pyblock3

Release 1.0

The pyblock3 developers

Feb 27, 2024

CONTENTS:

CHAPTER

ONE

PYBLOCK3 USAGE

1.1 Symmetry Labels

SZ represents a collection of three quantum numbers (particle number, projected spin, point group irreducible representation).

The group algebra for *SZ* is also defined:

```
>>> from pyblock3.algebra.symmetry import SZ
>>> a = SZ(0, 0, 0)
>>> b = SZ(1, 1, 2)
>>> a + b
< N=1 SZ=1/2 PG=2 >
>>> b + b
< N=2 SZ=1 PG=0 >
>>> -b
< N=-1 SZ=-1/2 PG=2 >
```

BondInfo represents a map from *SZ* to number of states. The union (*__or__*), intersection (*__and__*), addition (*__add__*) and tensor product (*__xor__*) of two *BondInfo* are also defined:

```
>>> from pyblock3.algebra.symmetry import SZ, BondInfo
>>> bi = BondInfo({SZ(0, 0, 0): 1, SZ(1, 1, 2): 2})
>>> ci = BondInfo({SZ(1, 1, 2): 2, SZ(-1, -1, 2): 2})
>>> bi | ci
< N=-1 SZ=-1/2 PG=2 > = 2 < N=0 SZ=0 PG=0 > = 1 < N=1 SZ=1/2 PG=2 > = 2
>>> bi & ci
< N=1 SZ=1/2 PG=2 > = 2
>>> bi + ci
< N=-1 SZ=-1/2 PG=2 > = 2 < N=0 SZ=0 PG=0 > = 1 < N=1 SZ=1/2 PG=2 > = 4
>>> bi ^ ci
< N=-1 SZ=-1/2 PG=2 > = 2 < N=0 SZ=0 PG=0 > = 4 < N=1 SZ=1/2 PG=2 > = 2 < N=2 SZ=1 PG=0 >
↪ = 4
```

1.2 Block-Sparse Tensor

A `SubTensor` is a `numpy.ndarray` with a `tuple` of `SZ` for quantum labels associated. It can be initialized using a `numpy.ndarray.shape` and a `tuple` of `SZ`.

```
>>> from pyblock3.algebra.core import SubTensor
>>> from pyblock3.algebra.symmetry import SZ
>>> x = SubTensor.zeros((4,3), q_labels=(SZ(0, 0, 0), SZ(1, 1, 2)))
>>> x
(Q=) (< N=0 SZ=0 PG=0 >, < N=1 SZ=1/2 PG=2 >) (R=) array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> x[1,:] = 1
>>> x
(Q=) (< N=0 SZ=0 PG=0 >, < N=1 SZ=1/2 PG=2 >) (R=) array([[0., 0., 0.],
       [1., 1., 1.],
       [0., 0., 0.],
       [0., 0., 0.]])
>>> x.q_labels
(< N=0 SZ=0 PG=0 >, < N=1 SZ=1/2 PG=2 >)
```

A `SparseTensor` represents a block-sparse tensor, which contains a list of `SubTensor`. A quantum-number conserving `SparseTensor` can be initialized using a `BondInfo`, `pattern` and `dq`. `pattern` is a string of '+' or '-', indicating how to combine `SZ` to get `dq`. `dq` is the conserved quantum number. For 1D `SparseTensor`, the initialization method does not require quantum-number conservation.

```
>>> from pyblock3.algebra.core import SparseTensor
>>> from pyblock3.algebra.symmetry import SZ, BondInfo
>>> x = BondInfo({SZ(0, 0, 0): 1, SZ(1, 1, 2): 2, SZ(-1, -1, 2): 2})
>>> SparseTensor.random((x, x), pattern='++', dq=SZ(0, 0, 0))
0 (Q=) (< N=-1 SZ=-1/2 PG=2 >, < N=1 SZ=1/2 PG=2 >) (R=) array([[0.89718406, 0.85419892],
       [0.65863698, 0.98023596]])
1 (Q=) (< N=0 SZ=0 PG=0 >, < N=0 SZ=0 PG=0 >) (R=) array([[0.69742141]])
2 (Q=) (< N=1 SZ=1/2 PG=2 >, < N=-1 SZ=-1/2 PG=2 >) (R=) array([[0.50722408, 0.34099007],
       [0.40760832, 0.8430552 ]])
```

Note that the resulting tensor has three non-zero blocks, for each block, the quantum numbers adds to `dq`, which is `SZ(0, 0, 0)`. So this is a quantum-number-conserving block-sparse tensor.

`SparseTensor` supports most common *numpy.ndarray* operations:

```
>>> import numpy as np
>>> x = SparseTensor.random((x, x), pattern='++', dq=SZ(0, 0, 0))
>>> y = 2 * x
>>> np.linalg.norm(y)
3.386356824229238
>>> np.linalg.norm(x)
1.693178412114619
>>> np.tensordot(x, y, axes=1)
0 (Q=) (< N=-1 SZ=-1/2 PG=2 >, < N=-1 SZ=-1/2 PG=2 >) (R=) array([[0.37106833, 0.
↪92381267],
       [0.23117763, 1.27553566]])
1 (Q=) (< N=0 SZ=0 PG=0 >, < N=0 SZ=0 PG=0 >) (R=) array([[1.25199691]])
```

(continues on next page)

(continued from previous page)

```
2 (Q=) (< N=1 SZ=1/2 PG=2 >, < N=1 SZ=1/2 PG=2 >) (R=) array([[0.66650452, 0.63606196],
      [0.61864202, 0.98009947]])
```

A `FermionTensor` contains two `SparseTensor`, including blocks with odd `dq` and even `dq`, respectively.

`FlatSparseTensor` is another representation of block-sparse tensor, where quantum-number are combined together to a single integer, and floating-point contents of all blocks are merged to one single "flattened" `numpy.ndarray`.

`FlatSparseTensor` has the same interface as `SparseTensor`, but `FlatSparseTensor` provides much faster C++ implementation for functions like *tensordot* and *tranpose*. For debugging purpose, `FlatSparseTensor` also has pure python implementation, which can be activated by setting `ENABLE_FAST_IMPLS = False` in `flat.py`.

1.3 MPO Construction

References:

- Knowles, P. J., Handy, N. C. A determinant based full configuration interaction program. *Computer Physics Communications* 1989, **54**, 75-83.
- Chan, G. K.-L.; Keselman, A.; Nakatani, N.; Li, Z.; White, S. R. Matrix product operators, matrix product states, and ab initio density matrix renormalization group algorithms. *The Journal of Chemical Physics* 2016, **145**, 014102.
- Ren, J., Li, W., Jiang, T., & Shuai, Z. A general automatic method for optimal construction of matrix product operators using bipartite graph theory. *The Journal of Chemical Physics* 2020, **153**, 084118.
- Hubig, C., McCulloch, I. P., & Schollwöck, U. Generic construction of efficient matrix product operators. *Physical Review B* 2017, **95**, 035129.

1.3.1 From FCIDUMP

MPO can be constructed from a FCIDUMP file:

```
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP

fd = 'data/H8.STO6G.R1.8.FCIDUMP'
hamil = Hamiltonian(FCIDUMP(pg='d2h').read(fd), flat=False)
mpo = hamil.build_qc_mpo()
```

This will build an `MPS` object (representing MPO) using a list of `FermionTensor`.

If `flat` parameter is set to `True` in `Hamiltonian`, the code will use more efficient C++ code for building MPO, and the resulting MPO is an `MPS` object with a list of `FlatFermionTensor` included.

```
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP

fd = 'data/H8.STO6G.R1.8.FCIDUMP'
hamil = Hamiltonian(FCIDUMP(pg='d2h').read(fd), flat=True)
mpo = hamil.build_qc_mpo()
```

One can also use `mpo.to_flat()` to transform a `FermionTensor` MPO to a `FlatFermionTensor` MPO.

1.3.2 From Hamiltonian expression (pure python)

A slower but more general way of build MPO is from Hamiltonian expression.

