'finite state machine'. The **states** or **keys** of this finite state machine life on the MPO bonds *between* the *Ws*. They label the indices of the virtul bonds of the MPOs, i.e., the indices on legs `wL` and `wR`. They can be anything hash-able like a `str`, `int` or a tuple of them.

The **edges** of the graph are the entries `W[keyL, keyR]`, which itself are onsite operators on the local Hilbert space. The indices *keyL* and *keyR* correspond to the legs `'wL', 'wR'` of the MPO. The entry `W[keyL, keyR]` connects the state `keyL` on bond `(i-1, i)` with the state `keyR` on bond `(i, i+1)`.

The keys `'IdR'` (for 'idenity left') and `'IdR'` (for 'identity right') are reserved to represent only `'Id'` (=identity) operators to the left and right of the bond, respectively.

Todo: might be useful to add a "cleanup" function which removes operators cancelling each other and/or unused states. Or better use a 'compress' of the MPO?

Parameters

sites [list of `Site`] Local sites of the Hilbert space.

bc [{'finite', 'infinite'}] MPO boundary conditions.

max_range [int | np.inf | None] Maximum range of hopping/interactions (in unit of sites) of the MPO. `None` for unknown.

Attributes

L Number of physical sites; for infinite boundaries the length of the unit cell.

sites [list of `Site`] Defines the local Hilbert space for each site.

chinfo [`ChargeInfo`] The nature of the charge.

bc [{'finite', 'infinite'}] MPO boundary conditions.

max_range [int | np.inf | None] Maximum range of hopping/interactions (in unit of sites) of the MPO. `None` for unknown.

states [list of set of keys] `states[i]` gives the possible keys at the virtual bond `(i-1, i)` of the MPO.

graph [list of dict of dict of list of tuples] For each site *i* a dictionary `{keyL: {keyR: [(opname, strength)]}}` with `keyL in vertices[i]` and `keyR in vertices[i+1]`.

_grid_legs [None | list of LegCharge] The charges for the MPO

Methods

add(self, i, keyL, keyR, opname, strength[, ...])	Insert an edge into the graph.
add_missing_IdL_IdR(self)	Add missing identity ('Id') edges connecting `'IdL'->'IdL'` and ``'IdR'->'IdR'``.
add_string(self, i, j, key[, opname, ...])	Insert a bunch of edges for an 'operator string' into the graph.
build_MPO(self[, Ws_qtotal, leg0])	Build the MPO represented by the graph (*self*).
from_term_list(term_list, sites, bc)	Initialize form a list of operator terms and prefactors.
from_terms(onsite_terms, coupling_terms, ...)	Initialize an *MPOGraph* from OnsiteTerms and CouplingTerms.
has_edge(self, i, keyL, keyR)	True if there is an edge from *keyL* on bond (i-1, i) to *keyR* on bond (i, i+1).
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

classmethod from_terms(*onsite_terms*, *coupling_terms*, *sites*, *bc*)
Initialize an *MPOGraph* from OnsiteTerms and CouplingTerms.

Parameters

onsite_terms [*OnsiteTerms*] Onsite terms to be added to the new *MPOGraph*.

coupling_terms :class:`~tenpy.networks.terms.CouplingTerms` | :class:`~tenpy.networks.terms.MultiCoupli
Coupling terms to be added to the new *MPOGraph*.

sites [list of *Site*] Local sites of the Hilbert space.

bc `['finite' | 'infinite']` MPO boundary conditions.

Returns

graph [*MPOGraph*] Initialized with the given terms.

See also:

from_term_list equivalent for other representation terms.

classmethod from_term_list(*term_list*, *sites*, *bc*)
Initialize form a list of operator terms and prefactors.

Parameters

term_list [`TermList`] Terms to be added to the MPOGraph.

sites [list of *Site*] Local sites of the Hilbert space.

bc `['finite' | 'infinite']` MPO boundary conditions.

Returns

graph [*MPOGraph*] Initialized with the given terms.

See also:

from_terms equivalent for other representation of terms.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

property L
Number of physical sites; for infinite boundaries the length of the unit cell.

add(*self*, *i*, *keyL*, *keyR*, *opname*, *strength*, *check_op=True*, *skip_existing=False*)
Insert an edge into the graph.

Parameters

i [int] Site index at which the edge of the graph is to be inserted.

keyL [hashable] The state at bond (i-1, i) to connect from.

keyR [hashable] The state at bond (i, i+1) to connect to.

opname [str] Name of the operator.

strength [str] Prefactor of the operator to be inserted.

check_op [bool] Whether to check that 'opname' exists on the given *site*.

skip_existing [bool] If `True`, skip adding the graph node if it exists (with same keys and *opname*).

add_string(*self*, *i*, *j*, *key*, *opname='Id'*, *check_op=True*, *skip_existing=True*)
Insert a bunch of edges for an 'operator string' into the graph.

Terms like $S_i^z S_j^z$ actually stand for $S_i^z \otimes \prod_{i<k<j} \mathbb{1}_k \otimes S_j^z$. This function adds the $\mathbb{1}$ terms to the graph.

Parameters

i, j: int An edge is inserted on all bonds between *i* and *j*, $i < j$. *j* can be larger than `L`, in which case the operators are supposed to act on different MPS unit cells.

key: hashable The state at bond (i-1, i) to connect from and on bond (j-1, j) to connect to. Also used for the intermediate states. No operator is inserted on a site $i < k < j$ if `has_edge(k, key, key)`.

opname [str] Name of the operator to be used for the string. Useful for the Jordan-Wigner transformation to fermions.

skip_existing [bool] Whether existing graph nodes should be skipped.

Returns

label_j [hashable] The *key* on the left of site j to connect to. Usually the same as the parameter *key*, except if `j - i > self.L`, in which case we use the additional labels `(key, 1)`, `(key, 2)`, ... to generate couplings over multiple unit cells.

add_missing_IdL_IdR(*self*)
Add missing identity ('Id') edges connecting `'IdL'->'IdL' and ``'IdR'->'IdR'`.

For `bc='infinite'`, insert missing identities at *all* bonds. For `bc='finite' | 'segment'` only insert `'IdL'->'IdL'` to the left of the rightmost existing 'IdL' and `'IdR'->'IdR'` to the right of the leftmost existing 'IdR'.

This function should be called *after* all other operators have been inserted.

has_edge(*self*, *i*, *keyL*, *keyR*)
True if there is an edge from *keyL* on bond (i-1, i) to *keyR* on bond (i, i+1).

build_MPO(*self*, *Ws_qtotal=None*, *leg0=None*)
Build the MPO represented by the graph (*self*).

Parameters

Ws_qtotal [None | (list of) charges] The *qtotal* for each of the Ws to be generated., default (`None`) means 0 charge. A single qtotal holds for each site.

leg0 [None | `npc.LegCharge`] The charges to be used for the very first leg (which is a gauge freedom). If `None` (default), use zeros.

Returns

mpo [*MPO*] the MPO which self represents.

Functions

grid_insert_ops(site, grid)	Replaces entries representing operators in a grid of `W[i]` with npc.Arrays.

grid_insert_ops

- full name: tenpy.networks.mpo.grid_insert_ops
- parent module: *tenpy.networks.mpo*
- type: function

tenpy.networks.mpo.**grid_insert_ops**(*site*, *grid*)
Replaces entries representing operators in a grid of `W[i]` with npc.Arrays.

Parameters

site [*site*] The site on which the grid acts.

grid [list of list of *entries*] Represents a single matrix *W* of an MPO, i.e. the lists correspond to the legs `'vL'`, `'vR'`, and entries to onsite operators acting on the given *site*. *entries* may be `None`, *Array*, a single string or of the form `[('opname', strength), ...]`, where `'opname'` labels an operator in the *site*.

Returns

grid [list of list of {None | *Array*}] Copy of *grid* with entries `[('opname', strength), ...]` replaced by `sum([strength*site.get_op('opname') for opname, strength in entry])` and entries `'opname'` replaced by `site.get_op('opname')`.

Module description

Matrix product operator (MPO).

An MPO is the generalization of an *MPS* to operators. Graphically:

```
|      ^        ^        ^
|      |        |        |
|  ->- W[0] ->- W[1] ->- W[2] ->- ...
|      |        |        |
|      ^        ^        ^
|
```

So each 'matrix' has two physical legs `p, p*` instead of just one, i.e. the entries of the 'matrices' are local operators. Valid boundary conditions of an MPO are the same as for an MPS (i.e. `'finite' | 'segment' | 'infinite'`). (In general, you can view the MPO as an MPS with larger physical space and bring it into canoncial form. However, unlike for an MPS, this doesn't simplify calculations. Thus, an MPO has no *form*.)

We use the following label convention for the *W* (where arrows indicate *qconj*):

```
|             p*
|             ^
|             |
|     wL ->- W ->- wR
|             |
|             ^
|             p
```

If an MPO describes a sum of local terms (e.g. most Hamiltonians), some bond indices correspond to 'only identities to the left/right'. We store these indices in *IdL* and *IdR* (if there are such indices).

Similar as for the MPS, a bond index `i` is *left* of site *i*, i.e. between sites `i-1` and `i`.

terms

- full name: tenpy.networks.terms
- parent module: *tenpy.networks*
- type: module

Classes

CouplingTerms(L)	Operator names, site indices and strengths representing two-site coupling terms.
MultiCouplingTerms(L)	Operator names, site indices and strengths representing general *M*-site coupling terms.
OnsiteTerms(L)	Operator names, site indices and strengths representing onsite terms.
TermList(terms, strength)	A list of terms (=operator names and sites they act on) and associated strengths.

CouplingTerms

- full name: tenpy.networks.terms.CouplingTerms
- parent module: *tenpy.networks.terms*
- type: class

class tenpy.networks.terms.**CouplingTerms**(*L*)
Bases: object

Operator names, site indices and strengths representing two-site coupling terms.

Parameters

L [int] Number of sites.

Attributes

L [int] Number of sites.

coupling_terms [dict of dict] Filled by *add_coupling_term()*. Nested dictionaries of the form `{i: {('opname_i', 'opname_string'): {j: {'opname_j': strength}}}}`. Note that always `i < j`, but entries with `j >= L` are allowed for `bc_MPS == 'infinite'`, in which case they indicate couplings between different iMPS unit cells.

Methods

add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_to_graph(self, graph)	Add terms from `coupling_terms` to an MPO-Graph.
coupling_term_handle_JW(self, strength, ...)	Helping function to call before `add_multi_coupling_term()`.
max_range(self)	Determine the maximal range in `coupling_terms`.
plot_coupling_terms(self, ax, lat[, ...])	"Plot coupling terms into a given lattice.
remove_zeros(self[, tol_zero])	Remove entries close to 0 from `coupling_terms`.
to_TermList(self)	Convert `onsite_terms` into a *TermList*.
to_nn_bond_Arrays(self, sites)	Convert the `coupling_terms` into Arrays on nearest neighbor bonds.

max_range(*self*)
Determine the maximal range in `coupling_terms`.

Returns

max_range [int] The maximum of `j - i` for the *i*, *j* occuring in a term of `coupling_terms`.

add_coupling_term(*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*)
Add a two-site coupling term on given MPS sites.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

coupling_term_handle_JW(*self*, *strength*, *term*, *sites*, *op_string=None*)
Helping function to call before `add_multi_coupling_term()`.

Parameters

strength [float] The strength of the coupling term.

term [[(str, int), (str, int)]] List of two tuples `(op, i)` where *i* is the MPS index of the site the operator named *op* acts on.

sites [list of *Site*] Defines the local Hilbert space for each site. Used to check whether the operators need Jordan-Wigner strings.

op_string [None | str] Operator name to be used as operator string *between* the operators, or `None` if the Jordan Wigner string should be figured out.

Returns

strength, i, j, op_i, op_j, op_string: Arguments for *`MultiCouplingTerms.add_multi_coupling_term()`* such that the added term corresponds to the parameters of this function.

plot_coupling_terms(*self*, *ax*, *lat*, *style_map='default'*, *common_style={'linestyle': '--'}*, *text=None*, *text_pos=0.4*)

"Plot coupling terms into a given lattice.

This function plots the `coupling_terms`

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

lat [*`Lattice`*] The lattice for plotting the couplings, most probably the `M.lat` of the corresponding model `M`, see `lat`.

style_map [function | None] Function which get's called with arguments `i, j, op_i, op_string, op_j, strength` for each two-site coupling and should return a keyword-dictionary with the desired plot-style for this coupling. By default (`None`), the *linewidth* is given by the absolute value of *strength*, and the linecolor depends on the phase of *strength* (using the *hsv* colormap).

common_style [dict] Common style, which overwrites values of the dictionary returned by style_map. A `'label'` is only used for the first plotted line.

text: format_string | None If not `None`, we add text labeling the couplings in the plot. Available keywords are `i, j, op_i, op_string, op_j, strength` as well as `strength_abs, strength_angle, strength_real`.

text_pos [float] Specify where to put the text on the line between *i* (0.0) and *j* (1.0), e.g. *0.5* is exactly in the middle between *i* and *j*.

See also:

`tenpy.models.lattice.Lattice.plot_sites` plot the sites of the lattice.

add_to_graph(*self*, *graph*)

Add terms from `coupling_terms` to an MPOGraph.

Parameters

graph [*`MPOGraph`*] The graph into which the terms from `coupling_terms` should be added.

to_nn_bond_Arrays(*self*, *sites*)

Convert the `coupling_terms` into Arrays on nearest neighbor bonds.

Parameters

sites [list of *`Site`*] Defines the local Hilbert space for each site. Used to translate the operator names into *`Array`*.

Returns

H_bond [list of {*`Array`* | None}] The `coupling_terms` rewritten as `sum_i H_bond[i]` for MPS indices `i`. `H_bond[i]` acts on sites `(i-1, i)`, `None` represents 0. Legs of each `H_bond[i]` are `['p0', 'p0*', 'p1', 'p1*']`.

remove_zeros(*self*, *tol_zero=1e-15*)
Remove entries close to 0 from `coupling_terms`.

Parameters

tol_zero [float] Entries in `coupling_terms` with *strength* < *tol_zero* are considered to be zero and removed.

to_TermList(*self*)
Convert `onsite_terms` into a *TermList*.

Returns

term_list [*TermList*] Representation of the terms as a list of terms.

MultiCouplingTerms

- full name: tenpy.networks.terms.MultiCouplingTerms
- parent module: *tenpy.networks.terms*
- type: class

class tenpy.networks.terms.**MultiCouplingTerms**(*L*)
Bases: *tenpy.networks.terms.CouplingTerms*

Operator names, site indices and strengths representing general *M*-site coupling terms.

Generalizes the `coupling_terms` of *CouplingTerms* to *M*-site couplings. The structure of the nested dictionary `coupling_terms` is similar, but we allow an arbitrary recursion depth of the dictionary.

Parameters

L [int] Number of sites.

Attributes

L [int] Number of sites.

coupling_terms [dict of dict] Nested dictionaries of the following form:

```
{i: {('opname_i', 'opname_string_ij'):
        {j: {('opname_j', 'opname_string_jk'):
                {k: {('opname_k', 'opname_string_kl'):
                    ...
                        {l: {'opname_l':
                                strength
                        }   }
                    ...
                }   }
        }   }
}   }
```

For a M-site coupling, this involves a nesting depth of `2*M` dictionaries. Note that always `i < j < k < ... < l`, but entries with `j,k,l >= L` are allowed for the case of `bc_MPS == 'infinite'`, when they indicate couplings between different iMPS unit cells.

