simple-dmrg Documentation

Release 1.0

James R. Garrison and Ryan V. Mishmash

Nov 03, 2017

Contents

1 Authors **3**

2 Contents **5**

2.1 Using the code 5
2.2 Exercises 6
2.3 Python cheatsheet 7
2.4 Additional information on DMRG 13
2.5 Source code 13

Source code: https://github.com/simple-dmrg/simple-dmrg/

Documentation: http://simple-dmrg.readthedocs.org/

The goal of this tutorial (given at the 2013 summer school on quantum spin liquids, in Trieste, Italy) is to present the density-matrix renormalization group (DMRG) in its traditional formulation (i.e. without using matrix product states). DMRG is a numerical method that allows for the efficient simulation of quantum model Hamiltonians. Since it is a low-entanglement approximation, it often works quite well for one-dimensional systems, giving results that are nearly exact.

Typical implementations of DMRG in C++ or Fortran can be tens of thousands of lines long. Here, we have attempted to strike a balance between clear, simple code, and including many features and optimizations that would exist in a production code. One thing that helps with this is the use of Python. We have tried to write the code in a very explicit style, hoping that it will be (mostly) understandable to somebody new to Python. (See also the included *Python cheatsheet*, which lists many of the Python features used by `simple-dmrg`, and which should be helpful when trying the included *exercises*.)

The four modules build up DMRG from its simplest implementation to more complex implementations and optimizations. Each file adds lines of code and complexity compared with the previous version.

1. *Infinite system algorithm* (~180 lines, including comments)
2. *Finite system algorithm* (~240 lines)
3. *Conserved quantum numbers* (~310 lines)
4. *Eigenstate prediction* (~370 lines)

Throughout the tutorial, we focus on the spin-1/2 Heisenberg XXZ model, but the code could easily be modified (or expanded) to work with other models.

CHAPTER 1

Authors

- James R. Garrison (UCSB)
- Ryan V. Mishmash (UCSB)

CHAPTER 2

Contents

2.1 Using the code

The requirements are:

- Python 2.6 or higher (Python 3 works as well)
- numpy and scipy

Download the code using the Download ZIP button on github, or run the following command from a terminal:

```
$ wget -O simple-dmrg-master.zip https://github.com/simple-dmrg/simple-dmrg/archive/
↪master.zip
```

Within a terminal, execute the following to unpack the code:

```
$ unzip simple-dmrg-master.zip
$ cd simple-dmrg-master/
```

Once the relevant software is installed, each program is contained entirely in a single file. The first program, for instance, can be run by issuing:

```
$ python simple_dmrg_01_infinite_system.py
```

Note: If you see an error that looks like this:

```
SyntaxError: future feature print_function is not defined
```

then you are using a version of Python below 2.6. Although it would be best to upgrade, it may be possible to make the code work on Python versions below 2.6 without much trouble.

2.2 Exercises

2.2.1 Day 1

1. Consider a reduced density matrix ρ corresponding to a maximally mixed state in a Hilbert space of dimension md. Compute the truncation error associated with keeping only the largest m eigenvectors of ρ. Fortunately, the reduced density matrix eigenvalues for ground states of local Hamiltonians decay much more quickly!
2. Explore computing the ground state energy of the Heisenberg model using the infinite system algorithm. The exact Bethe ansatz result in the thermodynamic limit is $E/L = 0.25 - \ln 2 = -0.443147$. Note the respectable accuracy obtained with an extremely small block basis of size $m \sim 10$. Why does the DMRG work so well in this case?
3. Entanglement entropy:
   (a) Calculate the bipartite (von Neumann) entanglement entropy at the center of the chain during the infinite system algorithm. How does it scale with L?
   (b) Now, using the finite system algorithm, calculate the bipartite entanglement entropy for every bipartite splitting. How does it scale with subsystem size x?

   Hint: To create a simple plot in python:

```
>>> from matplotlib import pyplot as plt
>>> x_values = [1, 2, 3, 4]
>>> y_values = [4, 2, 7, 3]
>>> plt.plot(x_values, y_values)
>>> plt.show()
```

   (c) From the above, estimate the central charge c of the "Bethe phase" (1D quasi-long-range Néel phase) of the 1D Heisenberg model, and in light of that, think again about your answer to the last part of exercise 2.

   The formula for fitting the central charge on a system with open boundary conditions is:

$$S = \frac{c}{6} \ln \left[\frac{L}{\pi} \sin \left(\frac{\pi x}{L} \right) \right] + A$$

   where S is the von Neumann entropy.

   Hint: To fit a line in python:

```
>>> x_values = [1, 2, 3, 4]
>>> y_values = [-4, -2, 0, 2]
>>> slope, y_intercept = np.polyfit(x_values, y_values, 1)
```

4. XXZ model:
   (a) Change the code (ever so slightly) to accommodate spin-exchange anisotropy: $H = \sum_{\langle ij \rangle} \left[\frac{J}{2}(S_i^+ S_j^- + \text{h.c.}) + J_z S_i^z S_j^z \right]$.
   (b) For $J_z/J > 1$ ($J_z/J < -1$), the ground state is known to be an Ising antiferromagnet (ferromagnet), and thus fully gapped. Verify this by investigating scaling of the entanglement entropy as in exercise 3. What do we expect for the central charge in this case?

2.2.2 Day 2

1. Using `simple_dmrg_03_conserved_quantum_numbers.py`, calculate the "spin gap" $E_0(S_z = 1) - E_0(S_z = 0)$. How does the gap scale with $1/L$? Think about how you would go about computing the spectral gap in the $S_z = 0$ sector: $E_1(S_z = 0) - E_0(S_z = 0)$, i.e., the gap between the ground state and first excited state *within* the $S_z = 0$ sector.
2. Calculate the total weight of each S_z sector in the enlarged system block after constructing each block of ρ. At this point, it's important to fully understand *why* ρ is indeed block diagonal, with blocks labeled by the total quantum number S_z for the enlarged system block.
3. Starting with `simple_dmrg_02_finite_system.py`, implement a spin-spin correlation function measurement of the free two sites at each step in the finite system algorithm, i.e., calculate $\langle \vec{S}_i \cdot \vec{S}_{i+1} \rangle$ for all i. In exercise 3 of yesterday's tutorial, you should have noticed a strong period-2 oscillatory component of the entanglement entropy. With your measurement of $\langle \vec{S}_i \cdot \vec{S}_{i+1} \rangle$, can you now explain this on physical grounds?

   Answer: `finite_system_algorithm(L=20, m_warmup=10, m_sweep_list=[10, 20, 30, 40, 40])` with $J = J_z = 1$ should give $\langle \vec{S}_{10} \cdot \vec{S}_{11} \rangle = -0.363847565413$ on the last step.
4. Implement the "ring term" $H_{\text{ring}} = K \sum_i S_i^z S_{i+1}^z S_{i+2}^z S_{i+3}^z$. Note that this term is one of the pieces of the SU(2)-invariant four-site ring-exchange operator for sites (i, $i+1$, $i+2$, $i+3$), a term which is known to drive the J_1-J_2 Heisenberg model on the two-leg triangular strip into a quasi-1D descendant of the spinon Fermi sea ("spin Bose metal") spin liquid [see http://arxiv.org/abs/0902.4210].

   Answer: `finite_system_algorithm(L=20, m_warmup=10, m_sweep_list=[10, 20, 30, 40, 40])` with $K = J = 1$, should give $E/L = -0.40876250668$.

2.3 Python cheatsheet

[designed specifically for understanding and modifying simple-dmrg]

For a programmer, the standard, online Python tutorial is quite nice. Below, we try to mention a few things so that you can get acquainted with the `simple-dmrg` code as quickly as possible.

Python includes a few powerful internal data structures (lists, tuples, and dictionaries), and we use `numpy` (numeric python) and `scipy` (additional "scientific" python routines) for linear algebra.