One can set the explicit expression for Hamiltonian for Hubbard model and quantum chemistry:

```
def build_hubbard(u=2, t=1, n=8, cutoff=1E-9):
    fcidump = FCIDUMP(pg='c1', n_sites=n, n_elec=n, twos=0, ipg=0, orb_sym=[0] * n)
    hamil = Hamiltonian(fcidump, flat=False)

    def generate_terms(n_sites, c, d):
        for i in range(0, n_sites):
            for s in [0, 1]:
                if i - 1 >= 0:
                    yield t * c[i, s] * d[i - 1, s]
                if i + 1 < n_sites:
                    yield t * c[i, s] * d[i + 1, s]
            yield u * (c[i, 0] * c[i, 1] * d[i, 1] * d[i, 0])

    return hamil, hamil.build_mpo(generate_terms, cutoff=cutoff).to_sparse()
```

```
def build_qc(filename, pg='d2h', cutoff=1E-9):
    fcidump = FCIDUMP(pg=pg).read(fd)
    hamil = Hamiltonian(fcidump, flat=False)

    def generate_terms(n_sites, c, d):
        for i in range(0, n_sites):
            for j in range(0, n_sites):
                for s in [0, 1]:
                    t = fcidump.t(s, i, j)
                    if abs(t) > cutoff:
                        yield t * c[i, s] * d[j, s]
        for i in range(0, n_sites):
            for j in range(0, n_sites):
                for k in range(0, n_sites):
                    for l in range(0, n_sites):
                        for sij in [0, 1]:
                            for skl in [0, 1]:
                                v = fcidump.v(sij, skl, i, j, k, l)
                                if abs(v) > cutoff:
                                    yield (0.5 * v) * (c[i, sij] * c[k, skl] * d[l, skl]␣
↪* d[j, sij])

    return hamil, hamil.build_mpo(generate_terms, cutoff=cutoff, const=hamil.fcidump.
↪const_e).to_sparse()
```

Then the MPO can be built by:

```
hamil, mpo = build_hubbard(n=4)
```

or

```
fd = 'data/H8.STO6G.R1.8.FCIDUMP'
hamil, mpo = build_qc(fd, cutoff=1E-12)
```

1.3.3 From Hamiltonian expression (fast)

If C++ optimized code and numba are available, when there are very large number of terms in Hamiltonian, the MPO building process can be accelerated:

First, we can use numba optimized functions to set the Hamiltonian terms:

```
import numpy as np
import numba as nb

flat = True

SPIN, SITE, OP = 1, 2, 16384
@nb.njit(nb.types.Tuple((nb.float64[:], nb.int32[:, :]))(nb.int32, nb.float64, nb.
↪float64))
def generate_hubbard_terms(n_sites, u, t):
    OP_C, OP_D = 0 * OP, 1 * OP
    h_values = []
    h_terms = []
    for i in range(0, n_sites):
        for s in [0, 1]:
            if i - 1 >= 0:
                h_values.append(t)
                h_terms.append([OP_C + i * SITE + s * SPIN, OP_D + (i - 1) * SITE + s *␣
↪SPIN, -1, -1])
            if i + 1 < n_sites:
                h_values.append(t)
                h_terms.append([OP_C + i * SITE + s * SPIN, OP_D + (i + 1) * SITE + s *␣
↪SPIN, -1, -1])
            h_values.append(0.5 * u)
            h_terms.append([OP_C + i * SITE + s * SPIN, OP_C + i * SITE + (1 - s) * SPIN,
                            OP_D + i * SITE + (1 - s) * SPIN, OP_D + i * SITE + s *␣
↪SPIN])
    return np.array(h_values, dtype=np.float64), np.array(h_terms, dtype=np.int32)

@nb.njit(nb.types.Tuple((nb.float64[:], nb.int32[:, :]))
        (nb.int32, nb.float64[:, :], nb.float64[:, :, :, :], nb.float64))
def generate_qc_terms(n_sites, h1e, g2e, cutoff=1E-9):
    OP_C, OP_D = 0 * OP, 1 * OP
    h_values = []
    h_terms = []
    for i in range(0, n_sites):
        for j in range(0, n_sites):
            t = h1e[i, j]
            if abs(t) > cutoff:
                for s in [0, 1]:
                    h_values.append(t)
                    h_terms.append([OP_C + i * SITE + s * SPIN,
                                    OP_D + j * SITE + s * SPIN, -1, -1])
    for i in range(0, n_sites):
        for j in range(0, n_sites):
            for k in range(0, n_sites):
                for l in range(0, n_sites):
                    v = g2e[i, j, k, l]
```

(continues on next page)

(continued from previous page)

```
                    if abs(v) > cutoff:
                        for sij in [0, 1]:
                            for skl in [0, 1]:
                                h_values.append(0.5 * v)
                                h_terms.append([OP_C + i * SITE + sij * SPIN,
                                                OP_C + k * SITE + skl * SPIN,
                                                OP_D + l * SITE + skl * SPIN,
                                                OP_D + j * SITE + sij * SPIN])
    return np.array(h_values, dtype=np.float64), np.array(h_terms, dtype=np.int32)

def build_hubbard(u=2, t=1, n=8, cutoff=1E-9):
    fcidump = FCIDUMP(pg='c1', n_sites=n, n_elec=n,
                      twos=0, ipg=0, orb_sym=[0] * n)
    hamil = Hamiltonian(fcidump, flat=flat)
    terms = generate_hubbard_terms(n, u, t)
    return hamil, hamil.build_mpo(terms, cutoff=cutoff).to_sparse()

def build_qc(filename, pg='d2h', cutoff=1E-9, max_bond_dim=-1):
    fcidump = FCIDUMP(pg=pg).read(filename)
    hamil = Hamiltonian(fcidump, flat=flat)
    terms = generate_qc_terms(
        fcidump.n_sites, fcidump.h1e, fcidump.g2e, cutoff)
    return hamil, hamil.build_mpo(terms, cutoff=cutoff, max_bond_dim=max_bond_dim,␣
↪const=hamil.fcidump.const_e).to_sparse()
```

Then the MPO can be built by:

```
hamil, mpo = build_hubbard(n=16, cutoff=cutoff)
```

or

```
fd = 'data/H8.STO6G.R1.8.FCIDUMP'
hamil, mpo = build_qc(fd, cutoff=cutoff, max_bond_dim=-1)
```

1.3.4 From pyscf

The FCIDUMP can also be initialized using integral arrays, such as those obtained from pyscf. Here is an example for H10 (STO6G).

Note that running pyblock3 and pyscf in the same python script with openMP activated may cause some conflicts in parallel MKL library, in some cases. One need to check number of threads used by pyblock3 during DMRG, to make sure that number of openMP threads is correct.

```
from pyscf import gto, scf, lo, symm, ao2mo
from pyblock3.fcidump import PointGroup
# H chain
N = 10
BOHR = 0.52917721092  # Angstroms
mpg = 'c1'  # point group: d2h or c1
R = 1.8 * BOHR
mol = gto.M(atom=[['H', (0, 0, i * R)] for i in range(N)],
            basis='sto6g', verbose=0, symmetry=mpg)
```

(continues on next page)

(continued from previous page)

```
pg = mol.symmetry.lower()
mf = scf.RHF(mol)
ener = mf.kernel()
print("SCF Energy = %20.15f" % ener)

if pg == 'd2h':
    fcidump_sym = ["Ag", "B3u", "B2u", "B1g", "B1u", "B2g", "B3g", "Au"]
elif pg == 'c1':
    fcidump_sym = ["A"]

mo_coeff = mf.mo_coeff
n_mo = mo_coeff.shape[1]
orb_sym_str = symm.label_orb_symm(mol, mol.irrep_name, mol.symm_orb, mo_coeff)
xorb_sym = np.array([fcidump_sym.index(i) + 1 for i in orb_sym_str])
h1e = mo_coeff.T @ mf.get_hcore() @ mo_coeff
g2e = ao2mo.restore(1, ao2mo.kernel(mol, mo_coeff), n_mo)
ecore = mol.energy_nuc()
na = nb = mol.nelectron // 2

orb_sym = [PointGroup[mpg][i] for i in xorb_sym]
fd = FCIDUMP(pg='c1', n_sites=n_mo, n_elec=na + nb, twos=na - nb, ipg=0, uhf=False,
             h1e=h1e, g2e=g2e, orb_sym=orb_sym, const_e=ecore, mu=0)
hamil = Hamiltonian(fd, flat=True)
mpo = hamil.build_qc_mpo()
```

1.4 MPS Algebra

Construct MPO (set `flat=False` if you want to test pure python code):

```
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP

fd = 'data/HUBBARD-L8.FCIDUMP'
hamil = Hamiltonian(FCIDUMP(pg='c1').read(fd), flat=True)
mpo = hamil.build_qc_mpo()
```

Construct (random initial) MPS:

```
bond_dim = 100
mps = hamil.build_mps(bond_dim)
```

Expectation value:

```
import numpy as np
np.dot(mps, mpo @ mps)
```

Block-sparse tensor algebra:

```
np.tensordot(mps[0], mps[1], axes=1)
```

MPS canonicalization:

```
print("MPS = ", mps.show_bond_dims())
mps = mps.canonicalize(center=0)
mps /= mps.norm()
print("MPS = ", mps.show_bond_dims()
```

Check norm after normalization:

```
np.dot(mps, mps)
```

MPO Compression:

```
print("MPO = ", mpo.show_bond_dims())
mpo, _ = mpo.compress(left=True, cutoff=1E-12, norm_cutoff=1E-12)
print("MPO = ", mpo.show_bond_dims())
```

DMRG:

```
from pyblock3.algebra.mpe import MPE
dmrg = MPE(mps, mpo, mps).dmrg(bdims=[bond_dim], noises=[1E-6, 0], dav_thrds=[1E-3],␣
↪iprint=2, n_sweeps=10)
ener = dmrg.energies[-1]
print("Energy = %20.12f" % ener)
```