Methods

add_coupling_term(self, strength, i, j, ...)	Add a two-site coupling term on given MPS sites.
add_multi_coupling_term(self, strength, ...)	Add a multi-site coupling term.
add_to_graph(self, graph[, _i, _d1, _label_left])	Add terms from `coupling_terms` to an MPO-Graph.
coupling_term_handle_JW(self, strength, ...)	Helping function to call before *add_multi_coupling_term()*.
max_range(self)	Determine the maximal range in `coupling_terms`.
multi_coupling_term_handle_JW(self, ...[, ...])	Helping function to call before *add_multi_coupling_term()*.
plot_coupling_terms(self, ax, lat[, ...])	"Plot coupling terms into a given lattice.
remove_zeros(self[, tol_zero, _d0])	Remove entries close to 0 from `coupling_terms`.
to_TermList(self)	Convert `onsite_terms` into a *TermList*.
to_nn_bond_Arrays(self, sites)	Convert the `coupling_terms` into Arrays on nearest neighbor bonds.

add_multi_coupling_term(*self*, *strength*, *ijkl*, *ops_ijkl*, *op_string='Id'*)
Add a multi-site coupling term.

Parameters

strength [float] The strength of the coupling term.

ijkl [list of int] The MPS indices of the sites on which the operators acts. With *i, j, k, ...* = *ijkl*, we require that they are ordered ascending, `i < j < k < ...` and that `0 <= i < N_sites`. Inidces >= N_sites indicate couplings between different unit cells of an infinite MPS.

ops_ijkl [list of str] Names of the involved operators on sites *i, j, k, ...*.

op_string [(list of) str] Names of the operator to be inserted between the operators, e.g., op_string[0] is inserted between *i* and *j*. A single name holds for all in-between segments.

multi_coupling_term_handle_JW(*self*, *strength*, *term*, *sites*, *op_string=None*)
Helping function to call before *add_multi_coupling_term()*.

Handle/figure out Jordan-Wigner strings if needed.

Parameters

strength [float] The strength of the term.

term [list of (str, int)] List of tuples `(op_i, i)` where *i* is the MPS index of the site the operator named *op_i* acts on. We **require** the operators to be sorted (strictly ascending) by sites. If necessary, call *order_combine_term()* beforehand.

sites [list of *Site*] Defines the local Hilbert space for each site. Used to check whether the operators need Jordan-Wigner strings.

op_string [None | str] Operator name to be used as operator string *between* the operators, or `None` if the Jordan Wigner string should be figured out.

Returns

strength, ijkl, ops_ijkl, op_string : Arguments for *MultiCouplingTerms.add_multi_coupling_term()* such that the added term corresponds to the parameters of this function.

max_range (*self*)
Determine the maximal range in `coupling_terms`.

Returns

max_range [int] The maximum of `j - i` for the *i*, *j* occuring in a term of `coupling_terms`.

add_to_graph (*self*, *graph*, *_i=None*, *_d1=None*, *_label_left=None*)
Add terms from `coupling_terms` to an MPOGraph.

Parameters

graph [*MPOGraph*] The graph into which the terms from `coupling_terms` should be added.

_i, _d1, _label_left [None] Should not be given; only needed for recursion.

remove_zeros (*self*, *tol_zero=1e-15*, *_d0=None*)
Remove entries close to 0 from `coupling_terms`.

Parameters

tol_zero [float] Entries in `coupling_terms` with *strength* < *tol_zero* are considered to be zero and removed.

_d0 [None] Should not be given; only needed for recursion.

to_TermList (*self*)
Convert `onsite_terms` into a *TermList*.

Returns

term_list [*TermList*] Representation of the terms as a list of terms.

add_coupling_term (*self*, *strength*, *i*, *j*, *op_i*, *op_j*, *op_string='Id'*)
Add a two-site coupling term on given MPS sites.

Parameters

strength [float] The strength of the coupling term.

i, j [int] The MPS indices of the two sites on which the operator acts. We require `0 <= i < N_sites` and `i < j`, i.e., *op_i* acts "left" of *op_j*. If j >= N_sites, it indicates couplings between unit cells of an infinite MPS.

op1, op2 [str] Names of the involved operators.

op_string [str] The operator to be inserted between *i* and *j*.

coupling_term_handle_JW (*self*, *strength*, *term*, *sites*, *op_string=None*)
Helping function to call before *add_multi_coupling_term()*.

Parameters

strength [float] The strength of the coupling term.

term [[(str, int), (str, int)]] List of two tuples `(op, i)` where *i* is the MPS index of the site the operator named *op* acts on.

sites [list of *Site*] Defines the local Hilbert space for each site. Used to check whether the operators need Jordan-Wigner strings.

op_string [None | str] Operator name to be used as operator string *between* the operators, or `None` if the Jordan Wigner string should be figured out.

Returns

strength, i, j, op_i, op_j, op_string: Arguments for *MultiCouplingTerms.add_multi_coupling_term()* such that the added term corresponds to the parameters of this function.

plot_coupling_terms(*self*, *ax*, *lat*, *style_map='default'*, *common_style={'linestyle': '--'}*, *text=None*, *text_pos=0.4*)

"Plot coupling terms into a given lattice.

This function plots the `coupling_terms`

Parameters

ax [`matplotlib.axes.Axes`] The axes on which we should plot.

lat [*Lattice*] The lattice for plotting the couplings, most probably the `M.lat` of the corresponding model `M`, see `lat`.

style_map [function | None] Function which get's called with arguments `i, j, op_i, op_string, op_j, strength` for each two-site coupling and should return a keyword-dictionary with the desired plot-style for this coupling. By default (`None`), the *linewidth* is given by the absolute value of *strength*, and the linecolor depends on the phase of *strength* (using the *hsv* colormap).

common_style [dict] Common style, which overwrites values of the dictionary returned by style_map. A `'label'` is only used for the first plotted line.

text: format_string | None If not `None`, we add text labeling the couplings in the plot. Available keywords are `i, j, op_i, op_string, op_j, strength` as well as `strength_abs, strength_angle, strength_real`.

text_pos [float] Specify where to put the text on the line between *i* (0.0) and *j* (1.0), e.g. *0.5* is exactly in the middle between *i* and *j*.

See also:

tenpy.models.lattice.Lattice.plot_sites plot the sites of the lattice.

to_nn_bond_Arrays(*self*, *sites*)

Convert the `coupling_terms` into Arrays on nearest neighbor bonds.

Parameters

sites [list of *Site*] Defines the local Hilbert space for each site. Used to translate the operator names into *Array*.

Returns

H_bond [list of {*Array* | None}] The `coupling_terms` rewritten as `sum_i H_bond[i]` for MPS indices `i`. `H_bond[i]` acts on sites `(i-1, i)`, `None` represents 0. Legs of each `H_bond[i]` are `['p0', 'p0*', 'p1', 'p1*']`.

OnsiteTerms

- full name: tenpy.networks.terms.OnsiteTerms
- parent module: *tenpy.networks.terms*
- type: class

class tenpy.networks.terms.**OnsiteTerms**(*L*)
Bases: object

Operator names, site indices and strengths representing onsite terms.

Represents a sum of onsite terms where the operators are only given by their name (in the form of a string). What the operator represents is later given by a list of *Site* with *get_op()*.

Parameters

L [int] Number of sites.

Attributes

L [int] Number of sites.

onsite_terms [list of dict] Filled by meth:*add_onsite_term*. For each index *i* a dictionary `{'opname': strength}` defining the onsite terms.

Methods

add_onsite_term(self, strength, i, op)	Add a onsite term on a given MPS site.
add_to_graph(self, graph)	Add terms from `onsite_terms` to an MPOGraph.
add_to_nn_bond_Arrays(self, H_bond, sites, ...)	Add `self.onsite_terms` into nearest-neighbor bond arrays.
remove_zeros(self[, tol_zero])	Remove entries close to 0 from `onsite_terms`.
to_Arrays(self, sites)	Convert the `onsite_terms` into a list of np_conserved Arrays.
to_TermList(self)	Convert `onsite_terms` into a *TermList*.

add_onsite_term(*self*, *strength*, *i*, *op*)
Add a onsite term on a given MPS site.

Parameters

strength [float] The strength of the term.

i [int] The MPS index of the site on which the operator acts. We require `0 <= i < L`.

op [str] Name of the involved operator.

add_to_graph(*self*, *graph*)
Add terms from `onsite_terms` to an MPOGraph.

Parameters

graph [*MPOGraph*] The graph into which the terms from `onsite_terms` should be added.

to_Arrays(*self*, *sites*)
Convert the `onsite_terms` into a list of np_conserved Arrays.

Parameters

sites [list of *Site*] Defines the local Hilbert space for each site. Used to translate the operator names into *Array*.

Returns

onsite_arrays [list of *Array*] Onsite terms represented by *self*. Entry *i* of the list lives on `sites[i]`.

remove_zeros(*self*, *tol_zero=1e-15*)
Remove entries close to 0 from `onsite_terms`.

Parameters

tol_zero [float] Entries in `onsite_terms` with *strength* < *tol_zero* are considered to be zero and removed.

add_to_nn_bond_Arrays(*self*, *H_bond*, *sites*, *finite*, *distribute=(0.5, 0.5)*)
Add `self.onsite_terms` into nearest-neighbor bond arrays.

Parameters

H_bond [list of {*Array* | None}] The `coupling_terms` rewritten as `sum_i H_bond[i]` for MPS indices `i`. `H_bond[i]` acts on sites `(i-1, i)`, `None` represents 0. Legs of each `H_bond[i]` are `['p0', 'p0*', 'p1', 'p1*']`. Modified *in place*.

sites [list of *Site*] Defines the local Hilbert space for each site. Used to translate the operator names into *Array*.

distribute [(float, float)] How to split the onsite terms (in the bulk) into the bond terms to the left (`distribute[0]`) and right (`distribute[1]`).

finite [bool] Boundary conditions of the MPS, `MPS.finite`. If finite, we distribute the onsite term of the

to_TermList(*self*)
Convert `onsite_terms` into a *TermList*.

Returns

term_list [*TermList*] Representation of the terms as a list of terms.

TermList

- full name: tenpy.networks.terms.TermList
- parent module: *tenpy.networks.terms*
- type: class

class tenpy.networks.terms.**TermList**(*terms*, *strength*)
Bases: object

A list of terms (=operator names and sites they act on) and associated strengths.

A representation of terms, similar as *OnsiteTerms*, *CouplingTerms* and *MultiCouplingTerms*.

This class does not store operator strings between the sites. Jordan-Wigner strings of fermions are added during conversion to (Multi)CouplingTerms.

Parameters

terms [list of list of (str, int)] List of terms where each *term* is a list of tuples `(opname, i)` of an operator name and a site *i* it acts on. For Fermions, the order is the order in the mathematic sense, i.e., the right-most/last operator in the list acts last.

strengths [list of float/complex] For each term in *terms* an associated prefactor or strength (e.g. expectation value).

Attributes

terms [list of list of (str, int)] List of terms where each *term* is a tuple `(opname, i)` of an operator name and a site *i* it acts on.

strengths [1D ndarray] For each term in *terms* an associated prefactor or strength (e.g. expectation value).

Methods

order_combine(self, sites)	Order and combine operators in each term.
to_OnsiteTerms_CouplingTerms(self, sites)	Convert to *OnsiteTerms* and *CouplingTerms*

to_OnsiteTerms_CouplingTerms(*self*, *sites*)

Convert to *OnsiteTerms* and *CouplingTerms*

Performs Jordan-Wigner transformation for fermionic operators.

Parameters

sites [list of *Site*] Defines the local Hilbert space for each site. Used to check whether the operators need Jordan-Wigner strings. The length is used as *L* for the *onsite_terms* and *coupling_terms*.

Returns

onsite_terms [*OnsiteTerms*] Onsite terms.

coupling_terms [*CouplingTerms* | *MultiCouplingTerms*] Coupling terms. If *self* contains terms involving more than two operators, a *MultiCouplingTerms* instance, otherwise just *CouplingTerms*.

order_combine(*self*, *sites*)

Order and combine operators in each term.

Parameters

sites [list of *Site*] Defines the local Hilbert space for each site. Used to check whether the operators anticommute (= whether they need Jordan-Wigner strings) and for multiplication rules.

See also:

order_and_combine_term does it for a single term.

Functions

order_combine_term(term, sites)	Combine operators in a term to one terms per site.

order_combine_term

- full name: tenpy.networks.terms.order_combine_term
- parent module: *tenpy.networks.terms*
- type: function

tenpy.networks.terms.**order_combine_term**(*term*, *sites*)
Combine operators in a term to one terms per site.

Takes in a term of operators and sites they acts on, commutes operators to order them by site and combines operators acting on the same site with *multiply_op_names()*.

Parameters

term [a list of (opname_i, i) tuples] Represents a product of onsite operators with site indices *i* they act on. Needs not to be ordered and can have multiple entries acting on the same site.

sites [list of *Site*] Defines the local Hilbert space for each site. Used to check whether the operators anticommute (= whether they need Jordan-Wigner strings) and for multiplication rules.

Returns

combined_term : Equivalent to *term* but with at most one operator per site.

overall_sign [+1 | -1 | 0] Comes from the (anti-)commutation relations. When the operators in *term* are multiplied from left to right, and then multiplied by *overall_sign*, the result is the same operator as the product of *combined_term* from left to right.

Module description

Classes to store a collection of operator names and sites they act on, together with prefactors.

This modules collects classes which are not strictly speaking tensor networks but represent "terms" acting on them. Each term is given by a collection of (onsite) operator names and indices of the sites it acts on. Moreover, we associate a *strength* to each term, which corresponds to the prefactor when specifying e.g. a Hamiltonian.

purification_mps

- full name: tenpy.networks.purification_mps
- parent module: *tenpy.networks*
- type: module

Classes

PurificationMPS(sites, Bs, SVs[, bc, form, norm])	An MPS representing a finite-temperature ensemble using purification.

PurificationMPS

- full name: tenpy.networks.purification_mps.PurificationMPS
- parent module: *tenpy.networks.purification_mps*
- type: class

class tenpy.networks.purification_mps.**PurificationMPS**(*sites*, *Bs*, *SVs*, *bc='finite'*, *form='B'*, *norm=1.0*)

Bases: *tenpy.networks.mps.MPS*

An MPS representing a finite-temperature ensemble using purification.

Similar as an MPS, but each B has now the four legs `'vL', 'vR', 'p', 'q'`. From the point of algorithms, it is to be considered as a ususal MPS by combining the legs p and q, but all physical operators act only on the p part. For example, the right-canonical form is defined as if the legs 'p' and 'q' would be combined, e.g. a right-canonical B full-fills:

```
npc.tensordot(B, B.conj(),axes=[['vR', 'p', 'q'], ['vR*', 'p*', 'q*']]) == \
    npc.eye_like(B, axes='vL')  # up to round-off errors
```

For expectation values / correlation functions, all operators are to understood to act on p only, i.e. they act trivial on q, so we just trace over `'q', 'q*'`.

See also the docstring of the module for details.

Attributes

L	Number of physical sites; for an iMPS the len of the MPS unit cell.
chi	Dimensions of the (nontrivial) virtual bonds.
dim	List of local physical dimensions.
finite	Distinguish MPS vs iMPS.
nontrivial_bonds	Slice of the non-trivial bond indices, depending on `self.bc`.

Methods

add(self, other, alpha, beta[, cutoff])	Return an MPS which represents `alpha|self> + beta |others>`.
apply_local_op(self, i, op[, unitary, ...])	Apply a local (one or multi-site) operator to *self*.
average_charge(self[, bond])	Return the average charge for the block on the left of a given bond.
canonical_form(self[, renormalize])	Bring self into canonical 'B' form, (re-)calculate singular values.
canonical_form_finite(self[, renormalize, ...])	Bring a finite (or segment) MPS into canonical form (in place).