2.3.1 Basics

Unlike many languages where blocks are denoted by braces or special `end` statements, blocks in python are denoted by indentation level. Thus indentation and whitespace are significant in a python program.

It is possible to execute python directly from the commandline:

```
$ python
```

This will bring you into python's real-eval-print loop (REPL). From here, you can experiment with various commands and expressions. The examples below are taken from the REPL, and include the prompts (">>>" and "`...`") one would see there.

2.3.2 Lists, tuples, and loops

The basic sequence data types in python are lists and tuples.

A `list` can be constructed literally:

```
>>> x_list = [2, 3, 5, 7]
```

and a number of operations can be performed on it:

```
>>> len(x_list)
4

>>> x_list.append(11)
>>> x_list
[2, 3, 5, 7, 11]

>>> x_list[0]
2

>>> x_list[0] = 0
>>> x_list
[0, 3, 5, 7, 11]
```

Note, in particular, that python uses indices counting from zero, like C (but unlike Fortran and Matlab).

A `tuple` in python acts very similarly to a list, but once it is constructed it cannot be modified. It is constructed using parentheses instead of brackets:

```
>>> x_tuple = (2, 3, 5, 7)
```

Lists and tuples can contain any data type, and the data type of the elements need not be consistent:

```
>>> x = ["hello", 4, 8, (23, 12)]
```

It is also possible to get a subset of a list (e.g. the first three elements) by using Python's slice notation:

```
>>> x = [2, 3, 5, 7, 11]
>>> x[:3]
[2, 3, 5]
```

Looping over lists and tuples

Looping over a `list` or `tuple` is quite straightforward:

```
>>> x_list = [5, 7, 9, 11]
>>> for x in x_list:
...     print(x)
...
5
7
9
11
```

If you wish to have the corresponding indices for each element of the list, the `enumerate()` function will provide this:

```
>>> x_list = [5, 7, 9, 11]
>>> for i, x in enumerate(x_list):
...     print(i, x)
...
0 5
```

```
1 7
2 9
3 11
```

If you have two (or more) parallel arrays with the same number of elements and you want to loop over each of them at once, use the `zip()` function:

```
>>> x_list = [2, 3, 5, 7]
>>> y_list = [12, 13, 14, 15]
>>> for x, y in zip(x_list, y_list):
...     print(x, y)
...
2 12
3 13
5 14
7 15
```

There is a syntactic shortcut for transforming a list into a new one, known as a list comprehension:

```
>>> primes = [2, 3, 5, 7]
>>> doubled_primes = [2 * x for x in primes]
>>> doubled_primes
[4, 6, 10, 14]
```

2.3.3 Dictionaries

Dictionaries are python's powerful mapping data type. A number, string, or even a tuple can be a key, and any data type can be the corresponding value.

Literal construction syntax:

```
>>> d = {2: "two", 3: "three"}
```

Lookup syntax:

```
>>> d[2]
'two'
>>> d[3]
'three'
```

Modifying (or creating) elements:

```
>>> d[4] = "four"
>>> d
{2: 'two', 3: 'three', 4: 'four'}
```

The method `get()` is another way to lookup an element, but returns the special value `None` if the key does not exist (instead of raising an error):

```
>>> d.get(2)
'two'
>>> d.get(4)
```

Looping over dictionaries

Looping over the keys of a dictionary:

```
>>> d = {2: "two", 3: "three"}
>>> for key in d:
...     print(key)
...
2
3
```

Looping over the values of a dictionary:

```
>>> d = {2: "two", 3: "three"}
>>> for value in d.values():
...     print(value)
...
two
three
```

Looping over the keys and values, together:

```
>>> d = {2: "two", 3: "three"}
>>> for key, value in d.items():
...     print(key, value)
...
2 two
3 three
```

2.3.4 Functions

Function definition in python uses the `def` keyword:

```
>>> def f(x):
...     y = x + 2
...     return 2 * y + x
...
```

Function calling uses parentheses, along with any arguments to be passed:

```
>>> f(2)
10
>>> f(3)
13
```

When calling a function, it is also possibly to specify the arguments by name (e.g. `x=4`):

```
>>> f(x=4)
16
```

An alternative syntax for writing a one-line function is to use python's `lambda` keyword:

```
>>> g = lambda x: 3 * x
>>> g(5)
15
```

2.3.5 numpy arrays

`numpy` provides a multi-dimensional array type. Unlike lists and tuples, `numpy` arrays have fixed size and hold values of a single data type. This allows the program to perform operations on large arrays very quickly.

Literal construction of a 2x2 matrix:

```
>>> np.array([[1, 2], [3, 4]], dtype='d')
array([[ 1.,  2.],
       [ 3.,  4.]])
```

Note that `dtype='d'` specifies that the type of the array should be double-precision (real) floating point.

It is also possibly to construct an array of all zeros:

```
>>> np.zeros([3, 4], dtype='d')
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

And then elements can be added one-by-one:

```
>>> x = np.zeros([3, 4], dtype='d')
>>> x[1, 2] = 12
>>> x[1, 3] = 18
>>> x
array([[  0.,   0.,   0.,   0.],
       [  0.,   0.,  12.,  18.],
       [  0.,   0.,   0.,   0.]])
```

It is possible to access a given row or column by index:

```
>>> x[1, :]
array([  0.,   0.,  12.,  18.])
>>> x[:, 2]
array([  0.,  12.,   0.])
```

or to access multiple columns (or rows) at once:

```
>>> col_indices = [2, 1, 3]
>>> x[:, col_indices]
array([[  0.,   0.,   0.],
       [ 12.,   0.,  18.],
       [  0.,   0.,   0.]])
```

For matrix-vector (or matrix-matrix) multiplication use the `np.dot()` function:

```
>>> np.dot(m, v)
```

Warning: One tricky thing about `numpy` arrays is that they do not act as matrices by default. In fact, if you multiply two `numpy` arrays, python will attempt to multiply them element-wise!

To take an inner product, you will need to take the transpose-conjugate of the left vector yourself:

```
>>> np.dot(v1.conjugate().transpose(), v2)
```

Array storage order

Although a `numpy` array acts as a multi-dimensional object, it is actually stored in memory as a one-dimensional contiguous array. Roughly speaking, the elements can either be stored column-by-column ("column major", or "Fortran-style") or row-by-row ("row major", or "C-style"). As long as we understand the underlying storage order of an array, we can reshape it to have different dimensions. In particular, the logic for taking a partial trace in `simple-dmrg` uses this reshaping to make the system and environment basis elements correspond to the rows and columns of the matrix, respectively. Then, only a simple matrix multiplication is required to find the reduced density matrix.

2.3.6 Mathematical constants

`numpy` also provides a variety of mathematical constants:

```
>>> np.pi
3.141592653589793
>>> np.e
2.718281828459045
```

2.3.7 Experimentation and getting help

As mentioned above, python's REPL can be quite useful for experimentation and getting familiar with the language. Another thing we can do is to import the `simple-dmrg` code directly into the REPL so that we can experiment with it directly. The line:

```
>>> from simple_dmrg_01_infinite_system import *
```

will execute all lines *except* the ones within the block that says:

```
if __name__ == "__main__":
```

So if we want to use the finite system algorithm, we can (assuming our source tree is in the `PYTHONPATH`, which should typically include the current directory):

```
$ python
>>> from simple_dmrg_04_eigenstate_prediction import *
>>> finite_system_algorithm(L=10, m_warmup=8, m_sweep_list=[8, 8, 8])
```

It is also possible to get help in the REPL by using python's built-in `help()` function on various objects, functions, and types:

```
>>> help(sum)    # help on python's sum function

>>> help([])     # python list methods
>>> help({})     # python dict methods

>>> help({}.setdefault)    # help on a specific dict method

>>> import numpy as np
>>> help(np.log)           # natural logarithm
>>> help(np.linalg.eigh)   # eigensolver for hermitian matrices
```

2.4 Additional information on DMRG

Below is an incomplete list of resources for learning DMRG.

2.4.1 References

- "An introduction to numerical methods in low-dimensional quantum systems" by A. L. Malvezzi (2003) teaches DMRG concisely but in enough detail to understand the `simple-dmrg` code.
- U. Schollwöck has written two review articles on DMRG. The first (from 2005) focuses on DMRG in its traditional formulation, while the second (from 2011) describes it in terms of matrix product states.
- Steve White's papers, including the original DMRG paper (1992), a more in-depth paper (1993) which includes (among other things) periodic boundary conditions, and a later paper (1996) which describes eigenstate prediction, are quite useful.

2.4.2 Links

- The dmrg101 tutorial by Iván González, was prepared for the Taipai DMRG winter school.
- sophisticated-dmrg, a more "sophisticated" program based on this tutorial.

2.5 Source code

Formatted versions of the source code are available in this section. See also the github repository, which contains all the included code.

2.5.1 simple_dmrg_01_infinite_system.py

(Raw download)

```
#!/usr/bin/env python
#
# Simple DMRG tutorial.  This code contains a basic implementation of the
# infinite system algorithm
#
# Copyright 2013 James R. Garrison and Ryan V. Mishmash.
# Open source under the MIT license.  Source code at
# <https://github.com/simple-dmrg/simple-dmrg/>

# This code will run under any version of Python >= 2.6.  The following line
# provides consistency between python2 and python3.
from __future__ import print_function, division  # requires Python >= 2.6

# numpy and scipy imports
import numpy as np
from scipy.sparse import kron, identity
from scipy.sparse.linalg import eigsh  # Lanczos routine from ARPACK

# We will use python's "namedtuple" to represent the Block and EnlargedBlock
# objects
from collections import namedtuple
```

```python

Block = namedtuple("Block", ["length", "basis_size", "operator_dict"])
EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict"])

def is_valid_block(block):
    for op in block.operator_dict.values():
        if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
            return False
    return True

# This function should test the same exact things, so there is no need to
# repeat its definition.
is_valid_enlarged_block = is_valid_block

# Model-specific code for the Heisenberg XXZ chain
model_d = 2  # single-site basis size

Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d')  # single-site S^z
Sp1 = np.array([[0, 1], [0, 0]], dtype='d')  # single-site S^+

H1 = np.array([[0, 0], [0, 0]], dtype='d')  # single-site portion of H is zero

def H2(Sz1, Sp1, Sz2, Sp2):  # two-site part of H
    """Given the operators S^z and S^+ on two sites in different Hilbert spaces
    (e.g. two blocks), returns a Kronecker product representing the
    corresponding two-site term in the Hamiltonian that joins the two sites.
    """
    J = Jz = 1.
    return (
        (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↪transpose(), Sp2)) +
        Jz * kron(Sz1, Sz2)
    )

# conn refers to the connection operator, that is, the operator on the edge of
# the block, on the interior of the chain.  We need to be able to represent S^z
# and S^+ on that site in the current basis in order to grow the chain.
initial_block = Block(length=1, basis_size=model_d, operator_dict={
    "H": H1,
    "conn_Sz": Sz1,
    "conn_Sp": Sp1,
})

def enlarge_block(block):
    """This function enlarges the provided Block by a single site, returning an
    EnlargedBlock.
    """
    mblock = block.basis_size
    o = block.operator_dict

    # Create the new operators for the enlarged block.  Our basis becomes a
    # Kronecker product of the Block basis and the single-site basis.  NOTE:
    # `kron` uses the tensor product convention making blocks of the second
    # array scaled by the first.  As such, we adopt this convention for
    # Kronecker products throughout the code.
    enlarged_operator_dict = {
        "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
↪"conn_Sz"], o["conn_Sp"], Sz1, Sp1),
        "conn_Sz": kron(identity(mblock), Sz1),
        "conn_Sp": kron(identity(mblock), Sp1),
    }

    return EnlargedBlock(length=(block.length + 1),
                         basis_size=(block.basis_size * model_d),
                         operator_dict=enlarged_operator_dict)

def rotate_and_truncate(operator, transformation_matrix):
    """Transforms the operator to the new (possibly truncated) basis given by
    `transformation_matrix`.
    """
    return transformation_matrix.conjugate().transpose().dot(operator.dot(transformation_matrix))

def single_dmrg_step(sys, env, m):
    """Performs a single DMRG step using `sys` as the system and `env` as the
    environment, keeping a maximum of `m` states in the new basis.
    """
    assert is_valid_block(sys)
    assert is_valid_block(env)

    # Enlarge each block by a single site.
    sys_enl = enlarge_block(sys)
    if sys is env:  # no need to recalculate a second time
        env_enl = sys_enl
    else:
        env_enl = enlarge_block(env)

    assert is_valid_enlarged_block(sys_enl)
    assert is_valid_enlarged_block(env_enl)

    # Construct the full superblock Hamiltonian.
    m_sys_enl = sys_enl.basis_size
    m_env_enl = env_enl.basis_size
    sys_enl_op = sys_enl.operator_dict
    env_enl_op = env_enl.operator_dict
    superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) + kron(identity(m_sys_enl), env_enl_op["H"]) + \
                             H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_op["conn_Sz"], env_enl_op["conn_Sp"])

    # Call ARPACK to find the superblock ground state.  ("SA" means find the
    # "smallest in amplitude" eigenvalue.)
    (energy,), psi0 = eigsh(superblock_hamiltonian, k=1, which="SA")

    # Construct the reduced density matrix of the system by tracing out the
    # environment
    #
    # We want to make the (sys, env) indices correspond to (row, column) of a
    # matrix, respectively.  Since the environment (column) index updates most
    # quickly in our Kronecker product structure, psi0 is thus row-major ("C
    # style").
    psi0 = psi0.reshape([sys_enl.basis_size, -1], order="C")
    rho = np.dot(psi0, psi0.conjugate().transpose())

    # Diagonalize the reduced density matrix and sort the eigenvectors by
    # eigenvalue.
    evals, evecs = np.linalg.eigh(rho)
    possible_eigenstates = []
    for eval, evec in zip(evals, evecs.transpose()):
        possible_eigenstates.append((eval, evec))
    possible_eigenstates.sort(reverse=True, key=lambda x: x[0])  # largest eigenvalue␣
↪first

    # Build the transformation matrix from the `m` overall most significant
    # eigenvectors.
    my_m = min(len(possible_eigenstates), m)
    transformation_matrix = np.zeros((sys_enl.basis_size, my_m), dtype='d', order='F')
    for i, (eval, evec) in enumerate(possible_eigenstates[:my_m]):
        transformation_matrix[:, i] = evec

    truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
    print("truncation error:", truncation_error)

    # Rotate and truncate each operator.
    new_operator_dict = {}
    for name, op in sys_enl.operator_dict.items():
        new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)

    newblock = Block(length=sys_enl.length,
                     basis_size=my_m,
                     operator_dict=new_operator_dict)

    return newblock, energy

def infinite_system_algorithm(L, m):
    block = initial_block
    # Repeatedly enlarge the system by performing a single DMRG step, using a
    # reflection of the current block as the environment.
    while 2 * block.length < L:
        print("L =", block.length * 2 + 2)
        block, energy = single_dmrg_step(block, block, m=m)
        print("E/L =", energy / (block.length * 2))

if __name__ == "__main__":
    np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)

    infinite_system_algorithm(L=100, m=20)
```

2.5.2 simple_dmrg_02_finite_system.py

(Raw download)

```
#!/usr/bin/env python
#
# Simple DMRG tutorial.  This code integrates the following concepts:
#  - Infinite system algorithm
#  - Finite system algorithm
#
# Copyright 2013 James R. Garrison and Ryan V. Mishmash.
# Open source under the MIT license.  Source code at
# <https://github.com/simple-dmrg/simple-dmrg/>

```

```python
# This code will run under any version of Python >= 2.6.  The following line
# provides consistency between python2 and python3.
from __future__ import print_function, division  # requires Python >= 2.6