Check ground-state energy:

```
print('MPS energy = ', np.dot(mps, mpo @ mps))
```

Check that ground-state MPS is normalized:

```
print('MPS = ', mps.show_bond_dims())
print('MPS norm = ', mps.norm())
```

1.4.1 MPS Scaling

MPS scaling (by scaling the first MPS tensor):

```
mps.opts = {}
print('2 MPS = ', (2 * mps).show_bond_dims())
print((2 * mps).norm())
```

Check the first MPS tensor:

```
mps[0]
```

and

```
(2 * mps)[0]
```

1.4.2 MPS Addition

MPS addition will increase the bond dimension:

```
mps_add = mps + mps
print('MPS + MPS = ', mps_add.show_bond_dims())
print(mps_add.norm())
```

Check the overlap $< 2MPS|MPS + MPS >$:

```
mps_add @ (2 * mps)
```

1.4.3 MPS Canonicalization

Left canonicalization:

```
lmps = mps_add.canonicalize(center=mps_add.n_sites - 1)
print('L-MPS = ', lmps.show_bond_dims())
```

Right canonicalization:

```
rmps = mps_add.canonicalize(center=0)
print('R-MPS = ', rmps.show_bond_dims())
```

Check the overlap $< LMPS|RMPS >$:

```
lmps @ rmps
```

1.4.4 MPS Compression

Compression will first do canonicalization from left to right, then do SVD from right to left.

This can further decrease bond dimension of MPS.

```
print('MPS + MPS = ', mps_add.show_bond_dims())
mps_add, _ = mps_add.compress(cutoff=1E-9)
print('MPS + MPS = ', mps_add.show_bond_dims())
print(mps_add.norm())
```

1.4.5 MPS Subtraction

Subtractoin will also increase bond dimension:

```
mps_minus = mps - mps
print('MPS - MPS = ', mps_minus.show_bond_dims())
```

After compression, this is zero:

```
mps_minus, _ = mps_minus.compress(cutoff=1E-12)
print('MPS - MPS = ', mps_minus.show_bond_dims())
print(mps_minus.norm())
```

1.4.6 MPS Bond Dimension Truncation

Apply MPO two times to MPS:

```
hhmps = mpo @ (mpo @ mps)
print(hhmps.show_bond_dims())
print(np.sqrt(hhmps @ mps))
```

MPS compression can be used to reduce bond dimension (to FCI):

```
hhmps, cps_error = hhmps.compress(cutoff=1E-12)
print('error = ', cps_error)
print(hhmps.show_bond_dims())
print(np.sqrt(hhmps @ mps))
```

Truncation to bond dimension 100 will introduce a small error:

```
hhmps, cps_error = hhmps.compress(max_bond_dim=100, cutoff=1E-12)
print('error = ', cps_error)
print(hhmps.show_bond_dims())
print(np.sqrt(hhmps @ mps))
```

Truncation to bond dimension 30 will introduce a larger error:

```
hhmps, cps_error = hhmps.compress(max_bond_dim=30, cutoff=1E-12)
print('error = ', cps_error)
print(hhmps.show_bond_dims())
print(np.sqrt(hhmps @ mps))
```

1.4.7 MPO-MPO Contraction

One can also first contract two MPO:

```
h2 = mpo @ mpo
print(h2.show_bond_dims())
```

Check expectation value:

```
print(np.sqrt((h2 @ mps) @ mps))
```

1.4.8 MPO Bond Dimension Truncation

Compression MPO (keeping accuracy):

```
h2, cps_error = h2.compress(cutoff=1E-12)
print('error = ', cps_error)
print(h2.show_bond_dims())
print(np.sqrt((h2 @ mps) @ mps))
```

MPO Truncated to bond dimension 15:

```
h2, cps_error = h2.compress(max_bond_dim=15, cutoff=1E-12)
print('error = ', cps_error)
print(h2.show_bond_dims())
print(np.sqrt((h2 @ mps) @ mps))
```

MPO Truncated to bond dimension 12:

```
h2, cps_error = h2.compress(max_bond_dim=12, cutoff=1E-12)
print('error = ', cps_error)
print(h2.show_bond_dims())
print(np.sqrt((h2 @ mps) @ mps))
```

1.5 Determinant Tools

1.5.1 N2(10o, 7e) (STO3G)

Ground-state DMRG (N2 STO3G) with C++ optimized core functions:

```
import numpy as np
from pyblock3.algebra.mpe import MPE
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP

fd = 'data/N2.STO3G.FCIDUMP'
bond_dim = 250
hamil = Hamiltonian(FCIDUMP(pg='d2h').read(fd), flat=True)
mpo = hamil.build_qc_mpo()
mpo, _ = mpo.compress(cutoff=1E-9, norm_cutoff=1E-9)
mps = hamil.build_mps(bond_dim)

dmrg = MPE(mps, mpo, mps).dmrg(bdims=[bond_dim], noises=[1E-6, 0], dav_thrds=[1E-4],␣
↪iprint=2, n_sweeps=10)
ener = dmrg.energies[-1]
print("Energy = %20.12f" % ener)
```

Check MPO:

```
print('MPO = ', mpo.show_bond_dims())
mpo, error = mpo.compress(cutoff=1E-12)
print('MPO = ', mpo.show_bond_dims())
```

Check MPS:

```
print('MPS = ', mps.show_bond_dims())
print(mps.norm())
```

Check ground-state energy:

```
mps @ (mpo @ mps)
```

1.5.2 MPO-MPS Contraction

```
mps.opts = {}
hmps = mpo @ mps
print(hmps.show_bond_dims())
print(hmps.norm())
```

The result MPS can be compressed:

```
hmps, _ = hmps.compress(cutoff=1E-12)
print(hmps.show_bond_dims())
print(hmps.norm())
```

MPO truncation to bond dimension 50:

```
cmpo, cps_error = mpo.compress(max_bond_dim=50, cutoff=1E-12)
print('error = ', cps_error)
print(cmpo.show_bond_dims())
```

Apply contracted MPO to MPS:

```
hmps = cmpo @ mps
print(hmps.show_bond_dims())
print(hmps.norm())
```

1.5.3 Determinants

Using SliceableTensor:

```
smps = mps.to_non_flat().to_sliceable()
print(smps[0])
print('-'*20)
print(smps[0][:, 2:, 2])
print('-'*20)
print(smps[0][:, :2, 2].infos)
print('-'*20)
print(smps[0])
print('-'*20)
print(smps.amplitude([3, 3, 0, 3, 0, 3, 3, 3, 3, 0]))
```

If the determinant belongs to another symmetry sector, the overlap should be zero:

```
print(smps.amplitude([3, 3, 0, 0, 0, 3, 3, 3, 3, 0]))
```

Check the overlap for all doubly occupied determinants:

```
import itertools
coeffs = []
for ocp in itertools.combinations(range(10), 7):
    det = [0] * mps.n_sites
    for t in ocp:
        det[t] = 3
    tt = time.perf_counter()
```

(continues on next page)

(continued from previous page)

```
    coeffs.append(smps.amplitude(det))
    print(np.array(det), "%10.5f" % coeffs[-1])
```

Check the sum of probabilities:

```
print((np.array(coeffs) ** 2).sum())
```

1.6 One-Particle Density Matrix

1.6.1 N2(10o, 7e) (STO3G)

Ground-state DMRG (N2 STO3G) with C++ optimized core functions:

```
import numpy as np
from pyblock3.algebra.mpe import MPE
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP
from pyblock3.symbolic.expr import OpElement, OpNames
from pyblock3.algebra.symmetry import SZ

fd = 'data/N2.STO3G.FCIDUMP'
bond_dim = 500
hamil = Hamiltonian(FCIDUMP(pg='d2h').read(fd), flat=True)
mpo = hamil.build_qc_mpo()
mpo, _ = mpo.compress(cutoff=1E-9, norm_cutoff=1E-9)
mps = hamil.build_mps(bond_dim)

dmrg = MPE(mps, mpo, mps).dmrg(bdims=[bond_dim], noises=[1E-6, 0], dav_thrds=[1E-4],␣
↪iprint=2, n_sweeps=10)
ener = dmrg.energies[-1]
print("Energy = %20.12f" % ener)

print('energy error = ', np.abs(ener - -107.654122447525))
assert np.abs(ener - -107.654122447525) < 1E-6
```

Now we can calculate the 1pdm based on ground-state MPS, and compare it with the FCI result.

```
# FCI results
pdm1_std = np.zeros((hamil.n_sites, hamil.n_sites))
pdm1_std[0, 0] = 1.999989282592
pdm1_std[0, 1] = -0.000025398134
pdm1_std[0, 2] = 0.000238560621
pdm1_std[1, 0] = -0.000025398134
pdm1_std[1, 1] = 1.991431489457
pdm1_std[1, 2] = -0.005641787787
pdm1_std[2, 0] = 0.000238560621
pdm1_std[2, 1] = -0.005641787787
pdm1_std[2, 2] = 1.985471515555
pdm1_std[3, 3] = 1.999992764813
pdm1_std[3, 4] = -0.000236022833
```