Continued on next page

Table 148 – continued from previous page

canonical_form_infinite(self[, renormalize, ...])	Bring an infinite MPS into canonical form (in place).
charge_variance(self[, bond])	Return the charge variance on the left of a given bond.
compute_K(self, perm[, swap_op, trunc_par, ...])	Compute the momentum quantum numbers of the entanglement spectrum for 2D states.
convert_form(self[, new_form])	Tranform self into different canonical form (by scaling the legs with singular values).
copy(self)	Returns a copy of *self*.
correlation_function(self, ops1, ops2[, ...])	Correlation function `<psi|op1_i op2_j|psi>/<psi|psi>` of single site operators.
correlation_length(self[, target, tol_ev0, ...])	Calculate the correlation length by diagonalizing the transfer matrix.
entanglement_entropy(self[, n, bonds, ...])	Calculate the (half-chain) entanglement entropy for all nontrivial bonds.
entanglement_entropy_segment(self[, ...])	Calculate entanglement entropy for general geometry of the bipartition.
entanglement_spectrum(self[, by_charge])	return entanglement energy spectrum.
expectation_value(self, ops[, sites, axes])	Expectation value `<psi|ops|psi>/<psi|psi>` of (n-site) operator(s).
expectation_value_multi_sites(self, ...)	Expectation value `<psi|op0_{i0}op1_{i0+1}...opN_{i0+N}|psi>/<psi|psi>`.
expectation_value_term(self, term[, autoJW])	Expectation value `<psi|op_{i0}op_{i1}...op_{iN}|psi>/<psi|psi>`.
expectation_value_terms_sum(self, term_list)	Calculate expectation values for a bunch of terms and sum them up.
from_Bflat(sites, Bflat[, SVs, bc, dtype, ...])	Construct a matrix product state from a set of numpy arrays *Bflat* and singular vals.
from_full(sites, psi[, form, cutoff, ...])	Construct an MPS from a single tensor *psi* with one leg per physical site.
from_infiniteT(sites[, bc, form])	Initial state corresponding to infinite-Temperature ensemble.
from_product_state(sites, p_state[, bc, ...])	Construct a matrix product state from a given product state.
from_singlets(site, L, pairs[, up, down, ...])	Create an MPS of entangled singlets.
gauge_total_charge(self[, qtotal, vL_leg, ...])	Gauge the legcharges of the virtual bonds such that the MPS has a total *qtotal*.
get_B(self, i[, form, copy, cutoff, label_p])	Return (view of) *B* at site *i* in canonical form.
get_SL(self, i)	Return singular values on the left of site *i*
get_SR(self, i)	Return singular values on the right of site *i*
get_grouped_mps(self, blocklen)	contract blocklen subsequent tensors into a single one and return result as a new MPS.
get_op(self, op_list, i)	Given a list of operators, select the one corresponding to site *i*.
get_rho_segment(self, segment)	Return reduced density matrix for a segment.
get_theta(self, i[, n, cutoff, formL, formR])	Calculates the *n*-site wavefunction on `sites[i:i+n]`.
get_total_charge(self[, only_physical_legs])	Calculate and return the *qtotal* of the whole MPS (when contracted).
group_sites(self[, n, grouped_sites])	Modify *self* inplace to group sites.

Continued on next page

Table 148 – continued from previous page

group_split(self[, trunc_par])	Modify *self* inplace to split previously grouped sites.
increase_L(self[, new_L])	Modify *self* inplace to enlarge the unit cell.
mutinf_two_site(self[, max_range, n, legs])	Calculate the two-site mutual information $I(i:j)$.
norm_test(self)	Check that self is in canonical form.
overlap(self, other[, charge_sector, ...])	Compute overlap <self|other>.
permute_sites(self, perm[, swap_op, ...])	Applies the permutation perm to the state (inplace).
probability_per_charge(self[, bond])	Return probabilites of charge value on the left of a given bond.
set_B(self, i, B[, form])	Set *B* at site *i*.
set_SL(self, i, S)	Set singular values on the left of site *i*
set_SR(self, i, S)	Set singular values on the right of site *i*
swap_sites(self, i[, swapOP, trunc_par])	Swap the two neighboring sites *i* and *i+1* (inplace).
test_sanity(self)	Sanity check, raises ValueErrors, if something is wrong.

test_sanity(*self*)
Sanity check, raises ValueErrors, if something is wrong.

copy(*self*)
Returns a copy of *self*.

The copy still shares the sites, chinfo, and LegCharges of the _B, but the values of B and S are deeply copied.

classmethod from_infiniteT(*sites*, *bc='finite'*, *form='B'*)
Initial state corresponding to infinite-Temperature ensemble.

Parameters

sites [list of Site] The sites defining the local Hilbert space.

bc [{'finite', 'segment', 'infinite'}] MPS boundary conditions as described in MPS.

form [(list of) {'B' | 'A' | 'C' | 'G' | None | tuple(float, float)}] The canonical form of the stored 'matrices', see table in mps. A single choice holds for all of the entries.

Returns

infiniteT_MPS [PurificationMPS] Describes the infinite-temperature (grand canonical) ensemble, i.e. expectation values give a trace over all basis states.

entanglement_entropy_segment(*self, segment=[0], first_site=None, n=1, legs='p'*)
Calculate entanglement entropy for general geometry of the bipartition.

This function is similar as entanglement_entropy(), but for more general geometry of the region *A* to be a segment of a *few* sites.

This is acchieved by explicitly calculating the reduced density matrix of *A* and thus works only for small segments.

Parameters

segment [list of int] Given a first site *i*, the region A_i is defined to be [i+j for j in segment].

first_site [None | (iterable of) int] Calculate the entropy for segments starting at these sites. None defaults to range(L-segment[-1]) for finite or *range(L)* for infinite boundary conditions.

n [int | float] Selects which entropy to calculate; *n=1* (default) is the ususal von-Neumann entanglement entropy, otherwise the *n*-th Renyi entropy.

leg [‘p’, ‘q’, ‘pq’] Whether we look at the entanglement entropy in both (*pq*) or only one of auxiliar (*q*) and physical (*p*) space.

Returns

entropies [1D ndarray] `entropies[i]` contains the entropy for the the region `A_i` defined above.

mutinf_two_site(*self*, *max_range=None*, *n=1*, *legs='p'*)
Calculate the two-site mutual information $I(i:j)$.

Calculates $I(i:j) = S(i) + S(j) - S(i,j)$, where $S(i)$ is the single site entropy on site i and $S(i,j)$ the two-site entropy on sites i, j.

Parameters

max_range [int] Maximal distance `|i-j|` for which the mutual information should be calculated. `None` defaults to *L-1*.

n [float] Selects the entropy to use, see *entropy()*.

leg [‘p’, ‘q’, ‘pq’] Whether we look at the entanglement entropy in both (*pq*) or only one of auxiliar (*q*) and physical (*p*) space.

Returns

coords [2D array] Coordinates for the mutinf array.

mutinf [1D array] `mutinf[k]` is the mutual information $I(i:j)$ between the sites `i, j = coords[k]`.

swap_sites(*self*, *i*, *swapOP='auto'*, *trunc_par={}*)
Swap the two neighboring sites *i* and *i+1* (inplace).

Exchange two neighboring sites: form theta, ‘swap’ the physical legs and split with an svd. While the ‘swap’ is just a transposition/relabeling for bosons, one needs to be careful about the sign for fermions.

Parameters

i [int] Swap the two sites at positions *i* and *i+1*.

swap_op [`None`|`'auto'`|*Array*] The operator used to swap the phyiscal legs of the two-site wave function *theta*. For `None`, just transpose/relabel the legs, for `'auto'` also take care of fermionic signs. Alternative give an npc *Array* which represents the full operator used for the swap. Should have legs `['p0', 'p1', 'p0*', 'p1*']` whith `'p0', 'p1*'` contractible.

trunc_par [dict] Parameters for truncation, see *truncate()*. *chi_max* defaults to `max(self.chi)`.

Returns

trunc_err [*TruncationError*] The error of the represented state introduced by the truncation after the swap.

property L
Number of physical sites; for an iMPS the len of the MPS unit cell.

add(*self*, *other*, *alpha*, *beta*, *cutoff=1e-15*)
Return an MPS which represents `alpha|self> + beta |others>`.

Works only for 'finite', 'segment' boundary conditions. For 'segment' boundary conditions, the virtual legs on the very left/right are assumed to correspond to each other (i.e. self and other have the same state outside of the considered segment). Takes into account `norm`.

Parameters

other [`MPS`] Another MPS of the same length to be added with self.

alpha, beta [complex float] Prefactors for self and other. We calculate `alpha * |self> + beta * |other>`

cutoff [float | None] Cutoff of singular values used in the SVDs.

Returns

sum [`MPS`] An MPS representing `alpha|self> + beta |other>`. Has same total charge as *self*.

U_L, V_R [*Array*] Only returned for `'segment'` boundary conditions. The unitaries defining the new left and right Schmidt states in terms of the old ones, with legs `'vL', 'vR'`.

apply_local_op(*self*, *i*, *op*, *unitary=None*, *renormalize=False*, *cutoff=1e-13*)

Apply a local (one or multi-site) operator to *self*.

Note that this destroys the canonical form if the local operator is non-unitary. Therefore, this function calls *canonical_form()* if necessary.

Parameters

i [int] (Left-most) index of the site(s) on which the operator should act.

op [str | npc.Array] A physical operator acting on site *i*, with legs `'p', 'p*'` for a single-site operator or with legs `['p0', 'p1', ...], ['p0*', 'p1*', ...]` for an operator acting on *n*>=2 sites. Strings (like `'Id', 'Sz'`) are translated into single-site operators defined by `sites`.

unitary [None | bool] Whether *op* is unitary, i.e., whether the canonical form is preserved (`True`) or whether we should call *canonical_form()* (`False`). `None` checks whether `norm(op dagger(op) - identity)` is smaller than *cutoff*.

renormalize [bool] Whether the final state should keep track of the norm (False, default) or be renormalized to have norm 1 (True).

cutoff [float] Cutoff for singular values if *op* acts on more than one site (see *from_full()*). (And used as cutoff for a unspecified *unitary*.)

average_charge(*self*, *bond=0*)

Return the average charge for the block on the left of a given bond.

For example for particle number conservation, define $N_b = sum_{i<b} n_i$ for a given bond b. Then this function returns $< \psi | N_b | \psi >$.

Parameters

bond [int] The bond to be considered. The returned charges are summed over the sites left of *bond*.

Returns

average_charge [1D array] For each type of charge in `chinfo` the average value when summing the charge values over sites left of the given bond.

canonical_form(*self*, *renormalize=True*)
Bring self into canonical 'B' form, (re-)calculate singular values.

Simply calls *canonical_form_finite()* or *canonical_form_infinite()*.

canonical_form_finite(*self*, *renormalize=True*, *cutoff=0.0*)
Bring a finite (or segment) MPS into canonical form (in place).

If any site is in `form` `None`, it does *not* use any of the singular values *S* (for 'finite' boundary conditions, or only the very left *S* for 'segment' b.c.). If all sites have a *form*, it respects the *form* to ensure that one *S* is included per bond. The final state is always in right-canonical 'B' form.

Performs one sweep left to right doing QR decompositions, and one sweep right to left doing SVDs calculating the singular values.

Parameters

renormalize: bool Whether a change in the norm should be discarded or used to update `norm`.

cutoff [float | None] Cutoff of singular values used in the SVDs.

Returns

U_L, V_R [*Array*] Only returned for `'segment'` boundary conditions. The unitaries defining the new left and right Schmidt states in terms of the old ones, with legs `'vL'`, `'vR'`.

canonical_form_infinite(*self*, *renormalize=True*, *tol_xi=1000000.0*)
Bring an infinite MPS into canonical form (in place).

If any site is in `form` `None`, it does *not* use any of the singular values *S*. If all sites have a *form*, it respects the *form* to ensure that one *S* is included per bond. The final state is always in right-canonical 'B' form.

Proceeds in three steps, namely 1) diagonalize right and left transfermatrix on a given bond to bring that bond into canonical form, and then 2) sweep right to left, and 3) left to right to bringing other bonds into canonical form.

Parameters

renormalize: bool Whether a change in the norm should be discarded or used to update `norm`.

tol_xi [float] Raise an error if the correlation length is larger than that (which indicates a degenerate "cat" state, e.g., for spontaneous symmetry breaking).

charge_variance(*self*, *bond=0*)
Return the charge variance on the left of a given bond.

For example for particle number conservation, define $N_b = sum_{i<b} n_i$ for a given bond b. Then this function returns $< \psi | N_b^2 | \psi > - (< \psi | N_b | \psi >)^2$.

Parameters

bond [int] The bond to be considered. The returned charges are summed over the sites left of *bond*.

Returns

average_charge [1D array] For each type of charge in `chinfo` the variance of of the charge values left of the given bond.

property chi
Dimensions of the (nontrivial) virtual bonds.

compute_K(*self*, *perm*, *swap_op='auto'*, *trunc_par=None*, *canonicalize=1e-06*, *verbose=0*)
Compute the momentum quantum numbers of the entanglement spectrum for 2D states.

Works for an infinite MPS living on a cylinder, infinitely long in *x* direction and with periodic boundary conditions in *y* directions. If the state is invariant under 'rotations' around the cylinder axis, one can find the momentum quantum numbers of it. (The rotation is nothing more than a translation in *y*.) This function permutes some sites (on a copy of *self*) to enact the rotation, and then finds the dominant eigenvector of the mixed transfer matrix to get the quantum numbers, along the lines of [PollmannTurner2012], see also (the appendix and Fig. 11 in the arXiv version of) [CincioVidal2013].

Parameters

perm [1D ndarray | *Lattice*] Permuation to be applied to the physical indices, see *permute_sites()*. If a lattice is given, we use it to read out the lattice structure and shift each site by one lattice-vector in y-direction (assuming periodic boundary conditions). (If you have a *CouplingModel*, give its *lat* attribute for this argument)

swap_op [None | 'auto' | *Array*] The operator used to swap the phyiscal legs of a two-site wave function *theta*, see *swap_sites()*.

trunc_par [dict] Parameters for truncation, see *truncate()*. If not set, *chi_max* defaults to max(self.chi).

canonicalize [float] Check that *self* is in canonical form; call *canonical_form()* if *norm_test()* yields np.linalg.norm(self.norm_test()) > canonicalize.

verbose [float] Level of verbosity, print status messages if verbose > 0.

Returns

U [*Array*] Unitary representation of the applied permutation on left Schmidt states.

W [ndarray] 1D array of the form S**2 exp(i K), where *S* are the Schmidt values on the left bond. You can use np.abs() and np.angle() to extract the Schmidt values *S* and momenta *K* from *W*.

q [*LegCharge*] LegCharge corresponding to *W*.

ov [complex] The eigenvalue of the mixed transfer matrix <*psi*|*T*|*psi*> per *L* sites. An absolute value different smaller than 1 indicates that the state is not invariant under the permutation or that the truncation error *trunc_err* was too large!

trunc_err [*TruncationError*] The error of the represented state introduced by the truncation after swaps when performing the truncation.

convert_form(*self*, *new_form='B'*)
Tranform self into different canonical form (by scaling the legs with singular values).

Parameters

new_form [(list of) {'B' | 'A' | 'C' | 'G' | 'Th' | None | tuple(float, float)}] The form the stored 'matrices'. The table in module doc-string. A single choice holds for all of the entries.

Raises

ValueError [if trying to convert from a None form. Use *canonical_form()* instead!]

correlation_function(*self*, *ops1*, *ops2*, *sites1=None*, *sites2=None*, *opstr=None*, *str_on_first=True*, *hermitian=False*)
Correlation function <psi|op1_i op2_j|psi>/<psi|psi> of single site operators.

Given the MPS in canonical form, it calculates 2-site correlation functions. For examples the contraction for a two-site operator on site *i* would look like:

```
|           .--S--B[i]--B[i+1]--...--B[j]---.
|           |     |     |             |      |
|           |     |     |            op2     |
|           |    op1    |             |      |
|           |     |     |             |      |
|           .--S--B*[i]-B*[i+1]-...--B*[j]--.
```

Onsite terms are taken in the order `<psi | op1 op2 | psi>`.

If *opstr* is given and `str_on_first=True`, it calculates:

```
|            for i < j                              for i > j
|
|           .--S--B[i]---B[i+1]--...- B[j]---.     .--S--B[j]---B[j+1]--...-␣
↪B[i]---.
|           |     |     |             |      |     |     |      |            |␣
↪     |
|           |     opstr  opstr        op2    |     |     op2    |            |␣
↪     |
|           |     |     |             |      |     |     |      |            |␣
↪     |
|           |     op1   |             |      |     |     opstr  opstr       ␣
↪op1    |
|           |     |     |             |      |     |     |      |            |␣
↪     |
|           .--S--B*[i]--B*[i+1]-...- B*[j]--.     .--S--B*[j]--B*[j+1]-...-␣
↪B*[i]--.
```

For `i==j`, no *opstr* is included. For `str_on_first=False`, the *opstr* on site `min(i, j)` is always left out.