# numpy and scipy imports
import numpy as np
from scipy.sparse import kron, identity
from scipy.sparse.linalg import eigsh  # Lanczos routine from ARPACK

# We will use python's "namedtuple" to represent the Block and EnlargedBlock
# objects
from collections import namedtuple

Block = namedtuple("Block", ["length", "basis_size", "operator_dict"])
EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict"])

def is_valid_block(block):
    for op in block.operator_dict.values():
        if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
            return False
    return True

# This function should test the same exact things, so there is no need to
# repeat its definition.
is_valid_enlarged_block = is_valid_block

# Model-specific code for the Heisenberg XXZ chain
model_d = 2  # single-site basis size

Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d')  # single-site S^z
Sp1 = np.array([[0, 1], [0, 0]], dtype='d')  # single-site S^+

H1 = np.array([[0, 0], [0, 0]], dtype='d')  # single-site portion of H is zero

def H2(Sz1, Sp1, Sz2, Sp2):  # two-site part of H
    """Given the operators S^z and S^+ on two sites in different Hilbert spaces
    (e.g. two blocks), returns a Kronecker product representing the
    corresponding two-site term in the Hamiltonian that joins the two sites.
    """
    J = Jz = 1.
    return (
        (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↪transpose(), Sp2)) +
        Jz * kron(Sz1, Sz2)
    )

# conn refers to the connection operator, that is, the operator on the edge of
# the block, on the interior of the chain.  We need to be able to represent S^z
# and S^+ on that site in the current basis in order to grow the chain.
initial_block = Block(length=1, basis_size=model_d, operator_dict={
    "H": H1,
    "conn_Sz": Sz1,
    "conn_Sp": Sp1,
})

def enlarge_block(block):
    """This function enlarges the provided Block by a single site, returning an
    EnlargedBlock.
    """
    mblock = block.basis_size
    o = block.operator_dict

    # Create the new operators for the enlarged block.  Our basis becomes a
    # Kronecker product of the Block basis and the single-site basis.  NOTE:
    # `kron` uses the tensor product convention making blocks of the second
    # array scaled by the first.  As such, we adopt this convention for
    # Kronecker products throughout the code.
    enlarged_operator_dict = {
        "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o["conn_Sz"], o["conn_Sp"], Sz1, Sp1),
        "conn_Sz": kron(identity(mblock), Sz1),
        "conn_Sp": kron(identity(mblock), Sp1),
    }

    return EnlargedBlock(length=(block.length + 1),
                         basis_size=(block.basis_size * model_d),
                         operator_dict=enlarged_operator_dict)

def rotate_and_truncate(operator, transformation_matrix):
    """Transforms the operator to the new (possibly truncated) basis given by
    `transformation_matrix`.
    """
    return transformation_matrix.conjugate().transpose().dot(operator.dot(transformation_matrix))

def single_dmrg_step(sys, env, m):
    """Performs a single DMRG step using `sys` as the system and `env` as the
    environment, keeping a maximum of `m` states in the new basis.
    """
    assert is_valid_block(sys)
    assert is_valid_block(env)

    # Enlarge each block by a single site.
    sys_enl = enlarge_block(sys)
    if sys is env:  # no need to recalculate a second time
        env_enl = sys_enl
    else:
        env_enl = enlarge_block(env)

    assert is_valid_enlarged_block(sys_enl)
    assert is_valid_enlarged_block(env_enl)

    # Construct the full superblock Hamiltonian.
    m_sys_enl = sys_enl.basis_size
    m_env_enl = env_enl.basis_size
    sys_enl_op = sys_enl.operator_dict
    env_enl_op = env_enl.operator_dict
    superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) + kron(identity(m_sys_enl), env_enl_op["H"]) + \
                             H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_op["conn_Sz"], env_enl_op["conn_Sp"])

    # Call ARPACK to find the superblock ground state.  ("SA" means find the
    # "smallest in amplitude" eigenvalue.)
    (energy,), psi0 = eigsh(superblock_hamiltonian, k=1, which="SA")

    # Construct the reduced density matrix of the system by tracing out the
    # environment
    #
    # We want to make the (sys, env) indices correspond to (row, column) of a
    # matrix, respectively.  Since the environment (column) index updates most
    # quickly in our Kronecker product structure, psi0 is thus row-major ("C
    # style").
    psi0 = psi0.reshape([sys_enl.basis_size, -1], order="C")
    rho = np.dot(psi0, psi0.conjugate().transpose())

    # Diagonalize the reduced density matrix and sort the eigenvectors by
    # eigenvalue.
    evals, evecs = np.linalg.eigh(rho)
    possible_eigenstates = []
    for eval, evec in zip(evals, evecs.transpose()):
        possible_eigenstates.append((eval, evec))
    possible_eigenstates.sort(reverse=True, key=lambda x: x[0])  # largest eigenvalue␣
↪first

    # Build the transformation matrix from the `m` overall most significant
    # eigenvectors.
    my_m = min(len(possible_eigenstates), m)
    transformation_matrix = np.zeros((sys_enl.basis_size, my_m), dtype='d', order='F')
    for i, (eval, evec) in enumerate(possible_eigenstates[:my_m]):
        transformation_matrix[:, i] = evec

    truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
    print("truncation error:", truncation_error)

    # Rotate and truncate each operator.
    new_operator_dict = {}
    for name, op in sys_enl.operator_dict.items():
        new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)

    newblock = Block(length=sys_enl.length,
                     basis_size=my_m,
                     operator_dict=new_operator_dict)

    return newblock, energy

def graphic(sys_block, env_block, sys_label="l"):
    """Returns a graphical representation of the DMRG step we are about to
    perform, using '=' to represent the system sites, '-' to represent the
    environment sites, and '**' to represent the two intermediate sites.
    """
    assert sys_label in ("l", "r")
    graphic = ("=" * sys_block.length) + "**" + ("-" * env_block.length)
    if sys_label == "r":
        # The system should be on the right and the environment should be on
        # the left, so reverse the graphic.
        graphic = graphic[::-1]
    return graphic

def infinite_system_algorithm(L, m):
    block = initial_block
    # Repeatedly enlarge the system by performing a single DMRG step, using a
    # reflection of the current block as the environment.
    while 2 * block.length < L:
        print("L =", block.length * 2 + 2)
        block, energy = single_dmrg_step(block, block, m=m)
        print("E/L =", energy / (block.length * 2))

def finite_system_algorithm(L, m_warmup, m_sweep_list):
    assert L % 2 == 0  # require that L is an even number

    # To keep things simple, this dictionary is not actually saved to disk, but
    # we use it to represent persistent storage.
    block_disk = {}  # "disk" storage for Block objects

    # Use the infinite system algorithm to build up to desired size.  Each time
    # we construct a block, we save it for future reference as both a left
    # ("l") and right ("r") block, as the infinite system algorithm assumes the
    # environment is a mirror image of the system.
    block = initial_block
    block_disk["l", block.length] = block
    block_disk["r", block.length] = block
    while 2 * block.length < L:
        # Perform a single DMRG step and save the new Block to "disk"
        print(graphic(block, block))
        block, energy = single_dmrg_step(block, block, m=m_warmup)
        print("E/L =", energy / (block.length * 2))
        block_disk["l", block.length] = block
        block_disk["r", block.length] = block

    # Now that the system is built up to its full size, we perform sweeps using
    # the finite system algorithm.  At first the left block will act as the
    # system, growing at the expense of the right block (the environment), but
    # once we come to the end of the chain these roles will be reversed.
    sys_label, env_label = "l", "r"
    sys_block = block; del block  # rename the variable
    for m in m_sweep_list:
        while True:
            # Load the appropriate environment block from "disk"
            env_block = block_disk[env_label, L - sys_block.length - 2]
            if env_block.length == 1:
                # We've come to the end of the chain, so we reverse course.
                sys_block, env_block = env_block, sys_block
                sys_label, env_label = env_label, sys_label