(continues on next page)

(continued from previous page)

```
pdm1_std[3, 5] = 0.000163863520
pdm1_std[4, 3] = -0.000236022833
pdm1_std[4, 4] = 1.986371259953
pdm1_std[4, 5] = 0.018363506969
pdm1_std[5, 3] = 0.000163863520
pdm1_std[5, 4] = 0.018363506969
pdm1_std[5, 5] = 0.019649294772
pdm1_std[6, 6] = 1.931412559660
pdm1_std[7, 7] = 0.077134636900
pdm1_std[8, 8] = 1.931412559108
pdm1_std[9, 9] = 0.077134637190

pdm1 = np.zeros((hamil.n_sites, hamil.n_sites))
for i in range(hamil.n_sites):
    diop = OpElement(OpNames.D, (i, 0), q_label=SZ(-1, -1, hamil.orb_sym[i]))
    di = hamil.build_site_mpo(diop)
    for j in range(hamil.n_sites):
        djop = OpElement(OpNames.D, (j, 0), q_label=SZ(-1, -1, hamil.orb_sym[j]))
        dj = hamil.build_site_mpo(djop)
        # factor 2 due to alpha + beta spins
        pdm1[i, j] = 2 * np.dot((di @ mps).conj(), dj @ mps)

# 1pdm error is often approximately np.sqrt(error in energy)
print('max 1pdm error = ', np.abs(pdm1 - pdm1_std).max())
assert np.abs(pdm1 - pdm1_std).max() < 1E-6
```

1.7 Finite-Temperature DMRG

References:

- Feiguin, A. E., White, S. R. Finite-temperature density matrix renormalization using an enlarged Hilbert space. *Physical Review B* 2005, **72**, 220401.
- Feiguin, A. E., White, S. R. Time-step targeting methods for real-time dynamics using the density matrix renormalization group. *Physical Review B* 2005, **72**, 020404.

Here is an example for calculating $\exp(-\beta/2 \cdot H)|\psi\rangle$.

Imports:

```
from pyblock3.algebra.integrate import rk4_apply
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP
import numpy as np
import time
from functools import reduce

flat = False
cutoff = 1E-12
```

1.7.1 Ancilla Approach

```
fd = 'data/H8.STO6G.R1.8.FCIDUMP'
hamil = Hamiltonian(FCIDUMP(pg='d2h', mu=-1.0).read(fd), flat=flat)

mps = hamil.build_ancilla_mps()
mpo = hamil.build_ancilla_mpo(hamil.build_qc_mpo())
mpo.const = 0.0

print('MPS = ', mps.show_bond_dims())
print('MPO = ', mpo.show_bond_dims())
mpo, error = mpo.compress(cutoff=cutoff)
print('MPO = ', mpo.show_bond_dims(), error)

init_e = np.dot(mps, mpo @ mps) / np.dot(mps, mps)
print('Initial Energy = ', init_e)
print('Error          = ', init_e - 0.3124038410492045)

mps.opts = dict(max_bond_dim=200, cutoff=cutoff)
beta = 0.01
tt = time.perf_counter()
fmps = rk4_apply((-beta / 2) * mpo, mps)
ener = np.dot(fmps, mpo @ fmps) / np.dot(fmps, fmps)
print('time = ', time.perf_counter() - tt)
print('Energy = ', ener)
print('Error  = ', ener - 0.2408363230374028)
```

1.7.2 Matrix Product Ancilla Operator Approach

Here, the ancilla MPS is contracted to an MPO, where both the physical and the auxiliary dimension is located on one site each.

```
mps = hamil.build_ancilla_mps()
mps.tensors = [a.hdot(b) for a, b in zip(mps.tensors[0::2], mps.tensors[1::2])]
mpo = hamil.build_qc_mpo()
mpo.const = 0.0

print('MPS = ', mps.show_bond_dims())
print('MPO = ', mpo.show_bond_dims())
mpo, _ = mpo.compress(cutoff=cutoff)
print('MPO = ', mpo.show_bond_dims())

init_e = np.dot(mps, mpo @ mps) / np.dot(mps, mps)
print('Initial Energy = ', init_e)
print('Error          = ', init_e - 0.3124038410492045)

mps.opts = dict(max_bond_dim=200, cutoff=cutoff)
beta = 0.01
tt = time.perf_counter()
fmps = rk4_apply((-beta / 2) * mpo, mps)
ener = np.dot(fmps, mpo @ fmps) / np.dot(fmps, fmps)
print('time = ', time.perf_counter() - tt)
```

(continues on next page)

(continued from previous page)

```
print('Energy = ', ener)
print('Error  = ', ener - 0.2408363230374028)
```

1.7.3 Time-Step Targeting Approach

The more efficient way of imaginary time evolution is using Time-Step Targeting Approach:

```
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP
from pyblock3.algebra.mpe import MPE
import numpy as np
import time

flat = True
cutoff = 1E-12

fd = '../data/H8.STO6G.R1.8.FCIDUMP'
hamil = Hamiltonian(FCIDUMP(pg='d2h', mu=-1.0).read(fd), flat=flat)

mps = hamil.build_ancilla_mps()
mpo = hamil.build_qc_mpo()
mpo = hamil.build_ancilla_mpo(mpo)
mpo.const = 0.0

print('MPS = ', mps.show_bond_dims())
print('MPO = ', mpo.show_bond_dims())
mpo, error = mpo.compress(cutoff=cutoff)
print('MPO = ', mpo.show_bond_dims(), error)

init_e = np.dot(mps, mpo @ mps) / np.dot(mps, mps)
print('Initial Energy = ', init_e)
print('Error          = ', init_e - 0.3124038410492045)

beta = 0.05
mpe = MPE(mps, mpo, mps)
mpe.tddmrg(bdims=[500], dt=-beta / 2, iprint=2, n_sweeps=1, n_sub_sweeps=6)
mpe.tddmrg(bdims=[500], dt=-beta / 2, iprint=2, n_sweeps=9, n_sub_sweeps=2)
```

1.8 DDMRG++ for Green's Function

References:

- Ronca, E., Li, Z., Jimenez-Hoyos, C. A., Chan, G. K. L. Time-step targeting time-dependent and dynamical density matrix renormalization group algorithms with ab initio Hamiltonians. *Journal of Chemical Theory and Computation* 2017, **13**, 5560-5571.

The following example calculate the Green's function for H10 (STO6G):

$$G_{ij}(\omega) = \langle \Psi_0 | a_j^\dagger \frac{1}{\omega + \hat{H}_0 - E_0 + i\eta} a_i | \Psi_0 \rangle$$

where $|\Psi_0\rangle$ is the ground-state, $i = j = 4$ (`isite`), $\omega = -0.17, \eta = 0.05$.

```
import time
import numpy as np
from pyblock3.algebra.mpe import MPE
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.fcidump import FCIDUMP
from pyblock3.symbolic.expr import OpElement, OpNames
from pyblock3.algebra.symmetry import SZ

np.random.seed(1000)

fd = 'data/H10.STO6G.R1.8.FCIDUMP'
ket_bond_dim = 500
bra_bond_dim = 750

hamil = Hamiltonian(FCIDUMP(pg='d2h').read(fd), flat=True)

tx = time.perf_counter()
mpo = hamil.build_qc_mpo()
print('MPO (NC) =         ', mpo.show_bond_dims())
print('build mpo time = ', time.perf_counter() - tx)

tx = time.perf_counter()
mpo, _ = mpo.compress(left=True, cutoff=1E-9, norm_cutoff=1E-9)
print('MPO (compressed) = ', mpo.show_bond_dims())
print('compress mpo time = ', time.perf_counter() - tx)

mps = hamil.build_mps(ket_bond_dim, occ=occ)
print('MPS = ', mps.show_bond_dims())

bdims = [500]
noises = [1E-4, 1E-5, 1E-6, 0]
davthrds = None

dmrg = MPE(mps, mpo, mps).dmrg(bdims=bdims, noises=noises,
                               dav_thrds=davthrds, iprint=2, n_sweeps=20, tol=1E-12)
ener = dmrg.energies[-1]
print("Energy = %20.12f" % ener)

isite = 4
mpo.const -= ener
omega, eta = -0.17, 0.05

dop = OpElement(OpNames.D, (isite, 0), q_label=SZ(-1, -1, hamil.orb_sym[isite]))
bra = hamil.build_mps(bra_bond_dim, target=SZ.to_flat(
    dop.q_label + SZ.from_flat(hamil.target)))
dmpo = hamil.build_site_mpo(dop)
print('DMPO =          ', dmpo.show_bond_dims())
MPE(bra, dmpo, mps).linear(bdims=[bra_bond_dim], noises=noises,
                                  cg_thrds=davthrds, iprint=2, n_sweeps=20, tol=1E-12)

np.random.seed(0)

gbra = hamil.build_mps(bra_bond_dim, target=SZ.to_flat(
```

(continues on next page)

(continued from previous page)

```
    dop.q_label + SZ.from_flat(hamil.target)))
print('GFMPO =          ', mpo.show_bond_dims())
print(MPE(bra, dmpo, mps).greens_function(mpo, omega, eta, bdims=[bra_bond_dim],␣
↪noises=noises,
                                  cg_thrds=[1E-4] * 10, iprint=2, n_sweeps=10, tol=1E-4))
```

1.9 Real-Time Time-Dependent DMRG (RT-TD-DMRG)

References:

- Ronca, E., Li, Z., Jimenez-Hoyos, C. A., Chan, G. K. L. Time-step targeting time-dependent and dynamical density matrix renormalization group algorithms with ab initio Hamiltonians. *Journal of Chemical Theory and Computation* 2017, **13**, 5560-5571.