Strings (like `'Id', 'Sz'`) in the arguments are translated into single-site operators defined by the *Site* on which they act. Each operator should have the two legs `'p', 'p*'`.

Parameters

ops1 [(list of) { *Array* | str }] First operator of the correlation function (acting after ops2). `ops1[x]` acts on site `sites1[x]`. If less than `len(sites1)` operators are given, we repeat them periodically.

ops2 [(list of) { *Array* | str }] Second operator of the correlation function (acting before ops1). `ops2[y]` acts on site `sites2[y]`. If less than `len(sites2)` operators are given, we repeat them periodically.

sites1 [None | int | list of int] List of site indices; a single *int* is translated to `range(0, sites1)`. `None` defaults to all sites `range(0, L)`. Is sorted before use, i.e. the order is ignored.

sites2 [None | int | list of int] List of site indices; a single *int* is translated to `range(0, sites2)`. `None` defaults to all sites `range(0, L)`. Is sorted before use, i.e. the order is ignored.

opstr [None | (list of) { *Array* | str }] Ignored by default (`None`). Operator(s) to be inserted between `ops1` and `ops2`. If less than *L* operators are given, we repeat them periodically. If given as a list, `opstr[r]` is inserted at site *r* (independent of *sites1* and *sites2*).

str_on_first [bool] Whether the *opstr* is included on the site `min(i, j)`. Note the order, which is chosen that way to handle fermionic Jordan-Wigner strings correctly. (In other

words: choose `str_on_first=True` for fermions!)

hermitian [bool] Optimization flag: if `sites1 == sites2` and `Ops1[i]^\dagger == Ops2[i]` (which is not checked explicitly!), the resulting `C[x, y]` will be hermitian. We can use that to avoid calculations, so `hermitian=True` will run faster.

Returns

C [2D ndarray] The correlation function `C[x, y] = <psi|ops1[i] ops2[j]|psi>`, where `ops1[i]` acts on site `i=sites1[x]` and `ops2[j]` on site `j=sites2[y]`. If *opstr* is given, it gives (for `str_on_first=True`):

- For `i < j`: `C[x, y] = <psi|ops1[i] prod_{i <= r < j} opstr[r] ops2[j]|psi>`.
- For `i > j`: `C[x, y] = <psi|prod_{j <= r < i} opstr[r] ops1[i] ops2[j]|psi>`.
- For `i = j`: `C[x, y] = <psi|ops1[i] ops2[j]|psi>`.

The condition `<= r` is replaced by a strict `< r`, if `str_on_first=False`.

correlation_length (*self*, *target=1*, *tol_ev0=1e-08*, *charge_sector=0*)

Calculate the correlation length by diagonalizing the transfer matrix.

Assumes that *self* is in canonical form.

Works only for infinite MPS, where the transfer matrix is a useful concept. Assuming a single-site unit cell, any correlation function splits into $C(A_i, B_j) = A_i' T^{j-i-1} B_j'$ with some parts left and right and the $j-i-1$-th power of the transfer matrix in between. The largest eigenvalue is 1 (if self is properly normalized) and gives the dominant contribution of $A_i' E_1 * 1^{j-i-1} * E_1^T B_j' =< A >< B >$, and the second largest one gives a contribution $\propto \lambda_2^{j-i-1}$. Thus $\lambda_2 = \exp(-\frac{1}{\xi})$.

More general for a *L*-site unit cell we get $\lambda_2 = \exp(-\frac{L}{\xi})$, where the *xi* is given in units of 1 lattice spacing in the MPS.

Warning: For a higher-dimensional lattice (which the MPS class doesn't know about), the correct unit is the lattice spacing in x-direction, and the correct formula is $\lambda_2 = \exp(-\frac{L_x}{\xi})$, where *L_x* is the number of lattice spacings in the infinite direction within the MPS unit cell, e.g. the number of "rings" of a cylinder in the MPS unit cell. To get to these units, divide the returned *xi* by the number of sites within a "ring", for a lattice given in `N_sites_per_ring`.

Parameters

target [int] We look for the *target* + 1 largest eigenvalues.

tol_ev0 [float] Print warning if largest eigenvalue deviates from 1 by more than *tol_ev0*.

charge_sector [None | charges | `0`] Selects the charge sector in which the dominant eigenvector of the TransferMatrix is. `None` stands for *all* sectors, `0` stands for the zero-charge sector. Defaults to `0`, i.e., *assumes* the dominant eigenvector is in charge sector 0.

Returns

xi [float | 1D array] If *target`=1, return just the correlation length, otherwise an array of the `target* largest correlation lengths. It is measured in units of a single lattice spacing in the MPS language, see the warning above.

property dim

List of local physical dimensions.

entanglement_entropy(*self*, *n=1*, *bonds=None*, *for_matrix_S=False*)
Calculate the (half-chain) entanglement entropy for all nontrivial bonds.

Consider a bipartition of the sytem into $A = \{j : j <= i_b\}$ and $B = \{j : j > i_b\}$ and the reduced density matrix $\rho_A = tr_B(\rho)$. The von-Neumann entanglement entropy is defined as $S(A, n = 1) = -tr(\rho_A \log(\rho_A)) = S(B, n = 1)$. The generalization for `n != 1, n>0` are the Renyi entropies: $S(A, n) = \frac{1}{1-n} \log(tr(\rho_A^2)) = S(B, n = 1)$

This function calculates the entropy for a cut at different bonds *i*, for which the the eigenvalues of the reduced density matrix ρ_A and ρ_B is given by the squared schmidt values *S* of the bond.

Parameters

n [int/float] Selects which entropy to calculate; *n=1* (default) is the ususal von-Neumann entanglement entropy.

bonds [`None` | (iterable of) int] Selects the bonds at which the entropy should be calculated. `None` defaults to `range(0, L+1)[self.nontrivial_bonds]`.

for_matrix_S [bool] Switch calculate the entanglement entropy even if the *_S* are matrices. Since $O(\chi^3)$ is expensive compared to the ususal $O(\chi)$, we raise an error by default.

Returns

entropies [1D ndarray] Entanglement entropies for half-cuts. *entropies[j]* contains the entropy for a cut at bond `bonds[j]` (i.e. left to site `bonds[j]`).

entanglement_spectrum(*self*, *by_charge=False*)
return entanglement energy spectrum.

Parameters

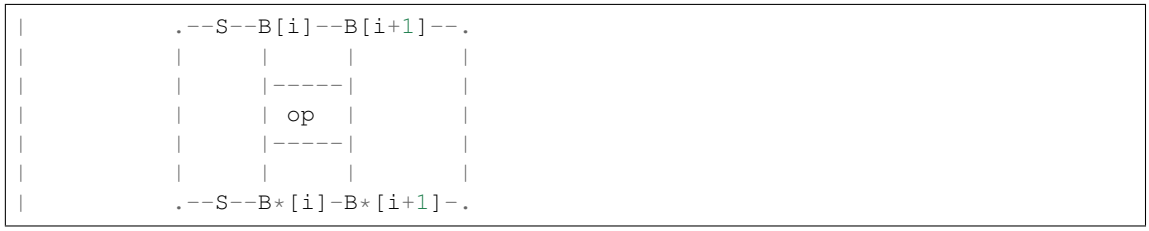
by_charge [bool] Wheter we should sort the spectrum on each bond by the possible charges.

Returns

ent_spectrum [list] For each (non-trivial) bond the entanglement spectrum. If *by_charge* is `False`, return (for each bond) a sorted 1D ndarray with the convention $S_i^2 = e^{-\xi_i}$, where S_i labels a Schmidt value and ξ_i labels the entanglement 'energy' in the returned spectrum. If *by_charge* is True, return a a list of tuples `(charge, sub_spectrum)` for each possible charge on that bond.

expectation_value(*self*, *ops*, *sites=None*, *axes=None*)
Expectation value `<psi|ops|psi>/<psi|psi>` of (n-site) operator(s).

Given the MPS in canonical form, it calculates n-site expectation values. For example the contraction for a two-site (*n* = 2) operator on site *i* would look like:



Parameters

ops [(list of) { *Array* | str }] The operators, for wich the expectation value should be taken, All operators should all have the same number of legs (namely *2 n*). If less than *self.L* operators are given, we repeat them periodically. Strings (like `'Id', 'Sz'`) are translated into single-site operators defined by `sites`.

sites [None | list of int] List of site indices. Expectation values are evaluated there. If `None` (default), the entire chain is taken (clipping for finite b.c.)

axes [None | (list of str, list of str)] Two lists of each n leg labels giving the physical legs of the operator used for contraction. The first n legs are contracted with conjugated B, the second n legs with the non-conjugated B. `None` defaults to `(['p'], ['p*'])` for single site operators ($n = 1$), or `(['p0', 'p1', ... 'p{n-1}'], ['p0*', 'p1*', .... 'p{n-1}*'])` for $n > 1$.

Returns

exp_vals [1D ndarray] Expectation values, `exp_vals[i] = <psi|ops[i]|psi>`, where `ops[i]` acts on site(s) `j, j+1, ..., j+{n-1}` with `j=sites[i]`.

Examples

One site examples (n=1):

```
>>> psi.expectation_value('Sz')
[Sz0, Sz1, ..., Sz{L-1}]
>>> psi.expectation_value(['Sz', 'Sx'])
[Sz0, Sx1, Sz2, Sx3, ... ]
>>> psi.expectation_value('Sz', sites=[0, 3, 4])
[Sz0, Sz3, Sz4]
```

Two site example (n=2), assuming homogeneous sites:

```
>>> SzSx = npc.outer(psi.sites[0].Sz.replace_labels(['p', 'p*'], ['p0', 'p0*
↪']),
                     psi.sites[1].Sx.replace_labels(['p', 'p*'], ['p1', 'p1*
↪']))
>>> psi.expectation_value(SzSx)
[Sz0Sx1, Sz1Sx2, Sz2Sx3, ... ]   # with len L-1 for finite bc, or L for␣
↪infinite
```

Example measuring <psi|SzSx|psi2> on each second site, for inhomogeneous sites:

```
>>> SzSx_list = [npc.outer(psi.sites[i].Sz.replace_labels(['p', 'p*'], ['p0',
↪'p0*']),
                           psi.sites[i+1].Sx.replace_labels(['p', 'p*'], ['p1
↪', 'p1*']))
                 for i in range(0, psi.L-1, 2)]
>>> psi.expectation_value(SzSx_list, range(0, psi.L-1, 2))
[Sz0Sx1, Sz2Sx3, Sz4Sx5, ...]
```

expectation_value_multi_sites (*self*, *operators*, *i0*)

Expectation value `<psi|op0_{i0}op1_{i0+1}...opN_{i0+N}|psi>/<psi|psi>`.

Calculates the expectation value of a tensor product of single-site operators acting on different sites next to each other. In other words, evaluate the expectation value of a term `op0_i0 op1_{i0+1} op2_{i0+2} ....`

Parameters

operators [List of { *Array* | str }] List of one-site operators. This method calculates the expectation value of the n-sites operator given by their tensor product.

i0 [int] The left most index on which an operator acts, i.e., `operators[i]` acts on site `i + i0`.

Returns

exp_val [float/complex] The expectation value of the tensorproduct of the given onsite operators, `<psi|operators[0]_{i0} operators[1]_{i0+1} ... |psi>/<psi|psi>`, where `|psi>` is the represented MPS.

expectation_value_term(*self*, *term*, *autoJW=True*)

Expectation value `<psi|op_{i0}op_{i1}...op_{iN}|psi>/<psi|psi>`.

Calculates the expectation value of a tensor product of single-site operators acting on different sites *i0*, *i1*, ... (not necessarily next to each other). In other words, evaluate the expectation value of a term `op0_i0 op1_i1 op2_i2 ...`.

For example the contraction of three one-site operators on sites *i0*, *i1=i0+1*, *i2=i0+3* would look like:

```
|           .--S--B[i0]---B[i0+1]--B[i0+2]--B[i0+3]--.
|           |     |        |         |         |      |
|           |    op1      op2        |        op3     |
|           |     |        |         |         |      |
|           .--S--B*[i0]--B*[i0+1]-B*[i0+2]-B*[i0+3]-.
```

Parameters

term [list of (str, int)] List of tuples `op, i` where *i* is the MPS index of the site the operator named *op* acts on. The order inside *term* determines the order in which they act (in the mathematical convention: the last operator in *term* is right-most, so it acts first on a Ket).

autoJW [bool] If True (default), automatically insert Jordan Wigner strings for Fermions as needed.

Returns

exp_val [float/complex] The expectation value of the tensorproduct of the given onsite operators, `<psi|op_i0 op_i1 ... op_iN |psi>/<psi|psi>`, where `|psi>` is the represented MPS.

See also:

correlation_function efficient way to evaluate many correlation functions.

Examples

```
>>> a = psi.expectation_value_term([('Sx', 2), ('Sz', 4)])
>>> b = psi.expectation_value_term([('Sz', 4), ('Sx', 2)])
>>> c = psi.expectation_value_multi_sites(['Sz', 'Id', 'Sz'], i0=2)
>>> assert a == b == c
```

expectation_value_terms_sum(*self*, *term_list*, *prefactors=None*)

Calculate expectation values for a bunch of terms and sum them up.

This is equivalent to the following expression:

```
sum([self.expectation_value_term(term)*strength for term, strength in term_
↪list])
```

However, for effiency, the term_list is converted to an MPO and the expectation value of the MPO is evaluated.

Note: Due to the way MPO expectation values are evaluated for infinite systems, it works only if all terms in the *term_list* start within the MPS unit cell.

Deprecated since version 0.4.0: *prefactor* will be removed in version 1.0.0. Instead, directly give just `TermList(term_list, prefactors)` as argument.

Parameters

term_list [*TermList*] The terms and prefactors (*strength*) to be summed up.

prefactors : Instead of specifying a *TermList*, one can also specify the term_list and strength separately. This is deprecated.

Returns

terms_sum [list of (complex) float] Equivalent to the expression `sum([self.expectation_value_term(term)*strength for term, strength in term_list])`.

_mpo : Intermediate results: the generated MPO. For a finite MPS, `terms_sum = _mpo.expectation_value(self)`, for an infinite MPS `terms_sum = _mpo.expectation_value(self) * self.L`

See also:

expectation_value_term evaluates a single *term*.

tenpy.networks.mpo.MPO.expectation_value expectation value density of an MPO.

property finite

Distinguish MPS vs iMPS.

True for an MPS (`bc='finite', 'segment'`), False for an iMPS (`bc='infinite'`).

classmethod from_Bflat(*sites*, *Bflat*, *SVs=None*, *bc='finite'*, *dtype=None*, *permute=True*, *form='B'*, *legL=None*)

Construct a matrix product state from a set of numpy arrays *Bflat* and singular vals.

Parameters

sites [list of *Site*] The sites defining the local Hilbert space.

Bflat [iterable of numpy ndarrays] The matrix defining the MPS on each site, with legs `'p', 'vL', 'vR'` (physical, virtual left/right).

SVs [list of 1D array | `None`] The singular values on *each* bond. Should always have length *L+1*. By default (`None`), set all singular values to the same value. Entries out of *nontrivial_bonds* are ignored.

bc [{'infinite', 'finite', 'segmemt'}] MPS boundary conditions. See docstring of `MPS`.

dtype [type or string] The data type of the array entries. Defaults to the common dtype of *Bflat*.

permute [bool] The `Site` might permute the local basis states if charge conservation gets enabled. If *permute* is True (default), we permute the given *Bflat* locally according to each site's `perm`. The *p_state* argument should then always be given as if *conserve=None* in the Site.

form [(list of) {`'B' | 'A' | 'C' | 'G' | None` | tuple(float, float)}] Defines the canonical form of *Bflat*. See module doc-string. A single choice holds for all of the entries.

leg_L [LegCharge | None] Leg charges at bond 0, which are purely conventional. If None, use trivial charges.