            # Perform a single DMRG step.
            print(graphic(sys_block, env_block, sys_label))
            sys_block, energy = single_dmrg_step(sys_block, env_block, m=m)

            print("E/L =", energy / L)

            # Save the block from this step to disk.
            block_disk[sys_label, sys_block.length] = sys_block

            # Check whether we just completed a full sweep.
            if sys_label == "l" and 2 * sys_block.length == L:
                break  # escape from the "while True" loop

if __name__ == "__main__":
    np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)

    #infinite_system_algorithm(L=100, m=20)
    finite_system_algorithm(L=20, m_warmup=10, m_sweep_list=[10, 20, 30, 40, 40])
```

2.5.3 simple_dmrg_03_conserved_quantum_numbers.py

(Raw download)

```
#!/usr/bin/env python
#
# Simple DMRG tutorial.  This code integrates the following concepts:
#  - Infinite system algorithm
#  - Finite system algorithm
#  - Conserved quantum numbers
#
# Copyright 2013 James R. Garrison and Ryan V. Mishmash.
# Open source under the MIT license.  Source code at
# <https://github.com/simple-dmrg/simple-dmrg/>

# This code will run under any version of Python >= 2.6.  The following line
# provides consistency between python2 and python3.
from __future__ import print_function, division  # requires Python >= 2.6

# numpy and scipy imports
import numpy as np
from scipy.sparse import kron, identity, lil_matrix
from scipy.sparse.linalg import eigsh  # Lanczos routine from ARPACK

# We will use python's "namedtuple" to represent the Block and EnlargedBlock
# objects
from collections import namedtuple

Block = namedtuple("Block", ["length", "basis_size", "operator_dict", "basis_sector_
↪array"])
EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict",
↪"basis_sector_array"])

def is_valid_block(block):
    if len(block.basis_sector_array) != block.basis_size:
        return False
    for op in block.operator_dict.values():
        if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
            return False
    return True

# This function should test the same exact things, so there is no need to
# repeat its definition.
is_valid_enlarged_block = is_valid_block

# Model-specific code for the Heisenberg XXZ chain
model_d = 2  # single-site basis size
single_site_sectors = np.array([0.5, -0.5])  # S^z sectors corresponding to the
                                             # single site basis elements

Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d')  # single-site S^z
Sp1 = np.array([[0, 1], [0, 0]], dtype='d')  # single-site S^+

H1 = np.array([[0, 0], [0, 0]], dtype='d')  # single-site portion of H is zero
```

```python

def H2(Sz1, Sp1, Sz2, Sp2):  # two-site part of H
    """Given the operators S^z and S^+ on two sites in different Hilbert spaces
    (e.g. two blocks), returns a Kronecker product representing the
    corresponding two-site term in the Hamiltonian that joins the two sites.
    """
    J = Jz = 1.
    return (
        (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↪transpose(), Sp2)) +
        Jz * kron(Sz1, Sz2)
    )

# conn refers to the connection operator, that is, the operator on the edge of
# the block, on the interior of the chain.  We need to be able to represent S^z
# and S^+ on that site in the current basis in order to grow the chain.
initial_block = Block(length=1, basis_size=model_d, operator_dict={
    "H": H1,
    "conn_Sz": Sz1,
    "conn_Sp": Sp1,
}, basis_sector_array=single_site_sectors)

def enlarge_block(block):
    """This function enlarges the provided Block by a single site, returning an
    EnlargedBlock.
    """
    mblock = block.basis_size
    o = block.operator_dict

    # Create the new operators for the enlarged block.  Our basis becomes a
    # Kronecker product of the Block basis and the single-site basis.  NOTE:
    # `kron` uses the tensor product convention making blocks of the second
    # array scaled by the first.  As such, we adopt this convention for
    # Kronecker products throughout the code.
    enlarged_operator_dict = {
        "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
↪"conn_Sz"], o["conn_Sp"], Sz1, Sp1),
        "conn_Sz": kron(identity(mblock), Sz1),
        "conn_Sp": kron(identity(mblock), Sp1),
    }

    # This array keeps track of which sector each element of the new basis is
    # in.  `np.add.outer()` creates a matrix that adds each element of the
    # first vector with each element of the second, which when flattened
    # contains the sector of each basis element in the above Kronecker product.
    enlarged_basis_sector_array = np.add.outer(block.basis_sector_array, single_site_
↪sectors).flatten()

    return EnlargedBlock(length=(block.length + 1),
                         basis_size=(block.basis_size * model_d),
                         operator_dict=enlarged_operator_dict,
                         basis_sector_array=enlarged_basis_sector_array)

def rotate_and_truncate(operator, transformation_matrix):
    """Transforms the operator to the new (possibly truncated) basis given by
    `transformation_matrix`.
    """
    return transformation_matrix.conjugate().transpose().dot(operator.
↪dot(transformation_matrix))
```

```

def index_map(array):
    """Given an array, returns a dictionary that allows quick access to the
    indices at which a given value occurs.

    Example usage:

    >>> by_index = index_map([3, 5, 5, 7, 3])
    >>> by_index[3]
    [0, 4]
    >>> by_index[5]
    [1, 2]
    >>> by_index[7]
    [3]
    """
    d = {}
    for index, value in enumerate(array):
        d.setdefault(value, []).append(index)
    return d

def single_dmrg_step(sys, env, m, target_Sz):
    """Performs a single DMRG step using `sys` as the system and `env` as the
    environment, keeping a maximum of `m` states in the new basis.
    """
    assert is_valid_block(sys)
    assert is_valid_block(env)

    # Enlarge each block by a single site.
    sys_enl = enlarge_block(sys)
    sys_enl_basis_by_sector = index_map(sys_enl.basis_sector_array)
    if sys is env:  # no need to recalculate a second time
        env_enl = sys_enl
        env_enl_basis_by_sector = sys_enl_basis_by_sector
    else:
        env_enl = enlarge_block(env)
        env_enl_basis_by_sector = index_map(env_enl.basis_sector_array)

    assert is_valid_enlarged_block(sys_enl)
    assert is_valid_enlarged_block(env_enl)

    # Construct the full superblock Hamiltonian.
    m_sys_enl = sys_enl.basis_size
    m_env_enl = env_enl.basis_size
    sys_enl_op = sys_enl.operator_dict
    env_enl_op = env_enl.operator_dict
    superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) +␣
↪kron(identity(m_sys_enl), env_enl_op["H"]) + \
                             H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_
↪op["conn_Sz"], env_enl_op["conn_Sp"])

    # Build up a "restricted" basis of states in the target sector and
    # reconstruct the superblock Hamiltonian in that sector.
    sector_indices = {} # will contain indices of the new (restricted) basis
                        # for which the enlarged system is in a given sector
    restricted_basis_indices = []  # will contain indices of the old (full) basis,␣
↪which we are mapping to
    for sys_enl_Sz, sys_enl_basis_states in sys_enl_basis_by_sector.items():
        sector_indices[sys_enl_Sz] = []
```

```python
        env_enl_Sz = target_Sz - sys_enl_Sz
        if env_enl_Sz in env_enl_basis_by_sector:
            for i in sys_enl_basis_states:
                i_offset = m_env_enl * i  # considers the tensor product structure of the superblock basis
                for j in env_enl_basis_by_sector[env_enl_Sz]:
                    current_index = len(restricted_basis_indices)  # about-to-be-added index of restricted_basis_indices
                    sector_indices[sys_enl_Sz].append(current_index)
                    restricted_basis_indices.append(i_offset + j)

    restricted_superblock_hamiltonian = superblock_hamiltonian[:, restricted_basis_indices][restricted_basis_indices, :]

    # Call ARPACK to find the superblock ground state.  ("SA" means find the
    # "smallest in amplitude" eigenvalue.)
    (energy,), restricted_psi0 = eigsh(restricted_superblock_hamiltonian, k=1, which="SA")