Here we use RT-TD-DMRG and Fast Fourier Transform (FFT) to calculate the same quantity defined in previous section, namely, the Green's function for H10 (STO6G) (for a wide range of frequencies):

$$G_{ij}(\omega) = \langle \Psi_0 | a_j^\dagger \frac{1}{\omega + \hat{H}_0 - E_0 + i\eta} a_i | \Psi_0 \rangle$$

where $|\Psi_0\rangle$ is the ground-state, $i = j = 4$ (`isite`), $\eta = 0.05$.

This is obtained from a Fourier transform from time domain to frequency domain:

$$\begin{aligned} G_{ij}(t) &= -i \langle \Psi_0 | a_j^\dagger \mathrm{e}^{-i(\hat{H}_0 - E_0)t} a_i | \Psi_0 \rangle \\ G_{ij}(\omega) &= \int_{-\infty}^{\infty} \mathrm{d}t \mathrm{e}^{-i\omega t} G_{ij}(t) \mathrm{e}^{-\eta t} \end{aligned}$$

where $\mathrm{e}^{-\eta t}$ is a broading factor.

```
import time
import numpy as np
from pyblock3.algebra.mpe import MPE
from pyblock3.hamiltonian import Hamiltonian
from pyblock3.algorithms.core import DecompositionTypes
from pyblock3.fcidump import FCIDUMP
from pyblock3.symbolic.expr import OpElement, OpNames
from pyblock3.algebra.symmetry import SZ

np.random.seed(1000)
```

First, we load the definition of a quantum chemistry Hamiltonian from a FCIDUMP file. Use `flat = True` to activate the efficient C++ backend.

```
fd = '../data/H10.STO6G.R1.8.FCIDUMP'
ket_bond_dim = 500
bra_bond_dim = 500

hamil = Hamiltonian(FCIDUMP(pg='d2h').read(fd), flat=True)
```

Then, we build the MPO `mpo` for the Hamiltonian. The compression of `mpo` can decrease the MPO bond dimension, which will then save some runtime during DMRG and time evolution algorithms.

```
tx = time.perf_counter()
mpo = hamil.build_qc_mpo()
print('MPO (NC) =          ', mpo.show_bond_dims())
print('build mpo time = ', time.perf_counter() - tx)

tx = time.perf_counter()
mpo, _ = mpo.compress(left=True, cutoff=1E-9, norm_cutoff=1E-9)
print('MPO (compressed) = ', mpo.show_bond_dims())
print('compress mpo time = ', time.perf_counter() - tx)
```

Now we build a random MPS, as the initial guess for the DMRG algorithm.

```
mps = hamil.build_mps(ket_bond_dim)
print('MPS = ', mps.show_bond_dims())
```

`MPE` (Matrix Product Expectation) is a bra-mpo-ket tensor network structure, with some partial contraction of environments stored internally. DMRG (sweep) algorithm can be invoked based on `MPE`. For DMRG algorithm, bra and ket are the same, both represented as the `mps` object.

```
bdims = [500]
noises = [1E-4, 1E-5, 1E-6, 0]
davthrds = None

dmrg = MPE(mps, mpo, mps).dmrg(bdims=bdims, noises=noises,
                               dav_thrds=davthrds, iprint=2, n_sweeps=20, tol=1E-12)
ener = dmrg.energies[-1]
print("Energy = %20.12f" % ener)
```

Now `ener` is the ground-state energy E_0 of the system. We substract this constant from MPO to let the `mpo` object represent $\hat{H}_0 - E_0$.

```
isite = 4
mpo.const -= ener
```

Here, `dop` is the destruction operator $\hat{a}_{4\alpha}$, defined using `OpElement`, where the first argument `OpNames.D` is the operator name, the second argument `(isite, 0)` is the orbital index (counting from zero) and spin index (0 = alpha, 1 = beta), and the last argument `q_label` is a quantum number, representing how this operator changes the quantum number of a state. Here $\hat{a}_{4\alpha}$ will decrease particle number by 1, decrease `2S_z` by 1, and change point group irrep by the point group irrep of orbital `isite` (which is 4 here). An MPO `dmpo` (bond dimension = 1) can be directly built from single site operator `dop` using `hamil.build_site_mpo()`.

```
dop = OpElement(OpNames.D, (isite, 0), q_label=SZ(-1, -1, hamil.orb_sym[isite]))
dmpo = hamil.build_site_mpo(dop)
print('DMPO =          ', dmpo.show_bond_dims())
```

Next, we need to construct an MPS `bra`, which is $\hat{a}_{4\alpha}|\Psi_0\rangle$ where $|\Psi_0\rangle$ is the ground-state `mps`. First we define `bra` as a random MPS with the correct quantum number. The quantum number of `bra` is simply the sum of the quantum number of `dop` and `mps`.

```
bra = hamil.build_mps(bra_bond_dim, target=SZ.to_flat(
    dop.q_label + SZ.from_flat(hamil.target)))
```

Then we use `MPE.linear()` to fit `bra` to `dmpo @ mps`. This is a sweep algorithm similar to DMRG. In principle, the following line (and the above line) can be replaced by simply `bra = dmpo @ mps; bra.fix_pattern()` (which

may be slower). Also note that `MPE.linear` may have some problems handling the constant term in MPO. If the mpo has a constant term (the `dmpo` here does not have a constant), one can do `MPE(bra, mpo - mpo.const, mps).linear(...); bra += mpo.const * mps`.

```
MPE(bra, dmpo, mps).linear(bdims=[bra_bond_dim], noises=noises,
                                cg_thrds=davthrds, iprint=2, n_sweeps=20, tol=1E-12)
```

Now we obtain a (deep) copy of `bra` to be `ket`. Later when we time evolve `ket`, `bra` will not be changed.

```
ket = bra.copy()
dt = 0.1
eta = 0.05
t = 100.0

nstep = int(t / dt)
```

Real time evolution can be performed using `MPE.tddmrg()`, with a imaginary `dt` argument. `normalize` should be set to `False`, so that we will not keep `ket` normalized, so that the constant prefactor in `ket` will transformed into `rtgf`, which is convenient. Note that since (in principle) real time evolution does not change the norm of the MPS, whether keeping the MPS normalized should not make a difference. When MPS is not explicitly normalized, the norm of MPS printed after each sweep can be used as an indicator of the accuracy of the algorithm. For imaginary time evolution, however, it is recommended to explicitly normalize MPS, since during the imaginary time evolution the prefactor in the MPS is not a constant. It can grow up rapidly, which may create some numerical problem.

After each time step, the overlap between `bra` and `ket`, which is $G_{44}(t)$, is calculated and stored in `rtgf`.

```
mpe = MPE(ket, mpo, ket)
rtgf = np.zeros((nstep, ), dtype=complex)
print('total step = ', nstep)
for it in range(0, nstep):
    cur_t = it * dt
    mpe.tddmrg(bdims=[500], dt=-dt * 1j, iprint=2, n_sweeps=1, n_sub_sweeps=2,␣
↪normalize=False)
    rtgf[it] = np.dot(bra.conj(), ket)
    print("=== T = %10.5f EX = %20.15f + %20.15f i" % (cur_t, rtgf[it].real, rtgf[it].
↪imag))
```

A single step of time evolution can also be written as (currently not completely supported), which can be significantly slower than `MPE.tddmrg()`.

```
# fmps = rk4_apply((-dt * 1j) * mpo, mps)
```

Finally, one can use FFT to transform back to frequency domain.

```
def gf_fft(eta, dt, rtgf):

    dt = abs(dt)
    npts = len(rtgf)

    frq = np.fft.fftfreq(npts, dt)
    frq = np.fft.fftshift(frq) * 2.0 * np.pi
    fftinp = -1j * rtgf * np.exp(-eta * dt * np.arange(0, npts))

    Y = np.fft.fft(fftinp)
    Y = np.fft.fftshift(Y)
```

(continues on next page)