Returns

mps [MPS] An MPS with the matrices *Bflat* converted to npc arrays.

classmethod from_full(*sites*, *psi*, *form=None*, *cutoff=1e-16*, *normalize=True*, *bc='finite'*, *outer_S=None*)

Construct an MPS from a single tensor *psi* with one leg per physical site.

Performs a sequence of SVDs of psi to split off the *B* matrices and obtain the singular values, the result will be in canonical form. Obviously, this is only well-defined for *finite* or *segment* boundary conditions.

Parameters

sites [list of *Site*] The sites defining the local Hilbert space.

psi [*Array*] The full wave function to be represented as an MPS. Should have labels 'p0', 'p1', ..., 'p{L-1}'. Additionally, it may have (or must have for 'segment' *bc*) the legs 'vL', 'vR', which are trivial for 'finite' *bc*.

form ['B' | 'A' | 'C' | 'G' | None] The canonical form of the resulting MPS, see module doc-string. None defaults to 'A' form on the first site and 'B' form on all following sites.

cutoff [float] Cutoff of singular values used in the SVDs.

normalize [bool] Whether the resulting MPS should have 'norm' 1.

bc ['finite' | 'segment'] Boundary conditions.

outer_S [None | (array, array)] For 'semgent' *bc* the singular values on the left and right of the considered segment, *None* for 'finite' boundary conditions.

Returns

psi_mps [MPS] MPS representation of *psi*, in canonical form and possibly normalized.

classmethod from_product_state(*sites*, *p_state*, *bc='finite'*, *dtype=<class 'numpy.float64'>*, *permute=True*, *form='B'*, *chargeL=None*)

Construct a matrix product state from a given product state.

Parameters

sites [list of *Site*] The sites defining the local Hilbert space.

p_state [iterable of {int | str | 1D array}] Defines the product state to be represented. If p_state[i] is str, then site i is in state self.sites[i].state_labels(p_state[i]). If p_state[i] is int, then site i is in state p_state[i]. If p_state[i] is an array, then site i wavefunction is p_state[i].

bc [{'infinite', 'finite', 'segmemt'}] MPS boundary conditions. See docstring of MPS.

dtype [type or string] The data type of the array entries.

permute [bool] The Site might permute the local basis states if charge conservation gets enabled. If *permute* is True (default), we permute the given *p_state* locally according to each site's perm. The *p_state* argument should then always be given as if *conserve=None* in the Site.

form [(list of) {'B' | 'A' | 'C' | 'G' | None | tuple(float, float)}] Defines the canonical form. See module doc-string. A single choice holds for all of the entries.

chargeL [charges] Leg charge at bond 0, which are purely conventional.

Returns

product_mps [MPS] An MPS representing the specified product state.

Examples

Example to get a Neel state for a TIChain:

```
>>> M = TFIChain({'L': 10})
>>> p_state = ["up", "down"] * (L//2)  # repeats entries L/2 times
>>> psi = MPS.from_product_state(M.lat.mps_sites(), p_state, bc=M.lat.bc_MPS)
```

The meaning of the labels "up","down" is defined by the Site, in this example a *SpinHalfSite*.

Extending the example, we can replace the spin in the center with one with arbitrary angles theta, phi in the bloch sphere:

```
>>> M = TFIChain({'L': 8, 'conserve': None})
>>> p_state = ["up", "down"] * (L//2)  # repeats entries L/2 times
>>> bloch_sphere_state = np.array([np.cos(theta/2), np.exp(1.j*phi)*np.
↪sin(theta/2)])
>>> p_state[L//2] = bloch_sphere_state   # replace one spin in center
>>> psi = MPS.from_product_state(M.lat.mps_sites(), p_state, bc=M.lat.bc_MPS,␣
↪dtype=np.complex)
```

Note that for the more general *SpinChain*, the order of the two entries for the bloch_sphere_state would be *exactly the opposite* (when we keep the the north-pole of the bloch sphere being the up-state). The reason is that the *SpinChain* uses the general *SpinSite*, where the states are orderd ascending from 'down' to 'up'. The *SpinHalfSite* on the other hand uses the order 'up', 'down' where that the Pauli matrices look as usual.

Moreover, note that you can not write this bloch state (for theta != 0, pi) when conserving symmetries, as the two physical basis states correspond to different symmetry sectors.

classmethod from_singlets(*site*, *L*, *pairs*, *up='up'*, *down='down'*, *lonely=[]*, *lonely_state='up'*, *bc='finite'*)

Create an MPS of entangled singlets.

Parameters

site [*Site*] The *site* defining the local Hilbert space, taken uniformly for all sites.

L [int] The number of sites.

pairs [list of (int, int)] Pairs of sites to be entangled; the returned MPS will have a singlet for each pair in *pairs*.

up, down [int | str] A singlet is defined as (|up down> - |down up>)/2**0.5, up and down give state indices or labels defined on the corresponding site.

lonely [list of int] Sites which are not included into a singlet pair.

lonely_state [int | str] The state for the lonely sites.

bc [{'infinite', 'finite', 'segmemt'}] MPS boundary conditions. See docstring of MPS.

Returns

singlet_mps [MPS] An MPS representing singlets on the specified pairs of sites.

gauge_total_charge(*self*, *qtotal=None*, *vL_leg=None*, *vR_leg=None*)

Gauge the legcharges of the virtual bonds such that the MPS has a total *qtotal*.

Parameters

qtotal [(list of) charges] If a single set of charges is given, it is the desired total charge of the MPS (which *get_total_charge()* will return afterwards). By default (`None`), use 0 charges, unless vL_leg and vR_leg are specified, in which case we adjust the total charge to match these legs.

vL_leg [None | LegCharge] Desired new virtual leg on the very left. Needs to have the same block strucuture as current leg, but can have shifted charge entries.

vR_leg [None | LegCharge] Desired new virtual leg on the very right. Needs to have the same block strucuture as current leg, but can have shifted charge entries. Should be *vL_leg.conj()* for infinite MPS, if *qtotal* is not given.

get_B (*self*, *i*, *form='B'*, *copy=False*, *cutoff=1e-16*, *label_p=None*)
Return (view of) *B* at site *i* in canonical form.

Parameters

i [int] Index choosing the site.

form [`'B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)] The (canonical) form of the returned B. For `None`, return the matrix in whatever form it is. If any of the tuple entry is None, also don't scale on the corresponding axis.

copy [bool] Whether to return a copy even if *form* matches the current form.

cutoff [float] During DMRG with a mixer, *S* may be a matrix for which we need the inverse. This is calculated as the Penrose pseudo-inverse, which uses a cutoff for the singular values.

label_p [None | str] Ignored by default (`None`). Otherwise replace the physical label `'p'` with `'p'+label_p'`. (For derived classes with more than one "physical" leg, replace all the physical leg labels accordingly.)

Returns

B [*Array*] The MPS 'matrix' *B* at site *i* with leg labels `'vL', 'p', 'vR'`. May be a view of the matrix (if `copy=False`), or a copy (if the form changed or `copy=True`).

Raises

ValueError [if self is not in canoncial form and *form* is not None.]

get_SL (*self*, *i*)
Return singular values on the left of site *i*

get_SR (*self*, *i*)
Return singular values on the right of site *i*

get_grouped_mps (*self*, *blocklen*)
contract blocklen subsequent tensors into a single one and return result as a new MPS.

blocklen = number of subsequent sites to be combined.

Returns

new MPS object with bunched sites.

get_op (*self*, *op_list*, *i*)
Given a list of operators, select the one corresponding to site *i*.

Parameters

op_list [(list of) {str | npc.array}] List of operators from which we choose. We assume that `op_list[j]` acts on site `j`. If the length is shorter than *L*, we repeat it periodically. Strings are translated using *get_op()* of site *i*.

i [int] Index of the site on which the operator acts.

Returns

op [npc.array] One of the entries in *op_list*, not copied.

get_rho_segment (*self*, *segment*)
Return reduced density matrix for a segment.

Note that the dimension of rho_A scales exponentially in the length of the segment.

Parameters

segment [iterable of int] Sites for which the reduced density matrix is to be calculated. Assumed to be sorted.

Returns

rho [*Array*] Reduced density matrix of the segment sites. Labels `'p0', 'p1', ..., 'pk', 'p0*', 'p1*', ..., 'pk*'` with `k=len(segment)`.

get_theta (*self*, *i*, *n=2*, *cutoff=1e-16*, *formL=1.0*, *formR=1.0*)
Calculates the *n*-site wavefunction on `sites[i:i+n]`.

Parameters

i [int] Site index.

n [int] Number of sites. The result lives on `sites[i:i+n]`.

cutoff [float] During DMRG with a mixer, *S* may be a matrix for which we need the inverse. This is calculated as the Penrose pseudo-inverse, which uses a cutoff for the singular values.

formL [float] Exponent for the singular values to the left.

formR [float] Exponent for the singular values to the right.

Returns

theta [*Array*] The n-site wave function with leg labels `vL, p0, p1, .... p{n-1}, vR`. In Vidal's notation (with s=lambda, G=Gamma): `theta = s**form_L G_i s G_{i+1} s ... G_{i+n-1} s**form_R`.

get_total_charge (*self*, *only_physical_legs=False*)
Calculate and return the *qtotal* of the whole MPS (when contracted).

Parameters

only_physical_legs [bool] For `'finite'` boundary conditions, the total charge can be gauged away by changing the LegCharge of the trivial legs on the left and right of the MPS. This option allows to project out the trivial legs to get the actual "physical" total charge.

Returns

qtotal [charges] The sum of the *qtotal* of the individual *B* tensors.

group_sites (*self*, *n=2*, *grouped_sites=None*)
Modify *self* inplace to group sites.

Group each *n* sites together using the *GroupedSite*. This might allow to do TEBD with a Trotter decomposition, or help the convergence of DMRG (in case of too long range interactions).

Parameters

n [int] Number of sites to be grouped together.

grouped_sites [None | list of *GroupedSite*] The sites grouped together.

See also:

group_split Reverts the grouping.

group_split(*self*, *trunc_par=None*)
Modify *self* inplace to split previously grouped sites.

Parameters

trunc_par [dict] Parameters for truncation, see *truncate()*. Defaults to `{'chi_max': max(self.chi)}`.

Returns

trunc_err [*TruncationError*] The error introduced by the truncation for the splitting.

See also:

group_sites Should have been used before to combine sites.

increase_L(*self*, *new_L=None*)
Modify *self* inplace to enlarge the unit cell.

For an infinite MPS, we have unit cells.

Parameters

new_L [int] New number of sites. Defaults to twice the number of current sites.

property nontrivial_bonds
Slice of the non-trivial bond indices, depending on `self.bc`.

norm_test(*self*)
Check that self is in canonical form.

Returns

norm_error: array, shape (L, 2) For each site the norm error to the left and right. The error `norm_error[i, 0]` is defined as the norm-difference between the following networks:

```
|   --theta[i]---.       --s[i]--.
|       |        |   vs         |
|   --theta*[i]--.       --s[i]--.
```

Similarly, `norm_errror[i, 1]` is the norm-difference of:

```
|   .--theta[i]---         .--s[i+1]--
|   |     |          vs    |
|   .--theta*[i]--         .--s[i+1]--
```

overlap(*self*, *other*, *charge_sector=0*, *ignore_form=False*, ***kwargs*)
Compute overlap `<self|other>`.

Parameters

other [MPS] An MPS with the same physical sites.

charge_sector [None | charges | 0] Selects the charge sector in which the dominant eigenvector of the TransferMatrix is. None stands for *all* sectors, 0 stands for the zero-charge sector. Defaults to 0, i.e., *assumes* the dominant eigenvector is in charge sector 0.

ignore_form [bool] If False (default), take into account the canonical form form at each site. If True, we ignore the canonical form (i.e., whether the MPS is in left, right, mixed or no canonical form) and just contract all the _B as they are. (This can give different results!)

****kwargs :** Further keyword arguments given to TransferMatrix.eigenvectors(); only used for infinite boundary conditions.

Returns

overlap [dtype.type] The contraction <self|other> * self.norm * other.norm (i.e., taking into account the norm of both MPS). For an infinite MPS, <self|other> is the overlap per unit cell, i.e., the largest eigenvalue of the TransferMatrix.

permute_sites(*self*, *perm*, *swap_op='auto'*, *trunc_par={}*, *verbose=0*)

Applies the permutation perm to the state (inplace).

Parameters

perm [ndarray[ndim=1, int]] The applied permutation, such that psi.permute_sites(perm)[i] = psi[perm[i]] (where [i] indicates the *i*-th site).

swap_op [None | 'auto' | *Array*] The operator used to swap the phyiscal legs of a two-site wave function *theta*, see *swap_sites()*.

trunc_par [dict] Parameters for truncation, see *truncate()*. *chi_max* defaults to max(self.chi).

verbose [float] Level of verbosity, print status messages if verbose > 0.

Returns

trunc_err [*TruncationError*] The error of the represented state introduced by the truncation after the swaps.

probability_per_charge(*self*, *bond=0*)

Return probabilites of charge value on the left of a given bond.

For example for particle number conservation, define $N_b = sum_{i<b} n_i$ for a given bond b. This function returns the possible values of *N_b* as rows of *charge_values*, and for each row the probabilty that this combination occurs in the given state.

Parameters

bond [int] The bond to be considered. The returned charges are summed on the left of this bond.

Returns

charge_values [2D array] Columns correspond to the different charges in *self.chinfo*. Rows are the different charge fluctuations at this bond

probabilities [1D array] For each row of *charge_values* the probablity for these values of charge fluctuations.

set_B (*self*, *i*, *B*, *form='B'*)
Set *B* at site *i*.

Parameters

i [int] Index choosing the site.

B [*Array*] The 'matrix' at site *i*. No copy is made! Should have leg labels `'vL', 'p', 'vR'` (not necessarily in that order).

form [`'B' | 'A' | 'C' | 'G' | 'Th' | None` | tuple(float, float)] The (canonical) form of the *B* to set. `None` stands for non-canonical form.

set_SL (*self*, *i*, *S*)
Set singular values on the left of site *i*

set_SR (*self*, *i*, *S*)
Set singular values on the right of site *i*

Module description

This module contains an MPS class representing an density matrix by purification.

Usually, an MPS represents a pure state, i.e. the density matrix is $\rho = |\psi><\psi|$, describing observables as $< O >= Tr(O|\psi><\psi|) =<\psi|O|\psi>$. Clearly, if $|\psi>$ is the ground state of a Hamiltonian, this is the density matrix at *T=0*.

At finite temperatures $T > 0$, we want to describe a non-pure density matrix $\rho = \exp(-H/T)$. This can be accieved by the so-called purification: in addition to the physical space *P*, we introduce a second 'auxiliar' space *Q* and define the density matrix of the physical system as $\rho = Tr_Q(|\phi><\phi|)$, where $|\phi>$ is a pure state in the combined phyisical and auxiliar system.

For $T = \infty$, the density matrix ρ_∞ is the identity matrix. In other words, expectation values are sums over all possible states $< O >= Tr_P(\rho_\infty O) = Tr_P(O)$. Saying that each `:` on top is to be connected with the corresponding `:` on the bottom, the trace is simply a contraction:

```
|          :   :   :   :   :   :
|          |   |   |   |   |   |
|          |-------------------|
|          |         O         |
|          |-------------------|
|          |   |   |   |   |   |
|          :   :   :   :   :   :
```

Clearly, we get the same result, if we insert an identity operator, written as MPO, on the top and bottom:

```
|          :   :   :   :   :   :
|          |   |   |   |   |   |
|          B---B---B---B---B---B
|          |   |   |   |   |   |
|          |-------------------|
|          |         O         |
|          |-------------------|
|          |   |   |   |   |   |
|          B*--B*--B*--B*--B*--B*
|          |   |   |   |   |   |
|          :   :   :   :   :   :
```

We use the following label convention:

```
|          q
|          ^
|          |
|   vL ->- B ->- vR
|          |
|          ^
|          p
```

You can view the *MPO* as an MPS by combining the *p* and *q* leg and defining every physical operator to act trivial on the *q* leg. In expecation values, you would then sum over over the *q* legs, which is exactly what we need. In other words, the choice $B = \delta_{p,q}$ with trivial (length-1) virtual bonds yields infinite temperature expectation values for operators action only on the *p* legs!