    # Construct each block of the reduced density matrix of the system by
    # tracing out the environment
    rho_block_dict = {}
    for sys_enl_Sz, indices in sector_indices.items():
        if indices: # if indices is nonempty
            psi0_sector = restricted_psi0[indices, :]
            # We want to make the (sys, env) indices correspond to (row,
            # column) of a matrix, respectively.  Since the environment
            # (column) index updates most quickly in our Kronecker product
            # structure, psi0_sector is thus row-major ("C style").
            psi0_sector = psi0_sector.reshape([len(sys_enl_basis_by_sector[sys_enl_Sz]), -1], order="C")
            rho_block_dict[sys_enl_Sz] = np.dot(psi0_sector, psi0_sector.conjugate().transpose())

    # Diagonalize each block of the reduced density matrix and sort the
    # eigenvectors by eigenvalue.
    possible_eigenstates = []
    for Sz_sector, rho_block in rho_block_dict.items():
        evals, evecs = np.linalg.eigh(rho_block)
        current_sector_basis = sys_enl_basis_by_sector[Sz_sector]
        for eval, evec in zip(evals, evecs.transpose()):
            possible_eigenstates.append((eval, evec, Sz_sector, current_sector_basis))
    possible_eigenstates.sort(reverse=True, key=lambda x: x[0])  # largest eigenvalue first

    # Build the transformation matrix from the `m` overall most significant
    # eigenvectors.  It will have sparse structure due to the conserved quantum
    # number.
    my_m = min(len(possible_eigenstates), m)
    transformation_matrix = lil_matrix((sys_enl.basis_size, my_m), dtype='d')
    new_sector_array = np.zeros((my_m,), dtype='d')  # lists the sector of each
                                                     # element of the new/truncated basis
    for i, (eval, evec, Sz_sector, current_sector_basis) in enumerate(possible_eigenstates[:my_m]):
        for j, v in zip(current_sector_basis, evec):
            transformation_matrix[j, i] = v
        new_sector_array[i] = Sz_sector
    # Convert the transformation matrix to a more efficient internal
    # representation.  `lil_matrix` is good for constructing a sparse matrix
    # efficiently, but `csr_matrix` is better for performing quick
    # multiplications.
    transformation_matrix = transformation_matrix.tocsr()

    truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
    print("truncation error:", truncation_error)

    # Rotate and truncate each operator.
    new_operator_dict = {}
    for name, op in sys_enl.operator_dict.items():
        new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)

    newblock = Block(length=sys_enl.length,
                     basis_size=my_m,
                     operator_dict=new_operator_dict,
                     basis_sector_array=new_sector_array)

    return newblock, energy

def graphic(sys_block, env_block, sys_label="l"):
    """Returns a graphical representation of the DMRG step we are about to
    perform, using '=' to represent the system sites, '-' to represent the
    environment sites, and '**' to represent the two intermediate sites.
    """
    assert sys_label in ("l", "r")
    graphic = ("=" * sys_block.length) + "**" + ("-" * env_block.length)
    if sys_label == "r":
        # The system should be on the right and the environment should be on
        # the left, so reverse the graphic.
        graphic = graphic[::-1]
    return graphic

def infinite_system_algorithm(L, m, target_Sz):
    block = initial_block
    # Repeatedly enlarge the system by performing a single DMRG step, using a
    # reflection of the current block as the environment.
    while 2 * block.length < L:
        current_L = 2 * block.length + 2  # current superblock length
        current_target_Sz = int(target_Sz) * current_L // L
        print("L =", current_L)
        block, energy = single_dmrg_step(block, block, m=m, target_Sz=current_target_
↪Sz)
        print("E/L =", energy / current_L)

def finite_system_algorithm(L, m_warmup, m_sweep_list, target_Sz):
    assert L % 2 == 0  # require that L is an even number

    # To keep things simple, this dictionary is not actually saved to disk, but
    # we use it to represent persistent storage.
    block_disk = {}  # "disk" storage for Block objects

    # Use the infinite system algorithm to build up to desired size.  Each time
    # we construct a block, we save it for future reference as both a left
    # ("l") and right ("r") block, as the infinite system algorithm assumes the
    # environment is a mirror image of the system.
    block = initial_block
    block_disk["l", block.length] = block
    block_disk["r", block.length] = block
    while 2 * block.length < L:
        # Perform a single DMRG step and save the new Block to "disk"
        print(graphic(block, block))
        current_L = 2 * block.length + 2  # current superblock length
        current_target_Sz = int(target_Sz) * current_L // L
        block, energy = single_dmrg_step(block, block, m=m_warmup, target_Sz=current_
↪target_Sz)
        print("E/L =", energy / current_L)
        block_disk["l", block.length] = block
        block_disk["r", block.length] = block

    # Now that the system is built up to its full size, we perform sweeps using
    # the finite system algorithm.  At first the left block will act as the
    # system, growing at the expense of the right block (the environment), but
    # once we come to the end of the chain these roles will be reversed.
    sys_label, env_label = "l", "r"
    sys_block = block; del block  # rename the variable
    for m in m_sweep_list:
        while True:
            # Load the appropriate environment block from "disk"
            env_block = block_disk[env_label, L - sys_block.length - 2]
            if env_block.length == 1:
                # We've come to the end of the chain, so we reverse course.
                sys_block, env_block = env_block, sys_block
                sys_label, env_label = env_label, sys_label

            # Perform a single DMRG step.
            print(graphic(sys_block, env_block, sys_label))
            sys_block, energy = single_dmrg_step(sys_block, env_block, m=m, target_
↪Sz=target_Sz)

            print("E/L =", energy / L)

            # Save the block from this step to disk.
            block_disk[sys_label, sys_block.length] = sys_block

            # Check whether we just completed a full sweep.
            if sys_label == "l" and 2 * sys_block.length == L:
                break  # escape from the "while True" loop

if __name__ == "__main__":
    np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)

    #infinite_system_algorithm(L=100, m=20, target_Sz=0)
    finite_system_algorithm(L=20, m_warmup=10, m_sweep_list=[10, 20, 30, 40, 40],␣
↪target_Sz=0)
```

2.5.4 simple_dmrg_04_eigenstate_prediction.py

(Raw download)

```
#!/usr/bin/env python
#
# Simple DMRG tutorial.  This code integrates the following concepts:
```

```python
#   - Infinite system algorithm
#   - Finite system algorithm
#   - Conserved quantum numbers
#   - Eigenstate prediction
#
# Copyright 2013 James R. Garrison and Ryan V. Mishmash.
# Open source under the MIT license.  Source code at
# <https://github.com/simple-dmrg/simple-dmrg/>

# This code will run under any version of Python >= 2.6.  The following line
# provides consistency between python2 and python3.
from __future__ import print_function, division  # requires Python >= 2.6

# numpy and scipy imports
import numpy as np
from scipy.sparse import kron, identity, lil_matrix
from scipy.sparse.linalg import eigsh  # Lanczos routine from ARPACK

# We will use python's "namedtuple" to represent the Block and EnlargedBlock
# objects
from collections import namedtuple

Block = namedtuple("Block", ["length", "basis_size", "operator_dict", "basis_sector_
↪array"])
EnlargedBlock = namedtuple("EnlargedBlock", ["length", "basis_size", "operator_dict",
↪"basis_sector_array"])

def is_valid_block(block):
    if len(block.basis_sector_array) != block.basis_size:
        return False
    for op in block.operator_dict.values():
        if op.shape[0] != block.basis_size or op.shape[1] != block.basis_size:
            return False
    return True

# This function should test the same exact things, so there is no need to
# repeat its definition.
is_valid_enlarged_block = is_valid_block

# Model-specific code for the Heisenberg XXZ chain
model_d = 2  # single-site basis size
single_site_sectors = np.array([0.5, -0.5])  # S^z sectors corresponding to the
                                             # single site basis elements

Sz1 = np.array([[0.5, 0], [0, -0.5]], dtype='d')  # single-site S^z
Sp1 = np.array([[0, 1], [0, 0]], dtype='d')  # single-site S^+