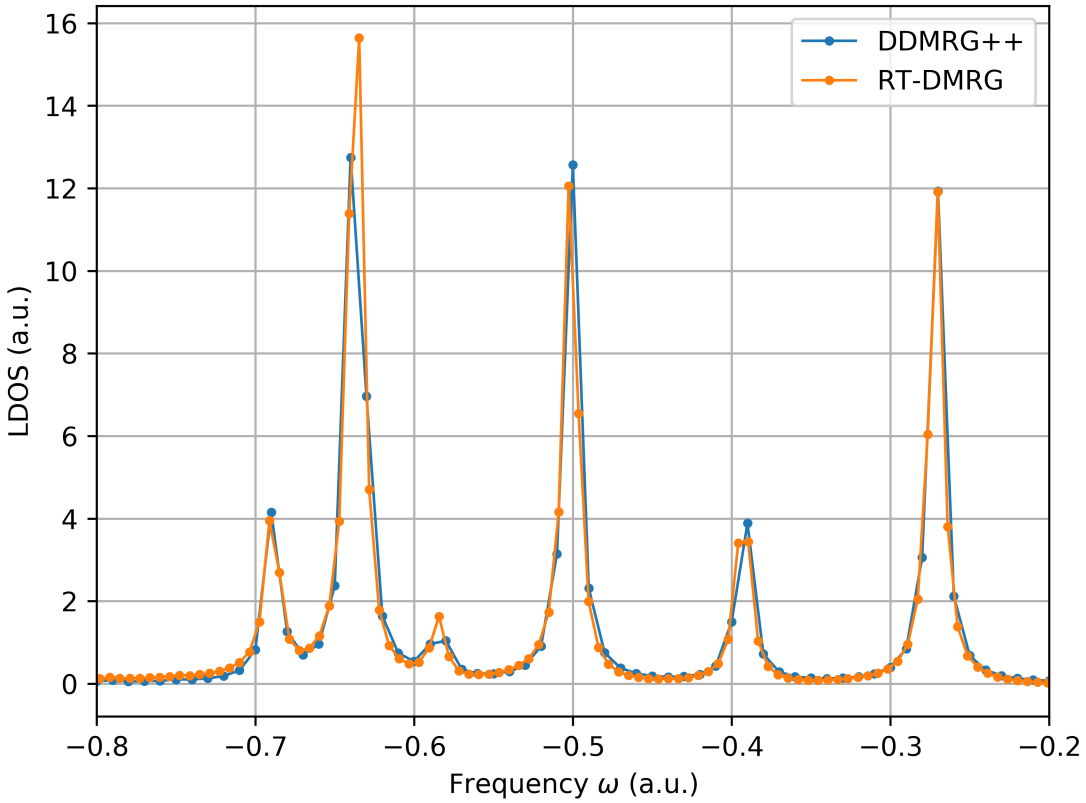
(continued from previous page)

```
    Y_real = Y.real * dt
    Y_imag = Y.imag * dt

    return frq, Y_real, Y_imag

frq, yreal, yimag = gf_fft(eta, dt, rtgf)
```

The following figure compares the results obtained from DDMRG++ and td-DMRG (with lowdin orbitals, dt = 0.1, eta = 0.005, t = 1000.0).



CHAPTER

TWO

PYBLOCK3 API REFERENCES

2.1 pyblock3.algebra

2.1.1 pyblock3.algebra.core

Basic definitions for block-sparse tensors and block-sparse tensors with fermion factors.

class pyblock3.algebra.core.**FermionTensor**(**args: Any*, ***kwargs: Any*)

Bases: NDArrayOperatorsMixin

block-sparse tensor with fermion factors.

Attributes:

odd
[SparseTensor] Including blocks with odd fermion parity.

even
[SparseTensor] Including blocks with even fermion parity.

conj()
Complex conjugate. Note that np.conj() is a ufunc (no need to be defined). But np.real and np.imag are array_functions

copy()

deflate(*cutoff=0*)

diag()

property dtype
Element datatype.

fuse(**idxs*, *info=None*, *pattern=None*)
Fuse several legs to one leg.

Args:

idxs
[tuple(int)] Leg indices to be fused. The new fused index will be idxs[0].

info
[BondFusingInfo (optional)] Indicating how quantum numbers are collected. If not specified, the direct sum of quantum numbers will be used. This will generate minimal and (often) incomplete fused shape.

pattern
[str (optional)] A str of '+'/'-'. Only required when info is not specified. Indicating how quantum numbers are linearly combined.

hdot(*b*, *out=None*)

Horizontally contract operator tensors (contracting connected virtual dimensions).

property imag

property infos

Return the quantum number layout of the FermionTensor, similar to numpy.ndarray.shape.

item()

Return scalar element.

kron_add(*b*, *infos=None*)

Direct sum of first and last legs. Middle legs are summed.

kron_product_info(**idxs*, *pattern=None*)

kron_sum_info(**idxs*, *pattern=None*)

left_canonicalize(*mode='reduced'*)

Left canonicalization (using QR factorization). Left canonicalization needs to collect all left indices for each specific right index. So that we will only have one R, but left dim of q is unchanged.

Returns:
q, r : (FermionTensor, SparseTensor (gauge))

left_svd(*full_matrices=True*)

Left svd needs to collect all left indices for each specific right index.

Returns:
l, s, r : (FermionTensor, SparseTensor (vector), SparseTensor (gauge))

lq(*mode='reduced'*)

property n_blocks

Number of (non-zero) blocks.

property nbytes

Number bytes in memory.

property ndim

Number of dimensions.

static ones(*bond_infos*, *pattern=None*, *dq=None*, *dtype=<class 'float'>*)

Create operator tensor with ones.

pdot(*b*, *out=None*)

qr(*mode='reduced'*)

static random(*bond_infos*, *pattern=None*, *dq=None*, *dtype=<class 'float'>*)

Create operator tensor with random elements.

property real

`right_canonicalize`(*mode='reduced'*)

Right canonicalization (using QR factorization).

Returns:
l, q : (SparseTensor (gauge), FermionTensor)

`right_svd`(*full_matrices=True*)

Right svd needs to collect all right indices for each specific left index.

Returns:
l, s, r : (SparseTensor (gauge), SparseTensor (vector), FermionTensor)

`shdot`(*b*, *out=None*)

Horizontally contract operator tensors (matrices) in symbolic matrix.

`symmetry_fuse`(*finfos*, *symm_map*)

`tensordot`(*b*, *axes=2*)

`to_dense`(*infos=None*)

`to_sliceable`(*infos=None*)

`to_sparse`()

`transpose`(*axes=None*)

`static truncate_svd`(*l*, *s*, *r*, *max_bond_dim=-1*, *cutoff=0.0*, *max_dw=0.0*, *norm_cutoff=0.0*)

Truncate tensors obtained from SVD.

Args:

l, s, r
[tuple(SparseTensor/FermionTensor)] SVD tensors.

max_bond_dim
[int] Maximal total bond dimension. If *k == -1*, no restriction in total bond dimension.

cutoff
[double] Minimal kept singular value.

max_dw
[double] Maximal sum of square of discarded singular values.

norm_cutoff
[double] Blocks with norm smaller than norm_cutoff will be deleted.

Returns:

l, s, r
[tuple(SparseTensor/FermionTensor)] SVD decomposition.

error
[float] Truncation error (same unit as singular value squared).

`unfuse`(*i*, *info*)

Unfuse one leg to several legs. May introduce some additional zero blocks.

Args:

i
[int] index of the leg to be unfused. The new unfused indices will be i, i + 1, ...

info
[BondFusingInfo] Indicating how quantum numbers are collected.

static zeros(*bond_infos*, *pattern=None*, *dq=None*, *dtype=<class 'float'>*)
Create operator tensor with zero elements.

class pyblock3.algebra.core.**SliceableTensor**(*reduced*, *infos=None*)
Bases: ndarray

Dense tensor of zero and non-zero blocks. For zero blocks, the elemenet is zero.

copy()

property density
Ratio of number of non-zero elements to total number of elements.

dot(*b*, *out=None*)

property dtype

tensordot(*b*, *axes=2*)

to_dense()
Convert to dense numpy.ndarray.

to_sparse()

class pyblock3.algebra.core.**SparseTensor**(**args: Any*, ***kwargs: Any*)
Bases: NDArrayOperatorsMixin

block-sparse tensor. Represented as a list of *SubTensor*.

property T
Transpose.

add(*b*)

conj()
Complex conjugate. Note that np.conj() is a ufunc (no need to be defined). But np.real and np.imag are array_functions

copy()

deflate(*cutoff=0*)
Remove zero blocks.

diag()

property dtype
Element datatype.

fuse(**idxs*, *info=None*, *pattern=None*)
Fuse several legs to one leg.

Args:

idxs
[tuple(int)] Leg indices to be fused. The new fused index will be min(idxs).

info
[BondFusingInfo (optional)] Indicating how quantum numbers are collected. If not specified, the direct sum of quantum numbers will be used. This will generate minimal and (often) incomplete fused shape.

pattern
[str (optional)] A str of '+'/'-'. Only required when info is not specified. Indicating how quantum numbers are linearly combined. len(pattern) == len(idxs)

hdot(*b*, *out=None*)

Horizontal contraction (contracting connected virtual dimensions).

property imag

property infos

Return the quantum number layout of the SparseTensor, similar to numpy.ndarray.shape.

item()

Return scalar element.

kron_add(*b*, *infos=None*)

Direct sum of first and last legs. Middle legs are summed.

kron_product_info(**idxs*, *pattern=None*)

Kron product of quantum numbers along selected indices, for fusing purpose.

kron_sum_info(**idxs*, *pattern=None*)

Kron sum of quantum numbers along selected indices, for fusing purpose.

left_canonicalize(*mode='reduced'*)

Left canonicalization (using QR factorization). Left canonicalization needs to collect all left indices for each specific right index. So that we will only have one R, but left dim of q is unchanged.