Now, you go a step further and also apply imaginary time evolution (acting only on *p* legs) to the initial infinite temperature state. For example, the normalized state $|\psi> \propto \exp(-\beta/2H)|\phi>$ yields expecation values

$$< O >= Tr(\exp(-\beta H)O)/Tr(\exp(-\beta H)) \propto < \phi| \exp(-\beta/2H) O \exp(-\beta/2H)|\phi> .$$

An additional real-time evolution allows to calculate time correlation functions:

$$< A(t)B(0) > \propto < \phi| \exp(-\beta H/2) \exp(+iHt) A \exp(-iHt) B \exp(-\beta H/2)|\phi>$$

See also [Karrasch2013] for additional tricks! On of their crucial observations is, that one can apply arbitrary unitaries on the auxiliar space (i.e. the *q*) without changing the result. This can actually be used to reduce the necessary virtual bond dimensions: From the definition, it is easy to see that if we apply $exp(-iHt)$ to the *p* legs of $|\phi>$, and $\exp(+iHt)$ to the *q* legs, they just cancel out! (They commute with $\exp(-\beta H/2)\ldots$) If the state is modified (e.g. by applying *A* or *B* to calculate correlation functions), this is not true any more. However, we still can find unitaries, which are 'optimal' in the sense of reducing the entanglement of the MPS/MPO to the minimal value. For a discussion of *Disentanglers* (implemented in `purification_tebd`), see [Hauschild2018].

Note: The classes `MPSEnvironment` and `TransferMatrix` should also work for the `PurificationMPS` defined here. For example, you can use `expectation_value()` for the expectation value of operators between different PurificationMPS. However, this makes only sense if the *same* disentangler was applied to the *bra* and *ket* PurificationMPS.

Note: The literature (e.g. section 7.2 of [Schollwoeck2011] or [Karrasch2013]) suggests to use a *singlet* as a maximally entangled state. Here, we use instead the identity $\delta_{p,q}$, since it is easier to generalize for *p* running over more than two indices, and allows a simple use of charge conservation with the above *qconj* convention. Moreover, we don't split the physical and auxiliar space into separate sites, which makes TEBD as costly as $O(d^6\chi^3)$.

Todo: One can also look at the canonical ensembles by defining the conserved quantities differently, see Barthel (2016), arXiv:1607.01696 for details. Idea: usual charges on *p*, trivial charges on *q*; fix total charge to desired value. I think it should suffice to implement another *from_infiniteT*.

7.2.5 tools

- full name: tenpy.tools
- parent module: *tenpy*
- type: module

Module description

A collection of tools: mostly short yet quite useful functions.

Some functions are explicitly imported in other parts of the library, others might just be useful when using the libary. Common to all tools is that they are not just useful for a single algorithm but fairly general.

Submodules

params	Tools to handle paramters for algorithms.
misc	Miscellaneous tools, somewhat random mix yet often helpful.
math	Different math functions needed at some point in the library.
fit	tools to fit to an algebraic decay.
string	Tools for handling strings.
process	Tools to read out total memory usage and get/set the number of threads.
optimization	Optimization options for this library.

params

- full name: tenpy.tools.params
- parent module: *tenpy.tools*
- type: module

Functions

get_parameter(params, key, default, descr[, ...])	Read out a parameter from the dictionary and/or provide default values.
unused_parameters(params[, warn])	Returns a set of the parameters which have not been read out with *get_parameters*.

get_parameter

- full name: tenpy.tools.params.get_parameter
- parent module: *tenpy.tools.params*
- type: function

tenpy.tools.params.**get_parameter**(*params*, *key*, *default*, *descr*, *asarray=False*)

Read out a parameter from the dictionary and/or provide default values.

This function provides a similar functionality as `params.get(key, default)`. *Unlike dict.get* this function writes the default value into the dictionary (i.e. in other words it's more similar to `params.setdefault(key, default)`).

This allows the user to save the modified dictionary as meta-data, which gives a concrete record of the actually used parameters and simplifies reproducing the results and restarting simulations.

Moreover, a special entry with the key `'verbose'` *in* the *params* can trigger this function to also print the used value. A higer *verbose* level implies more output. If *verbose* >= 100, it is printed every time it's used. If *verbose* >= 2., its printed for the first time time its used. and for *verbose* >= 1, non-default values are printed the first time they are used. otherwise only for the first use.

Internally, whether a parameter was used is saved in the set `params['_used_param']`. This is used in *unused_parameters()* to print a warning if the key wasn't used at the end of the algorithm, to detect mis-spelled parameters.

Parameters

params [dict] A dicionary of the parameters as provided by the user. If *key* is not a valid key, `params[key]` is set to *default*.

key [string] The key for the parameter which should be read out from the dictionary.

default : The default value for the parameter.

descr [str] A short description for verbose output, like 'TEBD', 'XXZ_model', 'truncation'.

asarray [bool] If True, convert the result to a numpy array with `np.asarray(...)` before returning.

Returns

value : `params[key]` if the key is in params, otherwise *default*. Converted to a numpy array, if *asarray*.

Examples

In the algorith *Engine* gets a dictionary of parameters. Beside doing other stuff, it calls `tenpy.models.model.NearestNeighborModel.calc_U_bond()` with the dictionary as argument, which looks similar like:

```
>>> def model_calc_U(U_param):
>>>     dt = get_parameter(U_param, 'dt', 0.01, 'TEBD')
>>>     # ... calculate exp(-i * dt* H) ....
```

Then, when you call *time_evolution* without any parameters, it just uses the default value:

```
>>> tenpy.algorithms.tebd.time_evolution(..., dict())  # uses dt=0.01
```

If you provide the special keyword `'verbose'` you can triger this function to print the used parameter values:

```
>>> tenpy.algorithms.tebd.time_evolution(..., dict(verbose=1))
parameter 'dt'=0.01 (default) for TEBD
```

Of course you can also provide the parameter to use a non-default value:

```
>>> tenpy.algorithms.tebd.time_evolution(..., dict(dt=0.1, verbose=1))
parameter 'dt'=0.1 for TEBD
```

unused_parameters

- full name: tenpy.tools.params.unused_parameters
- parent module: *tenpy.tools.params*
- type: function

tenpy.tools.params.**unused_parameters**(*params*, *warn=None*)

Returns a set of the parameters which have not been read out with *get_parameters*.

This function might be useful to check for typos in the parameter keys.

Parameters

params [dict] A dictionary of parameters which was given to (functions using) *get_parameter()*

warn [None | str] If given, print a warning "unused parameter for {warn!s}: {unused_keys!s}".

Returns

unused_keys [set] The set of keys of the params which was not used

Module description

Tools to handle paramters for algorithms.

See the doc-string of *get_parameter()* for details.

misc

- full name: tenpy.tools.misc
- parent module: *tenpy.tools*
- type: module

Functions

add_with_None_0(a, b)	Return `a + b`, treating *None* as zero.
any_nonzero(params, keys[, verbose_msg])	Check for any non-zero or non-equal entries in some parameters.
anynan(a)	check whether any entry of a ndarray *a* is 'NaN'.
argsort(a[, sort])	wrapper around np.argsort to allow sorting ascending/descending and by magnitude.

Continued on next page

Table 151 – continued from previous page

atleast_2d_pad(a[, pad_item])	Transform *a* into a 2D array, filling missing places with *pad_item*.
build_initial_state(size, states, filling[, ...])	
chi_list(chi_max[, dchi, nsweeps, verbose])	
inverse_permutation(perm)	reverse sorting indices.
lexsort(a[, axis])	wrapper around `np.lexsort`: allow for trivial case `a.shape[0] = 0` without sorting
list_to_dict_list(l)	Given a list *l* of objects, construct a lookup table.
pad(a[, w_l, v_l, w_r, v_r, axis])	Pad an array along a given *axis*.
setup_executable(mod, run_defaults[, ...])	Read command line arguments and turn into useable dicts.
to_array(a[, shape])	Convert *a* to an numpy array and tile to matching dimension/shape.
to_iterable(a)	If *a* is a not iterable or a string, return `[a]`, else return `a`.
transpose_list_list(D[, pad])	Returns a list of lists T, such that `T[i][j] = D[j][i]`.
zero_if_close(a[, tol])	set real and/or imaginary part to 0 if their absolute value is smaller than *tol*.

add_with_None_0

- full name: tenpy.tools.misc.add_with_None_0
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**add_with_None_0**(*a*, *b*)
Return `a + b`, treating *None* as zero.

Parameters

a, b : The two things to be added, or `None`.

Returns

sum : `a + b`, except if *a* or *b* is *None*, in which case the other variable is returned.

any_nonzero

- full name: tenpy.tools.misc.any_nonzero
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**any_nonzero**(*params*, *keys*, *verbose_msg=None*)
Check for any non-zero or non-equal entries in some parameters.

Parameters

params [dict] A dictionary of parameters.

keys [list of {key | tuple of keys}] For a single key, check `params[key]` for non-zero entries. For a tuple of keys, all the `params[key]` have to be equal (as numpy arrays).

verbose_msg [None | str] If params['verbose'] >= 1, we print *verbose_msg* before checking, and a short notice with the *key*, if a non-zero entry is found.

Returns

match [bool] False, if all params[key] are zero or *None* and True, if any of the params[key] for single *key* in *keys*,

of if any of the entries for a tuple of *keys*

anynan

- full name: tenpy.tools.misc.anynan
- parent module: *tenpy.tools.misc*
- type: function

`tenpy.tools.misc.`**`anynan`**`(`*a*`)`
check whether any entry of a ndarray *a* is 'NaN'.

argsort

- full name: tenpy.tools.misc.argsort
- parent module: *tenpy.tools.misc*
- type: function

`tenpy.tools.misc.`**`argsort`**`(`*a*, *sort=None*, ***kwargs*`)`
wrapper around np.argsort to allow sorting ascending/descending and by magnitude.

Parameters

a [array_like] the array to sort

sort `['m>', 'm<', '>', '<', None]` Specify how the arguments should be sorted.

sort	order
`'m>', 'LM'`	Largest magnitude first
`'m<', 'SM'`	Smallest magnitude first
`'>', 'LR', 'LA'`	Largest real part first
`'<', 'SR', 'SA'`	Smallest real part first
`'LI'`	Largest imaginary part first
`'Si'`	Smallest imaginary part first
`None`	numpy default: same as '<'

****kwargs :** further keyword arguments given directly to `numpy.argsort()`.

Returns

index_array [ndarray, int] same shape as *a*, such that `a[index_array]` is sorted in the specified way.

atleast_2d_pad

- full name: tenpy.tools.misc.atleast_2d_pad
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**atleast_2d_pad**(*a*, *pad_item=0*)
Transform *a* into a 2D array, filling missing places with *pad_item*.

Given a list of lists, turn it to a 2D array (pad with 0), or turn a 1D list to 2D.

Parameters

a [list of lists] to be converted into ad 2D array.

Returns

a_2D [2D ndarray] a converted into a numpy array.

Examples

```
>>> atleast_2d_pad([3, 4, 0])
array([[3, 4, 0]])
```

```
>>> atleast_2d_pad([[3, 4],[1, 6, 7]])
array([[ 3.,  4.,  0.],
       [ 1.,  6.,  7.]])
```

build_initial_state

- full name: tenpy.tools.misc.build_initial_state
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**build_initial_state**(*size*, *states*, *filling*, *mode='random'*, *seed=None*)

chi_list

- full name: tenpy.tools.misc.chi_list
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**chi_list**(*chi_max*, *dchi=20*, *nsweeps=20*, *verbose=0*)

inverse_permutation

- full name: tenpy.tools.misc.inverse_permutation
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**inverse_permutation**(*perm*)
reverse sorting indices.

Sort functions (as `LegCharge.sort()`) return a (1D) permutation *perm* array, such that `sorted_array = old_array[perm]`. This function inverts the permutation *perm*, such that `old_array = sorted_array[inverse_permutation(perm)]`.

Parameters

perm [1D array_like] The permutation to be reversed. *Assumes* that it is a permutation with unique indices. If it is, `inverse_permutation(inverse_permutation(perm)) == perm`.

Returns

inv_perm [1D array (int)] The inverse permutation of *perm* such that `inv_perm[perm[j]] = j = perm[inv_perm[j]]`.

lexsort

- full name: tenpy.tools.misc.lexsort
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**lexsort**(*a*, *axis=-1*)
wrapper around `np.lexsort`: allow for trivial case `a.shape[0] = 0` without sorting

list_to_dict_list

- full name: tenpy.tools.misc.list_to_dict_list
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**list_to_dict_list**(*l*)
Given a list *l* of objects, construct a lookup table.

This function will handle duplicate entries in *l*.

Parameters

l: iterable of iterabele of immutable A list of objects that can be converted to tuples to be used as keys for a dictionary.

Returns

lookup [dict] A dictionary with (key, value) pairs `(key):[i1,i2,...]` where `i1, i2, ...` are the indices where *key* is found in *l*: i.e. `key == tuple(l[i1]) == tuple(l[i2]) == ...`

pad

- full name: tenpy.tools.misc.pad
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**pad**(*a*, *w_l=0*, *v_l=0*, *w_r=0*, *v_r=0*, *axis=0*)
Pad an array along a given *axis*.

Parameters

a [ndarray] the array to be padded

w_l [int] the width to be padded in the front

v_l [dtype] the value to be inserted before *a*

w_r [int] the width to be padded after the last index

v_l [dtype] the value to be inserted after *a*

axis [int] the axis along which to pad

Returns

padded [ndarray] a copy of *a* with enlarged *axis*, padded with the given values.

setup_executable

- full name: tenpy.tools.misc.setup_executable
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**setup_executable**(*mod*, *run_defaults*, *identifier_list=None*)
Read command line arguments and turn into useable dicts.

Uses default values defined at: - model class for model_par - here for sim_par - executable file for run_par Alternatively, a model_defaults dictionary and identifier_list can be supplied without the model

NB: for setup_executable to work with a model class, the model class needs to define two things:

- defaults, a static (class level) dictionary with (key, value) pairs that have the name of the parameter (as string) as key, and the default value as value.
- identifier, a static (class level) list or other iterable with the names of the parameters to be used in filename identifiers.

Args: mod (model | dict): Model class (or instance) OR a dictionary containing model defaults run_defaults (dict): default values for executable file parameters identifier_list (ieterable, optional) | Used only if mod is a dict. Contains the identifier

variables

Returns: model_par, sim_par, run_par (dicts) : containing all parameters. args | namespace with raw arguments for some backwards compatibility with executables.

to_array

- full name: tenpy.tools.misc.to_array
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**to_array**(*a*, *shape=(None,)*)
Convert *a* to an numpy array and tile to matching dimension/shape.

This function provides similar functionality as numpys broadcast, but not quite the same: Only scalars are broadcasted to higher dimensions, for a non-scalar, we require the number of dimension to match. If the shape does not match, we repeat periodically, e.g. we tile `(3, 4) -> (6, 16)`, but `(4, 4) -> (6, 16)` will raise an error.

Parameters

a [scalar | array_like] The input to be converted to an array. A scalar is reshaped to the desired dimension.

shape [tuple of {None | int}] The desired shape of the array. An entry `None` indicates arbitrary len >=1. For int entries, tile the array periodically to fit the len.

Returns

a_array [ndarray] A copy of *a* converted to a numpy ndarray of desired dimension and shape.

to_iterable

- full name: tenpy.tools.misc.to_iterable
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**to_iterable**(*a*)
If *a* is a not iterable or a string, return `[a]`, else return `a`.

transpose_list_list

- full name: tenpy.tools.misc.transpose_list_list
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**transpose_list_list**(*D*, *pad=None*)
Returns a list of lists T, such that `T[i][j] = D[j][i]`.