H1 = np.array([[0, 0], [0, 0]], dtype='d')  # single-site portion of H is zero

def H2(Sz1, Sp1, Sz2, Sp2):  # two-site part of H
    """Given the operators S^z and S^+ on two sites in different Hilbert spaces
    (e.g. two blocks), returns a Kronecker product representing the
    corresponding two-site term in the Hamiltonian that joins the two sites.
    """
    J = Jz = 1.
    return (
        (J / 2) * (kron(Sp1, Sp2.conjugate().transpose()) + kron(Sp1.conjugate().
↪transpose(), Sp2)) +
        Jz * kron(Sz1, Sz2)
    )

# conn refers to the connection operator, that is, the operator on the edge of
# the block, on the interior of the chain.  We need to be able to represent S^z
# and S^+ on that site in the current basis in order to grow the chain.
initial_block = Block(length=1, basis_size=model_d, operator_dict={
    "H": H1,
    "conn_Sz": Sz1,
    "conn_Sp": Sp1,
}, basis_sector_array=single_site_sectors)

def enlarge_block(block):
    """This function enlarges the provided Block by a single site, returning an
    EnlargedBlock.
    """
    mblock = block.basis_size
    o = block.operator_dict

    # Create the new operators for the enlarged block.  Our basis becomes a
    # Kronecker product of the Block basis and the single-site basis.  NOTE:
    # `kron` uses the tensor product convention making blocks of the second
    # array scaled by the first.  As such, we adopt this convention for
    # Kronecker products throughout the code.
    enlarged_operator_dict = {
        "H": kron(o["H"], identity(model_d)) + kron(identity(mblock), H1) + H2(o[
↪"conn_Sz"], o["conn_Sp"], Sz1, Sp1),
        "conn_Sz": kron(identity(mblock), Sz1),
        "conn_Sp": kron(identity(mblock), Sp1),
    }

    # This array keeps track of which sector each element of the new basis is
    # in.  `np.add.outer()` creates a matrix that adds each element of the
    # first vector with each element of the second, which when flattened
    # contains the sector of each basis element in the above Kronecker product.
    enlarged_basis_sector_array = np.add.outer(block.basis_sector_array, single_site_
↪sectors).flatten()

    return EnlargedBlock(length=(block.length + 1),
                         basis_size=(block.basis_size * model_d),
                         operator_dict=enlarged_operator_dict,
                         basis_sector_array=enlarged_basis_sector_array)

def rotate_and_truncate(operator, transformation_matrix):
    """Transforms the operator to the new (possibly truncated) basis given by
    `transformation_matrix`.
    """
    return transformation_matrix.conjugate().transpose().dot(operator.
↪dot(transformation_matrix))

def index_map(array):
    """Given an array, returns a dictionary that allows quick access to the
    indices at which a given value occurs.

    Example usage:

    >>> by_index = index_map([3, 5, 5, 7, 3])
    >>> by_index[3]
    [0, 4]
    >>> by_index[5]
    [1, 2]
    >>> by_index[7]
    [3]
    """
    d = {}
    for index, value in enumerate(array):
        d.setdefault(value, []).append(index)
    return d

def single_dmrg_step(sys, env, m, target_Sz, psi0_guess=None):
    """Performs a single DMRG step using `sys` as the system and `env` as the
    environment, keeping a maximum of `m` states in the new basis.  If
    `psi0_guess` is provided, it will be used as a starting vector for the
    Lanczos algorithm.
    """
    assert is_valid_block(sys)
    assert is_valid_block(env)

    # Enlarge each block by a single site.
    sys_enl = enlarge_block(sys)
    sys_enl_basis_by_sector = index_map(sys_enl.basis_sector_array)
    if sys is env:  # no need to recalculate a second time
        env_enl = sys_enl
        env_enl_basis_by_sector = sys_enl_basis_by_sector
    else:
        env_enl = enlarge_block(env)
        env_enl_basis_by_sector = index_map(env_enl.basis_sector_array)

    assert is_valid_enlarged_block(sys_enl)
    assert is_valid_enlarged_block(env_enl)

    # Construct the full superblock Hamiltonian.
    m_sys_enl = sys_enl.basis_size
    m_env_enl = env_enl.basis_size
    sys_enl_op = sys_enl.operator_dict
    env_enl_op = env_enl.operator_dict
    superblock_hamiltonian = kron(sys_enl_op["H"], identity(m_env_enl)) +␣
↪kron(identity(m_sys_enl), env_enl_op["H"]) + \
                             H2(sys_enl_op["conn_Sz"], sys_enl_op["conn_Sp"], env_enl_
↪op["conn_Sz"], env_enl_op["conn_Sp"])

    # Build up a "restricted" basis of states in the target sector and
    # reconstruct the superblock Hamiltonian in that sector.
    sector_indices = {} # will contain indices of the new (restricted) basis
                        # for which the enlarged system is in a given sector
    restricted_basis_indices = []  # will contain indices of the old (full) basis,␣
↪which we are mapping to
    for sys_enl_Sz, sys_enl_basis_states in sys_enl_basis_by_sector.items():
        sector_indices[sys_enl_Sz] = []
        env_enl_Sz = target_Sz - sys_enl_Sz
        if env_enl_Sz in env_enl_basis_by_sector:
            for i in sys_enl_basis_states:
                i_offset = m_env_enl * i  # considers the tensor product structure of␣
↪the superblock basis
                for j in env_enl_basis_by_sector[env_enl_Sz]:
                    current_index = len(restricted_basis_indices)  # about-to-be-
↪added index of restricted_basis_indices
```

```
                    sector_indices[sys_enl_Sz].append(current_index)
                    restricted_basis_indices.append(i_offset + j)

    restricted_superblock_hamiltonian = superblock_hamiltonian[:, restricted_basis_indices][restricted_basis_indices, :]
    if psi0_guess is not None:
        restricted_psi0_guess = psi0_guess[restricted_basis_indices]
    else:
        restricted_psi0_guess = None

    # Call ARPACK to find the superblock ground state.  ("SA" means find the
    # "smallest in amplitude" eigenvalue.)
    (energy,), restricted_psi0 = eigsh(restricted_superblock_hamiltonian, k=1, which="SA", v0=restricted_psi0_guess)

    # Construct each block of the reduced density matrix of the system by
    # tracing out the environment
    rho_block_dict = {}
    for sys_enl_Sz, indices in sector_indices.items():
        if indices: # if indices is nonempty
            psi0_sector = restricted_psi0[indices, :]
            # We want to make the (sys, env) indices correspond to (row,
            # column) of a matrix, respectively.  Since the environment
            # (column) index updates most quickly in our Kronecker product
            # structure, psi0_sector is thus row-major ("C style").
            psi0_sector = psi0_sector.reshape([len(sys_enl_basis_by_sector[sys_enl_Sz]), -1], order="C")
            rho_block_dict[sys_enl_Sz] = np.dot(psi0_sector, psi0_sector.conjugate().transpose())

    # Diagonalize each block of the reduced density matrix and sort the
    # eigenvectors by eigenvalue.
    possible_eigenstates = []
    for Sz_sector, rho_block in rho_block_dict.items():
        evals, evecs = np.linalg.eigh(rho_block)
        current_sector_basis = sys_enl_basis_by_sector[Sz_sector]
        for eval, evec in zip(evals, evecs.transpose()):
            possible_eigenstates.append((eval, evec, Sz_sector, current_sector_basis))
    possible_eigenstates.sort(reverse=True, key=lambda x: x[0])  # largest eigenvalue first