Returns:
q, r : tuple(SparseTensor)

left_svd(*full_matrices=False*)

Left svd needs to collect all left indices for each specific right index.

Returns:
l, s, r : tuple(SparseTensor)

lq(*mode='reduced'*)

property n_blocks

Number of blocks.

property nbytes

Number bytes in memory.

property ndim

Number of dimensions.

norm()

normalize_along_axis(*axis*)

static ones(*bond_infos*, *pattern=None*, *dq=None*, *dtype=<class 'float'>*)

Create tensor from tuple of BondInfo with ones.

pdot(*b*, *out=None*)

Vertical contraction (all middle dims).

qr(*mode='reduced'*)

quick_deflate()

static random(*bond_infos*, *pattern=None*, *dq=None*, *dtype=<class 'float'>*)

Create tensor from tuple of BondInfo with random elements.

property real

right_canonicalize(*mode='reduced'*)

Right canonicalization (using QR factorization).

Returns:
l, q : tuple(SparseTensor)

right_svd(*full_matrices=False*)

Right svd needs to collect all right indices for each specific left index.

Returns:
l, s, r : tuple(SparseTensor)

subtract(*b*)

symmetry_fuse(*finfos*, *symm_map*)

Change from higher symmetry to lower symmetry.

Args:

finfos
[list(BondFusingInfo)] generated using BondFusingInfo.get_symmetry_fusing_info

symm_map
[lambda h: l] Map from higher symemtry irrep to lower symmetry irrep

tensor_svd(*idx=2*, *linfo=None*, *rinfo=None*, *pattern=None*, *full_matrices=False*)

Separate tensor in the middle, collecting legs as [0, idx) and [idx, ndim), then perform SVD.

Returns:
l, s, r : tuple(SparseTensor)

tensordot(*b*, *axes=2*)

to_dense(*infos=None*)

to_sliceable(*infos=None*)

to_sparse()

transpose(*axes=None*)

static truncate_svd(*l*, *s*, *r*, *max_bond_dim=-1*, *cutoff=0.0*, *max_dw=0.0*, *norm_cutoff=0.0*, *eigen_values=False*)

Truncate tensors obtained from SVD.

Args:

l, s, r
[tuple(SparseTensor)] SVD tensors.

max_bond_dim
[int] Maximal total bond dimension. If *k == -1*, no restriction in total bond dimension.

cutoff
[double] Minimal kept singular value.

max_dw
[double] Maximal sum of square of discarded singular values.

norm_cutoff
[double] Blocks with norm smaller than norm_cutoff will be deleted.

eigen_values
[bool] If True, treat *s* as eigenvalues.

Returns:

l, s, r
[tuple(SparseTensor)] SVD decomposition.

error
[float] Truncation error (same unit as singular value squared).

`unfuse`(*i*, *info*)

Unfuse one leg to several legs.

Args:

i
[int] index of the leg to be unfused. The new unfused indices will be i, i + 1, ...

info
[BondFusingInfo] Indicating how quantum numbers are collected.

`static zeros`(*bond_infos*, *pattern=None*, *dq=None*, *dtype=<class 'float'>*)

Create tensor from tuple of BondInfo with zero elements.

`class pyblock3.algebra.core.SubTensor`(*reduced*, *q_labels=None*)

Bases: `ndarray`

A block in block-sparse tensor.

Attributes:

q_labels
[tuple(SZ..)] Quantum labels for this sub-tensor block. Each element in the tuple corresponds one leg of the tensor.

`conj`()

`copy`()

`diag`()

`property imag`

`norm`()

`classmethod ones`(*shape*, *q_labels=None*, *dtype=<class 'float'>*)

`classmethod random`(*shape*, *q_labels=None*, *dtype=<class 'float'>*)

`property real`

tensordot(*b*, *axes=2*)

transpose(*axes=None*)

classmethod zeros(*shape*, *q_labels=None*, *dtype=<class 'float'>*)

pyblock3.algebra.core.**implements**(*np_func*)
Wrapper for overwritting numpy methods.

pyblock3.algebra.core.**method_alias**(*name*)
Make method callable from algebra.funcs.

2.1.2 pyblock3.algebra.flat

2.1.3 pyblock3.algebra.mpe

Partially contracted tensor network (MPE) with methods for DMRG-like sweep algorithm.

CachedMPE enables swapping with disk storage to reduce memory requirement.

class pyblock3.algebra.mpe.**CachedMPE**(*bra*, *mpo*, *ket*, *opts=None*, *do_canon=True*, *idents=None*, *tag='MPE'*, *scratch=None*, *maxsize=3*, *mpi=False*)

Bases: *MPE*

MPE for large system. Using disk storage to reduce memory usage.

copy()

property nbytes

class pyblock3.algebra.mpe.**MPE**(*bra*, *mpo*, *ket*, *opts=None*, *do_canon=True*, *idents=None*, *mpi=False*)

Bases: object

Matrix Product Expectation (MPE). Original and partially contracted tensor network <bra|mpo|ket>.

build_envs(*l=0*, *r=2*)
Canonicalize bra and ket around sites [l, r). Contract mpo around sites [l, r).

build_envs_no_contract(*l=0*, *r=2*)
Canonicalize bra and ket around sites [l, r).

copy()

copy_shell(*bra*, *mpo*, *ket*)

dmrg(*bdims*, *noises=None*, *dav_thrds=None*, *n_sweeps=10*, *tol=1e-06*, *max_iter=500*, *dot=2*, *iprint=2*, *forward=True*, ***kwargs*)

eigs(*iprint=False*, *fast=False*, *conv_thrd=1e-07*, *max_iter=500*, *extra_mpes=None*)
Return ground-state energy and ground-state effective MPE.

property expectation
<bra|mpo|ket> for the whole system.

greens_function(*mpo*, *omega*, *eta*, *bdims*, *noises=None*, *cg_thrds=None*, *n_sweeps=10*, *tol=1e-06*, *dot=2*, *iprint=2*)

linear(*bdims*, *noises=None*, *cg_thrds=None*, *n_sweeps=10*, *tol=1e-06*, *dot=2*, *iprint=2*)

multiply(*fast=False*)

property n_sites

property nbytes

rk4(*dt*, *fast=False*, *eval_ener=True*)

solve_gf(*ket*, *omega*, *eta*, *iprint=False*, *fast=False*, *conv_thrd=1e-07*)

tddmrg(*bdims*, *dt*, *n_sweeps=10*, *n_sub_sweeps=2*, *dot=2*, *iprint=2*, *forward=True*, *normalize=True*, ***kwargs*)

pyblock3.algebra.mpe.**implements**(*np_func*)

2.1.4 pyblock3.algebra.mps

1D tensor network for MPS/MPO.

class pyblock3.algebra.mps.**MPS**(**args: Any*, ***kwargs: Any*)

Bases: NDArrayOperatorsMixin

Matrix Product State / Matrix Product Operator.

Attributes:

tensors
[list(SparseTensor/FermionTensor)] A list of block-sparse tensors.

n_sites
[int] Number of sites.

const
[float] Constant term.

opts
[dict or None] Options indicating how bond dimension truncation should be done after MPO @ MPS, etc. Possible options are: max_bond_dim, cutoff, max_dw, norm_cutoff

dq
[SZ] Delta quantum of MPO operator

property T

amplitude(*det*)

Return overlap <MPS|det>. MPS tensors must be sliceable.

property bond_dim

canonicalize(*center*)

MPS canonicalization.

Args:

center
[int] Site index of canonicalization center.

compress(***opts*)

MPS bond dimension compression.

Args:

max_bond_dim
[int] Maximal total bond dimension. If *k == -1*, no restriction in total bond dimension.

cutoff
[double] Minimal kept singluar value.

max_dw
[double] Maximal sum of square of discarded singluar values.

conj()

copy()

dot(*b*, *out=None*)

property dtype

fix_pattern(*pattern=None*)

get_bipartite_entanglement()

matmul(*b*, *out=None*)

property n_sites

Number of sites

norm()

classmethod ones(*info*, *dtype=<class 'float'>*, *opts=None*)

Construct unfused MPS from MPSInfo, with identity matrix elements.

classmethod random(*info*, *low=0*, *high=1*, *dtype=<class 'float'>*, *opts=None*)

Construct unfused MPS from MPSInfo, with random matrix elements.

show_bond_dims()

simplify()

Reduce virtual bond dimensions for symbolic sparse tensors. Only works when tensor is SparseSymbolicTensor.

symmetry_fuse(*symm_map*, *info=None*)

to_ad_sparse()

to_flat()

to_non_flat()

to_sliceable(*info=None*)

Get a shallow copy of MPS with SliceableTensor.