Parameters

D [list of list] to be transposed

pad : Used to fill missing places, if D is not rectangular.

Returns

T [list of lists] transposed, rectangular version of *D*. constructed such that `T[i][j] = D[j][i] if i < len(D[j]) else pad`

zero_if_close

- full name: tenpy.tools.misc.zero_if_close
- parent module: *tenpy.tools.misc*
- type: function

tenpy.tools.misc.**zero_if_close**(*a*, *tol=1e-15*)
set real and/or imaginary part to 0 if their absolute value is smaller than *tol*.

Parameters

a [ndarray] numpy array to be rounded

tol [float] the threashold which values to consider as '0'.

Module description

Miscellaneous tools, somewhat random mix yet often helpful.

math

- full name: tenpy.tools.math
- parent module: *tenpy.tools*
- type: module

Functions

entropy(p[, n])	Calculate the entropy of a distribution.
gcd(a, b)	Computes the greatest common divisor (GCD) of two numbers.
gcd_array(a)	Return the greatest common divisor of all of entries in *a*
lcm(a, b)	Returns the least common multiple (LCM) of two positive numbers.
matvec_to_array(H)	transform an linear operator with a *matvec* method into a dense numpy array.
perm_sign(p)	Given a permutation *p* of numbers, returns its sign.
qr_li(A[, cutoff])	QR decomposition with cutoff to discard nearly linear dependent columns in *Q*.
rq_li(A[, cutoff])	RQ decomposition with cutoff to discard nearly linear dependent columns in *Q*.
speigs(A, k, *args, **kwargs)	Wrapper around `scipy.sparse.linalg.eigs()`, lifting the restriction `k < rank(A)-1`.
speigsh(A, k, *args, **kwargs)	Wrapper around `scipy.sparse.linalg.eigsh()`, lifting the restriction `k < rank(A)-1`.

entropy

- full name: tenpy.tools.math.entropy
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**entropy**(*p*, *n=1*)
Calculate the entropy of a distribution.

Assumes that p is a normalized distribution (`np.sum(p)==1.`).

Parameters

p [1D array] A normalized distribution.

n [1 | float | np.inf] Selects the entropy, see below.

Returns

entropy [float] Shannon-entropy $-\sum_i p_i \log(p_i)$ (n=1) or Renyi-entropy $\frac{1}{1-n}\log(\sum_i p_i^n)$ (n != 1) of the distribution *p*.

gcd

- full name: tenpy.tools.math.gcd
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**gcd**(*a*, *b*)
Computes the greatest common divisor (GCD) of two numbers.

Return 0 if both a, b are zero, otherwise always return a non-negative number.

gcd_array

- full name: tenpy.tools.math.gcd_array
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**gcd_array**(*a*)
Return the greatest common divisor of all of entries in *a*

lcm

- full name: tenpy.tools.math.lcm
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**lcm**(*a*, *b*)
Returns the least common multiple (LCM) of two positive numbers.

matvec_to_array

- full name: tenpy.tools.math.matvec_to_array
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**matvec_to_array**(*H*)
transform an linear operator with a *matvec* method into a dense numpy array.

Parameters

H [linear operator] should have *shape*, *dtype* attributes and a *matvec* method.

Returns

H_dense [ndarray, shape `(H.dim, H.dim)`] a dense array version of *H*.

perm_sign

- full name: tenpy.tools.math.perm_sign
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**perm_sign**(*p*)
Given a permutation *p* of numbers, returns its sign. (+1 or -1)

Assumes that all the elements are distinct, if not, you get crap.

Examples

```
>>> for p in itertools.permutations(range(3))]:
...     print('{p!s}: {sign!s}'.format(p=p, sign=perm_sign(p)))
(0, 1, 2): 1
(0, 2, 1): -1
(1, 0, 2): -1
(1, 2, 0): 1
(2, 0, 1): 1
(2, 1, 0): -1
```

qr_li

- full name: tenpy.tools.math.qr_li
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**qr_li**(*A*, *cutoff=1e-15*)
QR decomposition with cutoff to discard nearly linear dependent columns in *Q*.

Perform a QR decomposition with pivoting, discard columns where `R[i,i] < cuttoff`, reverse the permutation from pivoting and perform another QR decomposition to ensure that *R* is upper right.

Parameters

A [numpy.ndarray] Matrix to be decomposed as `A = Q.R`

Returns

Q, R [numpy.ndarray] Decomposition of *A* into isometry *Q^d Q = 1* and upper right *R* with diagonal entries larger than *cutoff*.

rq_li

- full name: tenpy.tools.math.rq_li
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**rq_li**(*A*, *cutoff=1e-15*)

RQ decomposition with cutoff to discard nearly linear dependent columns in *Q*.

Uses *qr_li()* on tranpose of *A*. Note that *R* is nonzero in the lowest left corner; *R* has entries below the diagonal for non-square *R*.

Parameters

A [numpy.ndarray] Matrix to be decomposed as `A = Q.R`

Returns

R, Q [numpy.ndarray] Decomposition of *A* into isometry *Q Q^d = 1* and upper right *R* with diagonal entries larger than *cutoff*. If `M, N = A.shape`, then `R.shape = M, K` and `Q.shape = K, N` with `K <= min(M, N)`.

speigs

- full name: tenpy.tools.math.speigs
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**speigs**(*A*, *k*, **args*, ***kwargs*)

Wrapper around `scipy.sparse.linalg.eigs()`, lifting the restriction `k < rank(A)-1`.

Parameters

A [MxM ndarray or like `scipy.sparse.linalg.LinearOperator`] the (square) linear operator for which the eigenvalues should be computed.

k [int] the number of eigenvalues to be computed.

***args, **kwargs :** further arguments are directly given to `scipy.sparse.linalg.eigs()`

Returns

w [ndarray] array of min(*k*, A.shape[0]) eigenvalues

v [ndarray] array of min(*k*, A.shape[0]) eigenvectors, `v[:, i]` is the *i*-th eigenvector. Only returned if `kwargs['return_eigenvectors'] == True`.

speigsh

- full name: tenpy.tools.math.speigsh
- parent module: *tenpy.tools.math*
- type: function

tenpy.tools.math.**speigsh**(*A*, *k*, **args*, ***kwargs*)
Wrapper around `scipy.sparse.linalg.eigsh()`, lifting the restriction `k < rank(A)-1`.

Parameters

A [MxM ndarray or like `scipy.sparse.linalg.LinearOperator`] The (square) hermitian linear operator for which the eigenvalues should be computed.

k [int] The number of eigenvalues to be computed.

***args, **kwargs :** Further arguments are directly given to `scipy.sparse.linalg.eigs()`.

Returns

w [ndarray] Array of min(*k*, A.shape[0]) eigenvalues.

v [ndarray] Array of min(*k*, A.shape[0]) eigenvectors, `v[:, i]` is the *i*-th eigenvector. Only returned if `kwargs['return_eigenvectors'] == True`.

Module description

Different math functions needed at some point in the library.

fit

- full name: tenpy.tools.fit
- parent module: *tenpy.tools*
- type: module

Functions

alg_decay(x, a, b, c)	define the algebraic decay.
alg_decay_fit(x, y[, npts, power_range, ...])	Fit y to the form `a*x**(-b) + c`.
alg_decay_fit_res(log_b, x, y)	Returns the residue of an algebraic decay fit of the form `x**(-np.exp(log_b))`.
alg_decay_fits(x, ys[, npts, power_range, ...])	Fit arrays of y's to the form a * x**(-b) + c.
lin_fit_res(x, y)	Returns the least-square residue of a linear fit y vs x.
linear_fit(x, y)	Perform a linear fit of y to ax + b.
plot_alg_decay_fit(plot_module, x, y, fit_par)	Given x, y, and fit_par (output from alg_decay_fit), produces a plot of the algebraic decay fit.

alg_decay

- full name: tenpy.tools.fit.alg_decay
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**alg_decay**(*x*, *a*, *b*, *c*)
define the algebraic decay.

alg_decay_fit

- full name: tenpy.tools.fit.alg_decay_fit
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**alg_decay_fit**(*x, y, npts=5, power_range=(0.01, 4.0), power_mesh=[60, 10]*)
Fit y to the form `a*x**(-b) + c`.

Returns a triplet [a, b, c].

npts specifies the maximum number of points to fit. If npts < len(x), then alg_decay_fit() will only fit to the last npts points. power_range is a tuple that gives that restricts the possible ranges for b. power_mesh is a list of numbers, which specifies how fine to search for the optimal b. E.g., if power_mesh = [60,10], then it'll first divide the power_range into 60 intervals, and then divide those intervals by 10.

alg_decay_fit_res

- full name: tenpy.tools.fit.alg_decay_fit_res
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**alg_decay_fit_res**(*log_b*, *x*, *y*)
Returns the residue of an algebraic decay fit of the form `x**(-np.exp(log_b))`.

alg_decay_fits

- full name: tenpy.tools.fit.alg_decay_fits
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**alg_decay_fits**(*x, ys, npts=5, power_range=(0.01, 4.0), power_mesh=[60, 10]*)
Fit arrays of y's to the form a * x**(-b) + c.

Returns arrays of [a, b, c].

lin_fit_res

- full name: tenpy.tools.fit.lin_fit_res
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**lin_fit_res**(*x*, *y*)
Returns the least-square residue of a linear fit y vs x.

linear_fit

- full name: tenpy.tools.fit.linear_fit
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**linear_fit**(*x*, *y*)
Perform a linear fit of y to ax + b.

Returns a, b, res.

plot_alg_decay_fit

- full name: tenpy.tools.fit.plot_alg_decay_fit
- parent module: *tenpy.tools.fit*
- type: function

tenpy.tools.fit.**plot_alg_decay_fit**(*plot_module*, *x*, *y*, *fit_par*, *xfunc=None*, *kwargs={}*, *plot_fit_args={}*)
Given x, y, and fit_par (output from alg_decay_fit), produces a plot of the algebraic decay fit.

plot_module is matplotlib.pyplot, or a subplot. x, y are the data (real, 1-dimensional np.ndarray) fit_par is a triplet of numbers [a, b, c] that describes and algebraic decay (see alg_decay()). xfunc is an optional parameter that scales the x-axis in the resulting plot. kwargs is a dictionary, whoses key/items are passed to the plot function. plot_fit_args is a dictionary that controls how the fit is shown.

Module description

tools to fit to an algebraic decay.

string

- full name: tenpy.tools.string
- parent module: *tenpy.tools*
- type: module

Functions

is_non_string_iterable(x)	Check if x is a non-string iterable, (e.g., list, tuple, dictionary, np.ndarray)
to_mathematica_lists(a)	convert nested *a* to string readable by mathematica using curly brackets '{...}'.
vert_join(strlist[, valign, halign, delim])	Join strings with multilines vertically such that they appear next to each other.

is_non_string_iterable

- full name: tenpy.tools.string.is_non_string_iterable
- parent module: *tenpy.tools.string*
- type: function

tenpy.tools.string.**is_non_string_iterable**(*x*)
Check if x is a non-string iterable, (e.g., list, tuple, dictionary, np.ndarray)

to_mathematica_lists

- full name: tenpy.tools.string.to_mathematica_lists
- parent module: *tenpy.tools.string*
- type: function

tenpy.tools.string.**to_mathematica_lists**(*a*)
convert nested *a* to string readable by mathematica using curly brackets '{...}'.

vert_join

- full name: tenpy.tools.string.vert_join
- parent module: *tenpy.tools.string*
- type: function

tenpy.tools.string.**vert_join**(*strlist*, *valign='t'*, *halign='l'*, *delim=' '*)
Join strings with multilines vertically such that they appear next to each other.

Parameters

strlist [list of str] the strings to be joined vertically

valing `['t', 'c', 'b']` vertical alignment of the strings: top, center, or bottom

halign `['l', 'c', 'r']` horizontal alignment of the strings: left, center, or right

delim [str] field separator between the strings

Returns

joined [str] a string where the strings of strlist are aligned vertically

Examples

```
>>> print vert_join(['a\nsample\nmultiline\nstring', str(np.arange(9).reshape(3,␣
↪3))],
...                 delim=' | ')
a         | [[0 1 2]
sample    |  [3 4 5]
multiline |  [6 7 8]]
string
```

Module description

Tools for handling strings.

process

- full name: tenpy.tools.process
- parent module: *tenpy.tools*
- type: module

Functions

load_omp_library([libs, verbose])	Tries to load openMP library.
memory_usage()	Return memory usage of the running python process.
mkl_get_nthreads()	wrapper around MKL `get_max_threads`.
mkl_set_nthreads(n)	wrapper around MKL `set_num_threads`.
omp_get_nthreads()	wrapper around OpenMP `get_max_threads`.
omp_set_nthreads(n)	wrapper around OpenMP `set_nthreads`.

load_omp_library

- full name: tenpy.tools.process.load_omp_library
- parent module: *tenpy.tools.process*
- type: function

`tenpy.tools.process.`**`load_omp_library`**(*libs=['libiomp5.so', None, 'libgomp.so.1'], verbose=True*)

Tries to load openMP library.

Parameters

libs : list of possible library names we should try to load (with ctypes.CDLL).

verbose [bool] wheter to print the name of the loaded library.

Returns

omp [CDLL | None] OpenMP shared libary if found, otherwise None. Once it was sucessfully imported, no re-imports are tried.

memory_usage

- full name: tenpy.tools.process.memory_usage
- parent module: *tenpy.tools.process*
- type: function

tenpy.tools.process.**memory_usage**()
Return memory usage of the running python process.

You can `pip install psutil` if you get only `-1.`.

Returns

mem [float] Currently used memory in megabytes. `-1.` if no way to read out.

mkl_get_nthreads

- full name: tenpy.tools.process.mkl_get_nthreads
- parent module: *tenpy.tools.process*
- type: function

tenpy.tools.process.**mkl_get_nthreads**()
wrapper around MKL `get_max_threads`.

Returns

max_threads [int] The maximum number of threads used by MKL. `-1` if unable to read out.

mkl_set_nthreads

- full name: tenpy.tools.process.mkl_set_nthreads
- parent module: *tenpy.tools.process*
- type: function

tenpy.tools.process.**mkl_set_nthreads**(*n*)
wrapper around MKL `set_num_threads`.

Parameters

n [int] the number of threads to use

Returns

success [bool] whether the shared library was found and set.

omp_get_nthreads

- full name: tenpy.tools.process.omp_get_nthreads
- parent module: *tenpy.tools.process*
- type: function

tenpy.tools.process.**omp_get_nthreads**()
wrapper around OpenMP `get_max_threads`.

Returns

max_threads [int] The maximum number of threads used by OpenMP (and thus MKL). `-1` if unable to read out.

omp_set_nthreads

- full name: tenpy.tools.process.omp_set_nthreads
- parent module: *tenpy.tools.process*
- type: function

tenpy.tools.process.**omp_set_nthreads**(*n*)
wrapper around OpenMP `set_nthreads`.

Parameters

n [int] the number of threads to use

Returns

success [bool] whether the shared library was found and set.

Module description

Tools to read out total memory usage and get/set the number of threads.

If your python is compiled against MKL (e.g. if you use *anaconda* as recommended in INSTALL), it will by default use as many threads as CPU cores are available. If you run a job on a cluster, you should limit this to the number of cores you reserved – otherwise your colleagues might get angry... A simple way to achieve this is to set a suitable enviornment variable before calling your python program, e.g. on the linux bash `export OMP_NUM_THREADS=4` for 4 threads. (MKL used OpenMP and thus respects its settings.)

Alternatively, this module provides *omp_get_nthreads()* and *omp_set_nthreads()*, which give their best to get and set the number of threads at runtime, while still being failsave if the shared OpenMP library is not found. In the latter case, you might also try the equivalent *mkl_get_nthreads()* and *mkl_set_nthreads()*.

optimization

- full name: tenpy.tools.optimization
- parent module: *tenpy.tools*
- type: module

Classes

OptimizationFlag	Options for the global 'optimization level' used for dynamical optimizations.
temporary_level(temporary_level)	Context manager to temporarily set the optimization level to a different value.