    # Build the transformation matrix from the `m` overall most significant
    # eigenvectors.  It will have sparse structure due to the conserved quantum
    # number.
    my_m = min(len(possible_eigenstates), m)
    transformation_matrix = lil_matrix((sys_enl.basis_size, my_m), dtype='d')
    new_sector_array = np.zeros((my_m,), dtype='d')  # lists the sector of each
                                                     # element of the new/truncated basis
    for i, (eval, evec, Sz_sector, current_sector_basis) in enumerate(possible_eigenstates[:my_m]):
        for j, v in zip(current_sector_basis, evec):
            transformation_matrix[j, i] = v
        new_sector_array[i] = Sz_sector
    # Convert the transformation matrix to a more efficient internal
    # representation.  `lil_matrix` is good for constructing a sparse matrix
    # efficiently, but `csr_matrix` is better for performing quick
    # multiplications.
    transformation_matrix = transformation_matrix.tocsr()

    truncation_error = 1 - sum([x[0] for x in possible_eigenstates[:my_m]])
    print("truncation error:", truncation_error)

    # Rotate and truncate each operator.
    new_operator_dict = {}
    for name, op in sys_enl.operator_dict.items():
        new_operator_dict[name] = rotate_and_truncate(op, transformation_matrix)

    newblock = Block(length=sys_enl.length,
                     basis_size=my_m,
                     operator_dict=new_operator_dict,
                     basis_sector_array=new_sector_array)

    # Construct psi0 (that is, in the full superblock basis) so we can use it
    # later for eigenstate prediction.
    psi0 = np.zeros([m_sys_enl * m_env_enl, 1], dtype='d')
    for i, z in enumerate(restricted_basis_indices):
        psi0[z, 0] = restricted_psi0[i, 0]
    if psi0_guess is not None:
        overlap = np.absolute(np.dot(psi0_guess.conjugate().transpose(), psi0).item())
        overlap /= np.linalg.norm(psi0_guess) * np.linalg.norm(psi0)  # normalize it
        print("overlap |<psi0_guess|psi0>| =", overlap)

    return newblock, energy, transformation_matrix, psi0

def graphic(sys_block, env_block, sys_label="l"):
    """Returns a graphical representation of the DMRG step we are about to
    perform, using '=' to represent the system sites, '-' to represent the
    environment sites, and '**' to represent the two intermediate sites.
    """
    assert sys_label in ("l", "r")
    graphic = ("=" * sys_block.length) + "**" + ("-" * env_block.length)
    if sys_label == "r":
        # The system should be on the right and the environment should be on
        # the left, so reverse the graphic.
        graphic = graphic[::-1]
    return graphic

def infinite_system_algorithm(L, m, target_Sz):
    block = initial_block
    # Repeatedly enlarge the system by performing a single DMRG step, using a
    # reflection of the current block as the environment.
    while 2 * block.length < L:
        current_L = 2 * block.length + 2  # current superblock length
        current_target_Sz = int(target_Sz) * current_L // L
        print("L =", current_L)
        block, energy, transformation_matrix, psi0 = single_dmrg_step(block, block,␣
↪m=m, target_Sz=current_target_Sz)
        print("E/L =", energy / current_L)

def finite_system_algorithm(L, m_warmup, m_sweep_list, target_Sz):
    assert L % 2 == 0  # require that L is an even number

    # To keep things simple, these dictionaries are not actually saved to disk,
    # but they are used to represent persistent storage.
    block_disk = {}  # "disk" storage for Block objects
    trmat_disk = {}  # "disk" storage for transformation matrices

    # Use the infinite system algorithm to build up to desired size.  Each time
    # we construct a block, we save it for future reference as both a left
    # ("l") and right ("r") block, as the infinite system algorithm assumes the
    # environment is a mirror image of the system.
    block = initial_block
    block_disk["l", block.length] = block
    block_disk["r", block.length] = block
    while 2 * block.length < L:
        # Perform a single DMRG step and save the new Block to "disk"
        print(graphic(block, block))
        current_L = 2 * block.length + 2  # current superblock length
        current_target_Sz = int(target_Sz) * current_L // L
        block, energy, transformation_matrix, psi0 = single_dmrg_step(block, block,␣
↪m=m_warmup, target_Sz=current_target_Sz)
        print("E/L =", energy / current_L)
        block_disk["l", block.length] = block
        block_disk["r", block.length] = block

    # Now that the system is built up to its full size, we perform sweeps using
    # the finite system algorithm.  At first the left block will act as the
    # system, growing at the expense of the right block (the environment), but
    # once we come to the end of the chain these roles will be reversed.
    sys_label, env_label = "l", "r"
    sys_block = block; del block  # rename the variable
    sys_trmat = None
    for m in m_sweep_list:
        while True:
            # Load the appropriate environment block from "disk"
            env_block = block_disk[env_label, L - sys_block.length - 2]
            env_trmat = trmat_disk.get((env_label, L - sys_block.length - 1))

            # If possible, predict an estimate of the ground state wavefunction
            # from the previous step's psi0 and known transformation matrices.
            if psi0 is None or sys_trmat is None or env_trmat is None:
                psi0_guess = None
            else:
                # psi0 currently looks e.g. like ===**--- but we need to
                # transform it to look like ====**-- using the relevant
                # transformation matrices and paying careful attention to the
                # tensor product structure.
                #
                # Keep in mind that the tensor product of the superblock is
                # (sys_enl_block, env_enl_block), which is equal to
                # (sys_block, sys_extra_site, env_block, env_extra_site).
                # Note that this does *not* correspond to left-to-right order
                # on the chain.
                #
                # First we reshape the psi0 vector into a matrix with rows
                # corresponding to the enlarged system basis and columns
                # corresponding to the enlarged environment basis.
                psi0_a = psi0.reshape((-1, env_trmat.shape[1] * model_d), order="C")
                # Now we transform the enlarged system block into a system
                # block, so that psi0_b looks like ====*-- (with only one
                # intermediate site).
                psi0_b = sys_trmat.conjugate().transpose().dot(psi0_a)
                # At the moment, the tensor product goes as (sys_block,
                # env_enl_block) == (sys_block, env_block, extra_site), but we
                # need it to look like (sys_enl_block, env_block) ==
                # (sys_block, extra_site, env_block).  In other words, the
                # single intermediate site should now be part of a new enlarged
                # system, not part of the enlarged environment.
                psi0_c = psi0_b.reshape((-1, env_trmat.shape[1], model_d), order="C").
↪transpose(0, 2, 1)
                # Now we reshape the psi0 vector into a matrix with rows
                # corresponding to the enlarged system and columns
                # corresponding to the environment block.
                psi0_d = psi0_c.reshape((-1, env_trmat.shape[1]), order="C")
                # Finally, we transform the environment block into the basis of
                # an enlarged block the so that psi0_guess has the tensor
                # product structure of ====**--.
                psi0_guess = env_trmat.dot(psi0_d.transpose()).transpose().reshape((-
↪1, 1))

            if env_block.length == 1:
                # We've come to the end of the chain, so we reverse course.
                sys_block, env_block = env_block, sys_block
                sys_label, env_label = env_label, sys_label
                if psi0_guess is not None:
                    # Re-order psi0_guess based on the new sys, env labels.
                    psi0_guess = psi0_guess.reshape((sys_trmat.shape[1] * model_d,␣
↪env_trmat.shape[0]), order="C").transpose().reshape((-1, 1))

            # Perform a single DMRG step.
            print(graphic(sys_block, env_block, sys_label))
            sys_block, energy, sys_trmat, psi0 = single_dmrg_step(sys_block, env_
↪block, m=m, target_Sz=target_Sz, psi0_guess=psi0_guess)

            print("E/L =", energy / L)

            # Save the block and transformation matrix from this step to disk.
            block_disk[sys_label, sys_block.length] = sys_block
            trmat_disk[sys_label, sys_block.length] = sys_trmat

            # Check whether we just completed a full sweep.
            if sys_label == "l" and 2 * sys_block.length == L:
                break  # escape from the "while True" loop

if __name__ == "__main__":
    np.set_printoptions(precision=10, suppress=True, threshold=10000, linewidth=300)

    #infinite_system_algorithm(L=100, m=20, target_Sz=0)
    finite_system_algorithm(L=20, m_warmup=10, m_sweep_list=[10, 20, 30, 40, 40],␣
↪target_Sz=0)
```