Args:

info
[MPSInfo, optional] MPSInfo containing the complete basis BondInfo. If not specified, the BondInfo will be generated from the MPS, which may be incomplete.

to_sparse()

to_symbolic()

classmethod zeros(*info*, *dtype=<class 'float'>*, *opts=None*)

Construct unfused MPS from MPSInfo, with zero matrix elements.

class pyblock3.algebra.mps.**MPSInfo**(*n_sites*, *vacuum*, *target*, *basis*)

Bases: object

BondInfo in every site in MPS (a) For constrution of initial MPS. (b) For tracking basis info in construction of SliceableTensor.

Attributes:

n_sites
[int] Number of sites

vacuum
[SZ] vacuum state

target
[SZ] target state

basis
[list(BondInfo)] BondInfo in each site

left_dims
[list(BondInfo)] Truncated states for left block

right_dims
[list(BondInfo)] Truncated states for right block

set_bond_dimension(*bond_dim*, *call_back=None*)

Truncated bond dimension based on FCI quantum numbers each FCI quantum number has at least one state kept

set_bond_dimension_fci(*call_back=None*)

FCI bond dimensions

set_bond_dimension_occ(*bond_dim*, *occ*, *bias=1*)

bond dimensions from occupation numbers

set_bond_dimension_thermal_limit()

Set bond dimension for MPS at thermal-limit state in ancilla approach.

pyblock3.algebra.mps.**implements**(*np_func*)

2.1.5 pyblock3.algebra.symmetry

Definition of quantum numbers.

class pyblock3.algebra.symmetry.**BondFusingInfo**(**args*, ***kwargs*)

Bases: *BondInfo*

collection of quantum labels with quantum label information for fusing/unfusing

Attributes:

self
[Counter] dict of quantum label and number of states

finfo
[dict(SZ -> dict(tuple(SZ) -> (int, tuple(int))))] For each fused q and unfused q, the starting index in fused dim and the shape of unfused block

pattern
[str] a str of '+'/'-' indicating how quantum numbers are combined

`static get_symmetry_fusing_info(info, symm_map)`

Fusing info for tranfrom to lower symmetry.

Args:

info
[BondInfo] BondInfo at higher symmetry

symm_map
[lambda h: l] Map from higher symemtry irrep to lower symmetry irrep

`static kron_sum(items, ref=None, pattern=None)`

Direct sum of combination of quantum numbers.

Args:

items
[list((tuple(SZ), tuple(int)))] The items to be summed. For every item, the q_labels and matrix shape are given. Repeated items are okay (will not be considered).

ref
[BondInfo (optional)] Reference fused BondInfo.

`static tensor_product(*infos, ref=None, pattern=None, trans=None)`

Direct product of a collection of BondInfo.

Args:

infos
[tuple(BondInfo)] BondInfo for each unfused leg.

ref
[BondInfo (optional)] Reference fused BondInfo.

`class pyblock3.algebra.symmetry.BondInfo(*args, **kwargs)`

Bases: `Counter`

collection of quantum labels

Attributes:

self
[Counter] dict of quantum label and number of bonds

`filter(other)`

`item()`

`keep_maximal()`

`property n_bonds`

Total number of bonds.

`property symm_class`

static tensor_product(*a*, *b*, *ref=None*)

truncate(*bond_dim*, *ref=None*)

truncate_no_keep(*bond_dim*, *ref=None*)

class pyblock3.algebra.symmetry.**SZ**(*n=0*, *twos=0*, *pg=0*)

Bases: object

non-spin-adapted spin label

static from_flat(*x*)

property is_fermion

static is_flat_fermion(*x*)

to_flat()

2.2 pyblock3.algorithms

2.2.1 pyblock3.algorithms.core

Common methods for sweep algorithms.

class pyblock3.algorithms.core.**DecompositionTypes**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

DensityMatrix = 1

SVD = 2

class pyblock3.algorithms.core.**NoiseTypes**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Perturbative = 1

Random = 2

class pyblock3.algorithms.core.**SweepAlgorithm**(*cutoff=1e-14*, *decomp_type=DecompositionTypes.DensityMatrix*, *noise_type=NoiseTypes.Perturbative*, *mpi=False*)

Bases: object

add_dm_noise(*dm*, *mpo*, *wfn*, *noise*, *forward*)

add_wfn_noise(*wfn*, *noise*, *forward*)

decomp_two_site(*mpo*, *wfns*, *forward*, *noise*, *bond_dim*, *weights=None*)

pyblock3.algorithms.core.**fmt_size**(*i*, *suffix='B'*)

2.2.2 pyblock3.algorithms.dmrg

class pyblock3.algorithms.dmrg.**DMRG**(*mpe*, *bdims*, *noises=None*, *dav_thrds=None*, *max_iter=500*, *iprint=2*, *cutoff=1e-14*, *extra_mpes=None*, *weights=None*, *init_site=None*)

Bases: *SweepAlgorithm*

Density Matrix Renormalization Group (DMRG).

solve(*n_sweeps=10*, *tol=1e-06*, *dot=2*, *forward=True*)

2.2.3 pyblock3.algorithms.green

class pyblock3.algorithms.green.**GreensFunction**(*mpe*, *mpo*, *omega*, *eta*, *bdims*, *noises=None*, *cg_thrds=None*, *iprint=2*)

Bases: *SweepAlgorithm*

DDMRG++ for solving Green's function.

solve(*n_sweeps=10*, *tol=1e-06*, *dot=2*)

2.2.4 pyblock3.algorithms.linear

class pyblock3.algorithms.linear.**Linear**(*mpe*, *bdims*, *noises=None*, *cg_thrds=None*, *iprint=2*)

Bases: *SweepAlgorithm*

Solving linear equation or compression in sweeps.

solve(*n_sweeps=10*, *tol=1e-06*, *dot=2*)

2.2.5 pyblock3.algorithms.tddmrg

class pyblock3.algorithms.tddmrg.**TDDMRG**(*mpe*, *bdims*, *iprint=2*, ***kwargs*)

Bases: *SweepAlgorithm*

Time-step targetting td-DMRG approach.

solve(*dt*, *n_sweeps=10*, *n_sub_sweeps=2*, *dot=2*, *forward=True*, *normalize=True*)

2.3 pyblock3.hamiltonian

2.3.1 pyblock3.hamiltonian

Quantum chemistry/general Hamiltonian object. For construction of MPS/MPO.

class pyblock3.hamiltonian.**Hamiltonian**(*fcidump*, *flat=False*)

Bases: `object`

Quantum chemistry/general Hamiltonian. For construction of MPS/MPO

Attributes:

basis
[list(BondInfo)] BondInfo in each site

orb_sym
[list(int)] Point group symmetry for each site

n_sites
[int] Number of sites

n_syms
[int] Number of Point group symmetries

fcidump
[FCIDUMP] one-electron and two-electron file

vacuum
[SZ] vacuum state

target
[SZ] target state

`build_ancilla_mpo`(*mpo*, *left=False*)

`build_ancilla_mps`(*target=None*)

`build_complex_mps`(*bond_dim*, *target=None*, *occ=None*, *bias=1*)

`build_complex_qc_mpo`(*cutoff=1e-12*, *max_bond_dim=-1*)

`build_identity_mpo`()

`build_mpo`(*gen*, *cutoff=1e-12*, *max_bond_dim=-1*, *const=0*)

`build_mps`(*bond_dim*, *target=None*, *occ=None*, *bias=1*, *dtype=<class 'float'>*)

`build_qc_mpo`()

`build_site_mpo`(*op*, *k=-1*)

`get_site_ops`(*m*, *op_names*, *cutoff=1e-20*)
Get dict for matrix representation of site operators in mat (used for SparseSymbolicTensor.ops)

2.3.2 pyblock3.fcidump

One-body/two-body integral object.

`class pyblock3.fcidump.FCIDUMP`(*pg='c1'*, *n_sites=0*, *n_elec=0*, *twos=0*, *ipg=0*, *uhf=False*, *h1e=None*, *g2e=None*, *orb_sym=None*, *const_e=0*, *mu=0*, *general=False*)

Bases: `object`

`build`(*gen*)

`parallelize`(*mpi=True*)

`read`(*filename*)
Read FCI options and integrals from FCIDUMP file.

Args:
filename : str

`t`(*s*, *i*, *j*)

v(*sij*, *skl*, *i*, *j*, *k*, *l*)

write(*filename*, *tol=1e-13*)

Write FCI options and integrals to FCIDUMP file.

Args:

filename : str tol : threshold for terms written into file

pyblock3.fcidump.**parallelize_g2e**(*n_sites*, *mrank*, *msize*, *g2e*)

pyblock3.fcidump.**parallelize_h1e**(*n_sites*, *mrank*, *msize*, *h1e*)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z