OptimizationFlag

- full name: tenpy.tools.optimization.OptimizationFlag
- parent module: *tenpy.tools.optimization*
- type: class

class tenpy.tools.optimization.**OptimizationFlag**

Bases: enum.IntEnum

Options for the global 'optimization level' used for dynamical optimizations.

Whether we optimize dynamically is decided by comparison of the global "optimization level" with one of the following flags. A higher level *includes* all the previous optimizations.

Level	Flag	Description
0	none	Don't do any optimizations, i.e., run many sanity checks. Used for testing.
1	default	Skip really unnecessary sanity checks, but also don't try any optional optimizations if they might give an overhead.
2	safe	Activate safe optimizations in algorithms, even if they might give a small overhead. Example: Try to compress the MPO representing the hamiltonian.
3	skip_arg_checks	Unsafe! Skip (some) class sanity tests and (function) argument checks.

Warning: When unsafe optimizations are enabled, errors will not be detected that easily, debugging is much harder, and you might even get segmentation faults in the compiled parts. Use this kind of optimization only for code which you succesfully ran before with (very) similar parmeters and disabled optimiztions! Enable this optimization only during the parts of the code where it is really necessary. Check whether it actually helps - if it doesn't, keep the optimization disabled!

temporary_level

- full name: tenpy.tools.optimization.temporary_level
- parent module: *tenpy.tools.optimization*
- type: class

class tenpy.tools.optimization.**temporary_level**(*temporary_level*)
Bases: object

Context manager to temporarily set the optimization level to a different value.

Parameters

temporary_level [int | OptimizationFlag | str | None] The optimization level to be set during the context. *None* defaults to the current value of the optimization level.

Examples

It is recommended to use this context manager in a `with` statement:

```
# optimization level default
with temporary_level(OptimizationFlag.safe):
    do_some_stuff()  # temporarily have Optimization level `safe`
    # you can even change the optimization level to something else:
    set_level(OptimizationFlag.skip_args_check)
    do_some_really_heavy_stuff()
# here we are back to the optimization level as before the ``with ...`` statement
```

Attributes

temporary_level [None | OptimizationFlag] The optimization level to be set during the context.

_old_level [OptimizationFlag] Optimization level to be restored at the end of the context manager.

Functions

get_level()	Return the global optimization level.
optimize([level_compare])	Called by algorithms to check whether it should (try to) do some optimizations.
set_level([level])	Set the global optimization level.
to_OptimizationFlag(level)	Convert strings and int to a valid OptimizationFlag.
use_cython([func, replacement, check_doc])	Decorator to replace a function with a Cython-equivalent from _npc_helper.pyx.

get_level

- full name: tenpy.tools.optimization.get_level
- parent module: *tenpy.tools.optimization*
- type: function

tenpy.tools.optimization.**get_level**()
Return the global optimization level.

optimize

- full name: tenpy.tools.optimization.optimize
- parent module: *tenpy.tools.optimization*
- type: function

tenpy.tools.optimization.**optimize**(*level_compare=<OptimizationFlag.default: 1>*)
Called by algorithms to check whether it should (try to) do some optimizations.

Parameters

level_compare [OptimizationFlag] At which level to start optimization, i.e., how safe the suggested optimization is.

Returns

optimize [bool] True if the algorithms should try to optimize, i.e., whether the global "optimization level" is equal or higher than the level to compare to.

set_level

- full name: tenpy.tools.optimization.set_level
- parent module: *tenpy.tools.optimization*
- type: function

tenpy.tools.optimization.**set_level**(*level=1*)
Set the global optimization level.

Parameters

level [int | OptimizationFlag | str | None] The new global optimization level to be set. `None` defaults to keeping the current level.

to_OptimizationFlag

- full name: tenpy.tools.optimization.to_OptimizationFlag
- parent module: *tenpy.tools.optimization*
- type: function

tenpy.tools.optimization.**to_OptimizationFlag**(*level*)
Convert strings and int to a valid OptimizationFlag.

`None` defaults to the current level.

use_cython

- full name: tenpy.tools.optimization.use_cython
- parent module: *tenpy.tools.optimization*
- type: function

tenpy.tools.optimization.**use_cython**(*func=None*, *replacement=None*, *check_doc=True*)
Decorator to replace a function with a Cython-equivalent from _npc_helper.pyx.

This is a decorator, which is supposed to be used in front of function definitions with an `@` sign, for example:

```
@use_cython
def my_slow_function(a):
    "some example function with slow python loops"
    result = 0.
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            #... heavy calculations ...
            result += np.cos(a[i, j]**2) * (i + j)
    return result
```

This decorator indicates that there is a Cython implementation in the file `tenpy/linalg/_npc_helper.pyx`, which should have the same signature (i.e. same arguments and return values) as the decorated function, and can be used as a replacement for the decorated function. However, if the cython code could not be compiled on your system (or if the environment variable `TENPY_OPTIMIZE` is set to negative values), we just pass the previous function.

Note: in case that the decorator is used for a class method, the corresponding Cython version needs to have an `@cython.binding(True)`.

Parameters

func [function] The defined function

replacement [string | None] The name of the function defined in `tenpy/linalg/_npc_helper.pyx` which should replace the decorated function. `None` defaults to the name of the decorated function, e.g., in the above example *my_slow_function*.

check_doc [bool] If True, we check that the cython version of the function has the exact same doc string (up to a possible first line containing the function signature) to exclude typos and inconsistent versions.

Returns

replacement_func [function] The function replacing the decorated function *func*. If the cython code can not be loaded, this is just *func*, otherwise it's the cython version specified by *replacement*.

Module description

Optimization options for this library.

Let me start with a quote of "Micheal Jackson" (a programmer, not the musician):

```
First rule of optimization: "Don't do it."
Second rule of optimization (for experts only): "Don't do it yet."
Third rule of optimization: "Profile before optimizing."
```

Luckily, following the third optimization rule, namely profiling code, is fairly simple in python, see the documentation. If you have a python skript running your code, you can simply call it with `python -m "cProfile" -s "tottime" your_skript.py`. Alternatively, save the profiling statistics with `python -m "cProfile" -o "profile_data.stat" your_skript.py` and run these few lines of python code:

```
import pstats
p = pstats.Pstats("profile_data.stat")
p.sort_stats('cumtime')  # sort by 'cumtime' column
p.print_stats(30)   # prints first 30 entries
```

That being said, I actually did profile and optimize (parts of) the library; and there are a few knobs you can turn to tweak the most out of this library, explained in the following.

1) Simply install the 'bottleneck' python package, which allows to optimize slow parts of numpy, most notably 'NaN' checking.

2) Figure out which numpy/scipy/python you are using. As explained in *Installation instructions*, we recommend to use the Python distributed provided by Intel or Anaconda. They ship with numpy and scipy which use Intels MKL library, such that e.g. `np.tensordot` is parallelized to use multiple cores.

3) In case you didn't do that yet: some parts of the library are written in both python and Cython with the same interface, so you can simply compile the Cython code, as explained in *Installation instructions*. Then everything should work the same way from a user perspective, while internally the faster, pre-compiled cython code from `tenpy/linalg/_npc_helper.pyx` is used. This should also be a safe thing to do. The replacement of the optimized functions is done by the decorator *use_cython()*.

4) One of the great things about python is its dynamical nature - anything can be done at runtime. In that spirit, this module allows to set a global "optimization level" which can be changed *dynamically* (i.e., during runtime) with *set_level()*. The library will then try some extra optimiztion, most notably skip sanity checks of arguments. The possible choices for this global level are given by the *OptimizationFlag*. The default initial value for the global optimization level can be adjusted by the environment variable *TENPY_OPTIMIZE*.

   Warning: When this optimizing is enabled, we skip (some) sanity checks. Thus, errors will not be detected that easily, and debugging is much harder! We recommend to use this kind of optimization only for code which you succesfully have run before with (very) similar parmeters! Enable this optimization only during the parts of the code where it is really necessary. The context manager *temporary_level* can help with that. Check whether it actually helps - if it doesn't, keep the optimization disabled! Some parts of the library already do that as well (e.g. DMRG after the first sweep).

5) You might want to try some different compile time options for the cython code, set in the *setup.py* in the top directory of the repository. Since the *setup.py* reads out the *TENPY_OPTIMIZE* environment variable, you can simple use an `export TENPY_OPTIMIZE=3` (in your bash/terminal) right before compilation. An `export TENPY_OPTIMIZE=0` activates profiling hooks instead.

Warning: This increases the probability of getting segmentation faults and anyway might not help that much; in the crucial parts of the cython code, these optimizations are already applied. We do *not* recommend using this!

7.2.6 version

- full name: tenpy.version
- parent module: *tenpy*
- type: module

Module description

Access to version of this library.

The version is provided in the standard python format `major.minor.revision` as string. Use `pkg_resources.parse_version` before comparing versions.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[TeNPyNotes] "Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy)" J. Hauschild, F. Pollmann, SciPost Phys. Lect. Notes 5 (2018), arXiv:1805.00055, doi:10.21468/SciPostPhysLectNotes.5

[TeNPySource] Source code available for download from GitHub, https://github.com/tenpy/tenpy

[TeNPyDoc] Online documentation, https://tenpy.github.io/

[TeNPyForum] Community forum for discussions, FAQ and announcements, https://tenpy.johannes-hauschild.de

[Schollwoeck2011] "The density-matrix renormalization group in the age of matrix product states" U. Schollwoeck, Annals of Physics 326, 96 (2011), arXiv:1008.3477 doi:10.1016/j.aop.2010.09.012

Extensive review, classic introduction.

[Verstraete2009] "Matrix Product States, Projected Entangled Pair States, and variational renormalization group methods for quantum spin systems" F. Verstraete and V. Murg and J.I. Cirac, Advances in Physics 57 2, 143-224 (2009) arXiv:0907.2796 doi:10.1080/14789940801912366

[Cirac2009] "Renormalization and tensor product states in spin chains and lattices" J. I. Cirac and F. Verstraete, Journal of Physics A: Mathematical and Theoretical, 42, 50 (2009) arXiv:0910.1130 doi:10.1088/1751-8113/42/50/504004

[CincioVidal2013] "Characterizing Topological Order by Studying the Ground States on an Infinite Cylinder" L. Cincio, G. Vidal, Phys. Rev. Lett. 110, 067208 (2013), arXiv:1208.2623 doi:10.1103/PhysRevLett.110.067208

[Eisert2013] "Entanglement and tensor network states" J. Eisert, Modeling and Simulation 3, 520 (2013) arXiv:1308.3318

[Orus2014] "A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States" R. Orus, Annals of Physics 349, 117-158 (2014) arXiv:1306.2164 doi:10.1016/j.aop.2014.06.013

[Hubig2019] "Time-evolution methods for matrix-product states" S. Paeckel, T. Köhler, A. Swoboda, S. R. Manmana, U. Schollwöck, C. Hubig, arXiv:1901.05824

[White1992] "Density matrix formulation for quantum renormalization groups" S. White, Phys. Rev. Lett. 69, 2863 (1992) doi:10.1103/PhysRevLett.69.2863, S. White, Phys. Rev. B 84, 10345 (1992) doi:10.1103/PhysRevB.48.10345

[Vidal2004] "Efficient Simulation of One-Dimensional Quantum Many-Body Systems" G. Vidal, Phys. Rev. Lett. 93, 040502 (2004), arXiv:quant-ph/0310089 doi:10.1103/PhysRevLett.93.040502

[Schollwoeck2005] "The density-matrix renormalization group" U. Schollwöck, Rev. Mod. Phys. 77, 259 (2005), arXiv:0409292 doi:10.1103/RevModPhys.77.259

[White2005] "Density matrix renormalization group algorithms with a single center site" S. White, Phys. Rev. B 72, 180403(R) (2005), arXiv:cond-mat/0508709 doi:10.1103/PhysRevB.72.180403

[McCulloch2008] "Infinite size density matrix renormalization group, revisited" I. P. McCulloch, arXiv:0804.2509

[Singh2009] "Tensor network decompositions in the presence of a global symmetry" S. Singh, R. Pfeifer, G. Vidal, Phys. Rev. A 82, 050301(R), arXiv:0907.2994 doi:10.1103/PhysRevA.82.050301

[Singh2010] "Tensor network states and algorithms in the presence of a global U(1) symmetry" S. Singh, R. Pfeifer, G. Vidal, Phys. Rev. B 83, 115125, arXiv:1008.4774 doi:10.1103/PhysRevB.83.115125

[Haegeman2011] "Time-Dependent Variational Principle for Quantum Lattices" J. Haegeman, J. I. Cirac, T. J. Osborne, I. Pizorn, H. Verschelde, F. Verstraete, Phys. Rev. Lett. 107, 070601 (2011), arXiv:1103.0936 doi:10.1103/PhysRevLett.107.070601

[Karrasch2013] "Reducing the numerical effort of finite-temperature density matrix renormalization group calculations" C. Karrasch, J. H. Bardarson, J. E. Moore, New J. Phys. 15, 083031 (2013), arXiv:1303.3942 doi:10.1088/1367-2630/15/8/083031

[Hubig2015] "Strictly single-site DMRG algorithm with subspace expansion" C. Hubig, I. P. McCulloch, U. Schollwoeck, F. A. Wolf, Phys. Rev. B 91, 155115 (2015), arXiv:1501.05504 doi:10.1103/PhysRevB.91.155115

[Haegeman2016] "Unifying time evolution and optimization with matrix product states" J. Haegeman, C. Lubich, I. Oseledets, B. Vandereycken, F. Verstraete, Phys. Rev. B 94, 165116 (2016), arXiv:1408.5056 doi:10.1103/PhysRevB.94.165116

[Hauschild2018] "Finding purifications with minimal entanglement" J. Hauschild, E. Leviatan, J. H. Bardarson, E. Altman, M. P. Zaletel, F. Pollmann, Phys. Rev. B 98, 235163 (2018), arXiv:1711.01288 doi:10.1103/PhysRevB.98.235163

[Resta1997] "Quantum-Mechanical Position Operator in Extended Systems" R. Resta, Phys. Rev. Lett. 80, 1800 (1997) doi:10.1103/PhysRevLett.80.1800

[Schuch2013] "Condensed Matter Applications of Entanglement Theory" N. Schuch, Quantum Information Processing. Lecture Notes of the 44th IFF Spring School (2013) arXiv:1306.5551

[PollmannTurner2012] "Detection of symmetry-protected topological phases in one dimension" F. Pollmann, A. Turner, Phys. Rev. B 86, 125441 (2012), arXiv:1204.0704 doi:10.1103/PhysRevB.86.125441

[Stoudenmire2011] "Studying Two Dimensional Systems With the Density Matrix Renormalization Group" E.M. Stoudenmire, Steven R. White, Ann. Rev. of Cond. Mat. Physics, 3: 111-128 (2012), arXiv:1105.1374 doi:10.1146/annurev-conmatphys-020911-125018

[Neupert2011] "Fractional quantum Hall states at zero magnetic field" Titus Neupert, Luiz Santos, Claudio Chamon, and Christopher Mudry, Phys. Rev. Lett. 106, 236804 (2011), arXiv:1012.4723 doi:10.1103/PhysRevLett.106.236804

[Yang2012] "Topological flat band models with arbitrary Chern numbers" Shuo Yang, Zheng-Cheng Gu, Kai Sun, and S. Das Sarma, Phys. Rev. B 86, 241112(R) (2012), arXiv:1205.5792, doi:10.1103/PhysRevB.86.241112

[Grushin2015] "Characterization and stability of a fermionic =1/3 fractional Chern insulator" Adolfo G. Grushin, Johannes Motruk, Michael P. Zaletel, and Frank Pollmann, Phys. Rev. B 91, 035136 (2015), arXiv:1407.6985 doi:10.1103/PhysRevB.91.035136

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Z